

RTPA: A Denotational Mathematics for Manipulating Intelligent and Computational Behaviors

Yingxu Wang, University of Calgary, Canada

ABSTRACT

Real-time process algebra (RTPA) is a denotational mathematical structure for denoting and manipulating system behavioral processes. RTPA is designed as a coherent algebraic system for intelligent and software system modeling, specification, refinement, and implementation. RTPA encompasses 17 metaprocesses and 17 relational process operations. RTPA can be used to describe both logical and physical models of software and intelligent systems. Logic views of system architectures and their physical platforms can be described using the same set of notations. When a system architecture is formally modeled, the static and dynamic behaviors performed on the architectural model can be specified by a three-level refinement scheme at the system, class, and object levels in a top-down approach. RTPA has been successfully applied in real-world system modeling and code generation for software systems, human cognitive processes, and intelligent systems.

Keywords: cognitive processes; denotational mathematics; knowledge engineering; intelligent systems; process algebra; RTPA; software engineering; system architectures; system behaviors; system modeling; system refinement; system specification; software systems

INTRODUCTION

The modeling and refinement of software and intelligent systems require new forms of denotational mathematics that possess enough expressive power for rigorously describing system architectures and behaviors. Real-time process algebra (RTPA) is a new denotational mathematical structure for software system modeling, specification, refinement, and implementation for both real-time and nonreal-time systems (Wang, 2002b, 2007d; Wang & King, 2000). An important finding in formal methods is that a software system can

be perceived and described as the *composition* of a set of *processes*, which are constructed on the basis of algebraic operations. Theories on process algebras can be traced back to Hoare's *Communicating Sequential Processes* (CSP) (Hoare, 1978, 1985) and Milner's *Calculus of Communicating Systems* (CCS) (Milner, 1980). The *process metaphor* of software systems has evolved from concurrent processes to real-time process systems in the area of operating system research and formal methods (Boucher & Gerth, 1987; Hoare, 1978, 1985; Milner, 1980, 1989; Reed & Roscoe, 1986; Schneider, 1991).

Definition 1. A process is an abstract model of a unit of meaningful system behaviors that represents a transition procedure of the system from one state to another by changing values of the sets of inputs, outputs, and/or internal variables.

It is recognized that generic computing problems are 3-D problems, known as those of the behavior, space, and time dimensions, which require a denotational mathematical means for addressing the requirements in all dimensions, particularly the time dimension (Wang, 2002b, 2003b, 2007d). However, conventional process models are hybrid (Hoare, 1985), which do not distinguish the concepts of fundamental metaprocesses and process operations between them. The CSP notation models a major part of elementary software behaviors that may be used in system specification and description. However, it lacks many useful processes that are perceived essential in system modeling, such as addressing, memory manipulation, timing, and system dispatch. CSP models all input and output (I/O) as abstract channel operations that are not expressive enough to denote complex system interactions, particularly for those of real-time systems.

A number of timed extensions and variations of process algebra have been proposed (Baeten & Bergstra, 1991; Boucher & Gerth, 1987; Cerone, 2000; Corsetti, Montanari, & Ratto, 1991; Dierks, 2000; Fecher, 2001; Gerber, Gunter, & Lee, 1992; Jeffrey, 1992; Klusener, 1992; Nicollin & Sifakis, 1991; Reed & Roscoe, 1986; Schneider, 1991; Vereijken, 1995). It is found that the existing work on process algebra and their timed variations can be extended and refined to a new form of denotational mathematics, RTPA, based on a set of algebraic process operations and laws (Wang, 2002b, 2003b, 2006a, 2006c, 2007d, 2008a). RTPA can be used to formally and precisely describe and specify architectures and behaviors of software systems on the basis of algebraic process notations and rules.

Definition 2. A process P in RTPA is a composed component of n metastatements s_i and s_j , $1 \leq i < n$, $j = i + 1$, according to certain composing relations r_{ij} i.e.:

$$P = (\dots((s_1) r_{12} s_2) r_{23} s_3) \dots r_{n-1,n} s_n) \quad (1)$$

where $r_{ij} \in \mathfrak{R}$, which is a set of relational process operators of RTPA that will be formally defined in Lemma 2.

Definition 2 indicates that the mathematical model of a process is a cumulative relational structure among computing operations. The simplest process is a single computational statement. Further discussion will be provided in the section on the unified mathematical model of programs.

Definition 3. RTPA is a denotational mathematical structure for algebraically denoting and manipulating system behavioral processes and their attributes by a triple, i.e.:

$$RTPA \triangleq (\mathfrak{T}, \mathfrak{P}, \mathfrak{R}) \quad (2)$$

where \mathfrak{T} is a set of 17 primitive types for modeling system architectures and data objects, \mathfrak{P} is a set of 17 metaprocesses for modeling fundamental system behaviors, and \mathfrak{R} is a set of 17 relational process operations for constructing complex system behaviors.

RTPA provides a coherent notation system and a formal engineering methodology for modeling both software and intelligent systems. RTPA can be used to describe both *logical* and *physical* models of systems, where logical views of the architecture of a software system and its operational platform can be described using the same set of notations. When the system architecture is formally modeled, the static and dynamic behaviors that perform on the system architectural model can be specified by a three-level refinement scheme at the system, class, and object levels in a top-down approach.

This article presents the RTPA structure and methodology for modeling software and intelligent system behaviors. The type system, process notations, process relations, and process composing rules of RTPA are described. The type system and formal type rules of RTPA are formally described, and their usage in data object and system architectural modeling are demonstrated. The fundamental processes and their denotational functionality in modeling both human and system behaviors are presented. A set of algebraic process operations is introduced for constructing complex processes and systems. Then, applications of RTPA in computing and intelligent system modeling and manipulation are explored with case studies. A unified mathematic model of software and programs is derived based on the RTPA theories. The system specification and refinement methodology of RTPA and case studies on real-world problems are provided, which demonstrate the descriptive power of RTAP as a precise and neat algebraic system for software engineering.

THE TYPE SYSTEM OF RTPA

Computational operations of systems can be classified into the categories of *data object*, *behavior*, and *resource* modeling and manipulations. Based on this view, programs are perceived as the coordination of data objects and behaviors in computing. Data object modeling is a process to creatively extract and abstractly represent a real-world problem by computing objects based on the constraints of given computing resources. Using types to model the natural world can be traced back to the mathematical thought of Bertrand Russell (Russell, 1961; Schilpp, 1946) and Godel (van Heijenoort, 1997). A type is a category of variables that share a common property, such as kinds of data, domain, and allowable operations. Types are an important logical property shared by data objects in programming. Although data in their most primitive form is a string of bits, types are found expressively convenient for data representation at the logical level in computing and software engineering.

The Type System for Data Objects Modeling in RTPA

A type is a set in which all member data objects share a common logical property or attribute. The maximum range of values that a variable can assume is a type, and a type is associated with a set of predefined or allowable operations. A type can be classified as *primitive* and *derived* (complex). The former is the most elemental of types that cannot further be divided into simpler ones; the latter is a compound form of multiple primitive types based on given type rules. Most primitive types are provided by programming languages while most user-defined types are derived ones.

Definition 4. *A type system is a set of predefined templates and manipulation rules for modeling data objects and system architectures.*

RTPA types, syntaxes, and properties are given in Table 1, where the 17 primitive types in computing and human cognitive process modeling have been elicited (Cardelli & Wegner, 1985; Martin-Lof, 1975; Mitchell, 1990; Wang, 2002a, 2002b, 2003b, 2006a, 2007d, 2007e). In Table 1, the first 11 primitive types are for mathematical and logical manipulation of data objects, and the remaining 6 are for system architectural modeling.

Lemma 1. *The primary types of computational objects state that the RTPA type system \mathcal{T} encompasses 17 primitive types elicited from fundamental computing needs, i.e.:*

$$\mathcal{T} \hat{=} \{ \mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST}, \\ @e\mathbf{S}, @t\mathbf{TM}, @int\odot, \odot s\mathbf{BL} \} \quad (3)$$

RTPA adopts the *type suffix convention* in which every variable x declared in a type \mathbf{T} , $x : \mathbf{T}$, by a bold type label attached to the variable in all invocations in the form $x\mathbf{T}$, where \mathbf{T} is any valid primitive as defined in Table 1 or a derived type based on the table. The advance of the type suffix convention is the improvement

of readability. Using the type suffixes, system analysts may easily identify if all variables in a statement or expression are equivalent or compatible without referring to earlier declarations, which may be scattered in a system model across hundreds of pages in a large-scale software. The type suffix convention also greatly simplifies type checking requirements during parsing the RTPA specifications by machines (Tan, Wang, & Noglah, 2004, 2006).

An essence of type theory is that types can be classified into the domains of *mathematical* (logical) D_m , *language defined* D_p , and *user defined* D_u , as shown in Table 1 (2007d).

Theorem 1. *The domain constraints of data objects states that the following relationship between the domains of any data object in computing is always held, i.e.:*

$$D_u \subseteq D_l \subseteq D_m \quad (4)$$

It is noteworthy that although a generic computing behavior is constrained by D_m , an executable program is constrained by D_p , and, at most time, it is further restricted by the user defined domain D_u , where $D_u \subseteq D_l$. According to Theorem 1, the following corollary can be derived.

Corollary 1. *The precedence of domain determinations and type inferences in computing and software engineering is always as follows:*

$$D_u \Rightarrow D_l \Rightarrow D_m \quad (5)$$

Advanced Types of RTPA

The most common and powerful derived type in computing is a *record*, also known as a *construct*, because its flexibility to accommodate different data fields in different primary or composed types. System architectures can be modeled on the basis of structured records. There are also a number of special advanced types introduced in RTPA, such as the *system* type, *dynamic run-time* type, and *event*, *interrupt*, and *status* types (Wang, 2002b, 2007d).

Definition 5. *The run-time type **RT** is a non-deterministic type at compile-time that can be dynamically bound during run-time with one of the predefined primitive types.*

The run-time type **RT** provides programmers a powerful means to express and handle highly flexible and nondeterministic computing objects in data modeling. Some languages such as Java and IDL (OMG, 2002) label the dynamic type **RT** as the *anytype*, for which a specific type may be bound until run time.

Definition 6. *An event is an advanced type in computing that captures the occurring of a predefined external or internal change of status, such as an action of users, an external change of environment, and an internal change of the value of a specific variable.*

The event types of RTPA can be classified into those of *operation* ($@e\mathbf{S}$), *time* ($@t\mathbf{TM}$), and *interrupt* ($@int\odot$), as shown in Table 1, where $@$ is the *event prefix*, and \mathbf{S} , \mathbf{TM} , and \odot the corresponding type suffixes, respectively.

A special set of complex types known as the system type **ST** is widely used for modeling system architectures in RTPA, particularly real-time, embedded, and distributed systems architectures. All the system types are non-trivial data objects in computing, rather than simple data or logical objects, which play a very important role in the whole lifecycle of complex system development including design, modeling, specification, refinement, comprehension, implementation, and maintenance of such systems.

Definition 7. *A system type **ST** is a system architectural type that models the architectural components of the system and their relations.*

A generic **ST** type is the *Component Logical Model* (CLM), as defined in RTPA (Wang, 2002b, 2007d). CLMs are powerful modeling means in system architectural modeling.

Table 1. RTPA primitive types and their domains

No.	Type	Syntax	D _m	D _i	Equivalence
1	Natural number	N	[0, +∞]	[0, 65535]	Arithmetic, mathematical, assignment
2	Integer	Z	[-∞, +∞]	[-32768, +32767]	
3	Real	R	[-∞, +∞]	[-2147483648, 2147483647]	
4	String	S	[0, +∞]	[0, 255]	String and character operations
5	Boolean	BL	[T, F]	[T, F]	Logical, assignment
6	Byte	B	[0, 256]	[0, 256]	Arithmetic, assignment, addressing
7	Hexadecimal	H	[0, +∞]	[0, max]	
8	Pointer	P	[0, +∞]	[0, max]	
9	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]	Timing, duration, arithmetic (A generic abbreviation: TI = {TI, D, DT})
10	Date	D = yy:MM:dd	yy : [0, 99] MM : [1, 12] dd : [1, 31]	yy : [0, 99] MM : [1, 12] dd : [1, 31]	
11	Date/Time	DT = 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]	
12	Run-time determinable type	RT	–	–	Operations suitable at run-time
13	System architectural type	ST	–	–	Assignment (field reference by ‘.’)
14	Random event	@eS	[0, +∞]	[0, 255]	String operations
15	Time event	@tTM	[0ms, 9999 yyy]	[0ms, 9999 yyy]	Logical
16	Interrupt event	@int⊙	[0, 1023]	[0, 1023]	Logical
17	Status	⊙sBL	[T, F]	[T, F]	Logical

Definition 8. CLMs are a record-structured abstract model of a system architectural component that represents a hardware interface, an internal logical model, and/or a common control structure of a system.

CLMs can be used for unifying user defined complex types in system modeling. A formal treatment of CLMs will be provided in Definition 12.

It is recognized that any form of intelligence is memory based (Wang, 2002a, 2003a, 2006b, 2007c, 2007d, 2007e). All data objects,

no matter language generated or user created, should be implemented as physical data objects and be bound to specific memory locations. Therefore, memory models play an important role in system modeling.

Definition 9. *The generic system memory model, MEMST, can be described as a special system type ST with a finite linear space, i.e.:*

$$\text{MEMST} \triangleq [\text{addr}_1\mathbf{H} \dots \text{addr}_2\mathbf{H}]\mathbf{RT} \quad (6)$$

where $\text{addr}_1\mathbf{H}$ and $\text{addr}_2\mathbf{H}$ are the start and end addresses of the memory space, and \mathbf{RT} is the type of each of the memory elements, which is usually in Byte \mathbf{B} in computing.

Another special system type is the I/O port type for modeling hardware architectures and their interfaces.

Definition 10. *The generic system I/O port model, PORTST, can be described as a system type ST with a finite linear space, i.e.:*

$$\text{PORTST} \triangleq [\text{ptr}_1\mathbf{H} \dots \text{ptr}_2\mathbf{H}]\mathbf{RT} \quad (7)$$

where $\text{ptr}_1\mathbf{H}$ and $\text{ptr}_2\mathbf{H}$ are the start and end addresses of the port space, and \mathbf{RT} is the type of each of the port I/O interfaces, which is usually in Byte \mathbf{B} in computing.

Formal Type Rules of RTPA

A type system specifies the data objects modeling and composing rules of a programming language as that of a grammar system which specifies the program behavior modeling and composing rules of the language. The basic *properties* of type systems are decidable, transparent, and enforceable (Cardelli & Wegner, 1985; Martin-Lof, 1975; Mitchell, 1990). Type systems should be *decidable* by a type checking system, which ensures that types of variables are both well-declared and referred. Type systems should be *transparent* that helps for diagnoses

of reasons for inconsistency between variables or variables and their declarations. Type systems should be *enforceable* in order to check type inconsistency as much as possible.

A *formal type system* is a collection of all type rules for a given programming language or formal notation system. A type rule is a mathematical relation and the constraints on a given type. Type rules are defined on the basis of a type environment.

Definition 11. *The type environment Θ_t of RTPA is a collection of all primitive types in the formal notation system, i.e.:*

$$\begin{aligned} \Theta_t &\triangleq \mathfrak{T} \\ &= \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST}, \\ &\quad @e\mathbf{S}, @t\mathbf{TM}, @int\mathbf{C}, @s\mathbf{BL}\} \end{aligned} \quad (8)$$

where \mathfrak{T} is the set of primary types as defined in Table 1.

Complex and derived types of RTPA can be described by composing type rules based on those of the primitive types.

As given in Definition 8, a CLM is a generic system architectural type for modeling and manipulating data objects and system architectures (Wang, 2002b, 2007d).

Definition 12. *The type rule of a CLM type, CLM, is a complex system type ST in RTPA derived from Θ_p , i.e.:*

$$\frac{\Theta_t \vdash \mathbf{ST}}{\Theta_t \vdash \text{CLM} : \mathbf{ST}} \quad (9)$$

The declaration of a variable, *ClmID*, with a given CLM type can be denoted by using the following type rule as given in Equation 10, where the ClmIDST is defined by the string type label ClmIDS with an n-field structure, each of them specifies a metavariable ID_i in type \mathbf{T}_i , and its constraints denoted by $\text{Constraint}(\text{ID}_i\mathbf{T}_i)$, which are a set of expressions.

Box 1.

$$\frac{\Theta_t \vdash \mathbf{ST}, \Theta_t \vdash \mathbf{T}, \Theta_t \vdash \mathit{CImID}:\mathbf{ST}, \Theta_t \vdash \mathit{ID}:\mathbf{T}}{\Theta_t \vdash \mathit{CImID}\mathbf{ST} \triangleq \mathit{CImIDS} :: \{\mathop{\bigvee}\limits_{i=1}^n \langle \mathit{ID}_i:\mathbf{T}_i \mid \mathit{Constraint}(\mathit{ID}_i:\mathbf{T}_i) \rangle\}} \quad (10)$$

A *process* in RTPA is a basic behavioral unit for modeling software system operations onto the data objects. A process can be a metaprocess or a complex process composed with multiple metaprocesses by relational process operators. Because processes are so frequently used in system modeling, a derived type in RTPA known as the process type can be introduced as a special system type.

Definition 13. *The type rule of a process type, PROC, is a complex system type ST in RTPA derived from Θ_p , i.e.:*

$$\frac{\Theta_t \vdash \mathbf{ST}}{\Theta_t \vdash \mathit{PROC} : \mathbf{ST}} \quad (11)$$

The declaration of a variable, *ProcID*, with the *PROC* type can be denoted by using the following type rule as given in Equation 11, where the *ProcIDST* is defined by the string type label *ProcIDS* with a set of n inputs and a set of m outputs in a specific type, as well as a set of q I/O constructs or CLMs in a specific **ST** type.

METAPROCESSES OF RTPA

On the basis of the process metaphor, this section elicits the most general and fundamental system behaviors in computing and intelligent systems. Computational operations in conventional process algebra, such as CSP (Hoare, 1985), Timed-CSP (Boucher & Gerth, 1987; Reed & Roscoe, 1986; Schneider, 1991), and other proposals are treated as a set of processes at the same level. This approach results in an exhaustive listing of processes. Whenever a new operation is identified or required in computing, the existing process system must be extended.

RTPA adopts the foundationalism in order to find the most primitive computational processes known as the *metaprocesses*. In this approach, complex processes are treated as derived processes from these metaprocesses, based on a set of algebraic process composition rules known as the *process relations*.

Definition 14. *A metaprocess in RTPA is a primitive computational operation that cannot be broken down to further individual actions or behaviors.*

Box 2.

$$\frac{\Theta_t \vdash \mathit{ProcID}:\mathbf{ST}}{\Theta_t \vdash \mathit{ProcID}\mathbf{ST} \triangleq \mathit{ProcIDS} \quad (\mathbf{I}::\langle \mathop{\bigvee}\limits_{i=1}^n \mathit{ID}_i:\mathbf{T}_i \rangle; \mathbf{O}::\langle \mathop{\bigvee}\limits_{j=1}^m \mathit{ID}_j:\mathbf{T}_j \rangle; \mathbf{CLM}::\langle \mathop{\bigvee}\limits_{k=1}^q \mathit{CImID}_k:\mathbf{ST}_k \rangle)} \quad (12)$$

A metaprocess is an elementary process that serves as a basic building block for modeling software behaviors. *Complex processes* can be composed from metaprocesses using *process relations*. In RTPA, a set of 17 metaprocesses has been elicited as shown in Table 2, from essential and primary computational operations commonly identified in existing formal methods and modern programming languages (Aho, Sethi, & Ullman, 1985; Higman, 1977; Hoare et al., 1987; Loudon, 1993; Wilson & Clark, 1988; Woodcock & Davies, 1996). Mathematical notations and syntaxes of the metaprocesses are formally described in Table 2, while formal semantics of the metaprocesses of RTPA may be found in Wang (2006c, 2008a).

Lemma 2. *The RTPA metaprocess system \mathfrak{P} encompasses 17 fundamental computational operations elicited from the most basic computing needs, i.e.:*

$$\mathfrak{P} = \{ :=, \blacklozenge, \Rightarrow, \Leftarrow, \neq, \succ, \prec, |\succ, |\prec, @, \triangle, \uparrow, \downarrow, !, \otimes, \boxtimes, \$ \} \tag{13}$$

As shown in Lemma 2 and Table 2, each metaprocess is a basic operation on one or more operands such as variables, memory elements, or I/O ports. Structures of the operands and their allowable operations are constrained by their types, as described in previous sections.

It is noteworthy that not all generally important and fundamental computational operations, as shown in Table 2, had been explicitly identified in conventional formal methods (e.g., the evaluation, addressing, memory allocation/release, timing/duration, and the system processes). However, all these are found necessary and essential in modeling system architectures and behaviors (Wang, 2007d).

ALGEBRAIC PROCESS OPERATIONS IN RTPA

The metaprocesses of RTPA developed in the preceding section identified a set of fundamental elements for modeling the most basic behaviors of computing and intelligent systems. It is

interesting to realize that there is only a small set of 17 metaprocesses in system modeling. However, via the combination of a number of the metaprocesses by certain algebraic operations, any architecture and behavior of real-time or nonreal-time systems can be sufficiently described (Wang, 2002b, 2003b, 2006a, 2007d).

Definition 15. *A process relation in RTPA is an algebraic operation and a compositional rule between two or more metaprocesses in order to construct a complex process.*

A set of 17 fundamental process relational operations has been elicited from fundamental algebraic and relational operations in computing in order to build and compose complex processes in the context of real-time software systems. Syntaxes and usages of the 17 RTPA process relations are formally described in Table 3. Deductive semantics of these process relations may be found in Wang (2006c, 2008a).

Lemma 3. *The software composing rules state that the RTPA process relation system \mathfrak{R} encompasses 17 fundamental algebraic and relational operations elicited from basic computing needs, i.e.:*

$$\mathfrak{R} = \{ \rightarrow, \curvearrowright, |, | \dots | \dots, R^*, R^+, R^l, \odot, \mapsto, ||, \mathbb{H}, |||, \gg, \ll, \downarrow_p, \downarrow_e, \downarrow_i \} \tag{14}$$

As modeled in Lemma 3 and Table 3, the first seven process relations—i.e., *sequential* (#1), *jump* (#2), *branch* (#3), *switch* (#4), and *iterations* (#5 through #7)—may be identified as the Basic Control Structures (BCSs) of system behaviors (Aho et al., 1985; Hoare et al., 1987; Wilson & Clark, 1988). To represent the modern programming structural concepts, CSP (Hoare, 1985) identified the following seven additional process relations such as *recursion* (#8), *function call* (#9), *parallel* (#10), *concurrency* (#11), *interleave* (#12), *pipeline* (#13), and *interrupt* (#14). However, these process relations or operations were treated as the same

Table 2. RTPA Metaprocesses

No.	Meta Process	Notation	Syntax
1	Assignment	$:=$	$y^T := x^T$
2	Evaluation	\blacklozenge	$\blacklozenge_{\tau} \text{exp}^T \rightarrow T$
3	Addressing	\Rightarrow	$id^T \Rightarrow \text{MEM}[ptr^P] T$
4	Memory allocation	\Leftarrow	$id^T \Leftarrow \text{MEM}[ptr^P] T$
5	Memory release	\Leftarrow	$id^T \Leftarrow \text{MEM}[\perp] T$
6	Read	\triangleright	$\text{MEM}[ptr^P] T \triangleright x^T$
7	Write	\triangleleft	$x^T \triangleleft \text{MEM}[ptr^P] T$
8	Input	$ \triangleright$	$\text{PORT}[ptr^P] T \triangleright x^T$
9	Output	$ \triangleleft$	$x^T \triangleleft \text{PORT}[ptr^P] T$
10	Timing	$\textcircled{!}$	$\textcircled{!}_{t} \text{TM} \textcircled{!}_{\tau} \text{TM}$ $\text{TM} = \text{yy:MM:dd}$ $ \text{hh:mm:ss:ms}$ $ \text{yy:MM:dd:hh:mm:ss:ms}$
11	Duration	\triangleq	$\textcircled{!}_{t} \text{TM} \triangleq \textcircled{!}_{\tau} \text{TM} + \Delta n \text{TM}$
12	Increase	\uparrow	$\uparrow(n^T)$
13	Decrease	\downarrow	$\downarrow(n^T)$
14	Exception detection	$!$	$!(@e\mathbf{S})$
15	Skip	\otimes	\otimes
16	Stop	\boxtimes	\boxtimes
17	System	\S	$\S(\text{SysID}\mathbf{ST})$

of the metaprocesses in existing formal methods. That is, the conventional notation systems are not an algebraic production system rather than an exhaustive instruction system, which do not distinguish the basic computational operations and their composing rules.

RTPA (Wang, 2002b) extends the BCS's and process relations to *time-driven dispatch* (#15), *event-driven dispatch* (#16), and *interrupt-driven dispatch* (#17) in order to model the top-level system behaviors, particularly those of real-time systems. The 17 process relations (BCSs) are regarded as the founda-

tion of programming and system behavioral design, because any complex process can be combinatory implemented by the algebraic process composing operations onto the set of the 17 metaprocesses. In Table 3, the big-R used in process relations #5 through #8 is a special calculus recently created for denoting iterative and recursive behaviors of software systems (Wang, 2008b).

Theorem 2. *The express power of algebraic modeling states that the total number of the possible computational behaviors (operations)*

Table 3. RTPA process relations and algebraic operations

No.	Process Relation	Notation	Syntax
1	Sequence	\rightarrow	$P \rightarrow Q$
2	Jump	\curvearrowright	$P \curvearrowright Q$
3	Branch	$ $	$\blacklozenge \text{exp} \mathbf{BL} = \mathbf{T} \rightarrow P$ $ \blacklozenge \sim \rightarrow Q$
4	Switch	$\begin{matrix} \\ \dots \\ \end{matrix}$	$\blacklozenge \text{exp} \mathbf{T} =$ $i \rightarrow P_i$ $ \sim \rightarrow \emptyset$ where $\mathbf{T} \in \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{S}\}$
5	While-loop	R^*	$\overset{\mathbf{f}}{R} P$ $\text{exp} \mathbf{BL} = \mathbf{T}$
6	Repeat-loop	R^+	$P \rightarrow \overset{\mathbf{f}}{R} P$ $\text{exp} \mathbf{BL} = \mathbf{T}$
7	For-loop	R^i	$\overset{n\mathbf{M}}{R} P(i\mathbf{M})$ $i\mathbf{M} = 1$
8	Recursion	\circ	$\overset{0}{R} P^{\mathbf{M}} \circ P^{\mathbf{M}-1}$ $i\mathbf{M} = n\mathbf{M}$
9	Function call	\mapsto	$P \mapsto F$
10	Parallel	\parallel	$P \parallel Q$
11	Concurrency	\P	$P \P Q$
12	Interleave	$\parallel\parallel$	$P \parallel\parallel Q$
13	Pipeline	\gg	$P \gg Q$
14	Interrupt	$\not\Leftarrow$	$P \not\Leftarrow Q$
15	Time-driven dispatch	\hookrightarrow_t	$@_t \mathbf{TM} \hookrightarrow_t P_i$
16	Event-driven dispatch	\hookrightarrow_e	$@_e \mathbf{S} \hookrightarrow_e P_i$
17	Interrupt-driven dispatch	\hookrightarrow_i	$@_{int_j} \odot \hookrightarrow_i P_j$

\mathcal{N} is a set of combinations between two arbitrary metaprocesses $\mathbb{P}_1, \mathbb{P}_2 \in \mathfrak{P}$ composed by each of the process relations $\mathbb{R} \in \mathfrak{R}$ in RTPA, i.e.:

$$\begin{aligned} \mathcal{N} &= \#\mathfrak{R} \cdot \overset{\# \mathfrak{P}}{C}_{\# \mathfrak{P}}^2 \\ &= 17 \cdot \frac{17!}{2!(17-2)!} \\ &= 17 \cdot 136 \\ &= 2,312 \end{aligned}$$

(15)

Theorem 2 demonstrates the expressive power of the algebraic structure of RTPA towards computational behavior modeling and programming. It is noteworthy that an ordinary high-level programming language may introduce about 150 to 300 individual instructions. However, the expressive power of RTPA is much higher than those of programming languages and other exhaustive formal notation systems, although it just adopts a small set of 17 metaprocesses and 17 process relations.

MANIPULATION OF COMPUTATIONAL BEHAVIORS BY RTPA

As presented in previous sections, RTPA provides a neat and powerful denotational mathematics structure, which is capable to be used as a generic notation system for system architecture and behavior modeling and specifications. This section describes the usage and methodology of RTPA for software system modeling and refinement. Its applications in formally modeling intelligent systems and human cognitive processes will be presented in the next section.

The Universal Mathematical Model of Programs Based on RTPA

Program modeling is on coordination of computational behaviors with given data objects. On the basis of RTPA, a generic program model can be described by a formal treatment of statements, processes, and complex processes from the bottom up in the program hierarchy.

Definition 16. A process P is the basic unit of an applied computational behavior that is composed by a set of statements $s_i, 1 \leq i \leq n-1$, with left-associated cumulative relations, i.e.:

$$P = \underset{i=1}{\overset{n-1}{R}} (s_i r_{ij} s_j), j = i+1$$

$$= (\dots(((s_1) r_{12} s_2) r_{23} s_3) \dots r_{n-1,n} s_n)$$

(16)

where $s_i \in P$ and $r_{ij} \in R$.

With the formal process model as defined above, the universal mathematical model of programs can be derived below.

Definition 17. A program \wp is a composition of a finite set of m processes according to the time-, event-, and interrupt-based process dispatching rules of RTPA, i.e.:

$$\wp = \underset{k=1}{\overset{m}{R}} (@e_k \hookrightarrow P_k)$$

(17)

Equations 16 and 17 indicate that a program is an *embedded relational algebraic* entity, where a statement s in a program is an instantiation of a metainstruction of a programming language that executes a basic unit of coherent function and leads to a predictable behavior.

Theorem 3. The Embedded Relational Model (ERM) states that a software system or a program \wp is a set of complex embedded relational processes, in which all previous processes of a given process form the context of the current process, i.e.:

$$\wp = \underset{k=1}{\overset{m}{R}} (@e_k \hookrightarrow P_k)$$

$$= \underset{k=1}{\overset{m}{R}} [@e_k \hookrightarrow \underset{i=1}{\overset{n-1}{R}} (p_i(k) \text{ † }_{ij} k) p_j(k)], j = i+1$$

(18)

The ERM model presented in Theorem 3 reveals that a program is a finite and nonempty set of embedded binary relations between a current statement and *all previous ones* that formed the *semantic context* or environment of computing. Theorem 3 provides a unified software model, which is a formalization of the well accepted but informal process metaphor for software systems in computing.

ADT Modeling and Specification in RTPA

Abstract data types (ADTs) are perfect software architectures at component level that will be

used to explain the modeling methodology of computing architectures and behaviors in RTPA. An ADT is a logical model of complex and/or user defined data type with a set of predefined operations on it. A queue as a typical ADT is presented in this subsection to demonstrate how the RTPA notation system is used to model and specify the architecture, static behaviors, and dynamic behaviors of software systems.

There are a number of approaches to the specification of ADTs, which can be classified into the logical and algebraic approaches. The logic approach is good at specifying the static properties of ADT operations, usually in forms of preconditions and post-conditions of operations of ADTs; while the algebraic approach is good at describing dynamic and run-time behaviors of ADTs.

A queue in RTPA is modeled as an algebraic entity, which has predefined operations on a set of data objects in given types. Unlike the conventional approaches to ADT specifications that treat ADTs as static data types, ADTs in RTPA are treated as dynamic finite state machines, which have both architectures and behaviors, in order to model both structural and operational components in system design.

Architectural Modeling in RTPA

At the top level, an RTPA specification of the queue, *QueueST*, has three parallel facets, which are architecture, static behaviors, and dynamic behaviors, as given below.

$$\begin{aligned} \text{Queue}^{\text{ST}} &\triangleq \text{Queue}^{\text{ST}}.\text{Architecture} \\ &\parallel \text{Queue}^{\text{ST}}.\text{StaticBehaviors} \\ &\parallel \text{Queue}^{\text{ST}}.\text{DynamicBehaviors} \end{aligned} \quad (19)$$

Then, *QueueST* can be further refined by detailed specifications according to the RTPA methodology (Wang, 2007d).

The key modeling methodology for system architectures in RTPA is CLMs, as described in the subsection of advanced types. Any system architecture and data object, including their control structures, internal logic model, and

hardware interface model can be rigorously described and specified by using a set of CLMs.

Example 1. *The architecture of QueueST modeled in RTPA is given in Figure 1, where both the architectural CLM and an access model are provided for the Queue.*

In Figure 1, the *access model* of *QueueST* is a logic model for supporting external invocation of the *QueueST* in operations, such as enqueue and service. The other parts of the model are designed for internal manipulation of the *QueueST*, such as creation, memory allocation, and memory release.

Static Behavior Modeling in RTPA

System static behaviors in RTPA are valid operations of systems that can be determined at compile-time, which describe the configuration of processes of the component and their relations. The set of static behaviors of *QueueST* can be modeled by a set of behavioral processes encompassing *create*, *release*, *enqueue*, *serve*, *clear*, *empty test*, and *full test*.

Example 2. *The detailed specification of one of the Queue's static behaviors, QueueST.serve, is given in Figure 2.*

Contrasting the static behavior model of *QueueST.serve* in RTPA as shown in Figure 2 and in conventional propositional logic (Stubbs & Webre, 1985), the advances of RTAP method and notations may be well demonstrated. Among them, the most important advantage is that an RTPA model may be seamlessly refined into code in a programming language in the succeeding phase of software engineering.

Dynamic Behavior Modeling in RTPA

System dynamic behaviors in RTPA are process relations determined at run time. According to the RTPA system modeling and refinement scheme, models of system static behaviors are process models of the system. To put the component processes into a live and interacting system, the dynamic behaviors of the system

Figure 1. The architectural model of the Queue in RTPA

```

Queue ST.Architecture  $\triangleq$  CLM : ST
    || AccessModel : ST
    || Events : S
    || Status : BL

Queue ST.Architecture.CLM  $\triangleq$  QueueIDS :
    (<Size : N | SizeN  $\geq$  0>,
     <Element : RT>,
     <CurrentPos : P | 0  $\leq$  CurrentPosP  $\leq$  SizeN-1>
    )

Queue ST.Architecture.AccessModel  $\triangleq$  QueueIDS(CurrentPosP)RT
    
```

Figure 2. The static behavioral model of the Queue in RTPA

```

Queue ST.Serve (<I :: QueueInstS>,
               <O ::  $\textcircled{S}$ QueueID.ServedBL, ElementRT>)  $\triangleq$ 
{
    QueueIDS := QueueInstS
     $\rightarrow$  (  $\textcircled{\blacklozenge}$   $\textcircled{S}$ QueueExistBL = T  $\wedge$  CurrentPosP > 0
         $\rightarrow$  (QueueID(1))RT  $\triangleright$  Element RT
             $\rightarrow$  QueueID(i)RT  $\triangleright$  QueueID(i-1)RT
             $\rightarrow$   $\downarrow$ (QueueID.CurrentPosP)
             $\rightarrow$   $\textcircled{S}$ QueueID.ServedBL := T
        |  $\textcircled{\blacklozenge}$   $\sim$ 
             $\rightarrow$   $\textcircled{S}$ QueueID.ServedBL := F
             $\rightarrow$  !(@'QueueIDExistBL = F  $\vee$  QueueEmptyBL = T)
    )
}
    
```

Figure 3. The dynamic behavioral model of the Queue in RTPA

```

Queue ST.DynamicBehaviors  $\triangleq$  {  $\S$   $\rightarrow$ 
    ( @CreateQueueS  $\hookrightarrow$  Queue.Create (<I :: QueueInstS, ElementInstRT, SizeInstN>;
        <O ::  $\textcircled{S}$ QueueID.AllocatedBL,  $\textcircled{S}$ QueueID.ExistBL>)
    | @ReleaseQueueS  $\hookrightarrow$  Queue.Release (<I :: QueueInstS>; <O ::  $\textcircled{S}$ QueueID.ReleasedBL>)
    | @EnqueueS  $\hookrightarrow$  Queue.Enqueue (<I :: QueueInstS, ElementInstRT>; <O ::  $\textcircled{S}$ QueueID.EnqueueedBL>)
    | @ServeS  $\hookrightarrow$  Queue.Serve (<I :: QueueInstS>; <O ::  $\textcircled{S}$ QueueID.ServedBL, ElementRT>)
    | @ClearS  $\hookrightarrow$  Queue.Clear (<I :: QueueInstS>; <O ::  $\textcircled{S}$ QueueID.ClearedBL>)
    | @QueueEmptyS  $\hookrightarrow$  Queue.EmptyTest (<I :: QueueInstS>; <O ::  $\textcircled{S}$ QueueID.FullBL>)
    | @QueueFullS  $\hookrightarrow$  Queue.FullTest (<I :: QueueInstS>; <O ::  $\textcircled{S}$ QueueID.FullBL>)
    )  $\rightarrow$   $\S$ 
}
    
```

in terms of process deployment and dispatch are yet to be specified.

Example 3. *The dynamic behaviors of QueueST are specified in RTPA as shown in Figure 3, where the process dispatch mechanisms of the Queue specifies detailed dynamic process relations at run time by a set of event-driven relations.*

Figures 1 through 3 model an ADT, *QueueST*, in a coherent system from three facets. With the RTPA specification and refinement methodology and the expressive power of the RTPA notation system, the features of ADTs as both static data types and dynamic finite machines can be specified formally and precisely.

MANIPULATION OF INTELLIGENT BEHAVIORS BY RTPA

RTPA may be used not only for modeling and description of computing behaviors but also for modeling and denoting the cognitive processes of the brain in cognitive informatics (Wang, 2002a, 2003a, 2003b, 2006a, 2006b, 2007a, 2007e, 2007f; Wang & Wang, 2006; Wang et al., 2006). A formal treatment of memorization as a cognitive process is presented in this section by a rigorous RTPA model based on the cognitive model of human memorization (Wang, 2007a).

The Cognitive Process of Memorization

Memorization as a cognitive process can be described by two phases: the *establishment* phase and the *reconstruction* phase. The former represents the target information in the form of an object-attribute-relation (OAR) model (Wang, 2007c) and creates a memory in long-term memory (LTM). The latter retrieves the memorized information and reconstructs it in the form of a concept in the short-term memory (STM). Therefore, memorization can be perceived as the transformation of informa-

tion and knowledge between STM and LTM, where the forward transformation from STM to LTM is for memory establishment, and the backward transformation from LTM to STM is for memory reconstruction.

Algorithm 1. *The cognitive process of memorization can be carried out by the following steps:*

- (0) Begin.
- (1) Encoding: This step generates a representation of a given concept by transferring it into a sub-OAR model;
- (2) Retention: This step updates the entire OAR in LTM with the sub-OAR for memorization by creating new synaptic connections between the sub-OAR and the entire OAR;
- (3) Rehearsal test: This step checks if the memorization result in LTM needs to be rehearsed. If yes, it continues to practice Steps (4) and (5); otherwise, it jumps to Step (7);
- (4) Retrieval: This step retrieves the memorized object in the form of sub-OAR by searching the entire OAR with clues of the initial concept;
- (5) Decoding: This step transfers the retrieved sub-OAR from LTM into a concept and represents it in STM;
- (6) Repetitive memory test: This step tests if the memorization process was succeeded or not by comparing the recovered concept with the original concept. If need, repetitive memorization will be called.
- (7) End.

It is noteworthy that the input of memorization is a structured concept formulated by learning or other cognitive processes (Wang, 2007d, 2007f; Wang et al., 2006).

Formal Description of the Memorization Process in RTPA

The cognitive process of memorization described in Algorithm 1 can be formally modeled using RTPA as given in Figure 4. According to

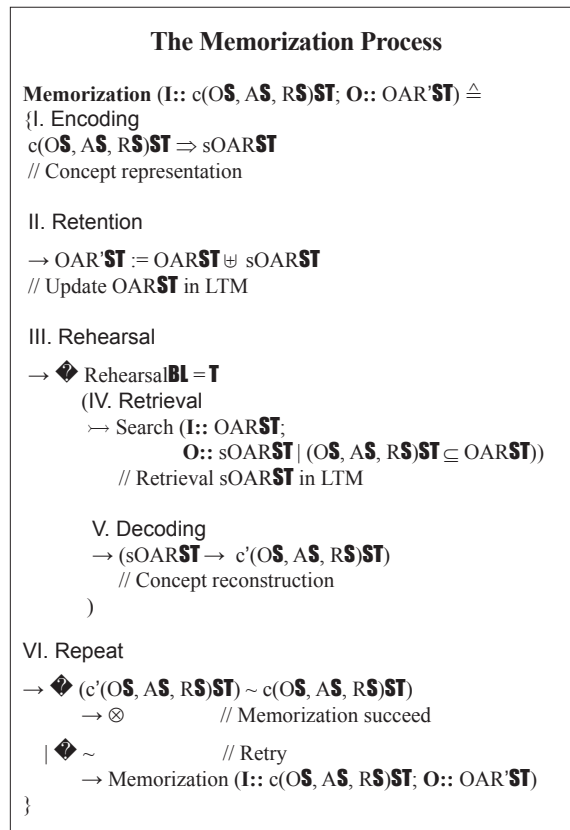
the LRMB model (Wang et al., 2006) and the OAR model (Wang, 2007c) of internal knowledge representation in the brain, the input of the memorization process is a structured concept $c(OS, AS, RS)ST$, which will be transformed to update the entire OAR model of knowledge in LTM in order to create a permanent memory. Therefore, the output of memorization is the updated $OAR'ST$ in LTM.

In the RTPA memorization process, as shown in Figure 4, the *encoding* subprocess is modeled as a function that maps the given concept cST into a sub-OAR, $sOARST$. The *retention* subprocess composes the $sOARST$ with the entire $OARST$ in LTM that maintains the whole knowledge of an individual. In order to check the memorization quality, rehearsals are usually needed. In a rehearsal, the *retrieval*

subprocess searches a related $sOARST$ in LTM by giving clues of previously memorized objects and attributes in cST . Then, the *decoding* subprocess transfers the $sOARST$ into a recovered concept $c'ST$. In the repetitive memory test subprocess, the reconstructed $c'ST$ will be compared with the original input of cST in order to determine if further memorization is recursively needed.

According to the 24-hour law of memorization as stated in Wang and Wang (2006), the memorization process may be completed with a period at least 24 hours by several cycles of repetitions. Although almost all steps in the process shown in Figure 4 are conscious, the key step of *retention* is subconscious or nonintentionally controllable. Based on the LRMB model (Wang et al., 2006), the memorization

Figure 4. Formal description of the memorization process in RTPA



process is closely related to learning (Wang, 2007f). In other words, memorization is a back-end process of learning, which retains learning results in LTM and retrieves them when rehearsals or applications are needed. The retrieve process is search-based by contents or **sOARST** matching.

The cognitive process of memorization formally modeled in RTPA provides a rigorous description of one of the important and complicated mental processes of the brain. This case study explains the second usage of RTPA in intelligent system modeling and human behavioral manipulation. Further models and applications of RTPA in intelligent system modeling may be referred to (Wang, 2007d; Wang & Ngolah, 2002, 2003). The applications of RTPA in modeling cognitive processes of the brain and natural intelligence may be found in Wang (2003b, 2007a, 2007f).

CONCLUSION

RTPA has been developed as a denotational mathematical means, which can be used as algebra-based, expressive, easy-to-comprehend, and language-independent notation system, and a practical specification and refinement method for software and intelligent system modeling. RTPA is capable to support top-down software system design and implementation by algebraic modeling and seamless refinement methodologies. The RTPA methodology covers the entire system lifecycle from high-level design to code generation in a coherent algebraic notation system.

This article has demonstrated that RTPA is not only useful as a generic notation and methodology for computing and software system modeling but is also good at modeling human cognitive processes and intelligent systems. A number of case studies on large-scale software system modeling and specifications have been carried out, such as the Telephone Switching System (TSS) (Wang, 2003b), the Lift Dispatching System (LDS) (Wang & Ngolah, 2002), and the Automated Teller Machine (ATM) (Wang & Zhang, 2003). The application results have encouragingly demonstrated that RTPA is a

powerful and practical algebraic system for both academics and practitioners in software and intelligent system engineering.

A set of support tools for RTPA have been developed (Ngolah, Wang, & Tan, 2005b, 2006; Tan & Wang, 2006; Tan, Wang, & Ngolah, 2004a, 2004b, 2005, 2006), which encompasses the RTPA parser, type checker, and code generator in C++ and Java. The RTPA code generator enables system specifications in RTPA to be automatically translated into fully executable code. The RTPA tools will support system architects, analysts, and practitioners for developing consistent and correct specifications and architectural models of large-scale software and intelligent systems, and the automatic generation of code based on formal models and rigorous specifications in the denotational mathematical notations.

ACKNOWLEDGMENT

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

REFERENCES

- Aho, A. V., Sethi, R. & Ullman, J. D. (1985). *Compilers: Principles, techniques, and tools*. New York: Addison-Wesley.
- Baeten, J. C. M., Bergstra, J. A. (1991). Real time process algebra. *Formal Aspects of Computing*, 3, 142-188.
- Boucher, A., & Gerth, R. (1987). A timed model for extended communicating sequential processes. In *Proceedings of ICALP'87* (LNCS 267). Springer.
- Cardelli, L., & 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, pp. 57-67). Berlin, Germany: Springer.
- Corsetti, E., Montanari, A., & Ratto, E. (1991). Dealing with different time granularities in formal specifica-

- tions of real-time systems. *The Journal of Real-Time Systems*, 3(2), 191-215.
- Dierks, H. (2000). A process algebra for real-time programs (LNCS 1783, pp. 66/76). Berlin, Germany: Springer.
- Fecher, H. (2001). A real-time process algebra with open intervals and maximal progress, *Nordic Journal of Computing*, 8(3), 346-360.
- Gerber, R., Gunter, E. L., & Lee, I. (1992). Implementing a real-time process algebra In M. Archer, J. J. Joyce, K. N. Levitt, & P. J. Windley (Eds.), *Proceedings of the International Workshop on the Theorem Proving System and its Applications* (pp. 144-145). Los Alamitos, CA: IEEE Computer Society Press.
- Higman, B. (1977). *A comparative study of programming languages* (2nd ed.). MacDonal.
- Hoare, C. A. R. (1978). Communicating sequential processes. *Communications of the ACM*, 21(8), 666-677.
- Hoare, C. A. R. (1985). *Communicating sequential processes*. London: Prentice Hall International.
- Hoare, C. A. R., Hayes, I. J., He, J., Morgan, C. C., Roscoe, A. W., Sanders, J. W., et al. (1987). Laws of programming, *Communications of the ACM*, 30(8), 672-686.
- Jeffrey, A. (1992). Translating timed process algebra into prioritized process algebra. In J. Vytopil (Ed.), *Proceedings of the Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems* (LNCS 571, pp. 493-506). Nijmegen, The Netherlands: Springer-Verlag.
- Klusener, A. S. (1992). Abstraction in real time process algebra. In J. W. de Bakker, C. Huizing, W. P. de Roever, & G. Rozenberg (Eds.), *Proceedings of Real-Time: Theory in Practice* (LNCS, pp. 325-352). Berlin, Germany: Springer.
- Louden K. C. (1993). *Programming languages: Principles and practice*. Boston.: PWS-Kent.
- Martin-Lof, P. (1975). An intuitionistic theory of types: Predicative part. In H. Rose & J. C. Shepherdson (Eds.), *Logic Colloquium 1973*. North-Holland.
- Milner, R. (1980). *A calculus of communicating systems* (LNCS 92). Springer-Verlag.
- Milner, R. (1989). *Communication and concurrency*. Englewood Cliffs, NJ: Prentice Hall.
- Mitchell, J. C. (1990). Type systems for programming languages. In J. van Leeuwen (Ed.), *Handbook of theoretical computer science* (pp.365-458). North-Holland.
- Nicollin, X., & Sifakis, J. (1991). An overview and synthesis on timed process algebras. In *Proceedings of the Third International Computer Aided Verification Conference* (pp. 376-398).
- OMG. (2002, July). *IDL syntax and semantics*. 1-74.
- Reed, G. M., & Roscoe, A. W. (1986). A timed model for communicating sequential processes. In *Proceedings of ICALP'86* (LNCS 226). Berlin, Germany: Springer-Verlag.
- Russell, B. (1961). *Basic writings of Bertrand Russell*. London: George Allen & Unwin Ltd.
- Schneider, S. A. (1991). *An operational semantics for timed CSP* (Programming Research Group Tech. Rep. TR-1-91). Oxford University.
- Schilpp, P. A. (1946). The philosophy of Bertrand Russell. *American Mathematical Monthly*, 53(4), 7210.
- Stubbs, D. F., & Webre, W. R. (1985). *Data structures with abstract data types and Pascal*. Monterey, CA: Brooks/Cole.
- Tan, X., Wang, Y., & Ngolah, C. F. (2004). A novel type checker for software system specifications in RTPA. In *Proceedings of the 17th Canadian Conference on Electrical and Computer Engineering (CCECE'04)*. (pp. 1549-1552). Niagara Falls, Ontario, Canada: IEEE CS Press.
- Tan, X., Wang, Y., & Ngolah, C. F. (2006). Design and implementation of an automatic RTPA code generator. In *Proceedings of the 19th Canadian Conference on Electrical and Computer Engineering (CCECE'06)* (pp. 1605-1608). Ottawa, Ontario, Canada: IEEE CS Press.
- van Heijenoort, J. (1997). *From Frege to Godel, a source book in mathematical logic 1879-1931*. Cambridge, MA: Harvard University Press.
- Vereijken, J. J. (1995). A process algebra for hybrid systems. In A. Bouajjani & O. Maler (Eds.), *In Proceedings of the Second European Workshop on Real-Time and Hybrid Systems*. Grenoble: France.
- Wang, Y. (2002a). On cognitive informatics (Keynote speech). In *Proceedings of the First IEEE International*

- Conference on Cognitive Informatics (ICCI'02)* (pp. 34-42). Calgary, Canada: IEEE CS Press.
- Wang, Y. (2002b). The real-time process algebra (RTPA). *Annals of Software Engineering: An International Journal*, 14, 235-274.
- Wang, Y. (2003a). On cognitive informatics. *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy*, 4(3), 151-167.
- Wang, Y. (2003b). Using process algebra to describe human and software system behaviors. *Brain and Mind*, 4(2), 199-213.
- Wang, Y. (2006a). Cognitive informatics and contemporary mathematics for knowledge representation and manipulation (Invited plenary talk). In *Proceedings of the First International Conference on Rough Set and Knowledge Technology (RSKT'06)* (LNAI 4062, pp. 69-78). Chongqing, China: Springer.
- Wang, Y. (2006b). Cognitive informatics—Towards the future generation computers that think and feel (Keynote speech). In *Proceedings of the Fifth IEEE International Conference on Cognitive Informatics (ICCI'06)* (pp. 3-7). Beijing, China: IEEE CS Press.
- Wang, Y. (2006c). On the informatics laws and deductive semantics of software. *IEEE Transactions on Systems, Man, and Cybernetics (C)*, 36(2), 161-171.
- Wang, Y. (2007a). Formal description of the cognitive process of memorization. In *Proceedings of the Sixth International Conference on Cognitive Informatics (ICCI'07)* (pp. 284-293). Lake Tahoe, CA: IEEE CS Press.
- Wang, Y. (2007b). Keynote speech, on theoretical foundations of software engineering and denotational mathematics. In *Proceedings of the Fifth Asian Workshop on Foundations of Software* (pp. 99-102). Xiamen, China.
- Wang, Y. (2007c). The OAR Model of neural informatics for internal knowledge representation in the brain. *The International Journal of Cognitive Informatics and Natural Intelligence*, 1(3), 64-75.
- Wang, Y. (2007d). *Software engineering foundations: A software science perspective*. In *CRC series in software engineering: Vol. 2*. CRC Press.
- Wang, Y. (2007e). The theoretical framework of cognitive informatics. *The International Journal of Cognitive Informatics and Natural Intelligence*, 1(1), 1-27.
- Wang, Y. (2007f). The Theoretical framework and cognitive process of learning. In *Proceedings of the Sixth International Conference on Cognitive Informatics (ICCI'07)* (pp. 470-479). Lake Tahoe, CA: IEEE CS Press.
- Wang, Y. (2008a). Deductive semantics of RTPA. *The International Journal of Cognitive Informatics and Natural Intelligence*, 2(2), 95-121.
- Wang, Y. (2008b). On the big-R notation for describing iterative and recursive behaviors. *The International Journal of Cognitive Informatics and Natural Intelligence*, 2(1), 17-28.
- Wang, Y., & King, G. (2000). Software engineering processes: Principles and applications In *CRC Series in Software Engineering: Vol. I*. CRC Press.
- Wang, Y., & Noglah, C. F. (2002). Formal specification of a real-time lift dispatching system. In *Proceedings of the 2002 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE'02)* (pp.669-674). Winnipeg, Manitoba, Canada.
- Wang, Y., & Noglah, C. F. (2003). Formal description of real-time operating systems using RTPA. In *Proceedings of the 2003 Canadian Conference on Electrical and Computer Engineering (CCECE'03)* (pp.1247-1250). Montreal, Canada: IEEE CS Press
- Wang, Y., & Wang, Y. (2006). Cognitive informatics models of the brain. *IEEE Transactions on Systems, Man, and Cybernetics (C)*, 36(2),203-207.
- Wang, Y., Wang, Y., Patel, S., & Patel, D. (2006). A layered reference model of the brain (LRMB). *IEEE Transactions on Systems, Man, and Cybernetics (C)*, 36(2),124-133.
- Wang, Y., & Zhang, Y. (2003). Formal description of an ATM system by RTPA. In *Proceedings of the 16th Canadian Conference on Electrical and Computer Engineering (CCECE'03)*(pp.1255-1258). Montreal, Canada: IEEE CS Press.
- Wilson, L. B., & Clark, R. G. (1988). *Comparative programming language*. Wokingham, England: Addison-Wesley.
- Woodcock, J., & Davies, J. (1996). *Using Z: Specification, refinement, and proof*. London: Prentice Hall International.

Yingxu Wang is professor of cognitive informatics and software engineering, director of International Center for Cognitive Informatics (ICfCI), and director of Theoretical and Empirical Software Engineering Research Center (TESERC) at the University of Calgary. He received a PhD in software engineering from The Nottingham Trent University, UK, in 1997, and a BSc in electrical engineering from Shanghai Tiedao University in 1983. He was a visiting professor in the Computing Laboratory at Oxford University and Department of Computer Science at Stanford University during 1995 and 2008, respectively, and has been a full professor since 1994. He is the editor-in-chief of the International Journal of Cognitive Informatics and Natural Intelligence (IJCINI), and editor-in-chief of the CRC Book Series in Software Engineering. He has published over 300 journal and conference papers and 11 books in software engineering and cognitive informatics, and won dozens of research achievement, best paper, and teaching awards in the last 28 years, particularly the IBC 21st Century Award for Achievement "in recognition of outstanding contribution in the field of Cognitive Informatics and Software Science," and the groundbreaking book on Software Engineering Foundations: A Software Science Perspective.