On the Cognitive Complexity of Software and its Quantification and Formal Measurement

Article *in* International Journal of Software Science and Computational Intelligence · April 2009

DOI: 10.4018/jssci.2009040103 · Source: DBLP

1 author:

Yingxu Wang
The University of Calgary
800 PUBLICATIONS **11,898** CITATIONS

On the Cognitive Complexity of Software and its Quantification and Formal Measurement

Yingxu Wang, University of Calgary, Canada

ABSTRACT

The quantification and measurement of functional complexity of software are a persistent problem in software engineering. Measurement models of software complexities have been studied in two facets in computing and software engineering, where the former is machine-oriented in the small; while the latter is human-oriented in the large. The cognitive complexity of software presented in this paper is a new measurement for cross-platform analysis of complexities, functional sizes, and cognition efforts of software code and specifications 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. A set of ten Basic Control Structures (BCS's) are elicited from software architectural and behavioral modeling and specifications. The cognitive weights of the 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 for software functional complexity and size measurement in software engineering. [Article copies are available for purchase from InfoSci-on-Demand.com]

Keywords: *BCS; Calibrated Weights; Cognitive Complexity; Cognitive Informatics; Cognitive Weights; Complexity Theory; Comparative Studies; Formal Models; Functional Size; Psychological Experiments; Software Engineering*

INTRODUCTION

One of the central problems in software engineering is the inherited complexity. The quantification and measurement of functional complexity of software systems have been a persistent fundamental problem in software engineering (Hartmanis and Stearns, 1965; Basili, 1980; Kearney et al., 1986; Melton, 1996; Fenton and Pfleeger, 1998; Lewis and Papadimitriou, 1998; Wang, 2003b, 2007a). The taxonomy of the complexity and size measures of software can be classified into the categories of computational complexity (time and space) (Hartmanis, 1994; McDermid, 1991), symbolic complexity (Lines of Code (LOC)) (Halstead, 1977; Albrecht and Gaffney, 1983;

McDermid, 1991), structural complexity (control flow, cyclomatic) (McCabe, 1976; Zuse, 1977), functional complexity (function points, cognitive complexity) (Albrecht, 1979; Wang, 2007a; Wang and Shao, 2003).

The most simple and intuitive measure of software complexity is the symbolic complexity, which is conventionally adopted as a measure in term of Lines of Code (LOC) (Halstead, 1977; Albrecht and Gaffney, 1983; McDermid, 1991). However, the functional complexity of software is so intricate and non-linear, which is too hard to be measured or even estimated in LOC. In order to improve the accuracy and measurability, McCabe proposed the cyclomatic complexity measure (McCabe, 1976) based on Euler's theorem (Lipschutz and Lipson, 1997) in the category of structural complexity. However, it only considered the internal loop architectures of software systems without taking into account of the throughput of the system in terms of data objects and many other important internal architectures such as the sequential, branch, and embedded constructs. Because the linear blocks of code are oversimplified as one unit as in graph theory, the cyclomatic complexity is not sensitive to linear structures and external data complexity as well as their impact on the basic structures. Albrecht (1979) introduced the concept of *function point* of software (Albrecht, 1979), which is a weighted product of a set of functional characteristics of software systems. However, the physical meaning of a unit function point is not rigorously modeled except a wide range of empirical studies. The *cognitive complexity* of software systems is introduced as a measure for the functional complexity in both software design and comprehension, which consists of the architectural and operational complexities. The cognitive complexity provides a novel and profound approach to explain and measure the functional complexity of software as well as the effort in software design and comprehension. The new approach perceives software functional complexity as a measure of cognitive complexity for human creative artifacts, which considers the effect of both internal structures of software and the I/O data objects under processing (Wang, 2007a; Wang and Shaw, 2003).

This paper presents a cognitive functional complexity of software as well as its mathematical models and formal measurement. The taxonomy and related work of software complexity measurement are explored systematically. A generic mathematical model of programs is created and the relative cognitive weights of fundamental software structures known as the Basic Control Structures (BCS's) are empirically calibrated based on a series of psychological experiments. The cognitive complexity of software systems is formally modeled as a product of the architectural and operational complexities of software. A set of comparative case studies is presented on applications of cognitive complexity in software engineering, which leads to a series of important findings on the basic properties of software cognitive complexity and its quantification and measurement.

TAXONOMY OF SOFTWARE COMPLEXITIES IN COMPUTING AND SOFTWARE ENGINEERING

The measurement models of software complexities have been studied in two facets in computing in the small and software engineering in the large. The orientation of software engineering complexity theories puts emphases on the problems of *functional complexity* that are human *cognition time* and *workload* oriented. While the computational complexity theories are focused on the problems of *high throughput complexity* that are computing *time efficiency* centered. In other words, software engineering measures system-level complexities, while computational science measures algorithmic complexities.

Although computational complexities, particularly *algorithm complexities*, are one of the focuses in computer science, software engineering is particularly interested in the *functional*

complexity of large-scale and real-world software systems. The computational complexity of algorithms puts emphases on the computability and efficiency of typical algorithms of massive data processing and high throughput systems, in which computing efficiency is dependent on and dominated by the problem sizes in terms of their number of inputs such as those in sorting and searching. However, there are more generic computational problems and software systems that are not dominated by this kind of input sizes rather than by internal architectural and operational complexities such as problem solving and process dispatching. This shows the differences of focuses or problem models in the complexity theories of software engineering and computing.

In software engineering, a problem with very high computational complexity may be quite simple for human comprehension, and vice versa. For instance, according to cognitive informatics (Wang, 2002b, 2003a, 2007b), human beings may comprehend a large loop of iteration, which is the major issue of computational complexity, by looking at only the beginning and termination conditions, and one or a few arbitrary internal loops on the basis of inductive inferences. However, humans are not good at dealing with functional complexities such as a long chain of interrelated operations, very abstract data objects, and their consistency maintenance. Therefore, the system complexity of large-scale software is the focus of software engineering.

A summary of software complexity and size measures is provided in Table 1. It is noteworthy that certain measures in Table 1 are suitable only for post-coding measurement after a software system has already been developed. While the remainder are good at both pre- or post-coding measurements, which may be applied in the processes of design and organizational planning. The following subsections comparatively review and analyze typical paradigms of software complexity and size measurements, such as the computational, symbolic, structural, and cognitive complexities.

Computational Complexity

Computational complexity theory is a fundamental and well established area in computing, which studies the solvability or computational load of algorithms and computing problems (Hartmanis and Stearns, 1965; Hartmanis, 1994; McDermid, 1991; Lewis and Papadimitriou, 1998).

Definition 1. *Computational complexity* is a special category of operational complexity of software that studies the taxonomy of problems in computing and their solvabilities, complexities, and efficiencies.

Computational complexity focuses on algorithm complexities, which can be modeled by its *time* or *space* complexity, particularly the former, proportional to the sizes of computational problems.

Taxonomy of Computational Problems

The *solvable problems* in computation are those that can be computed by *polynomial-time* consumption. The *nonsolvable problems* are those that cannot be solved in any practical sense by computers due to excessive time requirements. The taxonomy of problems in computation can be classified into the following classes. The class of solvable problems that is *polynomial-time* computational by *deterministic* Turing machines are called the *class P* problems. The class of problems that is polynomial-time computational by *nondeterministic* Turing machines are called the *class NP* problems. The class of problems that their answers are complementary to the NP problems is called the *NP complementary* (coNP) problems. The subclass of NP problems that serves as a meta-problem where other NP problems may be reduced to them in polynomial time, is called the *NP-complete* (NPc) problems. There is a special class of problems that can be reduced to known NP problems in polynomial time, which are usually referred to as the *NP-hard*

Table 1. Quantifications and measures of software complexities and sizes

No.	Category of measure	Paradigm	Measurability	Applicable process	Reference
1	Computational	Time complexity	Throughput in loops	Pre- or post-coding	(Lewis and Papadimitriou, 1998)
2		Space complexity	Source and target memory	Pre- or post-coding; language dependent	(Lewis and Papadimitriou, 1998)
3	Symbolic	Symbolic size	Lines of code (LOC)	Post coding; language dependent	(Albrecht and Gaffney, 1983)
4		Numbers of operators	Number of instructions	Post coding; language dependent	(Halstead, 1977)
5	Structural	Control flow complexity	Number of branches/ loops	Pre- or post-coding	(McDermid, 1991)
6		Cyclomatic complexity	Number of control loops	Pre- or post-coding	(McCabe, 1976)
7		Architectural complexity	Number of data objects	Pre- or post-coding	(Wang, 2007a)
8	Data	Number of operands	Number of variables	Post coding; language dependent	(Halstead, 1977)
9		Information complexity	Number of I/Os	Pre- or post-coding	(Zuse, 1997)
10	Functional	Function points	Weighted sum of certain system characteristics	Pre- or post-coding	(Albrecht, 1979)
11		Operational complexity	Weighted sun of system cognitive functions	Pre- or post-coding	(Wang, 2007a)
12		Cognitive complexity	A product of system operational and architectural complexities	Pre- or post-coding	(Wang, 2007a)
13	System	System relational complexity	Number of system relations	Pre- or post-coding	(Wang, 2007a)

(NPh) problems (McDermid, 1991; Lewis and Papadimitriou, 1998).

The relationship among various classes of problems in the taxonomy of computational problems can be illustrated as shown in Fig. 1. It is noteworthy that there are certain classes of problems that are unsolvable or with unknown solvability in computing. However, they might be solvable by human brains on the basis of the inductive power that reduces the large or even infinitive iterative complexity of computing to a simple or finite problem (Wang, 2007a). Therefore, software engineering complexity theories must deal with both machine-oriented computational complexities and human-oriented cognitive complexities. In many cases, these two

categories of complexities are fundamentally different in theories and practice.

Time Complexity of Algorithms

The time complexity of an algorithm for a given problem in computing is measured as an estimation of its computational complexity. The time complexity of an algorithm can be measured by analyzing the number of dominant operations in the algorithm, where each of the dominant operations is assumed to take an identical unit of time in operation.

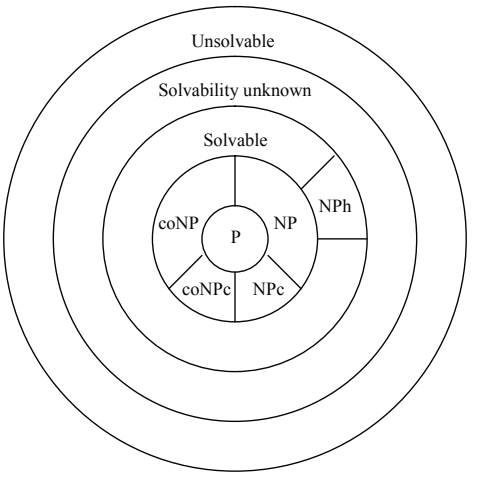
Definition 2. *The dominant operations in an algorithm are those statements within iterative structures that are proportional to the size of the problem or the number of inputs n of the algorithm.*

Definition 3. *For a given function f(x), its asymptotic function $f_a(x)$ is a function that satisfies:*

$$|f(x)| \le k|f_a(x)|, x > b \quad (1)$$

where k and b are a positive constant.

Figure 1. Taxonomy of problems in computation



Definition 4. *If a function f(x) has an asymptotic function $f_a(x)$, the function f(x) is said to be of order of $f_a(x)$, denoted by:*

$$f(x) = O(f_a(x)) \quad (2)$$

where O is known as the big O notation.

Definition 5. *For a given size of a problem, n, the time complexity $C_t(n)$ of an algorithm for solving the problem is a function of the maximum required number of dominant operations $O(f_a(n))$, i.e.:*

$$C_t(n) = O(f_a(n)) \quad (3)$$

where $f_a(n)$ is called the asymptotic function of $C_t(n)$.

It is noteworthy in Definition 5 that the maximum number of dominant operations $C_t(n)$ indicates the *worst case* scenario. An *average case* complexity is a mathematical expectation of $C_t(n)$.

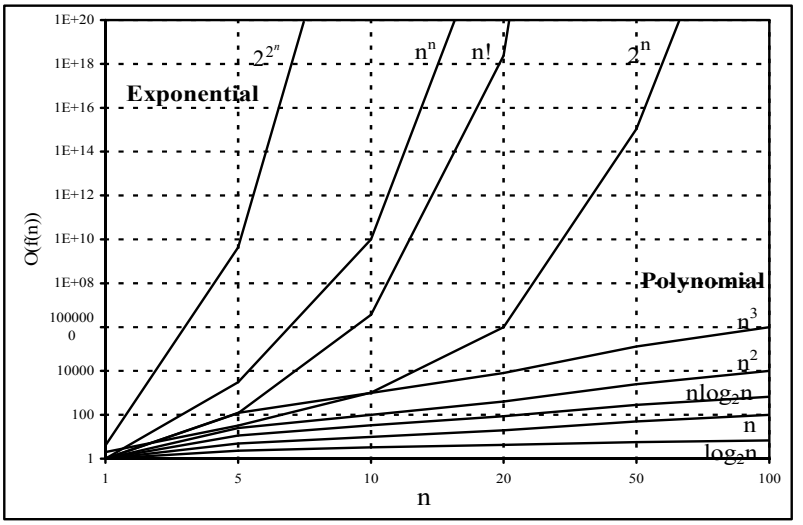
Example 1. *According to Definition 5, the time complexity $C_t(n)$ of the following functions $f_1(n)$ through $f_4(n)$ can be estimated as follows:*

- $f_1(n) = 5n^3 + 2n^2 - 6$
  $\Rightarrow C_{t1}(n) = O(f_{a1}(n)) = O(n^3)$
- $f_2(n) = 3n$
  $\Rightarrow C_{t2}(n) = O(f_{a2}(n)) = O(n)$
- $f_3(n) = 4\log_2 n + 10$
  $\Rightarrow C_{t3}(n) = O(f_{a3}(n)) = O(\log_2 n)$
- $f_4(n) = 2 + 8$
  $\Rightarrow C_{t4}(n) = O(f_{a4}(n)) = O(\varepsilon)$

where ε is a positive constant.

Typical asymptotic functions of algorithms and programs are shown in Fig. 2, where the computational loads in term of processing time of different functions may grow polynomially or exponentially as the size of problem *n*, usually the number of input items, increasing. If an algorithm can be reduced to a type of func-

Figure 2. Typical asymptotic functions of software time complexities



tion with polynomial complexity, it is always a computable problem; otherwise, it would be a very hard problem, particularly in the worst case when *n* is considerably large.

Space Complexity of Algorithms

Definition 6. *The space complexity of an algorithm for a given problem is the maximum required space for both working memory w and target code memory g, i.e.:*

$$C_m(n) = O(f(w+g)) \approx O(f(w)),\ w >> g \qquad (4)$$

where w refers to the memory for data objects under processing such as input/output and intermediate variables, and g refers to the memory for executable code.

Because the target code memory is static and determinable, algorithm space complexity is focused on the dynamic part of working memory complexity.

Symbolic Complexities of Software

The most simple and direct forward complexity measure of software systems is the symbolic complexity that can be represented by the number of lines of statements in a programming language (Halstead, 1977; Albrecht and Gaffney, 1983).

Definition 7. *The symbolic complexity of a software system S, $C_s(S)$, is the linear length of its static statements measured in the unit of Lines of Code (LOC), i.e.:*

$$C_s(S) = \sum_{k=1}^{n_c} C_s(k) \quad [\text{LOC}] \qquad (5)$$

where $C_s(k)$ represents the symbolic complexity of component k in S with n_c components.

It is noteworthy in measurement theory that the symbolic complexity is the measure, but LOC is the unit of the measure. It is in parallel to the measure of distance and its unit in meter. In the measure of symbolic complexity of software, variables, data objects declarations, and comments are not considered as a valid line

of instructions, and an instruction separated in multiple lines is usually counted as a single line. However, there is a lack of a unified standard, and the measure is language-dependent. Therefore, only post-coding complexity and size may be measured by the symbolic complexity. The measurement of design complexity before the implementation of code is impossible.

Control Flow Complexities of Software

Another approach to measure the code complexity of software systems can be via its control structures based on a Control Flow Graph (CFG) (McCabe, 1976; Wang, 2007a). When a program is abstracted by a CFG, well-defined graph theory may be adopted to analyze its complexity properties. For instance, the application of *Euler's theorem* (Lipschutz and Lipson, 1997) to model the complexity of CFGs is proposed by McCabe in 1976 known as the *cyclomatic complexity* of software.

Theorem 1. *Euler's theorem states that the following formula holds for the numbers of nodes n, of edges e, and of regions r for any connected planar graph or map G:*

$$n - e + r = 2 \tag{6}$$

The proof of Theorem 1 may refer to Lipschutz and Lipson (1997).

The McCabe *cyclomatic complexity* (McCabe, 1976) of a software system can be determined by applying Euler's theorem onto the CFGs of software systems.

Definition 8. *The cyclomatic complexity of a software system S, $Cr(S)$, is determined by the number of regions contained in a given CFG G_S, $r(G_S)$, provided that G is connected, i.e.:*

$$Cr(S) = r(G_S) = e - n + 2 \tag{7}$$

where, e is the number of edges in G_S representing branches and cycles, n number of nodes in G_S where a block of sequential code is reduced to a single node.

It can be observed that Eq. 7 is a derived application of Euler's theorem, which shows that the physical meaning of the McCabe cyclomatic complexity is the number of regions in a CFG for a given software system.

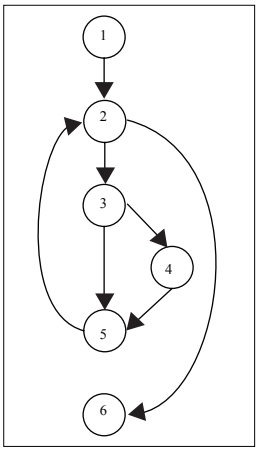
Example 2. *The cyclomatic complexity of the program MaxFinder as given in Fig. 3 can be determined by using Eq. 7 on the CFG as shown in Fig. 3 as follows:*

$$\begin{aligned} Cr(S) &= e - n + 2 \\ &= 7 - 6 + 2 \\ &= 3 \end{aligned}$$

Observing Fig. 3, it may be found that the result $Cr(S) = r(G_S) = 3$ is the number of regions in the CFG, providing there is always a region by linking the first node to the last node in the CFG. This finding indicates that the calculation as required in Eq. 7 may be replaced by a simple count of the number of regions in G_S.

Further, it may also be seen in Fig. 3 that the result $Cr(S) = r(G_S) = 3$ is the number of BCS's (as given in Definition 14) in the program

Figure 3. The CFG of the program MaxFinder



or its formal specification, providing a single sequential BCS is always taken into account by one. This finding reveals that the drawing of a CFG for a given program or software component is not necessary in the cyclomatic analysis. In other words, *r(CFG)* equals to the numbers of BCS's in a program (Wang, 2007a).

Incorporating the above findings with Definition 8, the following corollary is obtained.

Corollary 1. *The cyclomatic complexity of a connected software system Cr(S) can be determined by any of the following three methods:*

$$\begin{aligned} C_r(S) &\triangleq e - n + 2 \quad \text{// Method 1} \\ &= r(CFG) \quad \text{// Method 2} \\ &= \#(\text{BCS}) \quad \text{// Method 3} \end{aligned} \tag{8}$$

Functional Complexities of Software

In software engineering the most concerned complexity of software is the functional complexity. Functional complexity of software may be classified into two categories known as the functional points and cognitive complexity. The former is the functional complexity of software products assuming that a basic or common unit (point) of software function exist or may be definable. The latter is the functional complexity perceived by human beings in software design, implementation, and maintenance, which will be developed in the remaining sections of this paper.

Function points as a paradigm of functional size or complexity measure of software is proposed by Albrecht in 1979 (Albrecht, 1979), which is widely applied in the software industry.

Definition 9. *A function point (FP) is a virtual unit of software function that is determined by the weighted characteristic numbers of inputs, outputs, data objects, and internal processes, i.e.:*

$$\begin{aligned} Sfp &= f(Ni, No, Nq, Nlf, Nif, TCF) \\ &= UFP * TCF \\ &= (\sum_{k=1}^{5} w_k * N_k) * (0.65 + 0.01 \sum_{j=1}^{14} dj) \quad \text{[FP]} \end{aligned} \tag{9}$$

where UFP stands for the Unjustified Function Points, TCF the Technical Correction Factors, and the characteristic numbers denote the following:

- $N_1 = N_i$ = #(external inputs), $w_1 = 4$
- $N_2 = N_o$ = #(external outputs), $w_2 = 5$
- $N_3 = N_{lf}$ = #(internal logical files), $w_3 = 10$
- $N_4 = N_{if}$ = #(interface files), $w_4 = 7$
- $N_5 = N_q$ = #(external inquiries), $w_5 = 4$

According to Definition 9, function points are a subjective measure for the functional size/complexity of software, which are affected by an individual's selection of a variety of the weights. One of the fundamental issues in the function point measurement is that its basic unit of an FP is unclear in its physical meaning, because *Sfp* is a weighted sum of all the five external I/O quantities with adjusted application area factors, the basic unit, *Sfp* = 1 [FP], has no physical sense mapping onto real-world software entities.

The survey of software complexity theories and measures in the preceding subsections indicates that more sophisticate mathematical models and measures of software functional complexity for software engineering are yet to be sought. The following sections introduce the cognitive complexity theory of software based on the generic program model and the cognitive weights of program structures in terms of BCS's.

THE COGNITIVE WEIGHTS OF FUNDAMENTAL SOFTWARE STRUCTURES BASED ON THE GENERIC MATHEMATICAL MODEL OF PROGRAMS

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 elaborated. Then, the relative cognitive weights of BCS's are formally modeled and calibrated, which lead to the establishment of the cognitive complexity measurement of software systems in the next section.

The Mathematical Model of Programs

Despite of a 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 explaining the nature of programs and software systems (Wang, 2008a, 2008d).

Definition 10. *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.:*

$$P = \mathop{R}_{i=1}^{n-1}(p_i\ r_{ij}\ p_j), j = i+1$$
$$= (...(((p_1)\ r_{12}\ p_2)\ r_{23}\ p_3\ )...\ r_{n-1,n}\ p_n) \quad (10)$$

where the big-R notation (Wang, 2008b) is adopted to describe the nature of processes as the building blocks of programs.

Definition 11. *A program $\wp$ is a composition of a finite set of m processes according to the time-, event-, and interrupt-based process dispatching rules, i.e.:*

$$\wp = \mathop{R}_{k=1}^{m}(@e_k \hookrightarrow P_k) \quad (11)$$

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 2. *The Embedded Relational Model (ERM) states that an abstract program $\wp$ 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.:*

$$\wp = \mathop{R}_{k=1}^{m}(@e_k \hookrightarrow P_k)$$
$$= \mathop{R}_{k=1}^{m}[@e_k \hookrightarrow \mathop{R}_{i=1}^{n-1}(p_i(k)\ r_{ij}(k)\ p_j(k))], j = i+1 \quad (12)$$

Proof. *Theorem 2 can be directly proved on the basis of Definitions 10 and 11. Substituting P_k as given in Definition 11 with Eq. 10, a generic program $\wp$ obtains the form as a series of embedded relational processes as presented in Theorem 2*

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

Theorem 3. *The sum of the cognitive weights of all rij, w(rij), in the ERM model determines the operational complexity of a software system C_{op}, i.e.:*

$$C_{op} = \sum_{i=1}^{n-1} w(r_{ij}), j = i+1 \quad (13)$$

The EPM model of programs is developed on the basis of Real-Time Process Algebra (RTPA) (Wang, 2002a, 2007a, 2008c, 2008d, 2008e), where the definitions, syntaxes, and formal semantics of a set of meta-processes

and process relational operations have been introduced. 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 an algebraic composing operation between meta or complex processes.

Definition 12. *The RTPA meta-process system* $\mathfrak{P}$ *encompasses 17 fundamental computational operations elicited from the most basic computing needs, i.e.:*

$$\mathfrak{P} \triangleq \{:=, \blacklozenge, \Rightarrow, \Leftarrow, \Lsh, \gtrdot, \lessdot, |\gtrdot, |\lessdot, @, \triangleq, \uparrow, \downarrow, !, \otimes, \boxtimes, \S\} \quad (14)$$

where the meta-processes of RTPA stand for assignment, evaluation, addressing, memory allocation, memory release, read. write, input, output, timing, duration, increase, decrease, exception detection, skip, stop, and system, respectively (Wang, 2002a, 2008c).

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

Definition 13. *The RTPA process relation system* $\mathfrak{R}$ *encompasses 17 fundamental algebraic and relational operations elicited from basic computing needs, i.e.:*

$$\mathfrak{R} \triangleq \{\rightarrow, \curvearrowright, |, |\ldots|\ldots, R^*, R^+, R^i, \circlearrowleft, \rightarrowtail, \|, \fancyvarpi, |||, », \nleftarrow, \hookrightarrow_t, \hookrightarrow_e, \hookrightarrow_i\} \quad (15)$$

where the relational operators of RTPA stand for sequence, jump, branch, while-loop, repeat-loop, for-loop, recursion, function call, parallel, concurrence, interleave, pipeline, interrupt, time-driven dispatch, event-driven dispatch, and interrupt-driven dispatch, respectively (Wang, 2008c).

The Role of BCS's in Software Cognitive Complexity

Definition 14. *Basic Control Structures (BCS's) are a set of essential flow control mechanisms that are used to modele 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, i.e.:*

$$BCS = \{\rightarrow, |, |\ldots|, R^*, R^+, R^i, \circlearrowleft, \rightarrowtail, \| (\fancyvarpi), \nleftarrow\} \subset \mathfrak{R} \quad (16)$$

where || and $\fancyvarpi$ *are treated as equivalent.*

The ten BCS's are a subset of the most frequently used process relations $\mathfrak{R}$ as defined in RTPA, summarized in Table 2, which provide essential compositional rules for programming. Based on the set of BCS's, any complex computing function and process may be composed on the basis of the rules of algebraic relational operations.

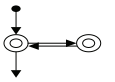
A formal semantics known as the *deductive semantics* of each of the above BCS's is provided in (Wang, 2006, 2008e). Since BCS's are the most highly reusable constructus in programming, according to Theorems 1 and 2, the cognitive weights of the ten fundamental BCS's play a crucial role in determining the cognitive complexity of software.

The Relative Cognitive Weights of Basic Software Structures

Definition 15. *The cognitive weight of a software system is the extent of relative difficulty or effort spent in time for comprehending the functions and semantics of the given program.*

Although it seems difficult to measure the cognitive weights of software at statement level since their variety and language dependency, it is found that it is feasible if the focus is put on the BCS's of software systems

Table 2. BCS's and their mathematical models

Category	BCS	Structure	Definition in RTPA
Sequence	Sequence (SEQ)		$P \rightarrow Q$
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^*)		$\overset{F}{R}_{\exp BL=T}(P)$
	Repeat-until (R^+)		$P \rightarrow \overset{F}{R}_{\exp BL=T}(P)$
	For-do (R^i)		$\overset{n}{R}_{i=1}(P(i))$
Embedment	Procedure call (PC)		P ↳ Q
	Recursion ($R^{\circlearrowleft}$)		P ↻ P
Concurrency	Parallel (PAR)		P ‖ Q
	Interrupt (INT)		P ↯ Q

(Wang, 2007a), because there are ten BCS's only in programming no matter what kind of programming language or formal notations is used. The ten BCS's are profound architectural attributes of software systems. Therefore, their relative cognitive weights are 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 (Osgood, 1953; Wang, 2005), 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.

Definition 16. *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_1, i.e.:

$$w_{BCS}(i) = \frac{t_{BCS}(i)}{t_{BCS}(1)} \text{ [CWU]}, \quad 1 \le i \le 10 \tag{17}$$

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

Definition 17. *The unit of cognitive weight of BCS's, CWU, is the relative time spent on the sequential BCS, i.e.:*

$$w_{BCS}(1) = \frac{t_{BCS}(1)}{t_{BCS}(1)} = 1 \text{ [CWU]} \tag{18}$$

Although, absolutely, different persons may comprehend the set of the ten BCS's in different speeds and efforts according to their experience and skills in a given programming language, the relative effort or the relative weight spent on each type of BCS's are statistically stable, assuming the relative weight of the *sequential* BCS is one according to Eqs. 17 and 18.

Definition 18. *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 for each of the ith, $1 \le i \le 10$, BCS:*

a. Record the start time T_1 in mm:ss;
b. Read a given test program, Test_i, for a specific BCS_i;
c. Derive the output(s) of the given program;
d. Record the end time T_2 in mm:ss;
e. Calculate the relative cognitive weight for the given construct, BCS_i, according to Eq. 17.

A set of 126 cognitive psychological experiments as designed according to Definition 17 have been carried out among undergraduate and graduate students as well as software engineers in the industry. Based on the experiment results, the equivalent cognitive weights of the ten fundamental BCS's are statistically calibrated as summarized in Table 3 (Wang, 2005).

The calibrated cognitive weights for the ten fundamental BCS's are illustrated in Fig. 4, where the relative cognitive weight of the sequential structure is defined as one unit, i.e. $w_1 \equiv 1$ (CWU).

THE COGNITIVE COMPLEXITIE OF SOFTWARE SYSTEMS

As reviewed in the ERM theory in Section 3, software functional complexity can be rigorously measured in a new way known as software cognitive complexity. This section develops the mathematical model and measurement methodology of cognitive complexity for software systems on the basis of software semantic structures (Wang, 2006, 2007a). A main notion is that the cognitive complexity of software is not a trivial one rather than a complex product of its operational and structural complexities.

The Operational Complexity of Software Systems

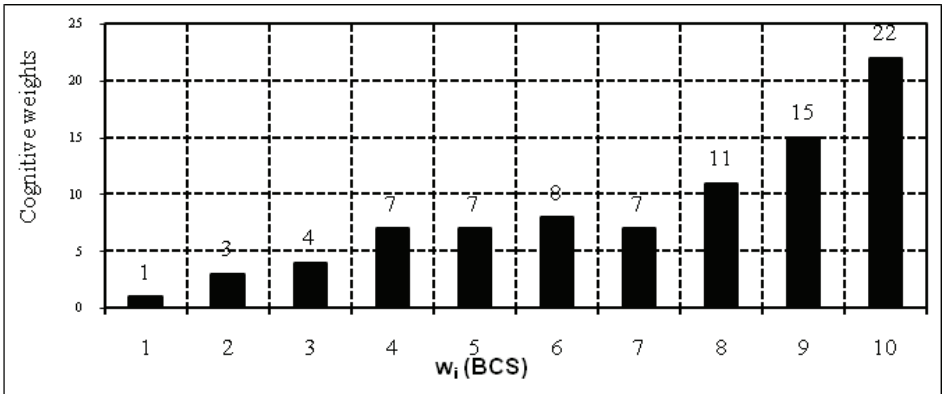
There are two structural patterns of BCS's in a given software system *S*: the *sequentially* and the *embedded* related BCS's. In the former, all the BCS's are in a linear layout in *S*, therefore the operational complexity of *S* is a sum of the cognitive weights of all linear BCS's. In the latter, some BCS's are embedded in others in *S*, hence the operational complexity of *S* is a product of the cognitive weights of inner BCS's and the weights of outer layer BCS's. In general, the two types of BCS architectural relations in *S* may be combined in various ways. Therefore, a general method for calculating the operational complexity of software can be derived as follows.

Table 3. Calibrated cognitive weights of BCS's

BCS	RTPA Notation	Description	Calibrated cognitive weight (w_i)
1	$\rightarrow$	Sequence	1
2	|	Branch	3
3	|…|	Switch	4
4	R^i	For-loop	7
5	R^*	Repeat-loop	7
6	R^*	While-loop	8
7	$\rightarrowtail$	Function call	7
8	↻	Recursion	11
9	|| or ⫴	Parallel	15
10	↯	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 4. The relative cognitive weights of BCS's of software systems



Definition 19. *The operational complexity of a software system S, $C_{op}(S)$, is determined by the sum of the cognitive weights of its n linear blocks composed by individual BCS's, where each block may consist of q layers of embedded BCS's, and within each of the layer there are m linear BCS's, i.e.:*

$$C_{op}(S) = \sum_{k=1}^{n_c} C_{op}(C_k) = \sum_{k=1}^{n_c} (\prod_{j=1}^{q_k} \sum_{i=1}^{m_{k,j}} w(k,j,i)) \quad [F] \tag{19}$$

If there is no embedded BCS in any of the n_c components in Eq. 19, i.e., $q = 1$, then

the operational complexity can be simplified as follows:

$$C_{op}(S) = \sum_{k=1}^{n_c} C_{op}(C_k) = \sum_{k=1}^{n_c} \sum_{i=1}^{m_k} w(k,i) \quad [\text{F}] \tag{20}$$

where $w(k, BCS)$ is given in Fig. 4.

Definition 20. *The unit of operational complexity of software systems is a single sequential operation called a function F, i.e.:*

$$C_{op}(S) = 1 \ [\text{F}] \Leftrightarrow \#(\text{SeqOP}(S)) = 1 \tag{21}$$

With the cognitive weight of sequential process relation defined as a unit of operational function of software systems, complex process relations can be quantitatively analyzed and measured.

Example 3. *The operational complexity of the algorithm of In-Between Sum, IBS_Algorithm***ST***, as given in Fig. 7, can be analyzed as follows:*

$$\begin{aligned} C_{op}(S) &= \sum_{k=1}^{n_c} \left(\prod_{j=1}^{q} \sum_{i=1}^{m} w(k,i,j)\right) \\ &= \sum_{k=1}^{n_c} \sum_{i=1}^{m} w(k,i) \\ &= w_{BCS}(\text{SEQ}) + \{w(\text{ITE}) \bullet 2w(\text{SEQ}) \\ &\quad + w(\text{ITE}) \bullet 2w(\text{SEQ})\} \\ &= 1 + (3 \bullet 2 + 3 \bullet 2) \\ &= 13 \quad [\text{F}] \end{aligned}$$

It is noteworthy that for a fully sequential software system where only w(sequence) = 1 [F] is involved, its operational complexity is reduced to the symbolic complexity in LOC.

Corollary 2. *The symbolic complexity $C_s(S)$ is a special case of the operational complexity $C_{op}(S)$, where the cognitive weights of all kinds of BCS's, $w_i(BCS)$, are simplified as a constant one, i.e.:*

$$\begin{aligned} S_{op}(S) &= \sum_{k=1}^{n_C} \sum_{i=1}^{m_k} w(k,i) \\ &= C_s(S), w(k,i) \equiv 1 \\ &= C_s(S) \quad [\text{LOC}] \end{aligned} \tag{22}$$

Corollary 2 presents an important finding on the relationship between conventional symbolic complexity and the operational complexity of software. It indicates that the symbolic measure in LOC is oversimplified so that it cannot represent the real functional complexities and sizes of software systems. Case studies summarized in Table 4 show that algorithms or programs with similar symbolic complexities may possess widely different functional complexities in terms of the operational and cognitive complexities.

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 21. *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.:*

$$C_a(S) = \text{OBJ}(S)) = \sum_{k=1}^{n_{CLM}} \text{OBJ}(CLM_k) + \sum_{k=1}^{n_C} \text{OBJ}(C_k) \quad [\text{O}] \tag{23}$$

where OBJ is a function that counts the number of data objects in a given Component Logical Model (CLM) (Wang, 2002a),which is equivalent to the number of global variables or components (number of local variables).

Definition 22. *The unit of architectural complexity of software systems is a single data object, modeled either globally or locally, called an object O, i.e.:*

$$C_a(S) = 1\ [O] \Leftrightarrow \#(\mathrm{OBJ}(S)) = 1 \tag{24}$$

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 as may be resulted in the computational complexity. This is determined on the basis of cognitive informatics where the inductive inference effort of a large even 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 data objects* in the set {X[1]**N**, X[2]**N**, …, X[*n*]**N**} is counted as three rather than *n*.

Example 4. *The architectural complexity of the MaxFinder algorithm as shown in Fig. 9 can be determined as follows:*

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

Example 5. *The CLM SysClock***ST** *given below encompasses 7 objects, therefore its architectural complexity is: C_a(SysClock**ST**) = 7 [O].*

The Cognitive Complexity of Software Systems

On the basis of the elaborations of software architectural and operational complexities, the cognitive complexity of software systems is introduced as a fundamental measure of the functional complexity and sizes 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 (Wang, 2006). 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 23. *The semantic function of a program $\wp$, $f_0(\wp)$, 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.:*

$$f_0(\wp) = f: T \times S \to V$$

$$= \begin{pmatrix} & \mathbf{s_1} & \mathbf{s_2} & \cdots & \mathbf{s_m} \\ \mathbf{t_0} & \perp & \perp & \cdots & \perp \\ \mathbf{t_1} & v_{11} & v_{12} & & v_{1m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{t_n} & v_{n1} & v_{n1} & \cdots & v_{nm} \end{pmatrix} \tag{25}$$

*Figure 5. The CLM architecture of a component: SysClock***ST**

```
SysClockST ≙ SysClockS ::
        ( <§t : N |  0 ≤ §tN  ≤ 1M>,
          <CurrentTime : hh:mm:ss:ms  | 00:00:00:000  ≤
          CurrentTime hh:mm:ss:ms ≤ 23:59:59:999>,
          <Timer : ss  | 0 ≤ Timerss ≤ 3600>,
          <MainClockPort : B | MainClockPortB = FFF0H >,
          <ClockInterval : N | TimeIntervalN = 1ms>,
          <InterruptCounter : N |  0 ≤ InterruptCounterN  ≤ 999>
        )
```

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 $1 \le j \le m$.

According to Definition 23, the semantic space of a program can be illustrated by a two dimensional plane as shown in Fig. 6.

Observing Fig. 6 and Eq. 25, 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 4. *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.:*

$$C_c(S) = C_{op}(S) \bullet C_a(S)$$
$$= \{\sum_{k=1}^{n_C} \sum_{i=1}^{\#(C_s(C_k))} w(k,i)\} \bullet$$
$$\{\sum_{k=1}^{n_{CLM}} \mathrm{OBJ}(CLM_k) + \sum_{k=1}^{n_C} \mathrm{OBJ}(C_k)\} \quad [\mathrm{FO}] \tag{26}$$

Corollary 3. *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 data objects, the larger the functional complexity of the system.*

Definition 24. *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.:*

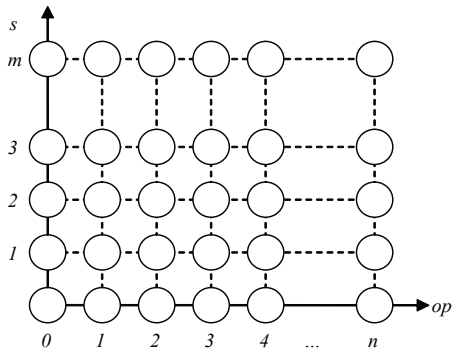
$$C_f = C_{op} \bullet C_a$$
$$= 1[\mathrm{F}] \bullet 1[\mathrm{O}]$$
$$= 1 \quad [\mathrm{FO}] \tag{27}$$

According to Definition 24, 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 determination of software functional sizes and design efforts in software engineering practice. It will be demonstrated in the next section that the cognitive complexity is the most distinguishable and accurate measure for the inherent functional complexity of software systems.

COMPARATIVE STUDIES ON THE COGNITIVE COMPLEXITY OF SOFTWARE SYSTEMS

On the basis of the classification of software complexities and the introduction of the cognitive complexity of software systems in preceding sections, a comparatively analysis between cognitive complexity in the facets of

Figure 6. The semantic space of software systems



the operational and architectural complexities, as well as conventional time, cyclomatic, and symbolic (LOC) complexities may provide useful insights. A set of case studies is carried out in order to examine the measurability and accuracy of various complexity and size measures for software systems in software engineering. The results demonstrate that cognitive complexity is the most sensitive measure for denoting the real complexities and functional sizes of software systems.

Comparative Case Studies on Different Complexity Models of Software Systems

Four sample software components in formal RTPA models are adopted to explain the comparative complexity analyses in Examples 6 through 8.

Example 6. *The formal specification of the algorithm of In-Between Sum (IBS), IBS_Algorithm***ST***, is specified in two different approaches as shown in Figs. 7 and 8, respectively.*

Example 7. *An algorithm, MaxFinder***ST***, is formally described in RTPA as shown in Fig. 9. 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***}.*

Example 8. *The self-index sort (SIS) algorithm, SIS***ST***, can be formally described in RTPA as shown in Fig. 10. Detailed explanations of the algorithm may be referred to (Wang, 1996).*

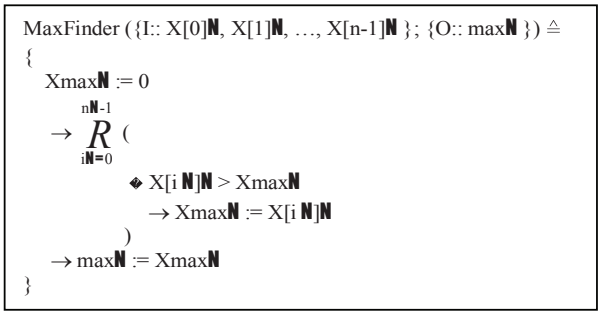
Figure 7. The formal model of the IBS algorithm (a) in RTPA

```
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 8. The formal model of the IBS algorithm (b) in RTPA

```
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 9. The formal model of the MaxFinder algorithm in RTPA



According to the methodologies of cognitive complexity measurement, particularly Eqs. 19, 23, and 26, the cognitive complexity of these four sample programs can be systematically analyzed as summarized in Table 4. Table 4 provides a systematical contrast of the measures of time, cyclomatic, symbolic, operational, architectural, and cognitive complexities.

Observing Table 4 it is noteworthy that the first three measurements, namely the time, cyclomatic, and symbolic complexities, cannot actually reflect the real complexity of software systems in design, representation, cognition, and/or comprehension in software engineering.

- Although the four example systems are with similar symbolic complexities, their operational and cognitive complexities are greatly different. This indicates that the symbolic complexity cannot be used to represent the operational or functional complexity of software systems.
- Symbolic complexity does not represent the throughput or the input size of problems.
- Time complexity does not work well for a system where is no loop and/or dominant operations, because theoretically in this case all statements in linear structures are treated as zero no matter how long they are. In addition, time complexity cannot 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 a more objective and rigorous measure of software system 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 cognitive complexity. However, symbolic and cyclomatic complexities cannot identify the functional differences very well.

The Symbolic vs. Cognitive Sizes of Software Systems

The cognitive complexity models developed so far have established a connection between the operational and architectural complexities of software. They also reveal that the symbolic complexity is only a special case of the operational complexity. The following investigation will establish another connection between the operational complexity and system relational complexity.

According to the generic system complexity theory (Wang, 2007a), when a system is treated as a black box, the relational complexity of the system can be estimated by the maximum possible pairwise relations between all components in the system.

Figure 10. The Formal model of the SIS sort algorithm in RTPA

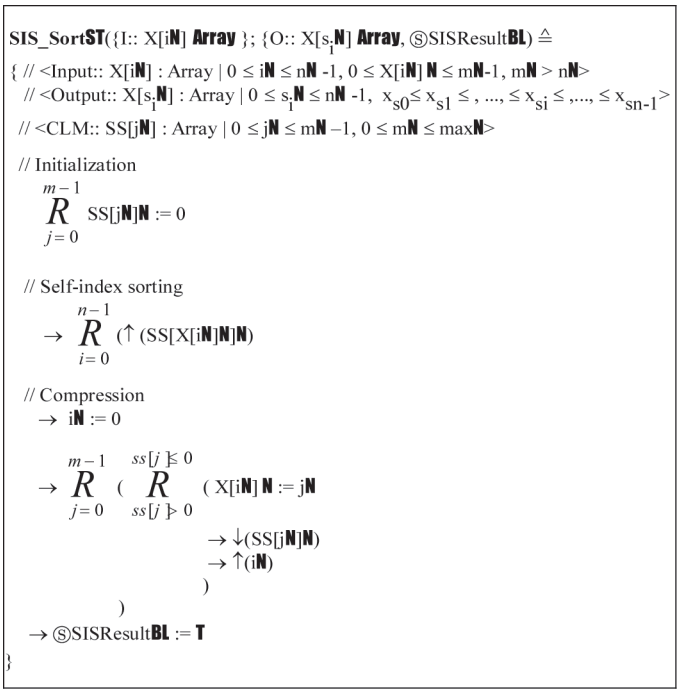


Table 4. Comparative measurements of software system complexities

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	5*	575
SIS_Sort	O(m+n)	5	8	163	11*	1,793

** The equivalent objects as defined in Definition 22*

Definition 25. *The relational complexity of software system S, $C_r(S)$, is the maximum number of relations n_r among components, i.e.:*

$$C_r(S) = n_r = n_c(n_c - 1) \quad [\mathrm{R}] \tag{28}$$

where the unit of the relational complexity is the number of relations R, which is equivalent to F as defined in the operational complexity.

It is noteworthy that $C_r(S)$ provides the maximum potential or the upper limit of internal relational complexity of a given software system. The relationship among the symbolic, relational, and operational complexities of software systems is plotted in Fig, 11 in logarithmic

scale. As shown in Fig. 11, the symbolic complexity of software $C_s(S)$ is the lower bound of the functional complexity of software and it is linearly proportional to the number of statements n, i.e., $O(n)$. The relational complexity $C_r(S)$ is the upper bound of functional complexity of software in the order of $O(n^2)$. Therefore, the real cognitive functional complexity represented by the operational complexity $C_{op}(S)$ is bounded between the curves of the symbolic and relational complexities.

Fig. 11 indicates that the floor of the operational complexity $C_{op}(S)$ of software systems is determined by the symbolic complexity $C_s(S)$ when only sequential relation is considered between all adjacent statements in a given program and all weights of the sequential relational operations in computing are simplified to one. The ceiling of the operational complexity $C_{op}(S)$ is determined by the relational complexity $C_r(S)$, when all potential relations among the components (statements) in computing are considered.

Corollary 4. *The operational complexity $C_{op}(S)$ of a software system S is constrained by the lower bound of the symbolic complexity $C_s(S)$ and the upper bound of the relational complexity $C_r(S)$, i.e.:*

$$O(n) \le C_{op}(S) \le O(n^2) \qquad (29)$$

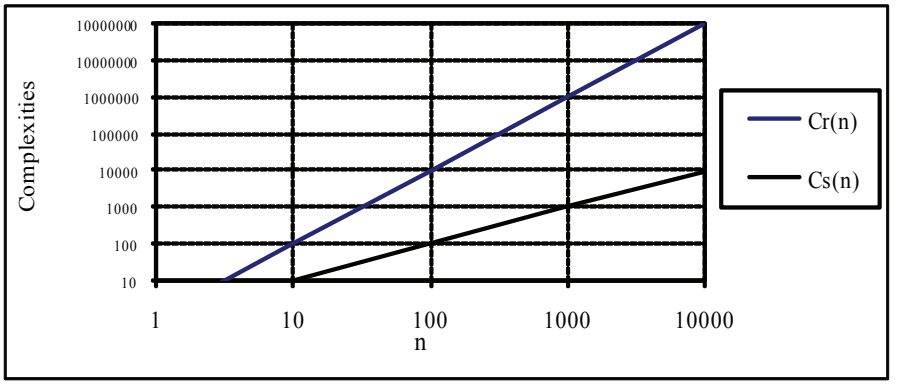
where n is the number of statements in S.

According to Corollary 4 and Fig. 11 it can be seen that the real complexity and size of software systems had probably been greatly underestimated when the conventional symbolic size measurement (LOC) is adopted, because it represents the minimum functional complexity of software. Therefore, the functional size of software systems measured by the cognitive complexity as described in Theorem 3 provides a more accurate size and complexity measurement for software systems in software engineering.

CONCLUSION

This paper has presented a new approach to model and quantifying the cognitive complexity of software. A comprehensive survey of computational theories and measures of software has been systematically analyzed. On this basis, the cognitive complexity of software is introduced as a product of its architectural and operational complexities on the basis of deductive semantics and abstract system theories. Ten basic control structures (BCS's) have been elicited according

Figure 11. The relational and symbolic complexities as the upper/lower bounds of functional complexity of software systems



to the generic mathematical model of programs as a structure of embedded relational processes. The cognitive weights of BCS's have been quantitatively derived and calibrated via a series of psychological experiments. Then, the cognitive complexity of software systems has been formally and rigorously elaborated. Robustness of the cognitive complexity measure has been analyzed with comparative case studies.

According to the cognitive complexity theory, it is found that the traditional time, cyclomatic, and symbolic complexities cannot actually reflect the real complexity of software systems in design, representation, cognition, and/or comprehension in software engineering. More fundamentally, although it is well applied in machine oriented estimations, conventional computational complexity might not be sensitive and distinguishable for human cognitive and creative work products in software engineering. The real complexity and size of software systems might have been greatly underestimated when the conventional symbolic size measurement is adopted, because it represents only the lower bound of the functional complexity of software systems. These findings indicate that the cognitive complexity needs to be adopted for rationally measuring the functional sizes of software systems in software engineering.

ACKNOWLEDGMENT

The author would like to acknowledge the Natural Science and Engineering Council of Canada (NSERC) for its partial support to this work. The author would like to thank the anonymous reviewers for their valuable suggestions and comments on this work.

REFERENCES

Albrecht, A.J. (1979), Measuring Application Development Productivity, Proc. of IBM Applications Development Joint SHARE/GUIDE Symposium, Oct., 83-92.

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.

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.

Fenton, N.E. and S.L. Pfleeger (1998), Software Metrics: A Rigorous and Practical Approach (2nd ed.), Brooks/Cole Pub Co.

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

Hartmanis, J. (1994), On Computational Complexity and the Nature of Computer Science, 1994 Turing Award Lecture, *Communications of the ACM*, Vol.37, No.10, pp.37-43.

Hartmanis, J. and R.E. Stearns (1965), On the Computational Complexity of Algorithms, *Trans. AMS,* 117, pp. 258-306.

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.

Lewis, H.R. and Papadimitriou, C.H. (1998), Elements of the Theory of Computation, 2nd ed., Prentice Hall International, Englewood Cliffs, NJ.

Lipschutz, S. and M. Lipson (1997), *Schaum's Outline of Theories and Problems of Discrete Mathematics*, 2nd ed., McGraw-Hill Inc., New York, NY.

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

McDermid, J.A. *ed.* (1991), *Software Engineer's Reference Book*, Butterworth-Heinemann Ltd., Oxford, UK.

Melton, A. *ed.* (1996), Software Measurement, International Thomson Computer Press.

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

Wang, Y. (1996), A New Sorting Algorithm: Self-Indexed Sort, *ACM SIGPLAN*, 31(3), March, pp. 28-36.

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

Wang, Y. (2002b), Keynote: *On Cognitive Informatics,* Proc. First IEEE International Conference on Cognitive Informatics (ICCI'02), Calgary, AB., Canada, IEEE CS Press, August, pp.34-42.

Wang, Y. (2003a), On Cognitive Informatics, *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy,* 4(2), 151-167.

Wang, Y. (2003b), The Measurement Theory for Software Engineering, *Proc. 2003 Canadian Conference on Electrical and Computer Engineering* (CCECE'03), IEEE CS Press, Montreal, Canada, May, pp.1321-1324.

Wang, Y. (2005), Keynote: 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.

Wang, Y. (2006), On the Informatics Laws and Deductive Semantics of Software, *IEEE Trans. on Systems, Man, and Cybernetics (C),* 36(2), March, pp. 161-171.

Wang, Y. (2007a), *Software Engineering Foundations: A Transdisciplinary and Rigorous Perspective,* CRC Book Series in Software Engineering, Vol. II, Auerbach Publications, NY, USA, July.

Wang, Y. (2007b), The Theoretical Framework of Cognitive Informatics, *International Journal of Cognitive Informatics and Natural Intelligence*, 1(1), Jan., pp. 1-27.

Wang, Y. (2008a), *On Contemporary Denotational Mathematics for Computational Intelligence, Transactions of Computational Science*, *2*, Springer, August, pp. 6-29.

Wang, Y. (2008b), On the Big-R Notation for Describing Iterative and Recursive Behaviors, *International Journal of Cognitive Informatics and Natural Intelligence,* 2(1), Jan., 17-28.

Wang, Y. (2008c), RTPA: A Denotational Mathematics for Manipulating Intelligent and Computational Behaviors, *International Journal of Cognitive Informatics and Natural Intelligence,* 2(2), April, pp. 44-62.

Wang, Y. (2008d), Mathematical Laws of Software, *Transactions of Computational Science*, 2, Springer, Aug., pp. 46-83.

Wang, Y. (2008e), Deductive Semantics of RTPA, *International Journal of Cognitive Informatics and Natural Intelligence,* 2(2), April, pp. 95-121.

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.

Zuse, H. (1997), A Framework of Software Measurement, Walter de Gruyter & Co., Berlin.

Yingxu Wang is professor of cognitive informatics and software engineering, director of International Center for Cognitive Informatics (ICfCI), and director of Theoretical and Empirical Software Engineering Research Center (TESERC) in Dept. of ECE at University of Calgary, Canada. He received a PhD in software engineering from the Notting¬ham Trent University, UK, in 1997, and a BSc in electrical engineering from Shanghai Tiedao University in 1983. He is/was a visiting professor in the Computing Laboratory at Oxford University in 1995, Dept. of Computer Science at Stanford University in 2008, and the Berkeley Initiative in Soft Computing (BISC) Lab at University of California, Berkeley in 2008, respectively. He has been a full professor since 1994. He is a Fellow of WIF, a P.Eng of Canada, a Senior Member of IEEE and ACM, and a member of ISO/IEC JTC1 and the Canadian Advisory Committee (CAC) for ISO. He is the founder and steering committee chair of the annual IEEE International Conference on Cognitive Informatics (ICCI). He is founding editor-in-chief of International Journal of Cognitive Informatics and Natural Intelligence *(*IJCINI*), founding editor-in-chief of* International Journal of Software Science and Computational Intel-

ligence *(*IJSSCI*), associate editor of* IEEE TSMC(A)*, and editor-in-chief of* CRC Book Series in Software Engineering. *He has published over 300 peer reviewed journal and conference papers and 12 books in cognitive informatics, software engineering, and computational intelligence. He is the recipient of dozens of research achievement, best paper, and teaching awards in the last 30 years.*