Empirical Studies on the Functional Complexity of Software in Large-Scale Software Systems

Yingxu Wang, University of Calgary, Canada
Vincent Chiew, University of Calgary, Canada

ABSTRACT

Functional complexity is one of the most fundamental properties of software because almost all other software attributes and properties such as functional size, development effort, costs, quality, and project duration are highly dependent on it. The functional complexity of software is a macro-scope problem concerning the semantic properties of software and human cognitive complexity towards a given software system; while the computational complexity is a micro-scope problem concerning algorithmic analyses towards machine throughput and time/space efficiency. This paper presents an empirical study on the functional complexity of software known as cognitive complexity based on large-scale samples using a Software Cognitive Complexity Analysis Tool (SCCAT). Empirical data are obtained with SCCAT on 7,531 programs and five formally specified software systems. The theoretical foundation of software functional complexity is introduced and the metric of software cognitive complexity is formally modeled. The functional complexities of a large-scale software system and the air traffic control systems (ATCS) are rigorously analyzed. A novel approach to represent software functional complexities and their distributions in software systems is developed. The nature of functional complexity of software in software engineering is rigorously explained. The relationship between the symbolic and functional complexities of software is quantitatively analyzed.

Keywords: Cognitive Complexity, Empirical Studies, Formal System Modeling, Functional Complexity, Mathematical Models, Measurement Tool, Metrics, Semantic Space, Software Science

1. INTRODUCTION

Although computational complexity of algorithms and programs have been well studied in computer science (Hartmanis & Stearns, 1965; Hartmanis, 1994; Zuse, 1997; Wang, 2009), the functional complexity of software in software engineering (McDermid, 1991; Wang, 2009a) is yet to be rigorously explored from both theoretical and empirical aspects. If it is perceived that

DOI: 10.4018/jssci.2011070103
the computational complexity is a micro-scope problem concerning algorithmic analyses towards machine’s throughput and efficiency, the functional complexity of software is a macro-scope problem concerning the semantic space of software and human cognitive complexity towards a given software system (Wang, 2003a, 2007b; Wang & Chiew, 2010; Wang et al., 2006). Conventional computational complexity theories are mainly focused on time and space properties of a given problem, which is usually a function of the input size $O(f(n))$ in the domains of real number ($\mathbb{R}$). However, software functional complexity is a two dimensional hyper-structure ($\mathbb{HS}$) between the interactions of the architectural data objects and the behavioral operations (Shao & Wang, 2003; Wang, 2009). Therefore, there is a practical need to study the functional properties of software and how they fundamentally affect human cognition, design, and manipulation in software engineering.

It is recognized that functional complexity is one of the most fundamental properties of software, because almost all other software properties and attributes such as functional size, development effort, costs, quality, and project duration, are highly dependent on it. The quantification and measurement of software functional complexity have been a persistent fundamental problem in software engineering (Hartmanis & Stearns, 1965; Basili, 1980; Kearney et al., 1986; Melton, 1996; Fenton & Pfleeger, 1998; Lewis & 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 & Gaffney, 1983; McDermid, 1991), structural complexity (control flow, cyclomatic) (McCabe, 1976; Zuse, 1977), functional complexity (function points, cognitive complexity) (Albrecht, 1979; Wang, 2007a, 2009; Shao & Wang, 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 & 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 & 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 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 defined instead of various empirical studies.

In order to improve the understanding of the nature of software functional complexity, the semantic properties of software have to be systematically studied (Wang, 2006, 2009, 2001). The two-dimensional semantic space of software indicates that the cognitive complexity of software is a measure for the functional complexity for both software design and comprehension, which is a product of the architectural and operational complexities of software. Cognitive complexity provides a profound approach to explain and measure the functional complexity of software as well as the associated effort in software engineering. The cognitive complexity metrics consider the effect of both internal structures of software and the I/O data objects under processing (Wang, 2009), which is a formal measurement for cross-platform analysis of complexities, sizes, as well as development efforts and costs of software systems in the phases of design, implementation, and maintenance in software engineering.
This paper presents an empirical study on the functional complexity of software known as cognitive complexity based on large-scale samples and the development of a Software Cognitive Complexity Analysis Tool (SCCAT). Empirical data are obtained with SCCAT on 7,531 shareware programs in late phase processes of software engineering and 5 formally specified software systems in early phase of software engineering. In the remainder of this paper, Section 2 presents the theoretical foundations of the functional complexity of software, which introduces the mathematical model and the metrics of software cognitive complexity. Section 3 analyzes the functional and cognitive complexities of a large-scale software system, the air traffic control systems (ATCS), based on the functional, architectural, and behavioral complexities and their distributions in the software system. On the basis of the formal models of software functional complexities and the empirical data of the ATCS system, the nature of functional complexity of software and empirical observations in software engineering are explained in Section 4. The relationship between the symbolic and functional complexities of software is rigorously revealed. The large-scale real world data provide solid evidences for the effectiveness of the functional complexity measurement of software systems and the practical applications of the cognitive complexity metrics in software engineering.

2. THEORETICAL FOUNDATIONS OF SOFTWARE FUNCTIONAL COMPLEXITY

It has been empirically observed that programmers do feel fundamentally different from that of machines on the functional complexity of software in software engineering. Further, the functional complexity is exponentially unpredictable when the symbolic size [LOC] is above a certain threshold of 100-200 [LOC] (Shao & Wang, 2003). The curiosity to explain the nature of software complexity leads to the study on the functional complexity of software in general and the cognitive complexity of software in particular.

2.1. The Mathematical Models of Functional Complexity of Software

It is observed that the nature of software functional complexity is underpinned by the structure of the semantic space of software (Wang, 2006, 2007a, 2008e). The characteristics of the semantic space can be formally described by the semantic function of programs.

**Definition 1.** The semantic function of a program $\phi$, $f_0(\phi)$, 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.:

$$f_0(\phi) = f : T \times S \rightarrow V$$

where

$$f_0(\phi) = f(t_0, t_1, \ldots, t_n) = f(s_1, s_2, \ldots, s_m)$$

and

$$V = \{v_1, v_2, \ldots, v_{nm}\}$$

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

Copyright © 2011, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
According to Definition 1, the semantic space of a program can be represented by a two-dimensional plane as shown in Figure 1. The semantic space of software indicates that software functional complexity is not a simple linear entity. Instead, it is complex hyper-structure that is proportional not only to the number of operations (operational complexity), but also to the number of data objects under operation (architectural complexity). That is, the functional complexities of software are a product of its operational and architectural complexities.

**Theorem 1.** The functional complexity of software is proportional to the size of the Cartesian plain of the semantic space of software, which is a product of the operational complexity and the object architectural complexity.

**Proof 1.** Theorem 1 can be directly proven on the basis of Definition 1.

Theorem 1 indicates that the functional complexity of software is not a one-dimensional linear structure as that of the symbolic complexity [LOC] suggests. In other words, the symbolic complexity in the unit of [LOC] may not be suitable and rigor for predicting and measuring software functional complexity.

### 2.2. The Metrics of Functional Complexity of Software

Definition 1 and Theorem 1 indicate that software functional complexity needs to be rigorously measured with a two-dimensional metric that considers both the operational and structural complexities as a product. With this notion, the measurement of cognitive complexity of software is derived in the following subsections.

It is recognized that there are commonly ten basic control structures (BCS’s) in software modeling as shown in Table 1 (Wang, 2005, 2007a) according to Real-Time Process Algebra (RTPA) (Wang, 2002, 2008a, 2008b, 2008c, 2008d, 2008e). The cognitive weights of the set of BCS’s have been quantitatively determined based on a series of empirical experiments (Shao & Wang, 2003) on their relative cognitive coefficients as given in Table 1. It is noteworthy that, although the absolute cognitive weight towards a BCS may be variously different from individual to individual in program design and comprehension, the relative cognitive weights between the ratio $w'(BCS_i)/w(BCS_1)$, $2 \leq i \leq 10$, remain stable. In other words, the relative ratio normalized on the most basic sequential BCS helps to eliminate the subjectivity among individuals in determining the objective cognitive weights of the ten BCS’s.
There are two structural patterns of BCS’s in a given software system $S$: the sequential and the embedded BCS’s. In the former, all BCS’s are related in a linear layout in $S$, therefore the operational complexity of $S$ is a sum of the cognitive weights of all linear BCS’s. However, in the latter, some BCS’s are embedded in others, hence the operational complexity of $S$ is a sum of the sums of the cognitive weights of inner BCS’s. In general, both types of BCS configurations in $S$ may be combined in various ways. Therefore, a general method for calculating the operational complexity of software can be derived as follows.

**Definition 2.** 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, $w(BCS)$, i.e.:

$$
C_{op}(S) = \sum_{k=1}^{n} C_{op}(C_k) = \sum_{k=1}^{n} \sum_{i=1}^{m_k} w(k, i) \ [F]
$$

(2)

The cognitive weights, $w(k, BCS)$, in Eq. 2 can be formally modeled and empirically studied as shown in Table 1.

**Definition 3.** The relative cognitive weight of a BCS, $w(BCS)$, is determined by a relative ratio between the tested comprehension effort $w'(BCS)$ and the reference comprehension effort of the sequential BCS, $w'(BCS_1) = 1$, i.e.:

$$
w(BCS_i) = \frac{w'(BCS_i \mid 2 \leq i \leq 10)}{w'(BCS_1 \mid w'(BCS_i) = w'(SEQ) \equiv 1)}
$$

(3)

**Definition 4.** The unit of operational complexity of a software system $S$ is a single sequential operation called a unit function $F$, i.e.:
With the cognitive weight of the sequential operator defined as a unit function for the operational complexity, general process relations in software structures can be quantitatively measured using Eq. 2. It is obvious that for a fully sequential software system where only \( w(\text{sequence}) = 1 \) \([F]\) is involved, its operational complexity is reduced to the symbolic complexity equivalent to LOC. However, in more generic cases, the actual operational complexity of software is much greater than that of the symbolic complexity in LOC.

According to Theorem 1, the second dimension that determines the functional complexity of software is the architectural complexity of a given software system. The architectural complexity is proportional to the number of its global and local data objects, such as inputs, outputs, data structures, and internal variables modeled in the program.

**Definition 5.** 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.:

\[
C_a(S) = \text{OBJ}(S) = \sum_{j=1}^{n_{\text{sys}}} \text{OBJ}(\text{UDM}_j) + \sum_{k=1}^{n_c} \text{OBJ}(C_k) \quad [O]
\]

where \( \text{OBJ} \) is a function that counts the number of data objects in a given data structure. The first item in Eq. 5 measures the number of global variables known as the unified data model (UDM). The second item is equivalent to the sum of the number of local variables in all \( n_c \) components.

**Definition 6.** The *unit of architectural complexity* of software is a single data object, modeled either globally or locally, called an object \( O \), i.e.:

\[
C_a(S) = 1 \ [O] \iff \#(\text{OBJ}(S)) = 1
\]

On the basis of the elaborations of software architectural and operational complexities, the cognitive complexity of software systems is introduced as follows as a fundamental measure of the functional complexity and sizes of software systems.

**Theorem 2.** 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.:

\[
C_c(S) = C_{op}(S) \cdot C_a(S) = \left\{ \frac{\sum_{j=1}^{n_{\text{sys}}} \text{OBJ}(\text{UDM}_j) + \sum_{k=1}^{n_c} \text{OBJ}(C_k)}{n_c} \right\} \cdot \left\{ \sum_{j=1}^{n_{\text{sys}}} \text{OBJ}(\text{UDM}_j) + \sum_{k=1}^{n_c} \text{OBJ}(C_k) \right\} \quad [FO]
\]
Theorem 2 indicates that the functional complexity, in term of the cognitive complexity, is a chain of embedded computational operations onto a set of data objects modeled in the software. Because the cognitive complexity of a software system is proportional to both its operational and architectural complexities, the more the architectural data objects and the higher the operational complicity applied on these data objects, the higher the functional complexity of the given system.

**Definition 7.** The unit of cognitive complexity of software is a single sequential operation onto a single data object called a function-object \( \text{FO} \), i.e.:

\[
C_f = C_{op} \cdot C_a = 1 \cdot [F] \cdot 1 \cdot [O] = 1 \cdot [\text{FO}]
\]  

(8)

According to Definition 7, the physical meaning of functional complexity is how many equivalent function-objects \( [\text{FOs}] \) modeled in a given software system. The cognitive complexity as a formal measure of software size and functionality enables the quantitative determination of software functional complexity and design efforts in software engineering. It will be demonstrated in the next section that the cognitive complexity is the most distinguishable and more accurate measure for the inherent functional complexity of software systems.

**Corollary 1.** Software functional size is orthogonal, which is proportional to both its operational and architectural complexities. The more the architectural data objects and the higher the operational complicity applied on the data objects, the larger the functional size of the system.

### 3. ANALYSES OF THE FUNCTIONAL COMPLEXITY OF THE AIR TRAFFIC CONTROL SYSTEM (ATCS)

An Air Traffic Control System (ATCS) is among the most demanding software systems and the most challenging software design technologies (Ross, 1986; Bass, Clements, & Kazman, 1998; Smith, 1993; Gibbs, 1994; Perry, 1997; Nolan, 1999; Peter & Pedrycz, 2000; de Neufville & Odoni, 2003; Ball et al., 2007; Braun & Gianopulos, 2009; Hansman & Odoni, 2009; Kontogiannis & Malakis, 2009; Debelack et al., 2010). In a case study, Len Bass et al. reported that the ATCS is hard real-time, safety-critical, and highly distributed. It is a hard real-time system because timing demands must be met absolutely. It is a safety-critical system because human lives may be lost for any system malfunction. It is highly distributed because it requires a large group of controllers to work cooperatively to guide aircrafts through the airway system (Bass, Clements, & Kazman, 1998; Sha et al., 1990; Tanenbaum, 1994; Wang, Zeng et al., 2010; Wang, Ngolah et al., 2010).

It is recognized that over 2/3 large-scale software projects have been failed (McDermid, 1991; Bass, Clements, & Kazman, 1998; Wang, 2007a). A major reason of the failures was because the highly complex software systems created by a team may eventually not be able to understand by any individual in the project. The complexity may easily grow out of the intellectual manageability of an individual in the team at any level when the system is integrated in the final phase. Not only at the system architect’s and managers’ level who may lose their cognitive ability for pinpointing and tracing the details of system behaviors, but also at the programmers’ level who may lose their cognitive ability for comprehending the intricate connections and relationship of a certain component with the remainder of the entire system. In addition, the highly dependent
interpersonal coordination requirement may result in an extremely high rework rate when the system design is not rigorously specified in a formal and precise model. These dilemmas are identified as the key causes of the failures in complex software development such as the ATCS, the real-time operating system, the digital telephone switching system, and the distributed banking system (Wang, 2007a).

The following subsections present the applications of the metrics of functional complexity, particularly the cognitive complexity, in the quantitative analyses and modeling of the ATCS system.

3.1. The Functional Architecture of the ATCS Software System

The ATCS controls aircrafts (flights) passing through the air space within an administrative zone or across multiple zones. There are multiple entities in ATCS such as the airport, flights, aircrafts, tower controller, approach controller, en route controller, runways, gates, radars, communication networks, ground services, and passengers. The conceptual model of ATCS can be abstracted by a number of interconnected air traffic administration zones as shown in Figure 2. Each administration zone is divided into three control areas known as the terminal, approach, and en route areas. Corresponding to them, the controllers in the areas are known as the tower (airport) controller, approach controller, and en route controller. In the air space, an aircraft’s heading direction is modeled in 360° clockwise starting from East as 0°. That is, East, South, West, and North are defined as 0° (3 O’clock), 90° (6 O’clock), 180° (9 O’clock), and 270° (12 O’clock) in ATCS, respectively. ATCS dispatching functions are carried out by wireless communications in which verbal instructions are sent to the pilots and acknowledged verbally by the pilots.

The functional architectural model of the ATCS software is shown in Figure 3, which encompasses five subsystems known as those of system management, airport control, en route control, approach control, and system dynamic behaviors. Each software subsystem of ATCS is refined by a number of components at the third level of the system model. There are two components in the system management subsystem such as system initialization and system clock. The airport control subsystem consists of three components known as the terminal control, landing control, and takeoff control. Within each component in the airport control subsystem, there are sets of three, five, and five subcomponents, respectively. The subsystem of en route control involves the components of en route flight detection, incoming en route registration, incoming en route monitor, and outgoing en route monitor. The approach control subsystem encompasses components of approach flight detection, incoming approach monitor, and outgoing approach monitor. In the last subsystem of dynamic behaviors, five components are modeled known as the process deployment, landing dispatching, takeoff dispatching, en route dispatching, and approach dispatching.

In Figure 3, the distributions of both the symbolic sizes $C_s$ [LOC] and functional complexities in term of the cognitive complexity $C_c$ [FO] are identified for each subsystems and components, respectively, in the brackets of each component, which will be derived in the following subsections.

3.2. The Architectural Attributes of ATCS

The architecture of the ATCS software system can be rigorously modeled by a set of unified data models (Wang, 2007a) with a coherent set of data objects and attributes.
Definition 8. A Unified Data Model (UDM) is a generic architectural model for a software system as well as its internal control structures, and its interfaces with hardware components, which can be rigorously modeled and refined as an \( n \)-tuple, i.e.:

\[
UDM \triangleq \bigotimes_{i=1}^{n} R(S_i \mid \forall e \in S_i, p(e))
\]  

where \( \bigotimes_{i=1}^{n} R \) is the big-R notation of RTPA that denotes a repetitive structure or operation; \( S_i \), \( 1 \leq i \leq n \), is a set that is equivalent to a type in computing for elements \( e \), in which all \( e \) share the property \( p(e) \).
According to Definitions 5 and 6, the architectural complexity of the ATCS system can be rigorously analyzed. As a result, the configurations of UDMs and attributes of ATCS is shown in Table 2 where the unit of architectural complexity is objects (O). The architectural model of the ATCS system as shown in Table 2 encompasses 14 UDMs, which can be categorized into five subsystems known as those of the system, the airport control structures, landing control, takeoff control, and system control structures. Each UDM encapsulates a set of architectural attributes identified by the number of objects that is equivalent to the number of fields modeled in the UDMs. Further details may be referred to the complete models of ATCS in RTPA (Wang, 2002, 2008e, 2008c). Based on the 14 UDMs with 197 objects, the average architectural complexity of the ATCS software system is 12 [O] ranged from minimum 3 [O] to maximum 40 [O]. It is noteworthy that although the entire ATCS system may be highly complicated, its architectural and data objects are relatively limited with 12 objects in average.

3.3. The Behavioral Attributes of ATCS

The behavioural functions of the ATCS software system can be rigorously modeled by a set of unified process models (Wang, 2007a) that operate onto the architectural UDMs of ATCS.

**Definition 9.** The Unified Process Model (UPM) of a program \( \wp \) is a composition of a finite set of \( m \) processes according to the event-based process dispatching rules, \( e_k \mathbin{s_i} \{ \mathbin{P} \mathbin{C} \}_{i=1}^{\infty} \), i.e.:

\[
UPM \triangleq \mathbb{A} = \bigoplus_{k=1}^{m} \mathbb{A}_{e_k s_i P_{C_j} r_{ij}(k) s_i(k)} \]

where \( s_i \) and \( s_j \) are one of the 17 RTPA meta-processes, \( r_{ij} \) is one of the 17 RTPA algebraic process operators, and \( e_k \) is an event.

The functional model of the ATCS system encompasses seven functional subsystems known as those of system management, airport (terminal) control, en route control, approach control, landing control, takeoff control, and system dynamic behaviors as shown in Table 3. The 27 behavioral UPMs in each of the seven subsystems are listed in the second column in Table 3. The behavioral attributes of each UPM are comparatively analyzed by the conventional symbolic complexity \( C_s [\text{LOC}] \) and cognitive complexity \( C_c [\text{FO}] \). The latter is a product of the operational complexity \( C_{op} [F] \) and the architectural complexity \( C_a [O] \). According to Definition 2, the statistics of distributions of behavioral attributes in the 27 UPMs of the seven subsystems is shown in Table 3. Further details may be referred to the complete models of ATCS in RTPA (Wang, 2008a, 2008c).

According to the empirical and derived data obtained in Table 3, the average symbolic complexity of the ATCS system is only 23 [LOC] per component. However, the corresponding average cognitive complexity of ATCS is 7,058 [FO] per UPM. The equivalency between the symbolic complexity and cognitive complexity in ATCS is as follows:

\[
\begin{align*}
1.000 \, [\text{LOC}] & = 307.845 \, [\text{FO}] \\
1.000 \, [\text{FO}] & = 0.003 \, [\text{LOC}]
\end{align*}
\]
Eq. 11 and Table 3 indicate that because of the higher the complexity of a given system, the higher the ratio of FO/LOC, the symbolic complexity of software may result in a significant under-estimation of the functional complexity of software. This explains the reasons of why a typical large-scale and real-time software system such as ATCS is highly complicated, because it represents a tremendous cognitive functional complexity of a software system to the architects, programs, quality engineers, managers, and users. This observation also provides evidence to J.V. Guttag’s assertion that “large software system are among the most complex systems engineered by man (Guttag, 2002).”

3.4. Analysis of the Distributions of Functions and Complexities in the ATCS System

On the basis of the quantitative analyses of the architectural and functional attributes of ATCS using SCCAT, a rich set of empirical data are obtained for illustrating the function and complexity distributions in the ATCS system. The data provide a novel view to visually represent the architectural and functional complexities and sizes of any software system.
### Table 3. Behavioral attributes of the ATCS system

<table>
<thead>
<tr>
<th>No.</th>
<th>Description</th>
<th>Symbolic Complexity $C_s[LOC]$</th>
<th>Functional Complexity $C_f(FO) = C_c(FO)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>System Management</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>System Initialization</td>
<td>32</td>
<td>197</td>
</tr>
<tr>
<td>1.2</td>
<td>System Clock</td>
<td>19</td>
<td>48</td>
</tr>
<tr>
<td>2</td>
<td>Airport Control (Terminal Control)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.1</td>
<td>Terminal Flight Scan</td>
<td>11</td>
<td>93</td>
</tr>
<tr>
<td>2.2</td>
<td>Ground Flight Scan</td>
<td>11</td>
<td>96</td>
</tr>
<tr>
<td>2.3</td>
<td>Separation Control</td>
<td>17</td>
<td>88</td>
</tr>
<tr>
<td>3</td>
<td>En Route Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td>En route Flight Detection</td>
<td>10</td>
<td>95</td>
</tr>
<tr>
<td>3.2</td>
<td>Incoming En Route Registration</td>
<td>23</td>
<td>110</td>
</tr>
<tr>
<td>3.3</td>
<td>Incoming En Route Monitor</td>
<td>24</td>
<td>125</td>
</tr>
<tr>
<td>3.4</td>
<td>Outgoing En Route Monitor</td>
<td>32</td>
<td>118</td>
</tr>
<tr>
<td>4</td>
<td>Approach Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.1</td>
<td>Approach Flight Detection</td>
<td>11</td>
<td>93</td>
</tr>
<tr>
<td>4.2</td>
<td>Incoming Approach Monitor</td>
<td>26</td>
<td>121</td>
</tr>
<tr>
<td>4.3</td>
<td>Outgoing Approach Monitor</td>
<td>25</td>
<td>121</td>
</tr>
<tr>
<td>5</td>
<td>Landing Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>Waiting for Landing</td>
<td>56</td>
<td>122</td>
</tr>
<tr>
<td>5.2</td>
<td>Landing</td>
<td>42</td>
<td>129</td>
</tr>
<tr>
<td>5.3</td>
<td>Landed</td>
<td>17</td>
<td>105</td>
</tr>
<tr>
<td>5.4</td>
<td>Arriving Gate</td>
<td>32</td>
<td>126</td>
</tr>
<tr>
<td>5.5</td>
<td>Special Landing Processes</td>
<td>15</td>
<td>129</td>
</tr>
<tr>
<td>6</td>
<td>Takeoff Control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.1</td>
<td>Waiting for Departure</td>
<td>15</td>
<td>125</td>
</tr>
<tr>
<td>6.2</td>
<td>Taxi for Departure</td>
<td>14</td>
<td>125</td>
</tr>
<tr>
<td>6.3</td>
<td>Ready to Takeoff</td>
<td>37</td>
<td>122</td>
</tr>
<tr>
<td>6.4</td>
<td>Takingoff</td>
<td>33</td>
<td>122</td>
</tr>
<tr>
<td>6.5</td>
<td>Takenoff</td>
<td>38</td>
<td>122</td>
</tr>
<tr>
<td>7</td>
<td>System Dynamic Behaviors</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7.1</td>
<td>Process Deployment</td>
<td>41</td>
<td>48</td>
</tr>
<tr>
<td>7.2</td>
<td>Landing Dispatching</td>
<td>11</td>
<td>111</td>
</tr>
<tr>
<td>7.3</td>
<td>Takeoff Dispatching</td>
<td>11</td>
<td>111</td>
</tr>
</tbody>
</table>

*continued on following page*
A perspective on the architectural configuration of ATCS is shown in Figure 4 based on the data obtained in Table 2. In Figure 4, the numbers of data objects in the seven subsystems of ATCS, such as the high-level structures of the system, airport control, landing control, takeoff control, and system control are illustrated in the three-dimensional distributions. It is obvious that the system control and airport control are the two largest architectural subsystems in ATCS. In particular, components 5.2 (the system control block) and 5.3 (the flight control block), and 2.1 (the runways) are the three most complex structural components in ATCS.

The perspective on the functional configuration of ATCS can be represented by the distributions of its symbolic and cognitive complexities, respectively. The functional distribution of ATCS measured by the symbolic complexity $C_s [\text{LOC}]$ is illustrated in Figure 5, which shows the numbers of lines of code distributed in the seven subsystems such as system management, airport (terminal) control, en route control, approach control, landing control, takeoff control, and system dynamic behaviors.

The functional distribution of ATCS measured by the functional complexity $C_c [\text{FO}]$ is illustrated in Figure 6. Figure 6 shows the distributions of cognitive complexity in term of function-objects in the seven subsystems of ATCS. The cognitive complexity distribution in a system
provides a new approach to represent a more accurate software functional complexity and their partitions in a software system. It also provides an intuitive indicator for showing where the main workload and costs are located in a software development project in software engineering.

Contrasting Figures 5 and 6, a number of interesting observations about the nature of software complexity distributions may be obtained. For example, the conventional symbolic complexity distribution in Figure 5 may suggest that the size/effort distribution in ATCS is in a descending pattern across subsystems (5, 6, 3, 7, 4, 1, 2). However, the functional measures in Figure 6 reveal a different view as that of (5, 6, 7, 3, 4, 2, 1). In other words, the real functional complexity, as well as effort and costs of ATCS, is distributed not only differently, but also at a much higher magnitude up to 190,466 [FO].

Figure 5. The functional distribution of ATCS in symbolic size ($C_s [LOC]$)

Figure 6. The functional distribution of ATCS in cognitive size ($C_c [FO]$)
4. FUNDAMENTAL FINDINGS ON SOFTWARE FUNCTIONAL COMPLEXITY

On the basis of the formal models of software functional complexities in Section 2 and the empirical studies on a number of large-scale software systems using the SCCAT tool in Section 3, the nature of the functional complexity can be better explained in software engineering. New perspectives on the relationship between the symbolic and functional complexities of software can also be rigorously analyzed.

4.1. The Nature of Functional Complexity of Software

The orthogonality of software functional complexity is formally explained in Theorem 1 and Corollary 1, which provides the theoretical foundation of software functional complexity. The orthogonal property of functional complexity shows that the more the architectural data objects and the higher the operational complicity onto these data objects, the higher the functional complexity of a given software system. In case any or both of the orthogonal dimensions is/are zero, there is no functional complexity in the given system.

**Corollary 2.** Software functionality is embodied by an orthogonal interaction (operation) between the data objects modeled in the architectures (UDMs) and the behavioral operations modeled by the computational processes (UPMs).

**Corollary 3.** The basic unit of software functional complexity is a function-object (FO) determined by two independent attributes known as the architectural complexity $C_a[O]$ and operational complexity $C_o[F]$.

**Corollary 4.** 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.:

$$C_{op}(S) = \sum_{k=1}^{m} \sum_{i=1}^{n} w(k, i)$$

$$= C_s(S), w(k, i) \equiv 1$$

$$= C_s(S) [LOC]$$

Corollary 4 presents an important property of the relationship between conventional symbolic complexity and the operational complexity of software. It reveals that the measure of symbolic complexity in LOC is oversimplified on software functional complexity. As a result, it cannot actually represent the real functional complexities and sizes of software systems. Further, real-world case studies show that programs with similar symbolic complexities may possess widely different functional complexities in terms of the cognitive complexities. According to Corollary 4, the actual complexity and size of software systems were significantly underestimated when the conventional symbolic measurement in [LOC] is adopted. Therefore, a more accurate and rational complexity measurement, the cognitive complexity, needs to be established in software engineering.

4.2. Empirical Findings on Software Functional Complexity

It is a profound finding that functional complexity of software in software engineering is fundamentally different from that of the time and space complexity of algorithms traditionally mod-
eled in computer science. Further, the cognitive complexity of software is significantly different from the points of view of human beings and machines in software engineering. Based on the empirical data obtained from the ATCS system and the SCCAT tool, the following observations and findings are identified on software functional complexities:

a) Although a software system would be extremely large, the average size of its components is usually quite small. For example, since there are 27 UPMs with 619 [LOC] in the ATCS system, the average symbolic complexity of ATCS is only 23 [LOC] per components corresponding to an average cognitive complexity of 7,058 [FO]. These empirical evidences provide solid support for the well known principles of modularization and component-based development in software engineering, which could not be explained using the symbolic complexity measure.

b) The symbolic complexity may dramatically under estimate the real complexity of software systems. For example, as shown in Table 4, with the same symbolic complexities \( C_s(2.1) = C_s(2.2) = 11 \) [LOC] for the UPMs of terminal flight scan and ground flight scan, their cognitive complexities would be largely different, i.e., \( C_c(2.1) = 2,418 \) [FO], while \( C_c(2.2) = 1,920 \) [FO].

c) The symbolic complexity is not sensitive to represent the functional complexities of software. For example, as shown in Table 4, although the difference \( \Delta C_s = C_s(6.1) - C_s(6.2) = 15 - 14 = 1 \) [LOC] for the UPMs of waiting for departure and taxi for departure, the real difference of their functional complexities would be as great as \( \Delta C_c = C_c(6.1) - C_c(6.2) = 4,875 - 6000 = -1,125 \) [FO].

d) Following observations (b) and (c), the effort \( E \) or workload of a software project may not be accurately estimated on the basis of the symbolic complexity \( C_s \) [LOC], because there is a lack of linear or predicitve correlation between \( C_s \) and \( E \). This finding helps to explain why so many software engineering projects, particularly large ones, were failed due to budget deficits, unacceptable delays, and/or quality sacrifices.

e) The configuration of complex software systems may be better expressed by the 3-D distributions of system functional complexity among all components in different subsystems as shown in Figures 3 and 6.

f) The optimal symbolic size \( C_s \) of a software component is constrained by a threshold of 100 - 200 [LOC] (Shao & Wang, 2003; Chew & Wang, 2006). No well structured component would be greater than the threshold. Otherwise, the cognitive complexity \( C_c \) and development effort \( E \) will be largely unpredictable.

The findings demonstrate that the cognitive complexity is highly distinguishable for modeling and measuring software functional complexities, sizes, and efforts of software projects in software engineering. Therefore, the functional complexity measure in term of the cognitive complexity is more accurate than that of the symbolic complexity in lines of code.

5. CONCLUSION

This paper has presented an empirical study on the functional complexity of software known as cognitive complexity based on a large set of large-scale samples. In order to demonstrate that programmers do perceive software functional complexities differently from that of computers, and to rationally model the functional sizes and complexities of software systems, the semantic space of software and its cognitive foundation have been studied. According to the cognitive
complexity theory, it is found that the traditional time, cyclomatic, and symbolic complexities do not work very well to actually reflect the real complexity of software systems in design, representation, cognition, and/or comprehension in software engineering. The cognitive complexity of software systems has been formally and rigorously elaborated with comparative case studies. The Software Cognitive Complexity Analysis Tool (SCCAT) has been developed to automatically analyze a comprehensive set of real-world programs and large-scale software systems.

This paper has revealed that the functional complexity of software is a product of its architectural and operational complexities on the basis of software semantic space theories. It has been found that programmers and computers are sensitive to different forms of software complexities, where the former focus on the functional complexity in software engineering based on the semantics of software; while the latter put emphases on the computational complexity based on time and space consumptions. A number of interesting characteristics on software functional complexities have been identified such as: a) Programs with similar symbolic complexities $C_s[LOC]$ can possess greatly different cognitive complexities $C_c[FO]$; b) The symbolic complexity $C_s[LOC]$ does not represent the throughput or the input size of problems, which may be captured by the architectural complexity measure $C_a[O]$; and c) Cognitive complexity $C_c[FO]$ provides a more accurate measurement for the real semantic complexity of software systems by coherently integrating both the operational and architectural complexities, which provide insights into human cognition in software engineering.

ACKNOWLEDGMENT

The authors 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


Yingxu Wang is professor of cognitive informatics, cognitive computing, and software engineering, President of International Institute of Cognitive Informatics and Cognitive Computing (IICICC), and Director of the Cognitive Informatics and Cognitive Computing Lab at the University of Calgary. 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 received a PhD in Software Engineering from the Nottingham Trent University, UK, and a BSc in Electrical Engineering from Shanghai Tiedao University. He has industrial experience since 1972 and has been a full professor since 1994. He was a visiting professor on sabbatical leaves 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 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 Intelligence (IJSSCI), Associate Editor of IEEE Trans on System, Man, and Cybernetics (Part A), associate Editor-in-Chief of Journal of Advanced Mathematics and Applications, and Editor-in-Chief of CRC Book Series in Software Engineering. Dr. Wang is the initiator of several cutting-edge research fields or subject areas such as cognitive informatics, abstract intelligence, cognitive computing, cognitive computers, denotational mathematics (i.e., concept algebra, inference algebra, system algebra, real-time process algebra, granular algebra, and visual semantic algebra), software science (on unified mathematical models and laws of software, cognitive complexity of software, and automatic code generators, coordinative work organization theory, built-in tests (BIs), and deductive semantics of languages), the layered reference model of the brain (LRMB), the mathematical model of consciousness, and the reference model of cognitive robots. He has published over 110 peer reviewed journal papers, 220+ peer reviewed full conference papers, and 16 books in cognitive informatics, software engineering, and computational intelligence. He is the recipient of dozens international awards on academic leadership, outstanding contributions, research achievement, best papers, and teaching in the last three decades.

Vincent Chiew is a PhD candidate in cognitive computing and software engineering with the Cognitive Informatics and Cognitive Computing Lab and the Theoretical and Empirical Software Engineering Center (TESERC) in dept. of electrical and computer engineering at the University of Calgary, Canada. He received an MS degree in software engineering from University of Calgary in 2003. He is senior member of IEEE and has more than 20 years experience in the software industry. His research interests are in software engineering, cognitive informatics, cognitive computing, and industrial applications.