Software Science: On the General Mathematical Models and Formal Properties of Software

ARTICLE · DECEMBER 2014

DOI: 10.1166/jama.2014.1060

1 AUTHOR:

Yingxu Wang
The University of Calgary
399 PUBLICATIONS **3,595** CITATIONS

Journal of
Advanced Mathematics and Applications
Vol. 3, 130–147, 2014

RESEARCH ARTICLE

Software Science: On the General Mathematical Models and Formal Properties of Software

Yingxu Wang

International Institute of Cognitive Informatics and Cognitive Computing (ICIC), Laboratory for Cognitive Informatics, Denotational Mathematics, and Software Science, Department of Electrical and Computer Engineering, Schulich School of Engineering University of Calgary, 2500 University Drive, NW, Calgary, Alberta, T2N 1N4, Canada

Software science is a discipline that studies the formal properties and mathematical models of software, general methodologies for rigorous and efficient software development, and coherent theories and laws underpinning software behaviors and software engineering practices. This paper presents a general mathematical model of software (GMMS). It reveals that software is not only an interactive dispatch structure at the top level driven by the trigger, timing, and interrupt events (E), but also a set of embedded relational processes at intermediate levels of components or subsystems. The GMMS model formally describes the abstract entities of software by structure models (*SMs*) and the functional behaviors of software by process models (*PMs*). As a result, the overall mathematical model of software systems (*SS*) is formally derived as a Cartesian product $SS = E \times PM \times SM$ for any form and size of software systems. Case studies demonstrate that novel, rigorous, and efficient methodologies for software engineering can be deductively developed based on the formal theories of software science.

Keywords: Software Science, Theoretical Foundations, General Mathematical Models, Structure Models (SMs), Process Models (PMs), System Models, Properties of Software, Embedded Processes, Dispatch Structures, Denotational Mathematics, Formal Methods, RTPA, Software Engineering, Applications.

1. INTRODUCTION

The nature of software is perceived quite differently in research and practice of computing and software engineering (von Neumann, 1946, 1963; Turing, 1950; Hoare, 1969, 1978; Dijkstra, 1976; Guttag, 1977; Higman, 1977; Hopcroft and Ullman, 1979; Cries, 1981; Mandrioli and Ghezzi, 1987; Wilson and Clark, 1988; Sommerville, 1995; Lewis and Papadimitriou, 1998; McDermid, 1991; Louden, 1993; Wang, 2006, 2007a, 2008b, f, 2009f; Wang and King, 2000; Wang and Patel, 2000, 2009). The concept of software is used to be treated as a listing of statements as for granted. Typical metaphors and perceptions about software are such as those of *mathematical entities* (Turing, 1950; Milner, 1980; Bishop, 1986; Hoare et al., 1987; Gowers, 2008; Wang, 2002, 2007a, 2008b, f), *computational logic* (Horn, 1951; Kowalski, 1988; Boolos, 2002), *automata or finite state machines* (FSMs) (von Neumann, 1963; Hopcroft and Ullman, 1979; Lewis and Papadimitriou, 1998), *algorithms* (Wirth, 1976; Baase, 1978; Bjorner and Jones, 1982; Knuth, 1997; Wang, 1996, 2008c), *behavioral processes* (Cerone, 2000; Wang, 2003b; Wang and King, 2000), *programming language entities* (Dijkstra, 1976; Higman, 1977; Gries, 1981; Horowitz, 1984; Bishop, 1986; Wilson and Clark, 1988; Louden, 1993; Wang, 2009a), *program classes and objects* (Dahl and Nygaard, 1967; Meyer, 1988; Wang, 2001a; Wang and Patel, 2004; Wang and Huang, 2008; Wang et al., 2000), *application products* (Gibbs, 1994; McDermid, 1991; Sommerville, 1995; Bass et al., 1998; Parnas, 2001; Wang, 2001b, 2005; Wang and Bryant, 2002; Wang et al., 1998, 1999a, b), *instructive information* (Llewellyn, 1987; McDermid, 1991; Wang, 2006), *intelligent behaviors* (Glorioso and Osorio, 1980; McDermid, 1991; Wang, 2003a, 2003b, 2007a, 2008d, 2009b), and *abstract systems* (Milner, 1980; Klir, 1992; Wang 2007a, 2008g, h, 2014b, 2015). Despite the rich repository of empirical knowledge on programming and software development, a rigorous theoretical framework of software science is yet to be sought.

DEFINITION 1. *Software* is an abstract representation of both executable *computing structures* as typed tuples and *instructive behaviours* as a *chain of embedded functions* interacting between the abstract and physical worlds.

 2156-7565/2014/3/130/018 doi:10.1166/jama.2014.1060

Software is a complex system that consists of a large set of intricately heterogeneous and interconnected components. Changes at one point of software may affect functions of the entire system due to propagations of interaction inconsistencies via highly coupled structures and intricately interactive functions. The necessary and sufficient conditions for software dependency in a system are the needs for *repeatability*, *programmability*, and *run-time determinability*, which transform a general computer platform to a specific intelligent system.

Any scientific discipline studies the structures and functions of an aspect of the natural or the abstract world (Aristotle, 384 BC–322 BC; Newton, 1729; Descartes, 1979; Russell, 1903). Fundamental theories of software science are type theories (Martin-Lof, 1975; Guttag, 1977; Cardelli and Wegner, 1985; Mitchell, 1990; Wang, 2007a; Wang et al., 2010e) and the process metaphor (Hoare, 1978, 1985; Miller, 1980; Wang, 2002, 2007a). The former indicate that the structural facet of software may be formally modeled by a set of numerical types where a category of software objects share common properties and allowable operations. The latter reveal that the functional facet of software may be formally described as a single or complex behavioral process. Hoare (1978), Milner (1980), and others developed various algebraic approaches to represent interacting and concurrent computing systems known as communicating sequential processes (CPS) (Hoare, 1978, 1985). Hoare and his colleagues studied laws of programs (Hoare et al., 1987), which have recently extended by Wang to the mathematical laws of software systems (Wang, 2008f). The real-time process algebra (RTPA) is developed as a denotational mathematics for rigorously modeling and manipulating intelligent behavioral processes of both software systems and humans (Wang, 2002, 2007b, 2008a, f). A comprehensive deductive semantics of RTPA is formally described in (Wang, 2008b, 2010b). The framework of denotational mathematics may be referred to (Wang, 2008e, 2009a, 2012a, b, d, e, 2011, 2013, 2014) towards software science, cognitive informatics (Wang, 2003a, 2007b, 2009d, e, 2010a, 2012c; Wang and Chiew, 2011; Wang et al., 2006, 2009b), and computational intelligence (Wang, 2009b, f; Wang and Chiew, 2010).

DEFINITION 2. Software engineering is applied software science that adopts engineering approaches, such as established formal methodologies, processes, measurements, tools, standards, organizational patterns, quality assurance systems and the like, in the development of large-scale software towards high productivity, low cost, controllable quality, and predictable schedule.

RESEARCH ARTICLE

Yingxu Wang is professor of denotational mathematics, cognitive informatics, software science, and brain science. He is President of International Institute of Cognitive Informatics and Cognitive Computing (ICIC), Director of Laboratory for Cognitive Informatics and Cognitive Computing, and Director of Laboratory for Denotational Mathematics and Software Science in Department of Electrical and Computer Engineering at the University of Calgary. He is a founding Fellow of ICIC, a Fellow of WIF (UK), a P.Eng. of Canada, a Senior Member of IEEE and ACM. He received a Ph.D. in Computer Science from the Nottingham Trent University, UK, and a B.Sc. 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 leave) in the Computing Laboratory at Oxford University in 1995, Department of Computer Science at Stanford University in 2008, the Berkeley Initiative in Soft Computing (BISC) Lab at University of California, Berkeley in 2008, and CSAIL at MIT (2012), respectively. 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 SMC (Systems), and Editor-in-Chief of Journal of Advanced Mathematics and Applications (JAMA). Dr. Wang is the initiator of a few cutting-edge research fields such as *Denotational Mathematics* (i.e., concept algebra, semantic algebra, inference algebra, process algebra, system algebra, granular algebra, visual semantic algebra, and fuzzy arithmetic/probability/calculus); *Cognitive Informatics* (theoretical framework of cognitive informatics, neuroinformatics, neurocomputing, the layered reference model of the brain (LRMB), the mathematical model of consciousness, and the cognitive learning engine); *Abstract Intelligence* (αI and mathematical models of the brain); *Cognitive Computing* (cognitive computers, cognitive robots, cognitive agents, and cognitive knowledge base); *Software Science* (general mathematical models of software, cognitive complexity of software, automatic code generators, the coordinative work organization theory, and built-in tests (BITs)); *Cognitive Linguistics* (theoretical framework of abstract languages, deductive syntax, cognitive semantics, deductive grammar of English, and cognitive complexity of text comprehension). He has served as general chairs/program chairs/keynote speakers in numerous international conferences. He has published 400+ peer reviewed papers and 28 books in cognitive informatics, denotational mathematics, cognitive computing, software science, and computational intelligence. He is the recipient of dozens international awards on academic leadership, outstanding contributions, best papers, and teaching in the last three decades.

The nature of software engineering and its theories and methodologies are determined by the nature of the objects under study, software, and the needs for mathematical, theoretical, and methodological means. It is reported that over 2/3 complex and large-scale software projects have been failed in the history of software engineering because the extremely high complexity of software systems created by a team may eventually not be able to understand by any individual in the team (McDermid, 1991; Sommerville, 1995; Bass et al., 1998; Wang, 2007a; Wang and King, 2000). The complexity of software may easily grow out of the intellectual manageability of an individual at any level in a project when the system is integrated in the final phase. A system architect may lose the cognitive ability for pinpointing and tracing the details of system behaviors. The programmers may lose their vision 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 key reasons of the failures in complex software development projects (Brooks, Jr., 1975; McDermid, 1991; Parnas, 2001; Wang, 2007a), which indicates that the programming approach to software engineering is insufficiently practical, and program languages are not good at dealing with complexity and consistency in software engineering.

It was recognized that software is not constrained by any known physical laws and principles (Hoare et al., 1987; Hartmanis, 1994; Wang, 2007a, 2008f, 2009f). The fact that we are still facing the same fundamental problems in software engineering as those we have recognized 50 years ago indicates an indispensable need for software science (Hoare, 1969; McDermid, 1991; Parnas, 2001; Wang, 2007a). Although software engineering has accumulated a rich set of empirical and heuristic principles, not all of them have been refined and formalized in order to form coherent theories for software science.

DEFINITION 3. Software science is a discipline that studies the formal properties and mathematical models of software, general methodologies for rigorous and efficient software development, and coherent theories and laws underpinning software behaviors and software engineering practices.

Based on the view of software science, the empirical discipline of software engineering may be clarified as that of the relationship between theoretical and applied physics. In software science, software is no longer seen as individual and diverse applications for solving particular problems; software engineering is not merely a labor-intensive code building activity; and software and computer scientists are no longer a practitioner for solving particular problems required by individual customers. Instead, the problems will be treated as a whole, at most a few categories of them, based on a set of general mathematical models and formal theories of software science.

This paper presents a general mathematical model of software (GMMS) towards software science on the basis of computer science, system science, information science, and denotational mathematics. In the remainder of this paper, Section 2 explores the theoretical discourse of software where any software system is formally represented by an abstract software model with sets of abstract structures, functions, events, and their interactions in the discourse of software. Section 3 formally represents the structure models of software by typed tuples. Section 4 rigorously denotes the behaviors models of software as embedded processes at the meta, complex, and system levels based on behavioral process algebra. Section 5 synthesizes the top level model of software system as a general mathematical model of interactive process dispatch systems where any concrete software system is formally treated as a derived instance of the GMMS model.

2. THE GENERAL MATHEMATICAL MODEL OF SOFTWARE (GMMS)

Although there are various perceptions on programs, algorithms, and software systems since the very beginning of computer science and software engineering, the concept of software is used to be treated as a listing of statements as for granted without deep studies towards the discipline of software science. This section explores the nature and insights of software as two-dimensional abstract entities interacting between its functions and structures. It leads to the formal description of the discourse of software and its general mathematical model.

2.1. The Discourse of Software

Upon the contemporary understanding about software and its properties in computer science, software engineering, information science, system science, cognitive computing, and computational intelligence, the discourse of software in software science can be rigorously introduced.

DEFINITION 4. Let $\mathfrak{V}$ be a finite set of *variables*, $\mathfrak{T}$ a finite set of *types*, $\mathfrak{I}$ a finite set of *instructions*, $\mathfrak{S}$ a finite set of *structures*, $\mathfrak{F}$ a finite set of *functions*, and $\widehat{\S}$ a finite set of *system dispatch functions*. Then, the *discourse of software*, $\mathfrak{U}$, is a 6-tuple, i.e.,:

$$\mathfrak{U} \triangleq (\mathfrak{V}, \mathfrak{T}, \mathfrak{I}, \mathfrak{S}, \mathfrak{F}, \widehat{\S}) \tag{1}$$

where the structure, function, and system of software are generated, respectively, based on the first three primitive attributes of $\mathfrak{U}$ as follows:

$$\begin{aligned} \mathfrak{S} &= \mathfrak{V} \times \mathfrak{T} \\ \mathfrak{F} &= \mathfrak{I} \times \mathfrak{S} \\ \widehat{\S} &= \mathfrak{V} \times \mathfrak{I} \times \mathfrak{S} \end{aligned} \tag{2}$$

2.2. The Abstract Software Model

On the basis of the universal discourse of software $\mathfrak{U}$, the general mathematical model of software can be derived.

DEFINITION 5. The *general mathematical model of software* (GMMS), $\wp$, is a 9-tuple in the discourse of software $\mathfrak{U}$, i.e.,:

$$\wp \triangleq (S, P, A, F, E, \S F, \Theta, R^i, R^o) \tag{3}$$

where

- S is a finite set of object *structures* of software $\wp$, $S = V \times T$, $V \subset \text{Þ}\mathfrak{V} \sqsubset \wp \sqsubset \mathfrak{U}$, $T \subset \text{Þ}\mathfrak{T} \sqsubset \mathfrak{U}$, $S \subset \text{Þ}\mathfrak{S} \sqsubset \wp \sqsubset \mathfrak{U}$ where V and T are sets of *variables* and *types*, respectively, Þ denotes a power set, and $\sqsubset$ is called a *system enclosure* where a set of components belongs to the hyper-structured system.
- P is a finite set of behavioral *processes* of software $\wp$, $P = I \times S$, $I \subset \text{Þ}\mathfrak{I} \sqsubset \mathfrak{U}$, $S \subset \text{Þ}\mathfrak{S} \sqsubset \wp \sqsubset \mathfrak{U}$, $P \subset \text{Þ}\mathfrak{F} \sqsubset \wp \sqsubset \mathfrak{U}$ where I is a set of *instructions*.
- A is a finite set of *architectures* of software $\wp$, $A = S \times S$, $A \subset \text{Þ}\mathfrak{S} \sqsubset \wp \sqsubset \mathfrak{U}$.
- F is a finite set of *functions* of software $\wp$, $F = P \times P$, $F \subset \text{Þ}\mathfrak{F} \sqsubset \wp \sqsubset \mathfrak{U}$.
- E is a finite set of *events* of software $\wp$, $E = V' \times T'$, $V' \subset \Theta \subset \text{Þ}\mathfrak{V} \sqsubset \wp \sqsubset \mathfrak{U}$, $T' \subset T \subset \text{Þ}\mathfrak{T} \sqsubset \wp \sqsubset \mathfrak{U}$, $E \subset \text{Þ}\mathfrak{S} \sqsubset \wp \sqsubset \mathfrak{U}$ where V' represents a set of event variables, T' a set of special types, $T' = \{Tr, TM, \odot\}$, known as those of *external trigger, system timing,* and *device interrupt* events.
- $\S F$ is a finite set of *system functions*, $\S F = E \times P \times S$, $\S F \subset \text{Þ}\widehat{\S} \sqsubset \mathfrak{U}$.
- Θ is the *environment* of software $\wp$, $\Theta = \{E', S'\}$, $E' \subset \text{Þ}\mathfrak{S} \sqsubset \mathfrak{U} \wedge E' \not\sqsubset \wp$, $S' \subset \text{Þ}\mathfrak{S} \sqsubset \mathfrak{U} \wedge S' \not\sqsubset \wp$, representing the sets of *external events* E' and *external structures* S' of another software $\wp'$, $\wp' \sqsubset \mathfrak{U} \wedge \wp' \neq \wp$.
- R^i is a finite set of input relations, $R^i = \Theta \times \wp$, $\Theta \not\sqsubset \wp$.
- R^o is a finite set of output relations, $R^o = \wp \times \Theta$, $\Theta \not\sqsubset \wp$.

The GMMS model in the discourse of universal software systems $\mathfrak{U}$ provides a theoretical framework of software science and theoretical software engineering. In the structural facet of software, it formally describes the object structures (S, Definition 4), events (E, Definitions 27/28), environments (Θ, Definition 25), and system architectures (A, Definition 23) by the unified structure models (*SMs*). In the functional facet of software, it rigorously expresses behavioral processes (P, Definition 16), functions (F, Definition 32), system functions ($\S F$, Definition 24), input relations (R^i, Definition 26), and output relations (R^o, Definition 26) by the unified process models (*PMs*) and its interactions with the *SMs* of a software system. Any program and/or software system can be formally modeled according to Definition 5 in $\mathfrak{U}$. Each facet of the formal model of software in GMMS will be further refined and elaborated in the remaining sections throughout the paper.

3. MATHEMATICAL MODELS OF SOFTWARE OBJECT STRUCTURES—TYPED TUPLES

The GMMS model in $\mathfrak{U}$ as created in the preceding section indicates that the structural model of software components is a typed tuple and the architectural model of software system at the top level is a composition of the structure models according to certain algebraic rules.

DEFINITION 6. The *object structure of software* $\wp$, S, in $\mathfrak{U}$ is an abstract model of software objects V such as a variable, constant, event, status, interface (port), memory, and complex objects identified by a type in T, i.e.,:

$$\begin{aligned} S \triangleq V \times T, \quad & V \subset \text{Þ}\mathfrak{V} \sqsubset \wp \sqsubset \mathfrak{U}, \\ & T \subset \text{Þ}\mathfrak{T} \sqsubset \mathfrak{U}, \\ & S \subset \text{Þ}\mathfrak{S} \sqsubset \wp \sqsubset \mathfrak{U} \end{aligned} \tag{4}$$

3.1. The Type Theory for Formally Modeling Object Structures of Software

Software objects as the operands of software operators are modeled as a finite set of variables and associated types. A *type* is a set in which all member data objects share a common logical property, domain constraints, and allowable operations. A *type system* specifies the modeling and manipulation rules of software objects (Martin-Lof, 1975; Guttag, 1977; Cardelli and Wegner, 1985; Mitchell, 1990; Wang, 2007a; Wang et al., 2010e). Types can be classified as *primitive* and *complex* types. The former are the most elemental types that cannot be further divided into simpler ones; and the latter are hybrid and derived types of multiple primitive types based on certain type rules.

DEFINITION 7. The *primitive types of software objects*, $\mathfrak{T}_p$, encompass a set of eight fundamental types that cannot be broken down further without losing their logical and semantic property, i.e.,:

$$\mathfrak{T}_p \triangleq \{\text{N, Z, R, S, BL, B, H, P}\} \tag{5}$$

where each of them represents the types of *natural number, integer, real number, string, Boolean, byte, hexadecimal,* and*pointer,* respectively.

The primitive types of software object structures are summarized in Table I where the notation, syntax, mathematical domain, language domain, and properties are specified.

DEFINITION 8. The *complex types of software structures*, $\mathfrak{T}_c$, encompass a set of ten composed types derived based on the primitive types, i.e.,:

$$\begin{aligned} \mathfrak{T}_c \triangleq \{&\mathbb{T}, \text{TI, D, TM, @e|S, @t|TM,} \\ &\text{@int}|\odot, \text{Ⓢ}s|\text{S, SM, PM}\} \end{aligned} \tag{6}$$

where each of them represents the type of *arbitrary, time, date, general time (data-time), trigger event, timing*

RESEARCH ARTICLE

Table I. Primitive types and domains of software objects.

No.	Type	Syntax	D_m	D_I	Equivalency
1	Natural number	N	[0, +∞]	[0, 65535]	Default arithmetic operations
2	Integer	Z	[-∞, +∞]	[-32768, +32767]	
3	Real number	R	[-∞, +∞]	[-2147483648, 2147483647]	
4	String	S	[0, +∞]	[0, 255]	Default character and string operations
5	Boolean	BL	[T, F]	[T, F]	Boolean constants {T|BL, F|BL}
6	Byte	B	[0, 256]	[0, 256]	Default binary operations
7	Hexadecimal	H	[0, +∞]	[0, max]	
8	Pointer	P	[0, +∞]	[0, max]	

event, interrupt event, operation status, system structure model (*SM*), and *behavioral process model* (*PM*), respectively.

The complex types of software objects as composed primitive types are summarized in Table II where the notation, syntax, mathematical domain, language domain, and properties are specified. Among the complex types in Table II, a pair of type prefixes @ and Ⓢ are introduced to denote an event and status, respectively, as special types of system variables. In addition, type suffixes are adopted for denotational convenience which are denoted by a primitive types prefixed by '|'. For example, the *interrupt suffix* is denoted by "|⊙".

DEFINITION 9. The *type suffix convention* is a denotational scheme to explicitly identify any software objects $o(x)$ such as variables, constants, events, status, structures, processes, and behaviors by an associated type $\mathbb{T}$ in both declaration and invocation, i.e.,:

$$\begin{cases} \text{Declaration: } o(x) \hat{=} x : \mathbb{T},\ \mathbb{T} \in \mathfrak{T}_p \cup \mathfrak{T}_c \\ \text{Invocation: } \ o(x) \hat{=} x|\mathbb{T},\ \mathbb{T} \in \mathfrak{T}_p \cup \mathfrak{T}_c \end{cases} \tag{7}$$

A formal type system is a collection of all type rules in $\mathfrak{U}$. A type rule is a mathematical relation and constraint on a given type. Type rules are defined on the basis of a type environment.

Table II. Complex types and domains of software objects.

No.	Type	Syntax	D_m	D_I	Equivalence
1	Arbitrary type	$\mathbb{T}$	–	–	Any dummy type determined at run-time
2	Time	TI = hh:mm:ss:ms	hh: [0, 23] mm: [0, 59] ss: [0, 59] ms: [0, 999]	hh: [0, 23] mm: [0, 59] ss: [0, 59] ms: [0, 999]	Default time manipulations
3	Date	D = yy:MM:dd	yy: [0, 99] MM: [1, 12] dd: [1, 31}	yy: [0, 99] MM: [1, 12] dd: [1, 31}	
4	Date/Time	TM = yyyy:MM:dd: hh:mm:ss:ms	yyyy: [0, 9999] MM:[1, 12] dd: [1, 31] hh: [0, 23] mm: [0, 59] ss: [0, 59] ms: [0, 999]	yyyy: [0, 9999] MM: [1, 12] dd: [1, 31] hh: [0, 23] mm: [0, 59] ss: [0, 59] ms: [0, 999]	
5	Trigger event	@ *e*|S	[0, +∞]	[0, 255]	Default system event capture and dispatch
6	Timing event	@ *t*|TM	[0|ms, 9999 |yyyy]	[0|ms, 9999 |yyyy]	
7	Interrupt event	@ *int*|⊙	[0, 1023]	[0, 1023]	
8	Operational status	Ⓢ *s*|BL	[T, F]	[T, F]	Default Boolean operations
9	System structure	SM	–	–	Default field reference: x|SM.y|$\mathbb{T}$
10	System process model	PM	–	–	Default process schema (Def. 23): P|SM (<I|SM>, <O|SM>, <M|SM>)

DEFINITION 10. The *type environment* Θ_t of software in $\mathfrak{U}$ is a collection of all formal primitive and complex types as specified in Tables I and II, i.e.,:

$$\begin{aligned}\Theta_t &\triangleq \mathfrak{T}_p \cup \mathfrak{T}_c \\ &= \{\text{N, Z, R, S, BL, B, H, P}\} \\ &\cup \{\mathbb{T}, \text{TI, D, TM, @e|S, @t|TM,} \\ &\quad \text{@int}|\odot, \text{Ⓢ}s|\text{S, SM, PM}\}\end{aligned} \tag{8}$$

The central principle of type theory is that the domains of types can be classified into those of *mathematical* D_m, *language defined* D_l, and *problem constrained* D_p domains as shown in Tables I and II.

THEOREM 1. *The domain constraint of software objects states that the following relationship between the domains of abstract types and concrete software objects is always held, i.e.,*:

$$D_p \subseteq D_l \subseteq D_m \tag{9}$$

PROOF. The relationships between the three domains of types are expressed in Eq. (10). For any given problem, because of the *constraints* on both the language domain k_l and the mathematical domain k_m cannot greater than one, Theorem 1 holds, i.e.,:

$$\begin{aligned}D_p &= k_l D_l = k_l k_m D_m, \quad k_l, k_m \le 1 \\ &\Rightarrow D_p \subseteq D_l \subseteq D_m \quad \square\end{aligned} \tag{10}$$

Empirically, for all primitive types as defined in Table I, $D_p \subset D_l \subset D_m$; while for all complex types as defined in Table II, $D_p = D_l = D_m$. Therefore, both categories of types obey the general domain constraints according to Theorem 1.

It is noteworthy that, although a generic specification of a software object is constrained by D_m, the executable program that embodies the specification is constrained by D_l. Then it is further restricted by the problem domain D_p, which is always a subset of D_l and D_m in order to reserve code and mathematical consistencies. Therefore, any structural object of software is constrained by the problem domain specified in given problem requirements.

COROLLARY 1. *The precedence of domain determination and type refinement in software object modeling is constrained by the following order*:

$$D_p \Rightarrow D_l \Rightarrow D_m \tag{11}$$

Although the mathematical domain D_m is the basis of system design in the phases of conceptual modeling, requirements analysis, and specification, the language domain D_l constrains system implementation on a certain platform and programming language. The problem domain D_p mainly constrains different instances of the system at run-time.

3.2. The Big-*R* Notation for Recurring Structures and Iterative Behaviors of Software

Iterative and recursive structures and behaviors are the most fundamental and essential mechanisms of software and computing, which make programs more efficient and expressive. However, the iterative and recursive constructs of software are the most diverse and confusable instructions in programming at both syntactic and semantic levels. The big-*R* notation (Wang, 2008c) is introduced to provide a unified and expressive treatment of iterations and recursions for rigorously modeling both recurring structures and iterative behaviors of software.

DEFINITION 11. The *big-R notation* is a mathematical operator for denoting:
(a) a finite set of *repetitive* object structures of software; and/or
(b) a finite set of *iterative* or *recursive* behavioral processes, i.e.,:

$$\text{(a)} \quad \overset{n}{\underset{i=1}{R}} S(i)|\text{SM} \tag{12a}$$

$$\text{(b)} \quad \overset{n}{\underset{i=1}{R}} P(i)|\text{PM} \tag{12b}$$

The big-*R* notation can be used to denote not only recurring constructs of software architectures and data objects, but also repetitive operational behaviors of software. According to Definition 11, structures of arbitrary software entities can be formally denoted by the big-*R* notation in a unique and efficient syntax.

EXAMPLE 1. The default system structure of memory, MEM|SM, and device interface port, PORT|SM, can be specified by the big-*R* notation, respectively, as follows:

$$\begin{aligned}\text{MEM}|\text{SM} &\triangleq \left[\overset{n}{\underset{i=0}{R}}(addr_i|\text{H}, data_i|\mathbb{T})\right]\Big|\text{SM} \\ \text{PORT}|\text{SM} &\triangleq \left[\overset{m}{\underset{i=0}{R}}(ptr_i|\text{P}, data_i|\mathbb{T}, IO_i|\text{BL}\right. \\ &\qquad = \{(\text{T}|\text{BL, intput}), \\ &\qquad\quad (\text{F}|\text{BL, output})\})\Big]\Big|\text{SM}\end{aligned} \tag{13}$$

where it is a convention that the internal memory space is denoted by a address $addr_i|\text{H}$, while the external port space is denoted by a pointer $ptr_i|\text{P}$, though they are mathematically equivalent; and the expression $IO_i|\text{BL} = \{(\text{T}|\text{BL, output}), (\text{F}|\text{BL, input})\}$ denotes a refinement of the port I/O specification.

Similarly, the behavior of an arbitrary software process can be formally denoted by the big-*R* notation in a unique syntax.

EXAMPLE 2. The default functions of memory read/write, MEM|PM, and port input/output, PORT|PM, can be uniquely denoted by the big-*R* notation,

RESEARCH ARTICLE

respectively, as follows by operating on the corresponding structure models as specified in Example 1:

$$\text{MEM}|\text{PM} \triangleq \begin{cases} \overset{n}{\underset{i=0}{R}} \langle data_i|\mathbb{T} := \text{MEM}|\text{SM}(addr_i|\text{H})\rangle & // \text{ Read} \\ \overset{n}{\underset{i=0}{R}} \langle \text{MEM}|\text{SM}(addr_i|\text{H}) := data_i|\mathbb{T}\rangle & // \text{ Write} \end{cases}$$

$$\text{PORT}|\text{PM} \triangleq \begin{cases} \overset{m}{\underset{i=0}{R}} \langle data_i|\mathbb{T} := \text{PORT}|\text{SM}(ptr_i|\text{P}, IO_i|\text{BL} = \text{T}|\text{BL})\rangle & // \text{ Input} \\ \overset{m}{\underset{i=0}{R}} \langle \text{PORT}|\text{SM}(ptr_i|\text{P}, IO_i|\text{BL} = \text{F}|\text{BL}) := data_i|\mathbb{T}\rangle & // \text{ Output} \end{cases} \tag{14}$$

3.3. Formal Structure Model of Software Objects by Typed Tuples

It is recognized that the structure model of software objects is a multidimensional composition of homogeneous or heterogeneous variables in primitive and/or complex types such as a set of structured variables, internal logical models, hardware interfaces, and control structures of software. The structure models provide a formal means for rigorously describing complex types and constructs in every phase of system lifecycles such as those of requirements representation, specification, modeling, implementation, and validation.

It is noteworthy that the structure model as a special system type is nontrivial software objects in software science and computing, which play an important role in system design and modeling as the effecter and holder of system operation results, as well as intermediate and contingent values of variables, within and beyond the lifecycle of the behavioral processes that invoke them.

Structure models of software can be rigorously described by the mathematical models of software objects and typed tuples at two levels from the bottom up.

Definition 12. A *software object*, O, is an abstract model of a *variable* v, $v \in V \subset \wp\mathfrak{V} \sqsubset \mathfrak{U}$, specified in a certain *type* $\mathbb{T}$ modified by a type constraint $\mu_v(\mathbb{T})$ that yields a tailored subdomain $\mathbb{T}'$ according to given problem-specific restriction $\mathbb{T}''$, i.e.,:

$$O \triangleq \langle v\colon \mathbb{T}|\mu_v(\mathbb{T}) = \mathbb{T}\backslash\mathbb{T}'' = \mathbb{T}'\rangle \tag{15}$$

where $\mu_v(\mathbb{T}) = \mathbb{T}\backslash\mathbb{T}'' = \mathbb{T}'$ tailors $\mathbb{T}$ in D_m into the subset $\mathbb{T}'$ in D_p, i.e., $v|\mathbb{T} \Rightarrow v|\mu_v(\mathbb{T}) = v|\mathbb{T}'$.

Definition 13. A *typed n-tuple*, τ^n, is a structured encapsulation of n software objects in homogeneous or heterogeneous types $\mathbb{T}_i$, $1 \le i \le n$, with certain constraints $\mu_{v_i|\mathbb{T}_i}(\mathbb{T}_i)$, i.e.,:

$$\begin{aligned} \tau^n &\triangleq \left(\overset{n}{\underset{i=1}{R}} O_i|\text{SM}\right) \\ &= \left(\overset{n}{\underset{i=1}{R}} \langle v_i\colon \mathbb{T}_i|\mu_{v_i|\mathbb{T}_i}(\mathbb{T}_i) = \mathbb{T}_i\backslash\mathbb{T}_i''\rangle\right) \end{aligned} \tag{16}$$

On the basis of the formal model of typed tuples, the structure model of software can be formally expressed as follows.

Theorem 2. *The structure model of software, SM, in* $\mathfrak{U}$ *states that any software object, as well as its internal control structures and interfaces with external hardware devices, can be uniformly represented and refined by the general mathematical model of typed n-tuple* τ^n*, i.e.,*:

$$\begin{aligned} SM \triangleq \tau^n &= \left(\overset{n}{\underset{i=1}{R}} \langle v_i\colon \mathbb{T}_i|\mu_{v_i|\mathbb{T}_i}(\mathbb{T}_i) = \mathbb{T}_i\backslash\mathbb{T}_i''\rangle\right) \\ &= (\langle v_1\colon \mathbb{T}_1|\mu_{v_1|\mathbb{T}_1}(\mathbb{T}_1) = \mathbb{T}_1\backslash\mathbb{T}_1''\rangle, \\ &\quad \langle v_2\colon \mathbb{T}_2|\mu_{v_2|\mathbb{T}_2}(\mathbb{T}_2) = \mathbb{T}_2\backslash\mathbb{T}_2''\rangle, \\ &\quad \ldots \\ &\quad \langle v_n\colon \mathbb{T}_n|\mu_{v_n|\mathbb{T}_n}(\mathbb{T}_n) = \mathbb{T}_n\backslash\mathbb{T}_n''\rangle \\ &\quad) \end{aligned} \tag{17}$$

where $v_i|\mathbb{T}_i$, $1 \le i \le n$, *is one of the software objects as an element of the SM in type* $\mathbb{T}_i$ *in general and in the problem-constrained domain* $\mu_{v_i|\mathbb{T}_i}(\mathbb{T}_i) = \mathbb{T}_i\backslash\mathbb{T}_i''$ *in particular.*

Proof. According to Definitions 6, 10, 12, and 13, the following relationship holds in $\mathfrak{U}$:

$$SM = \tau^n \supseteq (\mathfrak{T}_p \cup \mathfrak{T}_c) \supseteq \Theta_t \supseteq \mathbb{T}(O = v|\mathbb{T}) \tag{18}$$

Therefore, any primitive or complex software objects can be uniquely modeled by an SM or typed tuple in the form of Eq. (17). The necessary and sufficient conditions of SM are as follows:

(a) The *necessary* condition: Because any primitive variable $o|\mathbb{T}$ or any attribute of a complex software object $O|SM = \overset{n}{\underset{i=1}{R}} o_i|\mathbb{T}_i$ in $\mathfrak{U}$ needs to be abstractedly specified by a type $\mathbb{T} \subset \Theta_t = \mathfrak{T}_p \cup \mathfrak{T}_c$ and an associate constraint $\mu_{v|\mathbb{T}}(\mathbb{T})$, the necessity of the typed tuple model is fulfilled according to Definitions 6, 10, and 12;

(b) The *sufficient condition*: Because any complex software object $O|SM = \overset{n}{\underset{i=1}{R}} o_i|\mathbb{T}_i$ is a finite structure of limited number of elements in $\mathfrak{U}$, the sufficiency of the typed tuple model is fulfilled for denoting any simple or complex objects according to Definition 13. □

Theorem 2 reveals that the general structural model of software in $\mathfrak{U}$, such as a variable or constant in a primitive type and an internal control structure or interface of an external device as a complex model of software objects can be formally modeled by an SM.

Example 3. The port of a device interface for a keypad in a software system, $\text{PORT}_{\text{keypad}}|\text{SM}$, can be

formally specified as an SM according to Theorem 2 as follows:

$$\begin{aligned}&\text{PORT}_{\text{keypad}}|\text{SM}\\ &\quad\triangleq(\langle Ptr\colon \text{P}|Ptr|\text{P}=\text{2F01}|\text{H}\rangle,\\ &\qquad\langle DataInput\colon \text{B}|DataInput|\text{B}=0000\ 1111|\text{B}\rangle,\\ &\qquad\langle IO\_Control\colon \text{BL}|IO\_Control|\text{BL}=F|\text{BL}\rangle\\ &\quad)\end{aligned}\tag{19}$$

Example 4. A segment of the system memory, $\text{MEM}_1|\text{SM}$, in [FA00|H, FA0F|H] can be formally specified according to Theorem 2 as follows:

$$\begin{aligned}&\text{MEM}_1|\text{SM}\\ &\quad\triangleq(\langle(Addr_0\colon \text{H}|Addr_0|\text{H}=\text{FA00}|\text{H},\\ &\qquad Data_0\colon \text{B}|Data_0|\text{B}=1000\ 1100|\text{B})\rangle,\\ &\qquad\langle(Addr_1\colon \text{H}|Addr_1|\text{H}=\text{FA01}|\text{H},\\ &\qquad Data_1\colon \text{B}|Data_1|\text{B}=1000\ 1001|\text{B})\rangle,\\ &\qquad\ldots\\ &\qquad\langle(Addr_{15}\colon \text{H}|Addr_{15}|\text{H}=\text{FA0F}|\text{H},\\ &\qquad Data_{15}\colon \text{B}|\text{Data}_{15}|\text{B}=0000\ 0100|\text{B})\rangle\\ &\quad)\end{aligned}\tag{20}$$

Example 5. A set of structure models for the components of a digital *clock*, can be formally denoted as follows:

$$\begin{aligned}&\text{ComponentStructure}(Clock)|\text{SM}\\ &\quad\triangleq(\langle\text{Processor}|\text{SM}\rangle,\\ &\qquad\langle\text{Keypad}|\text{SM}\rangle,\\ &\qquad\langle\text{LED}|\text{SM}\rangle,\\ &\qquad\langle\text{Pulse}|\text{SM}\rangle,\\ &\qquad\langle\text{Bell}|\text{SM}\rangle,\\ &\qquad\langle\text{SCB}|\text{SM}\rangle\quad//\ \text{The system control block}\\ &\quad)\end{aligned}\tag{21}$$

where details of each SM in Eq. (21) can be refined according to Theorem 2 similarly as in those of Examples 3 and 4.

Example 5 indicates that the components of a software system can be formally described by SM of SMs in a hierarchical structure.

Corollary 2. *Any complex or primitive object structure of software, S, in* $\mathfrak{U}$ *can be formally modeled* (*declared*) *by an SM and invoked by its field access, i.e.,*:

$$\begin{cases}\text{Declaration: } S|SM\triangleq\left(\overset{n}{R}\langle S_i\colon \mathbb{T}_i|\mu_{S_i|\mathbb{T}_i}(\mathbb{T}_i)\rangle\right)\\ \text{Invocation:}\quad S|SM.S_i|\mathbb{T}_i\end{cases}\tag{22}$$

where the invocation of a declared SM will be elaborated in the process models in Section 4.

3.4. The Formal Structure Model of Software Architectures at the System Level by Relational Typed-Tuples

The architectural model of software at the system level is a composition of all SMs of its components according to a set of algebraic relational rules, which can be formally modeled by relational typed-tuples.

Definition 14. The set of *architectures* of a software $\wp$, *A*, in $\mathfrak{U}$ is a Cartesian product of sets of structures *S*, i.e.,:

$$\begin{aligned}A=S\times S,\quad S\subset \text{Þ}\Im\sqsubset\wp\sqsubset\mathfrak{U},\\ A\subset \text{Þ}\Im\sqsubset\wp\sqsubset\mathfrak{U}\end{aligned}\tag{23}$$

Definition 15. The set of *architectural relations*, $\mathbb{R}_a$, $\mathbb{R}_a\subseteq \text{Þ}\Im\sqsubset\mathfrak{U}$, between the structure models *SM*s of software components in a given system in $\mathfrak{U}$ encompasses nine basic structural composition operators elicited from the most fundamental software structures, i.e.,:

$$\mathbb{R}_a\triangleq\{\|,\rightarrow,\rightarrowtail,|\gg,|\ll,\oiint,\gg,\lightning,\hookrightarrow_e\}\tag{24}$$

where the architectural operators represent the relations of *parallel, sequential, embedded, input, output, I/O, concurrent, pipeline, interrupt*, and *dispatch* structures, respectively.

The syntaxes of the architectural relations $\mathbb{R}_a$ is a subset of the behavioral relations $\mathbb{R}$ as given in Table IV. Further details and their formal semantics may refer to Wang (2008b).

Corollary 3. *The general pattern of software system architectures,* $\mathbb{A}$, $\mathbb{A}\subset \text{Þ}\Im\sqsubset\mathfrak{U}$, *is an algebraically composed structure in* $\mathfrak{U}$ *by finite architectural relations between all components' SMs, i.e.,*:

$$\mathbb{A}\triangleq\overset{n}{R}(SM_i\ r_{ij}\ SM_j),\quad r_{ij}\in\mathbb{R}_a,\ j=i+1\tag{25}$$

Example 6. An informally described architecture of an abstract software system, *S*§, is shown in Figure 1. It can be formally modeled according to Corollary 3 as follows:
(a) Top level architecture of the system:

$$S\S\triangleq S_1\|S_2\|\cdots\|S_n$$

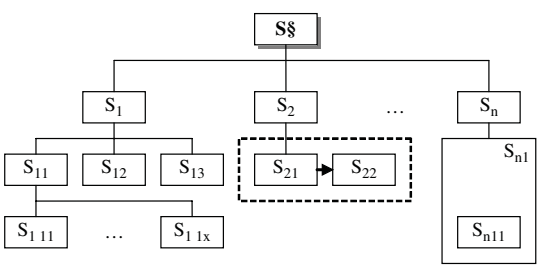

Fig. 1. The architecture of an abstract software system.

(b) Level 2 refinement:

$$\begin{cases} S_1 \triangleq S_{11} \parallel S_{12} \parallel S_{13} \\ \quad = (S_{111} \parallel \cdots \parallel S_{11n}) \parallel S_{12} \parallel S_{13} \\ S_2 \triangleq S_{21} \to S_{22} \\ \cdots \\ S_n \triangleq S_{n1} \rightarrowtail S_{n11} \end{cases} \tag{26}$$

(c) Level 3 refinement:

$$S_{11} \triangleq S_{111} \parallel \cdots \parallel S_{11n}$$

EXAMPLE 7. The composition of the architecture of a concrete digital clock system, Clock§.*Architecture*|SM, is formally described according to Corollary 3 based on the conceptual model as given in Figure 2 as follows:

$$\begin{aligned} &\textit{Clock}\S.\textit{Architecture}|\text{SM} \\ &\triangleq \{ \quad (\textit{Keypad}|\text{SM} \parallel \textit{Pulse}|\text{SM}) \\ &\quad | \gg (\textit{Procesor}|\text{SM} \parallel \textit{SCB}|\text{SM}) \\ &\quad | \ll (\textit{LED}|\text{SM} \parallel \textit{Bell}|\text{SM}) \\ &\quad \} \end{aligned} \tag{27}$$

where *Processor*|SM is embodied as the *Clock*§ system parallel with the *system control block*, *SCB*|SM, represented by system control structures, events, status, and global variables.

COROLLARY 4. *The general topological architecture of software systems, $S_\wp$, is an embedded hierarchical structure in* $\mathfrak{U}$ *where each kth layer of it, $S_\wp^k$, in the system hierarchy can be represented or refined by its next layer, $S_\wp^{k-1}$, i.e.,*:

$$\begin{aligned} S_\wp &\triangleq \overset{n}{R}_{k=1} S_\wp^k(S_\wp^{k-1}), \quad S_\wp^0 = O|SM = V|\mathbb{T} \\ &= S_\wp^n(S_\wp^{n-1}(\ldots(S_\wp^1(S_\wp^0)))) \end{aligned} \tag{28}$$

where $S_\wp^0$ is a known primitive software object represented by a typed variable.

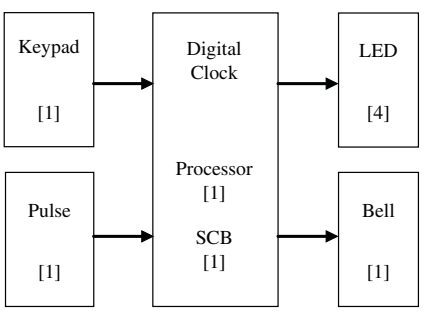

Fig. 2. The conceptual model of a digital clock.

4. MATHEMATICAL MODELS OF SOFTWARE FUNCTIONAL BEHAVIORS—EMBEDDED PROCESSES

Upon the contemporary understanding about software and its properties in computer science, software engineering, information science, system science, and computational intelligence, any meta and complex software behavior can be formally described by a functional process, which is recursively built by a unique mathematical structure known as the *chain of embedded functions of functions* (Wang, 2002, 2008a). Assume each statement, process, and process relation be treated as a function, then the entire behaviors of an abstract software system is a higher order function of functions that operates on low-level functions and structure models during program execution.

On the basis of the structure models of software presented in preceding section, the behavior models of software can be formally analyzed and modeled as a set of meta-processes at the instruction level, and then a set of relational operations of complex processes at the function level. The process model of meta and complex processes leads to the embedded process theory of software functions and behaviors as a Cartesian product between the sets of process models and structure models.

4.1. Meta-Processes of Software Behaviors

The essence of software behaviors is centric by the meta-process models at the lower level and the embedded composed processes at the higher level (Wang, 2008a). Behaviors of programs and software systems are observable computing processes and operational consequences on structural models of software objects in $\mathfrak{U}$. Typical software behaviors modeled by various instruction sets of computers and programming languages can be classified into eight categories known as those of data manipulations, arithmetical operations, logical operations, bitwise operations, program controls, memory manipulations, I/O manipulations, and time/interrupt manipulations. The eight categories of fundamental software behaviors defined on abstract data objects can be grouped into internal and external (interactive) behaviors (Wang, 2007a).

DEFINITION 16. The *meta-processes of software* $\wp$, P, in $\mathfrak{U}$ are abstract models of a set of fundamental computing instructions $\mathbb{I}$ operating on a set of software structures S, i.e.,:

$$\begin{aligned} P \triangleq \mathbb{I} \times S, \quad & \mathbb{I} \subset \text{Þ}\Im \sqsubset \mathfrak{U}, \\ & S \subset \text{Þ}\Im \sqsubset \wp \sqsubset \mathfrak{U}, \\ & P \subset \text{Þ}\Im \sqsubset \wp \sqsubset \mathfrak{U} \end{aligned} \tag{29}$$

According to Definition 16, the basic functional element of software is a statement or a meta-instruction commonly specified by programming languages. The statement as an instance of an instruction in a given programming language is the smallest functional unit of software, which

RESEARCH ARTICLE

specifies an explicit action and yields the change of one or more data objects logically modeled by simple variables or structure models.

DEFINITION 17. An *abstract instruction* or *statement*, i_s, is a meta-function p in $\mathfrak{U}$, that maps a set of input variables I_ν into a set of output variables O_ν, i.e.,:

$$i_s \triangleq p\colon I_\nu \to O_\nu, \quad I_\nu, O_\nu \subset S \subset \wp\mathfrak{G} \sqsubset \mathfrak{U}, \quad i_s, p \subset \wp P \subset \wp\Im \sqsubset \mathfrak{U} \tag{30}$$

A *meta-process* is a basic and common functional operator in $\mathfrak{U}$, which cannot be broken down to more detailed actions or behaviors. The most general and fundamental software behaviors can be mathematically modeled by a set of meta-processes. Then complex processes are derived by compositions of meta-processes according to a set of algebraic rules of process relations.

DEFINITION 18. The set of *meta-processes* of software, $\mathbb{P}$, $\mathbb{P} \subset \wp B \subset \wp\mathfrak{P} \sqsubset \mathfrak{U}$, encompasses 17 fundamental software operators elicited from the fundamental and essential computational needs, i.e.,:

$$\mathbb{P} \triangleq \{:=, \blacklozenge, \Rightarrow, \Leftarrow, \nLeftarrow, \gg, \ll, |\gg, |\ll, \overset{@}{=}, \triangleq, \uparrow, \downarrow, !, \otimes, \boxtimes, \S\} \tag{31}$$

where the meta-processes represent *assignment, evaluation, addressing, memory allocation, memory release, read. write, input, output, timing, duration, increase, decrease, exception detection, skip, stop*, and *system*, respectively.

Mathematical notations and syntaxes of the meta-processes of software are formally described in Table III, while their formal semantics may be referred to (Wang, 2008b). As shown in Definition 18 and Table III, each meta-process in $\mathbb{P}$ is a basic operation on one or more operands such as variables, memory elements, and I/O ports. Structures of operands (software objects) and their allowable operations are constrained by their types $\mathbb{T}$ as described in Section 3.1.

4.2. Algebraic Composition of Software Behaviors by Complex Processes

On the basis of the finite set of 17 meta-processes of software identified in Section 4.1, any complex behavior of software can be derived via relational compositions of two or multiple meta-processes according to certain algebraic composition rules.

DEFINITION 19. The set of *functions* of a software system $\wp$, F, in $\mathfrak{U}$ is a Cartesian product between sets of meta-processes P, i.e.,:

$$F = P \times P, \quad P \subset \wp\Im \sqsubset \wp \sqsubset \mathfrak{U}, \; F \subset \wp\widetilde{\Im} \sqsubset \wp \sqsubset \mathfrak{U} \tag{32}$$

According to Definition 19, as that the mathematical model of a basic software behavior is a meta-process, a complex behavior of software can be formally described by relational compositions of meta-processes.

Table III. Meta-processes of software ($\mathbb{P}$).

No.	Meta Process	Notation	Syntax
1	Assignment	:=	$y\|\mathbb{T} := x\|\mathbb{T}$
2	Evaluation	◆	◆ $exp\|\mathbb{T} \to$ Dom($\mathbb{T}$), $\mathbb{T} \in$ {BL, N, Z, B}
3	Addressing	⇒	$id\|\mathbb{T} \Rightarrow$ MEM [ptr|P] $\|\mathbb{T}$
4	Memory allocation	⇐	$id\|\mathbb{T} \Leftarrow$ MEM [ptr|P] $\|\mathbb{T}$
5	Memory release	⇍	$id\|\mathbb{T} \nLeftarrow$ MEM [⊥] $\|\mathbb{T}$
6	Read	≫	MEM [ptr|P] $\|\mathbb{T} \gg x\|\mathbb{T}$
7	Write	≪	$x\|\mathbb{T} \ll$ MEM [ptr|P] $\|\mathbb{T}$
8	Input	|≫	PORT[ptr|P]$\|\mathbb{T}$ |≫$x\|\mathbb{T}$
9	Output	|≪	$x\|\mathbb{T}$ |≪ PORT [ptr|P]$\|\mathbb{T}$
10	Timing	$\overset{@}{=}$	$@t\|$TM $\overset{@}{=}$ $\S t\|$TM
11	Duration	≙	$@t_n\|$TM ≙ $\S t_n\|$TM + Δn|TM
12	Increase	↑	↑($n\|\mathbb{T}$)
13	Decrease	↓	↓ ($n\|\mathbb{T}$)
14	Exception detection	!	! (@$e\|S$)
15	Skip	⊗	⊗
16	Stop	⊠	⊠
17	System	§	$SysID\|$§

DEFINITION 20. A *complex process*, P^*, in $\mathfrak{U}$ is an algebraic composition of two or more meta-processes P by a set of relational operators $\mathbb{R}$ in F, i.e.,:

$$P^* \triangleq \mathbb{R}(P, P), \quad P \subset \wp\Im \sqsubset \wp \sqsubset \mathfrak{U}, \quad \mathbb{R}, P^* \subset F \subset \wp\widetilde{\Im} \sqsubset \wp \sqsubset \mathfrak{U} \tag{33}$$

where the binary composition can be extended to multi-dimensional Cartesian product.

The process relations of software can be formally described by a set of relational operators on meta-processes as a set of process composition rules for building complex processes. A set of 17 process relational operators is elicited from software behavioral modeling and computing as follows.

DEFINITION 21. The set of *relational process operators* of software, $\mathbb{R}$, $\mathbb{R} \subset F \subset \wp\widetilde{\Im} \sqsubset \wp \sqsubset \mathfrak{U}$, encompasses 17 algebraic relational operators for building complex processes in $\mathfrak{U}$, i.e.,:

$$\mathbb{R} \triangleq \left\{\rightarrow, \curvearrowright, |, |..|.., R^*, R^+, R^i, \circlearrowleft, \rightarrowtail, \|, \oint\!\!\!\oint, |||, \gg, \lightning, \hookrightarrow_e, \hookrightarrow_t, \hookrightarrow_i\right\} \tag{34}$$

where the process operators denote *sequence, jump, branch, switch, while-loop, repeat-loop, for-loop,*

RESEARCH ARTICLE

recursion, function call, parallel, concurrence, interleave, pipeline, interrupt, event-driven dispatch, time-driven dispatch, and *interrupt-driven dispatch*, respectively.

As given in Definition 21 and Table IV, the 17 process relational operators represent a set of fundamental mechanisms of programming and system behavioral description, because any complex process can be combinatory built by the algebraic process operations on the set of the 17 meta-processes. In Table IV, the big-*R* notation used in process relations #5 through #8 is a special calculus as described in Section 3.2.

4.3. The Embedded Process Theory of Software Behaviors

A software behavior as a complex process is a chain of compositions of a list of meta-processes by predefined relational or composing rules $\mathbb{R}$ as given in Table IV. It is noteworthy that in a complex process the relations between meta-processes are a set of special embedded relational operators, in which the current process is not only related to the next process, but also related to all previous processes.

Table IV. Algebraic operations on meta-processes of software ($\mathbb{R}$).

No.	Process relational operator	Notation	Syntax
1	Sequence	$\rightarrow$	$P \rightarrow Q$
2	Jump	$\curvearrowright$	$P \curvearrowright Q$
3	Branch	$\mid$	$\blacklozenge exp \mid \mathrm{BL} = \mathrm{T} \mid \mathrm{BL} \rightarrow P$ $\mid \blacklozenge\sim \rightarrow Q$
4	Switch	$\mid$ … $\mid$	$\blacklozenge \mathrm{exp} \mid \mathbb{T} =$ $\mathrm{i} \rightarrow P_i$ $\mid \sim \rightarrow \oslash$ where $\mathbb{T} \in \{\mathrm{N}, \mathrm{Z}, \mathrm{B}, \mathrm{S}\}$
5	While-loop	R^*	$\overset{\mathrm{F}}{R}_{exp \mid \mathrm{BL}=\mathrm{T}} P \mid \mathrm{PM}$
6	Repeat-loop	R^+	$P \mid \mathrm{PM} \rightarrow \overset{\mathrm{F}}{R}_{exp \mid \mathrm{BL}=\mathrm{T}} P \mid \mathrm{PM}$
7	For-loop	R^i	$\overset{n}{R}_{i \mid \mathrm{N}=1} \mathrm{P}\,(i \mid \mathrm{N})$
8	Recursion	$\circlearrowleft$	$\overset{n \mid \mathrm{N}}{R}_{i \mid \mathrm{N}=1} (P^{i \mid \mathrm{N}} \mid \mathrm{PM} \circlearrowleft \mathrm{P}^{i \mid \mathrm{N}-1} \mid \mathrm{PM})$
9	Function call	$\rightarrowtail$	$P \rightarrowtail F$
10	Parallel	$\parallel$	$P \parallel Q$
11	Concurrence	$\Diamond\!\!\!\!\Diamond$	$\mathrm{P} \Diamond\!\!\!\!\Diamond Q$
12	Interleave	$\parallel\!\!\!\parallel$	$P \parallel\!\!\!\parallel Q$
13	Pipeline	$\gg$	$P \gg Q$
14	Interrupt	$\lightning$	$P \lightning Q$
15	Event-driven dispatch	$\hookrightarrow_e$	$@e_i \mid \mathrm{S} \hookrightarrow_e P_i$
16	Time-driven dispatch	$\hookrightarrow_t$	$@t_i \mid \mathrm{TM} \hookrightarrow_t P_i$
17	Interrupt-driven dispatch	$\hookrightarrow_i$	$@int_j \circledcirc \hookrightarrow_i P_i$

THEOREM 3. *The process model of software behaviors at the component level, PM, $PM \subset \text{Þ}F \subset \text{Þ}\widehat{\mathbb{S}} \sqsubset \mathbb{U}$, is a Cartesian product between a process p and a structure s, in* $\mathbb{U}$, *i.e.,*:

$$\begin{aligned} PM &\triangleq p \times s, \quad p \subset P^* \subset \text{Þ}F \subset \text{Þ}\widehat{\mathbb{S}} \sqsubset \mathbb{U}, \\ &\qquad s \subset S \subset \text{Þ}\mathfrak{S}, PM \subset \text{Þ}F \subset \text{Þ}\widehat{\mathbb{S}} \sqsubset \mathbb{U} \\ &= p \mid P^* \times s \mid SM \end{aligned} \tag{35}$$

PROOF. Theorem 3 can be directly proved according to Definitions 5 and 16. □

According to Theorem 3, many conventional technologies in programming and software engineering would increase complexity therefore be anti-productive. For instance, the widely used software modeling scheme known as object-oriented programming (OOP) is actually inefficient in both system modeling and implementation, because it hybridizes the independent facets of software structures and functions into the same class construct. Once a structure (data object) in a given class is invoked by multiple methods (processes) in other classes, or vice versa, when a method in a given class accesses multiple structures in other classes, the architecture of the software does immediately become an intricately unstructured system. This empirical practice has dramatically increased the complexity of software structures and behaviors as well as their inherence and interactions. Therefore, the OOP methods for software engineering are not in line with the insights of software science, which reveals that software is interactive behaviors between two totally different facets known as of the functions and structures of the software system.

DEFINITION 22. A *complex process* P^* in $\mathbb{U}$, $P^* \triangleq \mathbb{R}(p_i, p_j) \subset F \subset \text{Þ}\widehat{\mathbb{S}} \sqsubset \wp \sqsubset \mathbb{U}$, is an embedded relational composition of a series of *n meta-processes* p_i and p_j according to certain *relational operations* or composing rules γ_{ij}, $\gamma_{ij} \in \mathbb{R} \subset F \subset \text{Þ}\widehat{\mathbb{S}} \sqsubset \wp \sqsubset \mathbb{U}$, i.e.,:

$$\begin{aligned} P^* &= \overset{n-1}{\underset{i=0}{R}} (p_i\, \gamma_{ij}\, p_j), \quad j = i+1, \\ &\qquad p_i, p_j \subset \mathbb{P} \subset \text{Þ}\Im \sqsubset \mathbb{U}, \\ &\qquad \gamma_{ij} \in \mathbb{R} \subset F \subset \text{Þ}\widehat{\mathbb{S}} \sqsubset \wp \sqsubset \mathbb{U} \\ &= (\ldots(((p_0)\gamma_{01}p_1)\gamma_{12}p_2)\ldots\gamma_{n-1,n}\ p_n) \end{aligned} \tag{36}$$

where $1 \le i < n-1$, and $2 \le j = i+1 \le n$.

The structure of the abstract embedded process model, $P^* = \overset{n-1}{\underset{i=0}{R}}(p_i \gamma_{ij} p_j)$, can be illustrated in Figure 3, which indicates that any kth, $1 \le k < n-1$, function (statement) given in a process is a relational composition with all cumulated $k-1$ preceding functions, i.e., $P^* = (\ldots(((p_0)r_{01}p_1)r_{12}p_2)\ldots r_{k-1,k}p_k)$. In Definition 22, the sets of meta-processes $\mathbb{P}$ and relational operators $\mathbb{R}$ have been formally defined in Tables III and IV, respectively.

THEOREM 4. *The embedded relational model of processes (ERMP) states that the abstract process P^* is a general*

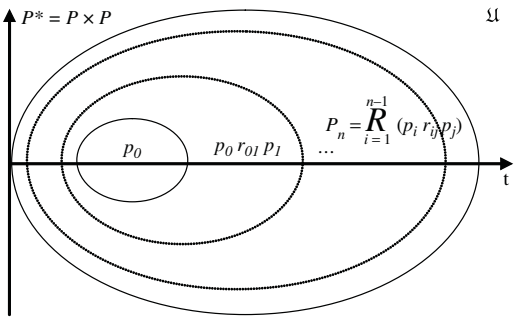

Fig. 3. The hyperstructure model of embedded processes of software.

model of software behaviors composed by a set of n meta-statements s_i, $1 \le i \le n$, *with left-associated embedded relations, i.e.,*:

$$\begin{aligned} P^* &= \overset{n-1}{\underset{i=0}{R}}(s_i r_{ij} s_j), \quad j = i+1,\ s_0 = \varnothing \\ &= (\ldots((((\varnothing) r_{01} s_1) r_{12} s_2) r_{23} s_3) \ldots r_{n-1,n} s_n) \end{aligned} \tag{37}$$

where s_i, $s_j \in \mathbb{P} \subset \text{Þ}\Im \sqsubset \mathfrak{U}$, *and* $r_{ij} \in \mathbb{R} \subset \text{Þ}\Im \sqsubset \mathfrak{U}$ *are given in Definitions* 18 *and* 21, *respectively.*

Proof. Given $s_0 = \varnothing$, Theorem 4 is proved in the following sequence:

$$\begin{aligned} P_0 &= s_0 r_{01} s_1 = r_{01}(\varnothing, s_1) = s_1 \\ P_1 &= s_1 r_{12} s_2 = r_{12}(s_1, s_2) = r_{12}(r_{01}(\varnothing, s_1), s_2) \\ P_2 &= s_2 r_{23} s_3 = r_{23}(s_2, s_3) = r_{23}(r_{12}(r_{01}(\varnothing, s_1), s_2), s_3) \\ &\ldots \\ P_{n-1} &= s_{n-1} r_{n-1,n} s_n = r_{n-1,n}(s_{n-1}, s_n) \\ &= r_{n-1,n}(r_{n-2,n-1}(\ldots(r_{23}(r_{12}(r_{01}(\varnothing, s_1), s_2), s_3) \\ &\qquad \ldots), s_{n-1}), s_n) \\ &= (\ldots((((\varnothing) r_{01} s_1) r_{12} s_2) r_{23} s_3) \ldots r_{n-1,n}\ s_n) \\ &= \overset{n-1}{\underset{i=0}{R}}(s_i r_{ij} s_j) = \overset{n-1}{\underset{i=0}{R}} P_i = P^* \quad \square \end{aligned} \tag{38}$$

Definition 23. The *schema of an abstract process,* $\mathscr{P}$, in $\mathfrak{U}$ is a general pattern of processes identified by the name of the process $P|$PM and three sets of default parameter structures known as those of *inputs* $I|$SM, *outputs* $O|$SM, and *global structure models* $GM|$SM, i.e.,:

$$\begin{aligned} &\mathscr{P} \triangleq P|\text{PM}(\langle I|\text{SM}\rangle, \langle O|\text{SM}\rangle, \langle GM|\text{SM}\rangle), \\ &\quad P \subset \text{Þ}P \subset \text{Þ}F \subset \text{Þ}\widetilde{\Im} \sqsubset \wp \sqsubset \mathfrak{U}, \\ &\quad I, O \subset \text{Þ}S \subset \text{Þ}A \subset \text{Þ}\widetilde{\Im} \sqsubset \wp \sqsubset \mathfrak{U}, \\ &\quad GM \subset \text{Þ}A \subset \text{Þ}\widetilde{\Im} \sqsubset \wp \sqsubset \mathfrak{U} \end{aligned} \tag{39}$$

where *GM* represents a set of global object structure models in the system whose lifecycle is usually longer than that of the invoking process.

Example 8. The schemas of the functions of the digital clock system, *Clock*§.*Functions*|PM, can be formally modeled according to Definition 23 as follows:

$$\begin{aligned} &\textit{Clock}\S.\textit{Functions}|\text{PM} \\ &\quad \triangleq (\textit{SetTime}|\text{PM}(\langle \text{I}::\ \textit{Key}_1|\text{E}\rangle, \\ &\qquad\qquad \langle \text{O}::\ \textit{LED}|\text{hh:mm}\rangle, \\ &\qquad\qquad \langle \textit{GM}::\ \textit{Keypad}|\text{SM}, \textit{LED}|\text{SM}\rangle), \\ &\qquad \textit{ShowTime}|\text{PM}(\langle \text{I}::\ \textit{Time}|\text{hh:mm:ss}\rangle, \\ &\qquad\qquad \langle \text{O}::\ \textit{LED}|\text{hh:mm}\rangle, \\ &\qquad\qquad \langle \textit{GM}::\ \textit{LED}|\text{SM}\rangle), \\ &\qquad \textit{TickTime}|\text{PM}(\langle \text{I}::\ ()\rangle, \langle \text{O}::\ ()\rangle, \\ &\qquad\qquad \langle \textit{GM}::\ \textit{Pulse}|\text{SM}, \textit{LED}|\text{SM}\rangle) \\ &\quad) \end{aligned} \tag{40}$$

Example 9. According to Theorem 4 and Definition 23, the process model of the show time process in the digital clock, *Clock*§.*ShowTime*|PM, can be formally refined as an embedded relational process model based on Example 8 as follows:

$$\begin{aligned} &\textit{ShowTime}|\text{PM}(\langle \text{I}::\ \textit{Time}|\text{hh:mm:ss}\rangle, \langle \text{O}::\ \textit{LED}|\text{hh:mm}\rangle, \\ &\qquad\qquad \langle \text{GM}::\ \textit{LED}|\text{SM}\rangle) \\ &\quad \triangleq \{ \rightarrow \textit{LED}|\text{SM.PORT}|\text{SM}(\textit{LED}_1|\text{P}) := \textit{Time}|\text{hx} \\ &\qquad \rightarrow \textit{LED}|\text{SM.PORT}|\text{SM}(\textit{LED}_2|\text{P}) := \textit{Time}|\text{xh} \\ &\qquad \rightarrow \textit{LED}|\text{SM.PORT}|\text{SM}(\textit{LED}_3|\text{P}) := \textit{Time}|\text{mx} \\ &\qquad \rightarrow \textit{LED}|\text{SM.PORT}|\text{SM}(\textit{LED}_4|\text{P}) := \textit{Time}|\text{xm} \\ &\quad \} \end{aligned} \tag{41}$$

where only the sequential process relation $\rightarrow$ in $\mathbb{R}$ is applied in this problem.

Corollary 5. *Software is a higher-order function of functions represented by a finite embedded sequence of left-associated processes rather than a linear list of statements or a system of combinational logic.*

Theorem 4 and Corollary 5 indicate that software, algorithms, and programs are a finite composition of embedded processes or a chain of function of functions according to the ERMP model.

Corollary 6. *The context of software is a dynamic environment embodied by its SMs,* $SMs \sqsubset A$, *which is determined by the outcomes of the embedded left-associated embedded processes where each current process is related to the cumulative results of all previous processes buffered in the SMs.*

As discovered in Theorem 4, the ERMP model is a novel denotational mathematical structure that reveals the

RESEARCH ARTICLE

fundamental difference of software from the conventional structures such as those of a list of statements, a sequence of functions, or automata. The uniqueness of the mathematical structure of the embedded relations is a chain of functions of functions, which is varying step-by-step during a program execution.

THEOREM 5. *The entire space of software behaviors,* Ω, *is a closure of all potential computational operations of relational process compositions* $\mathbb{R} \times (\mathbb{P} \times \mathbb{P})$ *between any pair of meta-processes,* $\mathbb{P} \subset \text{Þ}P \subset \text{Þ}\Im \sqsubset \mathfrak{U}$, *composed by each of the relational operators* $\mathbb{R} \subset \text{Þ}F \subset \text{Þ}\mathfrak{F} \sqsubset \mathfrak{U}$, *which yields* $\Omega = 4{,}913$.

PROOF. The closure of software behaviors Ω in $\mathfrak{U}$ can be proved based on Theorem 4 as well as Definitions 18 and 21 as follows:

$$\begin{aligned} \Omega &= |\mathbb{R} \times (\mathbb{P} \times \mathbb{P})| \\ &= |\mathbb{R}|\bullet(|\mathbb{P}|\bullet|\mathbb{P}|) \\ &= 17^3 = 4{,}913 \quad \square \end{aligned} \tag{42}$$

Theorem 5 demonstrates the expressive power of the process algebra based on the general ERMP model of software systems towards computational behavior modeling and programming. It is noteworthy that an ordinary programming language may empirically introduce only 150 to 300 individual instructions. However, the entire behavioral space of software in software science may generate upto 4,913 computational operations, though it only adopts fairly small finite sets of 17 meta-processes and 17 relational process operators.

5. MATHEMATICAL MODELS OF SOFTWARE SYSTEMS—INTERACTIVE PROCESS DISPATCH STRUCTURES

It is formally obtained in preceding sections that the mathematical model of software behaviors at the component level in $\mathfrak{U}$ is a finite chain of embedded processes modeled by PMs. It is also demonstrated that the algebraic model of software is a Cartesian product between PM and SM in the meta-processes. In order to extend the theories of software science to the system level based on embedded process models of software at the component level, the top-level software system will be formally synthesized as an event-driven dispatch structure on PMs according to Definition 5.

DEFINITION 24. The set of *system functions* of software $\wp$, $\S F$, $\S F \subset \text{Þ}\widehat{\S} \sqsubset \mathfrak{U}$, is an abstract model of a set of interactive relations between the sets of system events E, behavioral processes P^*, and object structures S in $\mathfrak{U}$, i.e.,:

$$\begin{aligned} \S F \triangleq E \times P \times S, \quad & E \subset \Theta \subset \text{Þ}\mathfrak{G} \sqsubset \wp \sqsubset \mathfrak{U}, \\ & P \subset \text{Þ}P \subset \text{Þ}\mathfrak{F} \sqsubset \wp \sqsubset \mathfrak{U}, \\ & S \subset \text{Þ}\mathfrak{S} \sqsubset \wp \sqsubset \mathfrak{U} \\ = E|\Theta \times P|\text{PM} \times S|\text{SM} & \end{aligned} \tag{43}$$

5.1. Mathematical Models of Events and Environment of Software Systems

DEFINITION 25. The *environment of software* $\wp$, Θ, $\Theta \subset \text{Þ}\mathfrak{G} \sqsubset \wp \sqsubset \mathfrak{U}$, encompasses two categories of objects known as sets of *external events* E' and *external structures* S' interacting with another *software systems* $\wp'$ in $\mathfrak{U}$, i.e.,:

$$\begin{aligned} \Theta = \{E', S'\}, \quad & E' \subset \text{Þ}\mathfrak{S} \sqsubset \mathfrak{U} \wedge E' \not\sqsubset \wp, \\ & S' \subset \text{Þ}\mathfrak{S} \sqsubset \mathfrak{U} \wedge S' \not\sqsubset \wp \end{aligned} \tag{44}$$

where E' includes the timing events related to a system clock that is treated as a default independent system.

Interactions between a given process or software to its environment can be classified as sets of inputs (I|SM) or outputs (O|SM), and the set of global object structures (GM|SM), $\mathscr{P} \triangleq P|\text{PM}(\langle I|\text{SM}\rangle, \langle O|\text{SM}\rangle, \langle GM|\text{SM}\rangle)$, as given in Definition 23.

DEFINITION 26. The *sets of input and output relations of software* $\wp$, R^i and R^o, $R^i \subset \Theta \times \wp \sqsubset \mathfrak{U}$, $R^o \subset \wp \times \Theta \sqsubset \mathfrak{U}$, are abstract models of system inputs and outputs in $\mathfrak{U}$, i.e.,:

$$\begin{aligned} R^i \triangleq \Theta \times \wp = E' \times \wp | S' \times \wp, & \\ \Theta \subset \text{Þ}\mathfrak{S} \sqsubset \wp \sqsubset \mathfrak{U}, \quad & E', S' \subset \Theta \\ R^o \triangleq \wp \times \Theta = \wp \times E' | \wp \times S', & \\ \Theta \subset \text{Þ}\mathfrak{S} \sqsubset \wp \sqsubset \mathfrak{U}, \quad & E', S' \subset \Theta \end{aligned} \tag{45}$$

An event is treated as a special type of system variables that represents the occurring of a predefined external or internal change of status, such as a request of users, a change of the environment, a device interrupt request, a change of time, and a change of internal status. System events as referred in Definitions 5, 24, 25, and 26 can be formally modeled as follows.

DEFINITION 27. An *event* of a software system $\wp$, E, $E \subset \Theta \subset \text{Þ}\mathfrak{G} \sqsubset \wp \sqsubset \mathfrak{U}$ is an abstract model of a set of special system variables such as an external trigger, a system timing, or a device interrupt in $\mathfrak{U}$, i.e.,:

$$\begin{aligned} E \triangleq V' \times T', \quad & V' \subset \Theta \subset \text{Þ}\mathfrak{V} \sqsubset \wp \sqsubset \mathfrak{U}, \\ & T' \subset \Theta_t \subset \text{Þ}\mathfrak{T} \sqsubset \wp \sqsubset \mathfrak{U} \end{aligned} \tag{46}$$

DEFINITION 28. The *taxonomy of system events*, E, $E \subset \Theta \subset \text{Þ}\mathfrak{G} \sqsubset \mathfrak{U}$, is classified into three categories known as the *trigger events* $@E_{tr}|S$, timing events $@E_t|\text{TM}$, and *device interrupt events* $@E_{\text{int}}|\odot$ in $\mathfrak{U}$, i.e.,:

$$E = \{@E_{tr}|S, @E_t|\text{TM}, @E_{\text{int}}|\odot\} \tag{47}$$

where @ is the prefix of an event, and the suffixes of $|S$, |TM, and $|\odot$ represent that external triggers are denoted in the type of string, timing events in the type of date/time, and device interrupts in the type of interrupt, respectively.

DEFINITION 29. The *trigger event*, $@E_{tr}|S$, is an differential change of external status to a software system, which can be either a *positive* trigger $@E^{+}_{tr}|S$ or a *negative* one $@E^{-}_{tr}|S$ in $\mathfrak{U}$, $@E_{tr}|S \subset E \subset \Theta \subset \wp\mathfrak{G} \sqsubset \wp \sqsubset \mathfrak{U}$, i.e.,:

$$@E_{tr}|S \triangleq \begin{cases} @E^{+}_{tr}|\mathrm{S} = \left(\dfrac{d(\sigma_e|\mathrm{BL})}{dt} = T|\mathrm{BL}\right) \\ \qquad \wedge (\sigma_c|\mathrm{BL} = T|\mathrm{BL}) \\ \qquad = (\sigma_c|\mathrm{BL} \oplus \sigma_l|\mathrm{BL} = T|\mathrm{BL}) \\ \qquad \wedge (\sigma_c|\mathrm{BL} = T|\mathrm{BL}) \\ @E^{-}_{tr}|S = \left(\dfrac{d(\sigma_e|\mathrm{BL})}{dt} = T|\mathrm{BL}\right) \\ \qquad \wedge (\sigma_c|\mathrm{BL} = F|\mathrm{BL}) \\ \qquad = (\sigma_c|\mathrm{BL} \oplus \sigma_l|\mathrm{BL} = T|\mathrm{BL}) \\ \qquad \wedge (\sigma_c|\mathrm{BL} = F|\mathrm{BL}) \end{cases} \tag{48}$$

where the event $\sigma_e|\mathrm{BL}$ is captured by a default *system scan mechanism*, $d(\sigma_e|\mathrm{BL})/dt$, from the input or interrupt ports. When the input status has been changed from false to true determined by the current scan $\sigma_c|\mathrm{BL}$ and last scan $\sigma_l|\mathrm{BL}$, it is identified as a positive trigger $@E^{+}_{tr}|S$; otherwise, the event is a negative trigger $@E^{-}_{tr}|S$.

DEFINITION 30. The *timing event*, $@E_t|\mathrm{TM}$, is an externally or internally generated event based on the global or local clock $\S t|\mathbb{T}$, which is classified as the absolute timing events $\S t|$hh:mm:ss:ms, duration events $\S t|$hh:mm:ss:ms$+\Delta n|\mathrm{TM}$, or timer events $@Timer_i(n|\mathrm{TM})|\mathrm{BL}$ in $\mathfrak{U}$, $@E_t|\mathrm{TM} \subset E \subset \Theta \subset \wp\mathfrak{G} \sqsubset \wp \sqsubset \mathfrak{U}$, i.e.,:

$$@E_t|\mathrm{TM} \triangleq \begin{cases} \S t|\text{hh:mm:ss:ms} \\ \S t|\text{hh:mm:ss:ms} + \Delta n|\mathrm{TM} \\ @Timer_i(n|\mathrm{TM})|\mathrm{BL} \\ \quad \triangleq \downarrow (Timer_i|\mathrm{SM}(n|\mathrm{TM})) = 0 \end{cases} \tag{49}$$

where the absolute timing event is a given system time $\S t|$hh:mm:ss:ms from hour through millisecond; the duration event $\S t|\mathrm{TM} + \Delta n|\mathrm{TM}$ is a relative point of time determine by the given interval $\Delta n|\mathrm{TM}$ based on the system clock at the initial time $\S t|\mathrm{TM}$; and the timer event represents a time-out indicator generated by a downward counter when its given value $n|\mathrm{TM}$ is reduced to zero by the cyclic updating of the system clock.

DEFINITION 31. The *interrupt event*, $@E_{\text{int}}|\odot$, is a device trigged event via the system interrupt capture mechanisms $PORT_{\text{int}}|\mathrm{SM}$ represented by a preassigned interrupt number $Int\#|\odot$ at a certain hexadecimal interrupt address $ptr_{\text{int}}|\mathrm{P}$ in $\mathfrak{U}$, $@E_{\text{int}}|\odot \subset E \subset \Theta \subset \wp\mathfrak{G} \sqsubset \wp \sqsubset \mathfrak{U}$, i.e.,:

$$@E_{\text{int}}|\odot \triangleq PORT_{\text{int}}|\mathrm{SM}(ptr_{\text{int}}|\mathrm{P}, Int\#|\odot) \tag{50}$$

where $|\odot$ denotes the type suffix of interrupt.

In the discourse of the software environment $\mathfrak{U}$, the external trigger events are captured by cyclic system scan from the interface ports of the system. The random interrupt events are captured by a specific system interrupt vector from the external devices. However, the timing events can be captured by either periodical polling or time-out interrupts of the system clock and software timers, respectively.

5.2. The Dispatch Theory of System Behaviors of Software

According to Definitions 5 and 24, the system behavior of software at the top level is an event-driven dispatch structure where system events are formally modeled in Section 5.1.

COROLLARY 7. *The overall logical model of software systems,* $\S\wp$, $\S\wp \subset \wp\widehat{\S} \sqsubset \mathfrak{U}$, *is a three dimensional structure yielded by the Cartesian product among the sets of system events E, system functions F, and system architectures A in* $\mathfrak{U}$, *i.e.,*:

$$\begin{aligned} \S\wp &= E \times F \times A, \quad E \subset \Theta \subset \wp\mathfrak{G} \sqsubset \wp \sqsubset \mathfrak{U}, \\ &\qquad F \subset \wp\widehat{\mathfrak{F}} \sqsubset \wp \sqsubset \mathfrak{U}, \quad A \subset \wp\widehat{\mathfrak{S}} \sqsubset \wp \sqsubset \mathfrak{U} \\ &= E|\Theta \times F|PM \times A|SM \end{aligned} \tag{51}$$

DEFINITION 32. A *process dispatch structure*, Ψ, in $\mathfrak{U}$ is an event-driven mechanism of a software system at the top level embodied by a Cartesian product, $E \times P^*$, between the set of events E captured by the system and the set of predesigned processes P^* of the system, i.e.,:

$$\begin{aligned} \Psi \triangleq \S \to \overset{n}{\underset{i=1}{R}}(@e_i|\mathrm{E} \hookrightarrow P_i|\mathrm{PM}|@e'_i|\mathrm{E} \hookrightarrow \varnothing) \\ \to \S, e_i \in E \wedge e'_i \notin E \end{aligned} \tag{52}$$

where $e_i \in E = \{@E_{tr}|\mathrm{S}, @E_t|\mathrm{TM}, @E_{\text{int}}|\odot\}$, $P_i \in P^*$ is a meta or complex process as formally modeled in Section 4, $E \times P^*$ is the set of process *dispatching rules*, and $\S$ denotes the software system as the overall controller for process dispatch.

On the basis of the process dispatch model specified in Eq. (52) and the abstract behavioral model of software at the component level as given in Theorem 4, the top-level model of a software system can be described as follows.

THEOREM 6. *The overall mathematical model of software systems (MMSS),* $\wp\S$, *in* $\mathfrak{U}$ *is a dispatch structure with finite sets of embedded relational processes at the subsystem and component levels, i.e.,*:

$$\begin{aligned} \wp\S &\triangleq \overset{m}{\underset{k=1}{R}}(@e_k|E \hookrightarrow P_k|PM) \\ &= \overset{m}{\underset{k=1}{R}}\left[@e_k|E \hookrightarrow \overset{n-1}{\underset{i=1}{R}}(p_i(k)r_{i,i+1}(k)p_{i+1}(k))|PM\right] \end{aligned} \tag{53}$$

RESEARCH ARTICLE

PROOF. Theorem 6 can be proved according Theorem 4 and Definitions 32. Substituting $P_k|$PM in Eq. (53) by Eq. (37) as given in Theorem 4, the general form of software system $\wp\S$ is obtained as an event-driven dispatch of chains of embedded relational processes. □

Theorem 6 and Definition 5 create a unified mathematical model of abstract software systems that can be deductively reduced to lower-level process and structure models from the top down in system analyses, and be inductively synthesized based on lower-level models from the bottom up.

EXAMPLE 10. The system dispatch process of the digital clock, *Clock*§.*SysDispatch*|PM, can be formally specified according to Theorem 6 on the basis of previous Examples 5, 7–9 as follows:

```
Clock§.SysDispatch|PM
    (<I:: SCB|SM.@CurrentEvent|S>,
     <O::( )>, <GM:: Clock§, SCB|SM>) ≙
{→SysEvent|N : = SCB|SM.#FuncKey|N
 →SCB|SM.#FuncKey|N: = 0
                        // Reset used event
 → ◆ SysEvent|N =
    (1:↳ SetTime|PM (<I:: @SetT|S>,
          <O::LED|hh:mm>,
        <GM:: Keypad|SM, LED|SM, SCB|SM>)
     |2:↳ SetAlarm|PM (<I:: @SetA|S>,
          <O:: LED|hh:mm>,
        <GM:: Keypad|SM, LED|SM, SCB|SM>)
     |3:↳ ShowAlarm|PM (<I:: @ShowA|S>,
          <O::LED|SM>,
        <GM:: Keypad|SM, LED|SM, SBC|SM>)
     |4:↳ ReleaseAlarm|PM (<I:: @ReleaseA|S>,
          <O:: Bell|BL>,
        <GM:: Keypad|SM, Bell|SM, SCB|SM>)
     | ~: →∅
    )
}                                                    (54)
```

where *SCB*|SM is the system control block that buffers system scan status of external events and other global control variables.

Example 10 demonstrates that the architectural and functional integration of any software system can be implemented according to the general mathematical model of software (GMMS, Definition 5) and the overall mathematical model of software systems (MMSS, Theorem 6).

COROLLARY 8. *The abstraction principle of software systems states that both the architectures and functions of any software system* $\wp\S$ *in* $\mathbb{U}$ *can be inductively integrated and composed with decreasing details at different layers,* $0 \le k \le n$*, from the bottom up.*

COROLLARY 9. *The refinement principle of software systems states that both the architectures and functions of any software system* $\wp\S$ *in* $\mathbb{U}$ *can be deductively specified and analyzed with increasing details at different layers,* $0 \le k \le n$*, from the top down.*

The software science theories represented by the algebraic software model GMMS, the embedded process model ERMS, and the event-driven system dispatch model MMSS as formally obtained in Definition 5, Theorems 4, and Theorem 6, respectively, can be deductively applied to efficiently generate any software application or program, particularly large-scale ones, based on formally modeled PMs and SMs of a given system. Experiments on transforming the formal software system models of a set of large-scale real-world systems into code have been reported on the *automated teller machine* (ATM) (Wang et al., 2010d), the *telephone switching system* (TSS) (Wang, 2009c), the *lift dispatching system* (LDS) (Wang et al., 2009a), the *real-time operating system* (RTOS+) (Wang et al., 2010b, c), the cognitive knowledge base (CKB) (Wang and Tian, 2013), and the *air traffic control system* (ATCS) (Wang et al., 2013a, b). The software science theories powered by the RTPA methodology enable machines to realize autonomic code generation (Wang et al., 2010a) in any programming language based on the formal models of software systems towards the ultimate aim of software science and software engineering.

6. CONCLUSIONS

A theoretical framework of software science has been rigorously presented. This work has revealed that the nature and mathematical models of software systems at the top level is an event-driven dispatch structure according to the MMSS model, while those at the intermediate component or subsystem levels are a finite set of embedded processes according to the EPMS model. A general mathematical model of software has been created that formally describes the object entities of software by a set of structure models (SMs), and the functional behaviors as a set of process models (PMs). Then, the entire mathematical model of software has been created as a Cartesian product $\wp\S \triangleq E \times PM \times SM$ based on the general mathematical model of software (GMMS). On the basis of the formal theories of software science, novel and rigorous methodologies and technologies of software engineering will be developed in empirical studies and practices in the software industry.

The rigorous theories, mathematical models, and formal methodologies of software science obtained in a series of basic researches have been successfully applied in real-world large-scale software engineering projects by groups of software engineers and graduate students. All pilot projects have consistently yielded significant improvement of software productivity by at least three folds due to the rigorous methodology, design efficiency, complexity

reduction, and inherited quality of the software science methodologies.

Acknowledgment: This work is supported in part by a discovery fund granted by the Natural Sciences and Engineering Research Council of Canada (NSERC). I would like to sincerely thank Professor Tony C. A. R. Hoare for his insight, advice, supports, enjoyable discussions, and hosting of my sabbatical leave at Oxford University as a visiting professor in 1995. I would like to thank the anonymous reviewers for their valuable comments on the previous version of this paper.

References

Aristotle, (384 BC–322 BC). (1989). Prior Analytics, translated by Robin Smith, Hackett.

Baase, S. (1978). *Computer Algorithms: Introduction to Design and Analysis* (Addison-Wesley, NY).

Bass, L., Clements, P., and Kazman, R. (1998). *Software Architecture in Practice* (Addison-Wesley, NY).

Bishop, J. (1986). *Data Abstraction in Programming Languages* (Addison-Wesley, Reading, MA).

Bjorner, D. and Jones, C. B. (1982). *Formal Specification and Software Development* (Prentice-Hall, Englewood Cliff, NJ).

Boolos, G. S., Burgess, J. P., and Jeffrey, R. C. (2002). *Computability and Logic*, 4th edn. (Cambridge University Press, NY).

Brooks, F. P. Jr. (1975). *The Mythical Man-Month: Essays on Software Engineering* (Addison-Wesley Longman, Inc., Boston).

Cardelli, L. and Wegner, P. (1985). On understanding types, data abstraction and polymorphism. *ACM Computing Surveys* 17(4): 471–522.

Cerone, A. (2000). *Process Algebra versus Axiomatic Specification of a Real-Time Protocol, LNCS #1816*, (Springer, Berlin), Vols. 57–67.

Dahl, O.-J. and Nygaard, K. (1967). Class and subclass declarations. *Simulation Programming Languages*, edited by Buxton, J. N., (North Holland, Amsterdam), 158–174.

Descartes, R. (1979). *Meditations on First Philosophy* (D. Cress trans., Indianapolis: Hackett Publishing Co. Inc.).

Gibbs, W. (1994). *Software's Chronic Crisis* (Scientific American), September, 86–95.

Dijkstra, E. W. (1976). *A Discipline of Programming* (Prentice Hall, Englewood Cliffs, NJ).

Glorioso, R. M. and Osorio, F. C. C. (1980). *Engineering Intelligent Systems: Concepts, Theories, and Applications* (Digital Press, USA).

Gowers, T. (ed.) (2008). *The Princeton Companion to Mathematics* (Princeton University Press, NJ).

Gries, D. (1981). *The Science of Programming* (Spinger-Verlag, Berlin).

Guttag, J. V. (1977). Abstract data types and the development of data structures. *Comm. ACM* 20(6): 396–404.

Hartmanis, J. (1994). On computational complexity and the nature of computer science, 1994 turing award lecture. *Communications of the ACM* 37(10): 37–43.

Higman, B. (1977). *A Comparative Study of Programming Languages*, 2nd edn. (MacDonald).

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Comm. ACM* 12(10): 576–580.

Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM* 21(8): 666–677.

Hoare, C. A. R. (1985). *Communicating Sequential Processes* (Prentice Hall International, Englewood Cliffs, NJ).

Hoare, C. A. R., Hayes, I. J., He, J., Morgan, C. C., Roscoe, A. W., Sanders, J. W., Sorensen, I. H., Spivey, J. M., and Sufrin, B. A. (1987). Laws of programming. *Communications of the ACM* 30(8): 672–686.

Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to Automata Theory, Languages, and Computation* (Addison-Wesley Publishing Co., MA).

Horn, A. (1951), On sentence which are true of direct unions of algebras. *J. of Symbolic Logic* 16: 14–21.

Horowitz, E. (1984). *Fundamentals of Programming Languages*, 2nd edn. (Computer Science Press, Rockville, MD).

Klir, G. J. (1992). *Facets of Systems Science* (Plenum, New York).

Knuth, D. E. (1997), Fundamental algorithms. *Series of The Art of Computer Programming* (Addison-Wesley, Reading, MA), Vol. 1.

Kowalski, R. A. (1988), The early years of logic programming. *Comm. ACM* 31(1): 38–43.

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

Llewellyn, J. A. (1987), *Information and Coding* (Chartwell-Bratt, Bromley, UK).

Louden K. C. (1993). *Programming Languages: Principles and Practice* (PWS-Kent Publishing Co., Boston).

Mandrioli, D. and Ghezzi, C. (1987). *Theoretical Foundations of Computer Science* (John Wiley & Sons, New York).

Martin-Lof, P. (1975). *An Intuitionistic Theory of Types: Predicative Part*, edited by Rose, H. and Shepherdson, J. C. (Logic Colloquium 1973, NorthHolland).

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

Meyer, B. (1988). *Object-Oriented Software Construction* (Prentice-Hall, Englewood Cliff, NJ).

Milner, R. (1980). *A Calculus of Communicating Systems* (Springer, LNCS #92, Berlin, Germany).

Mitchell, J. C. (1990), Type systems for programming languages. *Handbook of Theoretical Computer Science*, edited by J. van Leeuwen (North Holland), 365–458.

Newton, I. (1687). *The Mathematical Principles of Natural Philosophy*, 1st edn., Latin, 1687, English ed., 1729.

Parnas, D. L. (2001). *Software Fundamentals: Collected Papers by David L. Parnas*, edited by Hoffman, D. M. and Weiss, D. M. (Addison-Wesley, NJ).

Russell, B. (1903). *The Principles of Mathematics* (George Allen & Unwin, London).

Sommerville, I. (1995). *Software Engineering*, 5th edn. (Addison-Wesley, NY).

Turing, A. M. (1950), Computing machinery and intelligence. *Mind* (59): 433–460.

von Neumann, J. (1946), The principles of large-scale computing machines, reprinted in *Annals of History of Computers* 3(3): 263–273.

von Neumann, J. (1963). *General and Logical Theory of Automata*, edited by Taub, A. H., (Collected Works, Pergamon), Vol. 5, pp. 288–328.

Wang, Y. (2015), Towards the abstract system theory of system science for cognitive and intelligent systems. *Springer Journal of Complex and Intelligent Systems* 1(1), in press.

Wang, Y. (2014), On granular algebra: A denotational mathematics for modeling granular systems and granular computing. *J. Adv. Math. Appl.* 3(1): 60–73.

Wang, Y. (2013), On semantic algebra: A denotational mathematics for cognitive linguistics, machine learning, and cognitive computing, *J. Adv. Math. Appl.* 2(2): 145–161.

Wang, Y. (2012a), On denotational mathematics foundations for the next generation of computers: Cognitive computers for knowledge processing. *J. Adv. Math. Appl.* 1(1): 118–129.

Wang, Y. (2012b), Inference algebra (IA): A denotational mathematics for cognitive computing and machine reasoning (II). *International Journal of Cognitive Informatics and Natural Intelligence* 6(1): 21–46.

Wang, Y. (2012c), Keynote: Towards the next generation of cognitive computers: Knowledge versus data computers. *Proc. 12th International Conference on Computational Science and Applications (ICCSA'12)*, Salvador, Brazil, Springer, June, 18–21.

Wang, Y. (2012d), In search of denotational mathematics: Novel mathematical means for contemporary intelligence, brain, and knowledge sciences. *J. Adv. Math. Appl.* 1(1): 4–25.

Wang, Y. (2012e), Editorial: Contemporary mathematics as a metamethodology of science, engineering, society, and humanity, *J. Adv. Math. Appl.* 1(1): 1–3.

Wang, Y. (2011), Inference algebra (IA): A denotational mathematics for cognitive computing and machine reasoning (I), *International Journal of Cognitive Informatics and Natural Intelligence* 5(4): 61–82.

Wang, Y. (2010a), Cognitive robots: A reference model towards intelligent authentication. *IEEE Robotics and Automation* 17(4): 54–62.

Wang, Y. (2010b), On formal and cognitive semantics for semantic computing. *International Journal of Semantic Computing* 4(2): 203–237.

Wang, Y. (2009a), Paradigms of denotational mathematics for cognitive informatics and cognitive computing. *Fundamenta Informaticae* 90(3): 282–303.

Wang, Y. (2009b), On cognitive computing. *International Journal of Software Science and Computational Intelligence* 1(3): 1–15.

Wang, Y. (2009c), The formal design model of a telephone switching system (TSS). *International Journal of Software Science and Computational Intelligence* 1(3): 92–116.

Wang, Y. (2009d), On the cognitive complexity of software and its quantification and formal measurement. *International Journal of Software Science and Computational Intelligence* 1(2): 31–53.

Wang, Y. (2009e), A cognitive informatics reference model of autonomous agent systems (AAS), *International Journal of Cognitive Informatics and Natural Intelligence* 3(1): 1–16.

Wang, Y. (2009f), Editorial: Convergence of software science and computational intelligence. *International Journal of Software Science and Computational Intelligence* 1(1): i–xii.

Wang, Y. (2009g), On abstract intelligence: Toward a unified theory of natural, artificial, machinable, and computational intelligence. *International Journal of Software Science and Computational Intelligence* 1(1): 1–17.

Wang, Y. (2009a), A formal syntax of natural languages and the deductive grammar, *Fundamenta Informaticae* 90(4): 353–368.

Wang, Y. (2008a), RTPA: A denotational mathematics for manipulating intelligent and computing behaviors. *International Journal of Cognitive Informatics and Natural Intelligence* 2(2): 44–62.

Wang, Y. (2008b), Deductive semantics of RTPA. *International Journal of Cognitive Informatics and Natural Intelligence* 2(2): 95–121.

Wang, Y. (2008c), On the big-*R* notation for describing iterative and recursive behaviors. *International Journal of Cognitive Informatics and Natural Intelligence* 2(1): 17–23.

Wang, Y. (2008d), On contemporary denotational mathematics for computational intelligence. *Transactions on Computational Science* (2): 6–29.

Wang, Y. (2008e), On concept algebra: A denotational mathematical structure for knowledge and software modeling. *International Journal of Cognitive Informatics and Natural Intelligence* 2(2): 1–19.

Wang, Y. (2008f), Mathematical laws of software. *Transactions of Computational Science, Springer* 2: 46–83.

Wang, Y. (2008g), A hierarchical abstraction model for software engineering, *Proc. 30th IEEE International Conference on Software Engineering (ICSE'08)*, 2nd International Workshop on the Role of Abstraction in Software Engineering (ROA'08), ACM/IEEE CS Press, Leipzig, Germany, May, Vol. II, pp. 43–48.

Wang, Y. (2008h), On system algebra: A denotational mathematical structure for abstract system modeling. *Int'l Journal of Cognitive Informatics and Natural Intelligence* 2(2): 20–42.

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

Wang, Y. (2007b), The theoretical framework of cognitive informatics. *The International Journal of Cognitive Informatics and Natural Intelligence (IJCINI)*, (IPI Publishing, USA) 1(1), 1–27.

Wang, Y. (2006), On the informatics laws and deductive semantics of software. *IEEE Transactions on Systems, Man, and Cybernetics (C)* 36(2): 161–171.

Wang, Y. (2005), The development of the IEEE/ACM software engineering curricula. *IEEE Canadian Review* 51(2): 16–20.

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

Wang, Y. (2003b), Using process algebra to describe human and software system behaviors. *Brain and Mind* 4(2): 199–213.

Wang, Y. (2002), The real-time process algebra (RTPA). *Annals of Software Engineering* (14): 235–274.

Wang, Y. (2001a), Formal description of the UML architecture and extendibility. *International Journal of the Object, Hermes Science Publications, Paris* 6(4): 469–488.

Wang, Y. (2001b). A web-based european software process benchmarking server. *The 23rd IEEE International Conference on Software Engineering (ICSE'01)*, Toronto, May, 439–440.

Wang, Y. (1996). A new sorting algorithm: Self-indexed sort. *ACM SIGPLAN* 31(3): USA, 28–36.

Wang, Y. and Tian, Y. (2013). A formal knowledge retrieval system for cognitive computers and cognitive robotics. *International Journal of Software Science and Computational Intelligence* 5(2): 37–57.

Wang, Y. and Chiew, V. (2011). Empirical studies on the functional complexity of software in large-scale software systems. *International Journal of Software Science and Computational Intelligence* 3(3): 1–29.

Wang, Y. and Chiew, V. (2010). On the cognitive process of human problem solving. *Cognitive Systems Research: An International Journal, Elsevier* 11(1): 81–92.

Wang, Y. and Patel, S. (2009). Exploring the cognitive foundations of software engineering. *International Journal of Software Science and Computational Intelligence* 1(2): 1–19.

RESEARCH ARTICLE

Wang, Y. and Huang, J. (2008). Formal modeling and specification of design patterns using RTPA. *International Journal of Cognitive Informatics and Natural Intelligence* 2(1): 100–111.

Wang, Y. and Patel, S. (2004). On modeling object-oriented information systems. *International Journal of Software and System Modeling* 3(4): 258–261.

Wang, Y. and Bryant, A. (2002). Process-based software engineering: Building the infrastructure. *Annals of Software Engineering, Springer* 14: 9–37.

Wang, Y. and King, G. (2000). *Software Engineering Processes: Principles and Applications* (CRC Book Series in Software Engineering, CRC Press, USA), Vol. 1, 752pp.

Wang, Y. and Patel, D. (2000). Comparative software engineering: Review and perspectives. *Annals of Software Engineering, Springer* 10: 1–10.

Wang, Y., Pedrycz, W., Lu, J., and Luo, G. (2013a). Denotational mathematical models of an air traffic control system (ATCS-II): Process models of functions in RTPA. *J. Adv. Math. Appl.* 2(1): 82–110.

Wang, Y., Pedrycz, W., Lu, J., and Luo, G. (2013b), Denotational mathematical models of an air traffic control system (ATCS-I): Structure models of architectures in RTPA. *J. Adv. Math. Appl.* 2(1): 32–47.

Wang, Y., Tan, X., and Ngolah, C. F. (2010a), Design and implementation of an autonomic code generator based on RTPA (RTPA-CG). *International Journal of Software Science and Computational Intelligence* 2(2): 44–67.

Wang, Y., Ngolah, C. F., Zeng, G., Sheu, P. C.-Y., Choy, C. P., and Tian, Y. (2010b). The formal design models of a real-time operating system (RTOS+): Conceptual and architectural frameworks. *International Journal of Software Science and Computational Intelligence* 2(2): 105–122.

Wang, Y., Zeng, G., Ngolah, C. F., Sheu, P. C.-Y., Choy, C. P., and Tian, Y. (2010c). The formal design models of a real-time operating system (RTOS+): Static and dynamic behaviorc models. *International Journal of Software Science and Computational Intelligence* 2(3): 79–105.

Wang, Y., Zhang, Y., Sheu, P., Li, X., and Guo, H. (2010d). The formal design models of an automatic teller machine (ATM). *International Journal of Software Science and Computational Intelligence* 2(1): 102–131.

Wang, Y., Ngolah, C. F., Tan, X., Tian, Y., and Sheu, P. C.-Y. (2010e). The formal design models of a set of abstract data models (ADTs). *International Journal of Software Science and Computational Intelligence* 2(4): 72–104.

Wang, Y., Ngolah, F. C., Ahmadi, H., Sheu, P. C. Y., and Ying, S. (2009a). The formal design model of a lift dispatching system (LDS). *International Journal of Software Science and Computational Intelligence* 1(4): 111–137.

Wang, Y., Kinsner, W., and Zhang, D. (2009b). Contemporary cybernetics and its faces of cognitive informatics and computational intelligence. *IEEE Trans. on System, Man, and Cybernetics (Part B)*, 39(4): 1–11.

Wang, Y., Wang, Y., Patel, S., and Patel, D. (2006). A layered reference model of the brain (LRMB). *IEEE Trans. on Systems, Man, and Cybernetics (Part C)* 36(2): 124–133.

Wang Y., King, G., Fayad, M., Patel, D., Court, I., Staples, G., and Ross, M. (2000). On built-in tests reuse in object-oriented framework design. *ACM Computing Surveys* 32(1es): 7–12.

Wang, Y., King, G., Patel, D., Patel, S. and Dorling, A. (1999a), On coping with software dynamic 4 inconsistency at real-time by the built-in tests. *Annals of Software Engineering, Springer* 7: 283–296.

Wang Y., King, G., Dorling, A., Ross, M., Staples, G., and Court, I. (1999b). A worldwide survey on best practices towards software engineering process excellence. *ASQ Journal of Software Quality Professional* 2(1): 34–43.

Wang Y., King, G., Dorling, A., Patel, D., Court, I., Staples, G. and Ross, M. (1998). A worldwide survey on software engineering process excellence. *Proc. of IEEE 20th International Conference on Software Engineering (IEEE ICSE'98)* (Kyoto, April, IEEE Press), pp. 439–442.

Wilson, L. B. and Clark, R. G. (1988). *Comparative Programming Language* (Addison-Wesley Publishing Co., Wokingham, UK).

Wirth, N. (1976). *Algorithms + Data Structures = Programs* (Prentice-Hall, Englewood Cliff, NJ).

Received: 17 May 2013. Accepted: 15 January 2014.