

A new measure of software complexity based on cognitive weights

Une nouvelle métrique de complexité logicielle basée sur les poids cognitifs

Jingqiu Shao and Yingxu Wang

One of the central problems in software engineering is the inherent complexity. Since software is the result of human creative activity, cognitive informatics plays an important role in understanding its fundamental characteristics. This paper models one of the fundamental characteristics of software, complexity, by examining the cognitive weights of basic software control structures. Based on this approach a new concept of cognitive functional size of software is developed. Comparative case studies of the cognitive complexity and physical size of 20 programs are reported. The cognitive functional size provides a foundation for cross-platform analysis of complexity, size, and comprehension effort in the design, implementation, and maintenance phases of software engineering.

Un problème majeur en génie logiciel concerne sa complexité. Puisque les logiciels sont le résultat de la créativité humaine, les aspects cognitifs jouent un rôle essentiel dans ceux-ci. Cet article modélise une des caractéristiques essentielles des logiciels, à savoir leur complexité en examinant les poids cognitifs de leurs structures de commande de base. De là, une nouvelle métrique cognitive de taille du logiciel est mise au point. Des tests comparatifs sur 20 programmes de complexité et taille différentes sont discutés. Cette mesure cognitive permet l'analyse multi-plate-forme de complexité et de taille. Elle permet aussi d'évaluer les efforts reliés aux phases de spécification, design, implantation, et maintenance en génie logiciel.

Keywords: software engineering, software complexity, cognitive weight, formal description, RTPA, measurement

I. Introduction¹

An important issue encountered in software complexity analysis is the consideration of software as a human creative artifact and the development of a suitable measure that recognizes this fundamental characteristic. The existing measures for software complexity can be classified into two categories: the macro and the micro measures of software complexity.

Major macro complexity measures of software have been proposed by Basili and by Kearney et al. The former considered software complexity as “the resources expended” [1]. The latter viewed the complexity in terms of the degree of difficulty in programming [2].

The micro measures are based on program code, disregarding comments and stylistic attributes. This type of measure typically depends on program size, program flow graphs, or module interfaces such as Halstead’s software science metrics [3] and the most widely known cyclomatic complexity measure developed by McCabe [4]. However, Halstead’s software metrics merely calculate the number of operators and operands; they do not consider the internal structures of software components; while McCabe’s cyclomatic measure does not consider I/Os of software systems.

In cognitive informatics, it is found that the functional complexity of software in design and comprehension is dependent on three fundamental factors: internal processing, input and output [5]–[6]. Cognitive complexity, the new measure for software complexity presented in this paper, is a measure of the cognitive and psychological complexity of

software as a human intelligence artifact. Cognitive complexity takes into account both internal structures of software and the I/Os it processes. In this paper, the weights of cognitive complexity for fundamental software control structures will be defined in Section II. On the basis of cognitive weight, the cognitive functional size (CFS) of software is introduced in Section III. Real-time process algebra (RTPA) as a formal method for describing and measuring software complexity is introduced in Section IV. Robustness of the cognitive complexity measure is analyzed in Section V with a number of comparative case studies and experimental results.

II. The cognitive weight of software









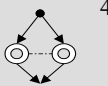
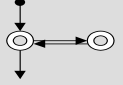
To comprehend a given program, we naturally focus on the architecture and basic control structures (BCSs) of the software [5]. BCSs are a set of essential flow control mechanisms that are used for building logical software architectures [5]–[6]. Three BCSs are commonly identified: the *sequential*, *branch*, and *iteration* structures [7]. Although it can be proven that an iteration may be represented by the combination of sequential and branch structures, it is convenient to keep iteration as an independent BCS. In addition, two advanced BCSs in system modelling, known as *recursion* and *parallel*, have been described by Hoare et al. [7]. Wang [5]–[6], [8] extended the above set of BCSs to cover *function call* and *interrupt*.

Definition 1. The *cognitive weight* of software is the degree of difficulty or relative time and effort required for comprehending a given piece of software modelled by a number of BCSs.

The seven categories of BCSs described above are profound architectural attributes of software systems. These BCSs and their variations are modelled and illustrated in Table 1, where the equivalent cognitive

Jingqiu Shao and Yingxu Wang are with the Theoretical and Empirical Software Engineering Research Centre, Department of Electrical and Computer Engineering, University of Calgary, 2500 University Drive N.W., Calgary, Alberta T2N 1N4. E-mail: {shao,wangyx}@enel.ucalgary.ca

Table 1**Definition of BCSs and their equivalent cognitive weights (W_i)**

| Category | BCS | Structure | W_i | RTPA notation |
|--------------------|------------------------|---|-------|--|
| Sequence | Sequence (SEQ) |  | 1 | $P \rightarrow Q$ Note: Consider only one sequential structure in a component |
| Branch | If-then-[else] (ITE) |  | 2 | $(? \text{exp } \mathbf{BL} = \mathbf{T}) \rightarrow P$ $ (? \sim) \rightarrow Q$ |
| | Case (CASE) |  | 3 | $? \text{exp } \mathbf{RT} =$ $0 \rightarrow P_0$ $ 1 \rightarrow P_1$ $ \dots$ $ n-1 \rightarrow P_{n-1}$ $ \text{else} \rightarrow \emptyset$ |
| Iteration | For-do (R_i) |  | 3 | $R_{i=1}^n (P(i))$ |
| | Repeat-until (R_1) |  | 3 | $R_{\geq 1}^{\text{exp } \mathbf{BL} \neq \mathbf{T}} (P)$ |
| | While-do (R_0) |  | 3 | $R_{\geq 0}^{\text{exp } \mathbf{BL} \neq \mathbf{T}} (P)$ |
| Embedded component | Function call (FC) |  | 2 | $P \hookrightarrow F$ Note: Consider only user-defined functions |
| | Recursion (REC) |  | 3 | $P \cup P$ |
| Concurrency | Parallel (PAR) |  | 4 | $P \parallel Q$ |
| | Interrupt (INT) |  | 4 | P $\parallel \odot (@eS \nearrow Q \searrow \odot)$ |

weights (W_i) of each BCS for determining a component's functionality and complexity are defined based on empirical studies in cognitive informatics [5].

There are two different architectures for calculating W_{bc} : either all the BCSs are in a linear layout or some BCSs are embedded in others. For the former case, we may sum the weights of all n BCSs; for the latter, we can multiply the cognitive weights of inner BCSs with the weights of external BCSs. In a generic case, the two types of architectures are combined in various ways. Therefore, a general method can be defined as follows.

Definition 2. The *total cognitive weight* of a software component, W_c , is defined as the sum of the cognitive weights of its q linear blocks composed of individual BCSs. Since each block may consist of m lay-

ers of nesting BCSs, and each layer of n linear BCSs, the total cognitive weight, W_c , can be calculated by

$$W_c = \sum_{j=1}^q \left[\prod_{k=1}^m \sum_{i=1}^n W_c(j, k, i) \right]. \quad (1)$$

If there is no embedded BCS in any of the q blocks, i.e., $m = 1$, then (1) can be simplified as follows:

$$W_c = \sum_{j=1}^q \sum_{i=1}^n W_c(j, i). \quad (2)$$

III. The cognitive functional size of software

A component's cognitive functional size is found to be proportional to the total weighted cognitive complexity of all internal BCSs and the number of inputs (N_i) and outputs (N_o) [5]–[6]. In other words, CFS is a function of the three fundamental factors: W_c , N_i , and N_o . Thus, an equivalent cognitive unit of software can be defined as follows.

Definition 3. The *unit of cognitive weight* (CWU) of software, S_{η} , is defined as the cognitive weight of the simplest software component with only a single I/O and a linear structured BCS, i.e.,

$$\begin{aligned} S_{f0} &= f(N_{i/o}, W_{bc}) \\ &= (N_i + N_o) \cdot W_c \\ &= 1 \times 1 \\ &= 1 \text{ [CWU]}, \end{aligned} \quad (3)$$

where the symbol shown in square brackets is the unit of quantity (as in the remaining equations in this paper).

Equation (3) models a tangible and fundamental unit of software functional size. It is intuitive that the larger each of the above factors is, the greater is the CFS.

Definition 4. The *cognitive functional size* of a basic software component that only consists of one method, S_f , is defined as a product of the sum of inputs and outputs ($N_{i/o}$) and the total cognitive weight, i.e.,

$$\begin{aligned} S_f &= N_{i/o} \times W_c \\ &= (N_i + N_o) \cdot \left\{ \sum_{j=1}^q \left[\prod_{k=1}^m \sum_{i=1}^n W_c(j, k, i) \right] \right\} \text{ [CWU]}, \end{aligned} \quad (4)$$

where the unit of CFS is the equivalent cognitive weight unit (CWU) as defined in (3).

Based on (4), the CFS of a complex component with n_c methods, $S_f(c)$, can be derived as follows:

$$S_f(c) = \sum_{c=1}^{n_c} S_f(c) \text{ [CWU]}, \quad (5)$$

where $S_f(c)$ is the CFS of the c -th method that can be directly measured according to (4).

Thus, the CFS of a component-based software system \hat{S} with p components, \hat{S}_f , can be defined below:

$$\begin{aligned}\hat{S}_f &= \sum_{p=1}^{n_p} S_f(p) \\ &= \sum_{p=1}^{n_p} \sum_{c=1}^{n_c} S_f(p, c) \text{ [CWU]},\end{aligned}\quad (6)$$

where n_p is the number of components in a program.

Example 1. An algorithm of in-between sum, the *IBS algorithm*, is implemented in C as shown in Fig. 1. It can be seen that, for this given program, $N_i = 2$, $N_o = 1$. There are two internal structures: a sequential and a branch BCS. The cognitive weights of these two BCSs can be determined as follows:

$$\begin{aligned}\text{BCS}_1 \text{ (sequence): } W_1 &= 1, \\ \text{BCS}_2 \text{ (branch): } W_2 &= 2.\end{aligned}$$

It is noteworthy that only one sequential structure is considered for a given component. Thus, the total cognitive weight of this component is: $S_c = S_1 + S_2 = 1 + 2 = 3$. According to (3), the CFS of this algorithm can be derived as

$$\begin{aligned}S_f &= (N_i + N_o) \cdot W_c \\ &= (2 + 1) \times 3 \\ &= 9 \text{ [CWU]}.\end{aligned}$$

The above result shows that when both the internal architectural complexity and I/O turnover are considered, this algorithm's complexity is equivalent to 9 CWU.

For a large software system composed of n_p components or algorithms, the total cognitive functional complexity is the sum of all components according to (6).

IV. Formal description of software cognitive complexity by RTPA

Software quality, from a cognitive informatics point of view, is defined as the completeness, correctness, consistency, feasibility, and verifiability of the software in both specification and implementation, with no misinterpretation and no ambiguity [5]. Therefore, quality software relies on the explicit description of three dimensional behaviours known as the architecture, static behaviours, and dynamic behaviours [8]. Program comprehension is then a cognitive process to understand a given software system in terms of these three dimensions and their relationships.

Formal methods provide a rigorous way to describe software systems in order to ensure a higher quality of design and implementation. It is found that the complexity of software can be analyzed on the basis of formal specifications early in the development process before code is available. In this section, real-time process algebra [8] is used to demonstrate how the new measurement of software complexity can be analyzed based on formal specifications.

Conventional formal methods are based on logic and set theories that lack the capability to describe software architectures and dynamic behaviours. RTPA is designed to describe and specify architectures and behaviours of software systems formally and precisely. RTPA models 16 meta processes and 16 process relations, as partially shown in Table 1 [8]. A meta process is an elementary and primary process that serves as a common and basic building block for a software system. Complex processes can be derived from meta processes by a set of process relations that serves as the process combinatory rules. Details about RTPA may be obtained in [8] and [9].

```
// =====
// Algorithm of In-Between Sum (IBS)
// =====

#include <stdio.h>
#include <stdlib.h>
/* Calculates the sum of all the numbers between A and B.
   The input is limited between (MIN_RANGE, MAX_RANGE).*/
#define MIN_RANGE 0
#define MAX_RANGE 30000

int main()
{
    long a,b,sum;

    // Input A and B
    printf("\n Input the first number A: "); // BCS1
    scanf("%l", &a);
    printf("\n Input the second number B:");
    scanf("%l", &b);

    // Check A and B
    if ((MIN_RANGE<a && a<MAX_RANGE) // BCS2
        && (MIN_RANGE<b && b<MAX_RANGE) && a<b)
        // Calculate the sum in-between
        {
            b--;
            sum = (b*(b+1))/ 2 - (a*(a+1))/2;
            // Present output
            printf("%s\n: The in-between sum of a and b is:, %li\n",
                sum );
        }
    else
        printf("Invalid input");

    return 0;
}
```

Figure 1: Source code of the IBS algorithm (Implementation 1).

In Section III the algorithm of in-between sum has been used as an example to illustrate how to apply the CFS to measure software complexity. By applying RTPA, we can specify the algorithm explicitly as shown in Fig. 2. We use the same example to demonstrate how the complexity of the IBS algorithm specified in RTPA can be measured by CFS.

From the above specification, we can see that it is easy to capture the inputs, outputs, and BCSs when a formal specification is used. According to Table 1, RTPA provides expressive notations to identify and denote the BCSs of software, such as R (iteration), \rightarrow (sequence), and $? \dots |? \sim \dots$ (branch). There are two BCSs in this component: one is a sequence and the other is a branch. Only one sequential structure is considered in an entire component as a convention. Based on the above analysis, we obtain $W_{bcst} = 1 + 2 = 3$, $N_i = 2$, and $N_o = 1$. Thus, the CFS of this component can be derived as follows:

$$\begin{aligned}S_f &= (N_i + N_o) \cdot W_c \\ &= (2 + 1) \times 3 \\ &= 9 \text{ [CWU]}.\end{aligned}$$

RTPA has been developed as an expressive, easy-to-comprehend, and language-independent notation system, and a specification and refinement method for software description and specification [9]. Based on the above analysis, the cognitive functional size is feasibly obtained to measure software complexity. Due to the precision and

```

IBS_Algorithm ({!:: AN, BN}; {O:: ⊙IBSResultBL, IBSumN}) ≜
{
  → 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 2: RTPA specification of the IBS algorithm.

rigorousness of RTPA, the measurement of the complexity of software systems can be obtained in an early phase of system development. Since RTPA is language-neutral, any implementation of a given specification will result in the same equivalent CFS.

V. Robustness of the cognitive complexity measure

A. Case studies of CFS

To analyze the robustness of the new measure of cognitive functional size (S_f) and its relationship with the physical size of software (S_p), we have carried out a large set of case studies and experimental analyses. In this subsection, 20 Java programs are selected from Wiener and Pinson's book *Fundamentals of OOP and Data Structures in Java* [10]. Each program is analyzed in terms of the physical size (with the unit known as lines of code or LOC) and cognitive functional size (with the unit of cognitive weight or CWU) as shown in Table 2 and illustrated in Fig. 3.

The physical size (S_p) of software, or program length, can be used as a predictor of program characteristics such as effort and difficulty of maintenance. However, it characterizes only one specific aspect of size, namely the static length, because it takes no account of functionality. The cognitive functional size (S_f) is concerned with functional and cognitive complexity. An interesting relationship between S_f and S_p , known as coding efficiency (E_c) [6], can be derived as shown below:

$$E_c = S_f / S_p \text{ [CWU/LOC].} \quad (7)$$

In all of the case studies, the average coding efficiency for the 20 programs is

$$\begin{aligned} E_c &= S_f / S_p \\ &= 4216 / 1678 \\ &\approx 2.5 \text{ [CWU/LOC],} \end{aligned}$$

ranging from 0.5 to 6.6 CWU per LOC throughout the samples. In other words, 1.0 LOC = 2.5 CWU on average.

Fig. 3 demonstrates that the trends of both physical size and cognitive functional size follow basically the same pattern. As the physical size increases, so does the corresponding cognitive functional size. It is noteworthy that there are four points for which the cognitive functional size goes up sharply. This indicates that these four programs have higher code cognitive efficiency, using fewer lines of code to implement more complex software.

As indicated in Fig. 4, the physical size is not a good measure for predicting software complexity. For example, there are two cases with similar physical size around 112/113 LOC, but one has cognitive functional size below 200 CWU, while that of the other is above 700 CWU.

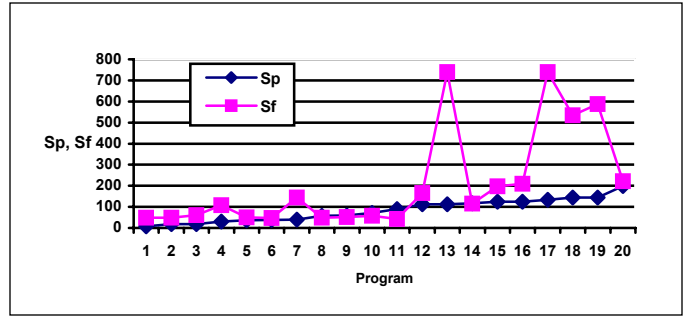


Figure 3: Plot of S_f and S_p of 20 sample programs.

Table 2
Analysis of the physical and functional sizes

| Number | Physical size (S_p) | Cognitive functional size (S_f) | Reference of source code |
|--------|-------------------------|-------------------------------------|--------------------------|
| 1 | 8 | 48 | pp. 430 |
| 2 | 18 | 48 | pp. 429 |
| 3 | 18 | 60 | pp. 434 |
| 4 | 30 | 108 | pp. 431 |
| 5 | 36 | 50 | pp. 258 |
| 6 | 38 | 47 | pp. 380 |
| 7 | 40 | 144 | pp. 323 |
| 8 | 58 | 49 | pp. 199 |
| 9 | 59 | 51 | pp. 203 |
| 10 | 72 | 57 | pp. 208 |
| 11 | 90 | 42 | pp. 404 |
| 12 | 112 | 166 | pp. 293 |
| 13 | 113 | 740 | pp. 219 |
| 14 | 117 | 115 | pp. 361 |
| 15 | 124 | 198 | pp. 230 |
| 16 | 124 | 209 | pp. 236 |
| 17 | 134 | 740 | pp. 387 |
| 18 | 144 | 587 | pp. 325 |
| 19 | 144 | 535 | pp. 344 |
| 20 | 199 | 222 | pp. 349 |
| Total | 1678 | 4216 | |

Generally, when S_p ranges from 90 to 200 LOC, S_f may vary dramatically. Therefore, a rule of thumb for deciding component complexity, i.e.,

$$S_p \leq 100 \text{ [LOC]}$$

or

$$60 \leq S_f \leq 100 \text{ [CWU]} \quad (8)$$

may be taken as a fundamental threshold to determine the suitable sizes of components during software system decomposition. Otherwise, the cognitive complexity may increase dramatically.

B. Robustness of the cognitive complexity measure

Using the specification of the IBS algorithm as shown in Fig. 2, we tested the robustness of S_f as a formal measure for cognitive complexity. Three programs based on the same algorithm were implemented in C, Java, and Pascal respectively as shown in Figs. 1, 6 and 7. The resulting physical and cognitive functional sizes are illustrated in Fig. 5.

Since all three implementations conform to the same algorithm specification, obviously they have the same cognitive functional size, known as $S_f = 9$ [CWU], as derived in Example 1. However, the physical sizes of different implementations, for the same cognitive functional size, vary within a wide range because the number of lines of code

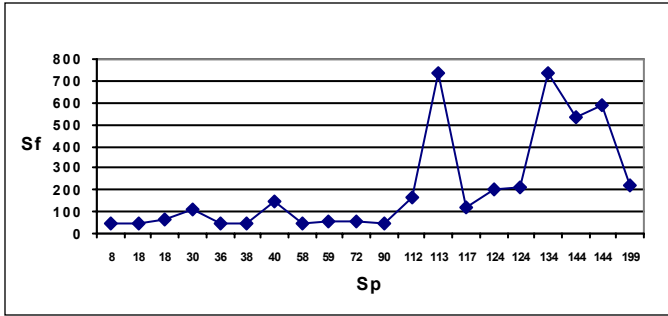


Figure 4: The cognitive complexity of software is not proportional to its physical size.

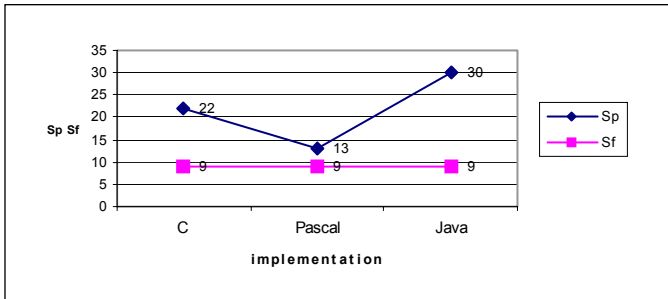


Figure 5: Physical size of implementations varies greatly compared with specifications with the same cognitive complexity.

```

// =====
// Algorithm of In-Between Sum (IBS)
// =====
Algorithm IBS (A, B: integer) : integer;
// In-between sum
const
    max = 65535;
begin
    readln (A, B);
    while not ((A > 0 and A < max) and (A > 0 and B < max)
        and (A < B))
    begin
        writeln ('Input error: A and/or B out of range,
            or A >= B. ');
        readln (A, B);
    end;
    IBS := ((B - 1) * B) / 2 - (A * (A + 1)) / 2;
    writeln ('IBS = ', IBS);
end.
    
```

Figure 6: Source code of the IBS algorithm (Implementation 2).

required to implement a program may vary greatly from programmer to programmer, and from language to language. This shows that the cognitive functional size based on the cognitive weights is a more robust measure of software complexity than the physical size, because the former possesses a fundamental merit that is independent of language, implementation, and programmer style. Therefore, the cognitive functional size and cognitive weights provide an objective, logical, and comparative measure for quantitatively analyzing and predicting software complexity and size in software engineering.

VI. Conclusions

Software complexity measures serve both as an analyzer and a predictor in quantitative software engineering. This paper has developed the cognitive functional size (CFS) on the basis of cognitive weights, per-

```

// =====
// Algorithm of In-Between Sum (IBS)
// =====
package IBSAlgorithm;
import java.io.*;

public class Untitled1 {
    public Untitled1() {
    }
    public static void main(String[] argv) throws Exception {

        Untitled1 tIBSAlgorithm = new Untitled1();
        String sUserInput;
        int A, B, sum=0;

        BufferedReader in = new BufferedReader(new
            InputStreamReader(System.in));
        String str = new String();

        System.out.println("Enter an integer A: ");
        str = in.readLine();
        A = Integer.parseInt(str);
        System.out.println("Entered an integer B: ");
        str = in.readLine();
        B = Integer.parseInt(str);

        if (!tIBSAlgorithm.ValidInput(A, B))
            System.out.println("Invalid input");
        else
        {
            sum = (B * (B - 1)) / 2 - (A * (A + 1)) / 2;
            System.out.println("In-Between-Sum is:" + sum);
        }
    }
    public boolean ValidInput (int A, int B)
    {
        int Max=30000;
        int Min=0;
        return ((Min<A && A<Max) && (Min<B && B<Max) &&
            (A<B));
    }
}
    
```

Figure 7: Source code of the IBS algorithm (Implementation 3).

mitting determination of software complexity from both architectural and cognitive aspects. Cognitive weights for basic control structures (BCSSs) have been introduced to measure the complexity of logical structures of software. Real-time process algebra (RTPA) has been adopted to describe and measure software complexity. A large set of case studies has been carried out to analyze the relationship between physical size and cognitive functional size of software. The cognitive functional size has been shown to be a fundamental measure of software artifacts based on the cognitive weight.

This work has produced three substantial findings: (a) The CFS of software in design and comprehension is dependent on three factors: internal processing structures, as well as the number of inputs and outputs. (b) The cognitive complexity measure (CFS) is more robust than the physical-size measure and independent of language/implementation. (c) CFS provides a foundation for cross-platform analysis of complexity and size of both software specifications and implementations for either design or comprehension purposes in software engineering.

Acknowledgements

The authors would like to acknowledge the Natural Sciences and Engineering Research Council (NSERC) of Canada for its support of this work. The authors would also like to thank the IEEE Canadian Conference on Electrical and Computer Engineering 2003 (CCECE'03) Program Committee for the Student Paper Competition award, as well as the reviewers and colleagues for their valuable comments and suggestions about this work.

References

- [1] V.R. Basili, *Qualitative Software Complexity Models: A Summary in Tutorial on Models and Methods for Software Management and Engineering*, Los Alamitos, Calif.: IEEE Computer Society Press, 1980.
- [2] J.K. Kearney, R.L. Sedlmeyer, W.B. Thompson, M.A. Gary, and M.A. Adler, *Software Complexity Measurement*, vol. 28, New York: ACM Press, 1986, pp. 1044–1050.
- [3] M.H. Halstead, *Elements of Software Science*, North Holland, N.Y.: Elsevier, 1977.
- [4] T.H. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, Dec. 1976, pp. 308–320.
- [5] Y. Wang, "On cognitive informatics: Keynote lecture," in *Proc. 1st IEEE Int. Conf. Cognitive Informatics (ICCI'02)*, Calgary, Alta., Aug. 2002, pp. 34–42.
- [6] ———, "Component-based software measurement," chap. 14 in *Business Component-Based Software Engineering*, ed. F. Barbier, Boston: Kluwer Academic Publishers, 2002, pp. 247–262.
- [7] C.A.R. Hoare, I.J. Hayes, J. He, C.C. Morgan, A.W. Roscoe, J.W. Sanders, I.H. Sorensen, J.M. Spivey, and B.A. Suffrin, "Laws of programming," *Comm. ACM*, vol. 30, no. 8, Aug. 1987, pp. 672–686.
- [8] Y. Wang, "The real-time process algebra (RTPA)," *Annals of Software Engineering*, vol. 14, Oct. 2002, pp. 235–274.
- [9] ———, "Using process algebra to describe human and software behaviors," *Brain and Mind*, vol. 4, no. 2, 2003, pp. 199–213.
- [10] R. Wiener and L.J. Pinson, *Fundamentals of OOP and Data Structures in Java*, Cambridge: Cambridge University Press, 2000.