

On Mathematical Laws of Software

Yingxu Wang

Theoretical and Empirical Software Engineering Research Centre (TESERC)
International Center for Cognitive Informatics (ICfCI)
Dept. of Electrical and Computer Engineering
Schulich Schools of Engineering, University of Calgary
2500 University Drive, NW, Calgary, Alberta, Canada T2N 1N4
Tel.: (403) 220 6141; Fax: (403) 282 6855
yingxu@ucalgary.ca

Abstract. Recent studies on the laws and mathematical constraints of software have resulted in fundamental discoveries in computing and software engineering toward exploring the nature of software. It was recognized that software is not constrained by any physical laws discovered in the natural world. However, software obeys the laws of mathematics, cognitive informatics, system science, and formal linguistics. This paper investigates into the mathematical laws of software and computing behaviors. A generic mathematical model of programs is created that reveals the nature of software as abstract processes and its uniqueness beyond other mathematical entities such as sets, relations, functions, and abstract concepts. A comprehensive set of mathematical laws for software and its behaviors is established based on the generic mathematical model of programs and the fundamental computing behaviors elicited in Real-Time Process Algebra (RTPA). A set of 95 algebraic laws of software behaviors is systematically derived, which encompasses the laws of meta-processes, process relations, and system compositions. The comprehensive set of mathematical laws of software lays a theoretical foundation for analyzing and modeling software behaviors and software system architectures, as well as for guiding rigorous practice in programming. They are also widely applicable for the rigorous modeling and manipulation of human cognitive processes and computational intelligent behaviors.

Keywords: Software science, software engineering, denotational mathematics, software, programs, computational intelligence, modeling, analysis, mathematical models, generic model of software, process models, algebraic laws, laws of meta-processes, laws of process relations, laws of process compositions, RTPA.

1 Introduction

The wander on laws of software science can be traced back to George Boole in his work on “*The Laws of Thought* [3].” Two fundamental discoveries in computing and software engineering, the *constrain laws* of software [9], [16] and the *process metaphor* [14], [15], [20], have fundamentally influenced the theoretical and empirical

research on exploring the nature of software. The former reveals that, although software is not constrained by any physical laws discovered in the nature world, it do obey the laws of mathematics [16], [21], [38], [39], [41], as well as cognitive informatics [33], [34], [36], [37], [40], system science [39], and formal linguistics and semantics [6], [7], [11], [13], [26], [31], [36], [39], [43]. The latter indicates that any software system and its behaviors can be modeled and described by a set of processes [14], which can be treated as a new mathematical entity beyond sets, relations, functions, and abstract concepts [38], [43], [44], [45], [46], [48].

In the exploration of the generic laws of programming, Hoare and his colleagues proposed a set of mathematic laws of programming [16]. This work covered a set of algebraic laws of programs for Dijkstra's *nondeterministic sequential programming language* [9]. Wang investigated the cognitive informatics laws of software in [36] and organizational laws of software engineering in [47], which enhanced the understanding toward the fundamental computing behaviors and their composing rules [2], [4], [8], [14], [15], [17], [20], [25], [27].

The latest reveal of the *generic program model* (GPM) in Wang (2007) [39] on the basis of Real-Time Process Algebra (RTPA) [32], [35], [39], [43], [46] have led to the establishment of a comprehensive set of 95 mathematical laws of fundamental software behaviors in the categories of meta-processes and process relations. The mathematical laws of software provide a solid foundation underlying software architectural and behavioral modeling and analyses. They can be used to reveal equivalency between process expressions, to simplify complicated process behaviors, to express interactive process relations, and to prove the correctness of software behaviors.

This paper investigates into the mathematical laws of software and computing behaviors. Emphases will be put on the laws of the most complicated real-time computing requirements and behaviors, such as laws for system dispatching, interrupt, timing, addressing, dynamic memory allocation, and Boolean/numeric evaluations. An algebraic treatment of fundamental behaviors of software systems is presented in Section 2, with the introduction of GPM and RTPA. Algebraic laws of software for the most fundamental and elementary computing behaviors are described in Section 3 known as the laws of meta-processes of software. Then, algebraic laws of software for constructing and composing complex process behaviors and architectures are described in Section 4 known as the laws of algebraic operations of software. The two categories of mathematical laws of software cover a comprehensive set of laws for software behaviors, architectures, and their interactions, which form a foundation for rigorous reasoning and modeling of software systems and computing mechanisms.

2 The Algebraic Treatment of Fundamental Behaviors of Software Systems in RTPA

On the basis of the process metaphor of software systems, abstract processes can be rigorously treated as a mathematical entity beyond sets, relations, functions, and abstract concepts. Real-Time Process Algebra (RTPA) is a denotational mathematical structure for denoting and manipulating system behavioral processes [32], [35], [36],

[39], [43], [46]. RTPA is designed as a coherent algebraic system for intelligent and software system modeling, specification, refinement, and implementation. RTPA encompasses 17 meta- processes 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.

Definition 1. RTPA is a denotational mathematical structure for algebraically denoting and manipulating system behavioural processes and their attributes by a triple, i.e.:

$$RTPA \triangleq (\mathfrak{T}, \mathfrak{P}, \mathfrak{R}) \quad (1)$$

where

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

Detailed descriptions of \mathfrak{T} , \mathfrak{P} , and \mathfrak{R} in RTPA will be extended in the following subsections.

2.1 The Generic Mathematical Model of Programs and Software Systems

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 2. A *process* P is the basic unit of an applied computational behavior that is composed by a set of statements p_i , $1 \leq i \leq n-1$, with left-associated cumulative relations, r_{ij} , i.e.:

$$\begin{aligned} P &= \mathbf{R}_{i=1}^{n-1} (p_i \ r_{ij} \ p_j), j = i+1 \\ &= (\dots(((p_1) \ r_{12} \ p_2) \ r_{23} \ p_3) \dots \ r_{n-1,n} \ p_n) \end{aligned} \quad (2)$$

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

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

Definition 3. 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.:

$$\wp = \mathbf{R}_{k=1}^m (@ e_k \hookrightarrow P_k) \quad (3)$$

Definitions 2 and 3 indicate that a program is an *embedded relational entity*, where a statement in a program is an instantiation of a meta-instruction of a programming language that executes a basic unit of coherent function and leads to a predictable behavior.

Theorem 1. The *Generic Program Model* (GPM) states that a software system or a program \wp is an algebraic structure with a set of embedded relational processes, in which all previous processes of a given process form the context of the current process, i.e.:

$$\begin{aligned} \wp &= \mathbf{R}_{k=1}^m (@ e_k \hookrightarrow P_k) \\ &= \mathbf{R}_{k=1}^m [@ e_k \hookrightarrow \mathbf{R}_{i=1}^{n-1} (p_i(k) r_{ij}(k) p_j(k))], j = i+1, p_i, p_j \in \mathfrak{P}, r_{ij} \in \mathfrak{R} \end{aligned} \quad (4)$$

Proof. Theorem 1 can be directly proved on the basis of Definitions 2 and 3. Substituting P_k in Definition 3 with Eq. 2, a generic program \wp obtains the form as a series of embedded relational processes as presented in Theorem 1.

The GPM model given in Theorem 1 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 1 provides a unified software model, which is a formalization of the well accepted but informal process metaphor for software systems in computing.

2.2 The Type System of 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 set of variables can assume is the domain of a type, and a type is always associated with a set of predefined or allowable operations in computing. A type can be classified as *primitive* and *derived* (complex) types. The former are the most elemental types that cannot further divided into simpler ones; the latter are a compound form of multiple primitive types based on given type rules. In computing, most primitive types are provided by programming languages; while most user defined types are derived ones.

Definition 4. A *type system* specifies data object modeling and manipulation rules in computing.

A set of 17 primitive types of RTPA in computing and human cognitive process modeling is elicited from works in [5], [19], [22], [28], [32], [35], [39], [43], [46], which

is summarized in Table 1. 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.

It is noteworthy that although a generic computing behavior is constrained by the *mathematical domain* D_m of types, an executable program is constrained by the *language-defined domain* D_l , and at most time, it is further restricted by the *user-defined domain* D_u , where $D_u \subseteq D_l \subseteq D_m$.

Table 1. RTPA Primitive Types and their Domains

No.	Type	Syntax	D_m	D_l
1	Natural number	N	$[0, +\infty]$	$[0, 65535]$
2	Integer	Z	$[-\infty, +\infty]$	$[-32768, +32767]$
3	Real	R	$[-\infty, +\infty]$	$[-2147483648, 2147483647]$
4	String	S	$[0, +\infty]$	$[0, 255]$
5	Boolean	BL	[T, F]	[T, F]
6	Byte	B	$[0, 255]$	$[0, 255]$
7	Hexadecimal	H	$[0, +\infty]$	$[0, \text{max}]$
8	Pointer	P	$[0, +\infty]$	$[0, \text{max}]$
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: $[0, 999]$
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	–	–
13	System architectural type	ST	–	–
14	Random event	@eS	$[0, +\infty]$	$[0, 255]$
15	Time event	@:TM	$[0\text{ms}, 9999\text{yyyy}]$	$[0\text{ms}, 9999\text{yyyy}]$
16	Interrupt event	@int@	$[0, 1023]$	$[0, 1023]$
17	Status	⊙sBL	[T, F]	[T, F]

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

$$\mathfrak{T} \triangleq \{\mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}, \mathbf{BL}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}, \mathbf{RT}, \mathbf{ST}, @e\mathbf{S}, @t\mathbf{TM}, @int\odot, \odot_s\mathbf{BL}\} \quad (5)$$

where the primitive types stand for *natural number, integer, real, string, Boolean, byte, hexadecimal, pointer, time, date, date/time, run-time determinable type, system architectural type, random event, time event, interrupt event, and system status*.

RTPA provides a coherent notation system and a rigorous mathematical structure for modeling both software and intelligent systems. RTPA can be used to describe both *logical and physical* models of systems, where logic 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 modelled, 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. Although CSP [14], [15], the timed-CSP [4], [10], [23], and other process algebra treated any computational operation as a process, RTPA distinguishes the concepts of meta-processes from those of complex and derived processes, which are composed by relational process operations on the meta-processes.

2.3 The Meta-processes of Software Behaviors in RTPA

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

Definition 5. A *meta-process* in RTPA is a primitive computational operation that cannot be broken down to further individual actions or behaviors.

A meta-process is an elementary process that serves as a basic building block for modeling software behaviors. In RTPA, a set of 17 meta-processes has been elicited as shown in Table 2, from essential and primary computational operations commonly identified in existing formal methods and modern programming languages [1], [12], [16], [18], [52], [53]. Mathematical notations and syntaxes of the meta-processes are formally described in Table 2, while formal semantics of the meta-processes of RTPA may be referred to [36], [39], [43].

Lemma 2. The *RTPA meta-process system* \mathfrak{P} encompasses 17 fundamental computational operations as defined in Table 2, i.e.:

$$\mathfrak{P} = \{:=, \blacklozenge, \Rightarrow, \Leftarrow, \Leftarrow\neq, >, <, |>, |<, @, \triangleq, \uparrow, \downarrow, !, \otimes, \boxtimes, \S\} \quad (6)$$

As shown in Lemma 2 and Table 2, each meta-process 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 the preceding subsection.

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. For instances, 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 [39].

Table 2. RTPA Meta-Processes

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

2.4 Process Operations of RTPA

Definition 6. A *process relation* in RTPA is an algebraic operation and a compositional rule between two or more meta-processes in order to construct a complex process.

A set of 17 fundamental process relations 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 referred to [36], [39], [43], [46].

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 as defined in Table 3, i.e.:

$$\mathfrak{R} = \{\rightarrow, \curvearrowright, |, | \dots | \dots, R^*, R^+, R^i, \circ, \rightsquigarrow, ||, \text{\textcircled{H}}, |||, \gg, \Leftarrow, \hookrightarrow_b, \hookrightarrow_e, \hookrightarrow_i\} \quad (7)$$

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	$ $	$\diamond \text{exp} \mathbf{B} \mathbf{L} = \mathbf{T} \rightarrow P$ $ \diamond \sim \rightarrow Q$
4	Switch	$ $ \dots $ $	$\diamond \text{exp} \mathbb{T} =$ $i \rightarrow P_i$ $ \sim \rightarrow \emptyset$ where $\mathbb{T} \in \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{S}\}$
5	While-loop	R^*	\mathbf{F} $R^* P$ $\text{exp} \mathbf{B} \mathbf{L} = \mathbf{T}$
6	Repeat-loop	R^+	\mathbf{F} $P \rightarrow R^+ P$ $\text{exp} \mathbf{B} \mathbf{L} = \mathbf{T}$
7	For-loop	R^i	$n \mathbf{N}$ $R^i P(i \mathbf{N})$ $i \mathbf{N} = 1$
8	Recursion	\circ	0 $R^{\circ} P^{i \mathbf{N}} \circ P^{i \mathbf{N} - 1}$ $i \mathbf{N} = n \mathbf{N}$
9	Function call	\rightsquigarrow	$P \rightsquigarrow F$
10	Parallel	$ $	$P Q$
11	Concurrence	$\text{\textcircled{H}}$	$P \text{\textcircled{H}} Q$
12	Interleave	$ $	$P Q$
13	Pipeline	\gg	$P \gg Q$
14	Interrupt	\Leftarrow	$P \Leftarrow Q$
15	Time-driven dispatch	\hookrightarrow_t	$@_t \mathbf{T} \mathbf{M} \hookrightarrow_t P_i$
16	Event-driven dispatch	\hookrightarrow_e	$@_e \mathbf{S} \hookrightarrow_e P_i$
17	Interrupt-driven dispatch	\hookrightarrow_i	$@_{int} \mathbf{I} \hookrightarrow_i P_i$

3 Laws of Meta-processes of Software Behaviors

A meta-process in RTPA is the most fundamental and elementary process that cannot be broken up further. Specific laws that constrain RTPA meta-processes in software engineering and computing are explored in this section. As a summary, the algebraic laws of the 17 meta-processes are listed in Table 4.

3.1 Laws of Assignments

Hoare wrote in 1969: “Assignment is undoubtedly the most characteristic feature of programming a digital computer, and one that most clearly distinguishes it from other branches of mathematics. It is surprising therefore that the axiom governing our reasoning about assignment is quite as simple as any to be found in elementary logic [13].”

Table 4. Algebraic Laws of Meta-Processes of Software Behaviors

No.	Process	Notation	Specific laws
1	Assignment	$:=$	L1 (Selectivity), L2 (Transitivity), L3 (Most recent effectiveness), L4 (Compositionality), L5 (Decompositionality)
2	Evaluation	\blacklozenge	L6 (Boolean evaluation), L7 (Ordinal evaluation), L8 (Ordinal power set evaluation), L9 (Numerical evaluation), L10 (Logical expressive equivalence), L11 (Shortcut of conjunctive evaluation), L12 (Shortcut of disjunctive evaluation)
3	Addressing	\Rightarrow	L13 (Definite memory addressing), L14 (Power set memory addressing)
4	Memory allocation	\Leftarrow	L15 (Memory allocation),
5	Memory release	\Leftarrow	L16 (Memory release)
6	Read	\triangleright	L17 (Memory read)
7	Write	\triangleleft	L18 (Memory write)
8	Input	\triangleright	L19 (Port input)
9	Output	\triangleleft	L20 (Port output)
10	Timing	$\@$ $\ $	L21 (System clock), L22 (Absolute timing), L23 (Relative timing)
11	Duration	\triangleq	L24 (Duration timing)
12	Increase	\uparrow	-
13	Decrease	\downarrow	-
14	Exception detection	!	-
15	Skip	\otimes	L25 (Jump equivalence), L26 (Skip absorption), L27 (Skew skip absorption)
16	Stop	\boxtimes	-
17	System	\S	L33 (Serial architecture representation), L46 (Recurrent CLM representation), L56 (Nested architecture representation), L63 (Parallel architecture representation), L77 (Coupled architecture representation)

The assignment process is transitive, and it constrained by the following laws of software.

Law 1. The law of *assignment selectivity* states that the assignment operation, $:=$, is selective on variables and/or values that share the same or equivalent type \mathbb{T} , i.e.:

$$\begin{aligned}
 y\mathbb{T} := x\mathbb{T} &\triangleq \blacklozenge(T(y) = T(x) \vee T(y) \simeq T(x)) \\
 &\rightarrow y\mathbb{T} = V(x)\mathbb{T} \\
 &| \blacklozenge\sim \\
 &\rightarrow !(@\text{TypeError}\mathbf{S}) \\
 &\rightarrow \emptyset
 \end{aligned} \tag{8}$$

where, \mathbb{T} is a predefined primitive type in RTPA, $\mathbb{T} \in \mathfrak{X}$, and $T(z)$ and $V(z)$ are a type and value evaluation function, respectively, on a given variable z .

Law 2. The law of *assignment transitivity* states that an assignment operation is transitive among multiple related variables that share the same or equivalent type \mathbb{T} , i.e.:

$$(y\mathbb{T} := z\mathbb{T}) \rightarrow (x\mathbb{T} := y\mathbb{T}) \triangleq x\mathbb{T} := z\mathbb{T} \tag{9}$$

Law 2 can be expressed in a more general form as assignment compositeness in Law 4.

Law 3. The law of *most recent effectiveness of assignments* states that n sequential assignment operations on the same variable are mutually exclusive, where only the last assignment is effective in the series of assignments, i.e.:

$$\mathop{R}_{i=1}^n y\mathbb{T} := x_i\mathbb{T} \triangleq (y\mathbb{T} := x_n\mathbb{T}) \tag{10}$$

Law 3 can be used to eliminate redundant or unnecessary assignments in programming or system specifications, particularly in a distributed environment.

Law 4. The law of *assignment compositionality* states that an assignment operation is compositional through multiple functions or expressions, i.e.:

$$\begin{aligned}
 (y\mathbb{T} := g(z\mathbb{T})\mathbb{T}) &\rightarrow (x\mathbb{T} := f(y\mathbb{T})\mathbb{T}) \\
 &\triangleq (x\mathbb{T} := f \circ g(z\mathbb{T})\mathbb{T})\mathbb{T} \\
 &= (x\mathbb{T} := f(g(z\mathbb{T})\mathbb{T})\mathbb{T})
 \end{aligned} \tag{11}$$

Law 4 is the foundation of programming and component-based system composition in software engineering. It is in line with the generic program model as presented in Section 2.1. It is obvious that Law 3 is a special case of Law 4 where both functions f and g are assignments.

An inverse expression of Law 4 forms the law of decompositionality for assignments.

Law 5. The law of *assignment decompositionality* states that an assignment operation is decompositional through multiple functions or expressions, i.e.:

$$\begin{aligned} x\mathbb{T} &:= f \circ g(z\mathbb{T})\mathbb{T}\mathbb{T} \\ &\triangleq x\mathbb{T} := f(g(z\mathbb{T})\mathbb{T})\mathbb{T} \\ &= (y\mathbb{T} := g(z\mathbb{T})\mathbb{T}) \rightarrow (x\mathbb{T} := f(y\mathbb{T})\mathbb{T}) \end{aligned} \quad (12)$$

Law 5 forms the foundation of programming and component-based system specification in software engineering.

3.2 Laws of Evaluations

The evaluation processes of expressions in computing can be classified as Boolean, ordinal, and numerical. The evaluation processes are constrained by the following laws of software.

Law 6. The law of *Boolean evaluation* states that a given Boolean expression $exp\mathbf{BL}$ can be evaluated exclusively by \diamond_{BL} , which results in one of the Boolean constants \mathbf{T} or \mathbf{F} , i.e.:

$$\diamond_{BL}(exp\mathbf{BL})\mathbf{BL} \triangleq \diamond_{BL}: exp\mathbf{BL} \rightarrow \{\mathbf{T}, \mathbf{F}\} \quad (13)$$

Typical branch constructs obey Law 6 where each branch is selected by a Boolean constant \mathbf{T} or \mathbf{F} .

Law 7. The law of *ordinal evaluation* states that a given natural number expression $exp\mathbf{N}$ can be evaluated ordinally by \diamond_N , which results in a unique ordinal number, i.e.:

$$\diamond_N(exp\mathbf{N})\mathbf{N} \triangleq \diamond_N: exp\mathbf{N} \rightarrow \mathbf{N} \quad (14)$$

Typical switch constructs obey Law 7 where each branch is selected by an ordinal number $n \in \mathbf{N}$.

Law 8. The law of *ordinal power set evaluation* states that a given natural number expression $exp\mathbf{N}$ can be evaluated ordinally by \diamond_{PN} , which results in a subset of natural numbers, i.e.:

$$\diamond_{PN}(exp\mathbf{N})\mathbf{PN} \triangleq \diamond_{PN}: exp\mathbf{N} \rightarrow \mathbf{PN} \quad (15)$$

A more general and flexible switch constructs obey Law 8 where each branch is selected by a subset of numbers $\mathbf{PN} \subseteq \mathbf{N}$.

Law 9. The law of *numerical evaluation* states that a given real expression $exp\mathbf{R}$ or integer expression $exp\mathbf{Z}$ can be evaluated numerically by \diamond_R , or \diamond_Z , which results in a real number or an integer, respectively, i.e.:

$$\diamond_R(exp\mathbf{R})\mathbf{R} \triangleq \diamond_R: exp\mathbf{R} \rightarrow \mathbf{R} \quad (16a)$$

$$\diamond_Z(exp\mathbf{Z})\mathbf{Z} \triangleq \diamond_Z: exp\mathbf{Z} \rightarrow \mathbf{Z} \quad (16b)$$

Law 9 is the mathematical foundation of measurement theories and software engineering measurement.

Law 10. The law of *logical expressive equivalence* states that a pair of Boolean expressions are equivalent, denoted by \simeq , iff: (a) Both expressions share the same set of variables; and either (b) The symbolic expressions can be transformed into the same form, or (c) The truth tables of both expressions are identical, i.e.:

$$\begin{aligned} exp_1(x_1\mathbf{BL}, x_2\mathbf{BL}, \dots, x_n\mathbf{BL})\mathbf{BL} &\simeq exp_2(x_1\mathbf{BL}, x_2\mathbf{BL}, \dots, x_n\mathbf{BL})\mathbf{BL} \\ &\triangleq \prod_{i=1}^n \forall x_i \in \mathbf{BL}, \diamond_{\mathbf{BL}}(exp_1\mathbf{BL})\mathbf{BL} = \diamond_{\mathbf{BL}}(exp_2\mathbf{BL})\mathbf{BL} \end{aligned} \quad (17)$$

According to Law 10, equivalent evaluation of Boolean expressions can be determined either by identical truth tables or by transformation of one expression into the other.

Law 11. The law of *shortcut of conjunctive evaluation* states that a conjunctive Boolean expression with multiple Boolean variables, $exp\mathbf{BL} = x_1\mathbf{BL} \wedge x_2\mathbf{BL} \wedge \dots \wedge x_n\mathbf{BL}$, can be evaluated directly as \mathbf{F} whenever at most one of them is false, i.e.:

$$\diamond_{\mathbf{BL}}(x_1\mathbf{BL} \wedge x_2\mathbf{BL} \wedge \dots \wedge x_n\mathbf{BL})\mathbf{BL} \triangleq \mathbf{F}, \exists x_i\mathbf{BL} = \mathbf{F}, 0 \leq i \leq n \quad (18)$$

Law 12. The law of *shortcut of disjunctive evaluation* states that a disjunctive Boolean expression with multiple Boolean variables, $exp\mathbf{BL} = x_1\mathbf{BL} \vee x_2\mathbf{BL} \vee \dots \vee x_n\mathbf{BL}$, can be evaluated directly as \mathbf{T} whenever at most one of them is true, i.e.:

$$\diamond_{\mathbf{BL}}(x_1\mathbf{BL} \vee x_2\mathbf{BL} \vee \dots \vee x_n\mathbf{BL})\mathbf{BL} \triangleq \mathbf{T}, \exists x_i\mathbf{BL} = \mathbf{T}, 0 \leq i \leq n \quad (19)$$

3.3 Laws of Addressing

Addressing is a fundamental computational process that maps a variable to a memory location or block. The addressing processes are constrained by the following laws of software.

Law 13. The law of *definite memory addressing* \Rightarrow states that a declared logical identifier $id\mathbf{B}$ in type byte can be associated to a unique physical memory address, i.e.:

$$\begin{aligned} id\mathbf{BL} &\Rightarrow \text{MEM}[ptr\mathbf{P}]\mathbf{B} \triangleq \\ \pi : id\mathbf{B} &\rightarrow ptr\mathbf{P} \rightarrow \text{MEM}[ptr\mathbf{P}]\mathbf{B} \end{aligned} \quad (20)$$

More generally, when the logical representation of an arbitrary typed identifier $id\mathbf{T}$ occupies more than one memory elements, the following laws can be introduced.

Law 14. The law of *power set memory addressing* states that a declared logical identifier $id\mathbf{T}$ can be mapped into a unique block of n continuous logical memory elements for representing a variable in type \mathbf{T} , i.e.:

$$\begin{aligned} id\mathbf{T} &\Rightarrow \text{MEM}[ptr\mathbf{P}]\mathbf{T} \triangleq \\ \pi : id\mathbf{T} &\rightarrow ptr\mathbf{P} \rightarrow \text{MEM}[ptr\mathbf{P}]\mathbf{T} \end{aligned} \quad (21)$$

where the power set of addressing results is determined by $ptr\mathbf{P} = [ptr\mathbf{P}, ptr\mathbf{P} + n\mathbf{N} - 1]$, and n is language implementation-specific.

Law 15. The law of *memory allocation* \Leftarrow states that a unique block of n continuous physical memory elements can be allocated to a declared logical identifier $id\mathbb{T}$, i.e.:

$$\begin{aligned} id\mathbb{T} \Leftarrow MEM[ptr\mathbf{P}]\mathbb{T} &\triangleq \\ \pi^{-1}: MEM[ptr\mathbf{P}]\mathbb{T} &\rightarrow id\mathbb{T} \end{aligned} \quad (22)$$

where the power set of allocation pointer is $ptr\mathbf{P} = [ptr\mathbf{P}, ptr\mathbf{P} + n\mathbf{N} - 1]$, and n is language implementation-specific.

Law 16. The law of *memory release* \Leftarrow states that a unique block of n continuous logical memory elements can be dissociated from a declared logical identifier $id\mathbb{T}$ by the following sequential processes, i.e.:

$$\begin{aligned} id\mathbb{T} \Leftarrow MEM[ptr\mathbf{P}]\mathbb{T} &\triangleq \\ (id\mathbb{T} \Rightarrow MEM[ptr\mathbf{P}]\mathbb{T} & \\ \rightarrow MEM[ptr\mathbf{P}, ptr\mathbf{P} + n\mathbf{N} - 1]\mathbb{T} := \perp & \\ \rightarrow ptr\mathbf{P} := \perp & \\ \rightarrow id\mathbb{T} := \perp & \\) & \end{aligned} \quad (23)$$

where the first step denotes a readdressing process in order to establish the association between the identifier, the memory block, and the pointers, and \perp denotes a value undefined.

3.4 Laws of I/O Manipulations

Input/output (I/O) processes are important computational operations for modeling interactive behaviors of systems. The I/O manipulation processes are constrained by the following laws of software.

Law 17. The law of *memory read* states that the read operation on memory, \triangleright , is equivalent to an assignment operation, where the allocated memory element pointed by $ptr\mathbf{P}$ is treated as a logical variable, i.e.:

$$MEM[ptr\mathbf{P}]\mathbb{T} \triangleright x\mathbb{T} \triangleq x\mathbb{T} := MEM[ptr\mathbf{P}]\mathbb{T} \quad (24)$$

Law 18. The law of *memory write* states that the write operation on memory, \triangleleft , is equivalent to an assignment operation, where the allocated memory element pointed by $ptr\mathbf{P}$ is treated as a logical variable, i.e.:

$$x\mathbb{T} \triangleleft MEM[ptr\mathbf{P}]\mathbb{T} \triangleq MEM[ptr\mathbf{P}]\mathbb{T} := x\mathbb{T} \quad (25)$$

Law 19. The law of *port input* states that the input from a port, $|>$, is equivalent to an assignment operation, where the allocated port buffer pointed by $ptr\mathbf{P}$ is treated as a logical variable, i.e.:

$$\text{PORT}[ptr\mathbf{P}]_{\mathbb{T}} |> x_{\mathbb{T}} \triangleq x_{\mathbb{T}} := \text{PORT}[ptr\mathbf{P}]_{\mathbb{T}} \quad (26)$$

Law 20. The law of *port output* states that the output to a port, $|<$, is equivalent to an assignment operation, where the allocated port buffer pointed by $ptr\mathbf{P}$ is treated as a logical variable, i.e.:

$$x_{\mathbb{T}} |< \text{PORT}[ptr\mathbf{P}]_{\mathbb{T}} \triangleq \text{PORT}[ptr\mathbf{P}]_{\mathbb{T}} := x_{\mathbb{T}} \quad (27)$$

3.5 Laws of Time Manipulations

Time manipulation is a necessary dimension in computing that is supplemental to the logic and space dimensions for modeling interactive system behaviors. The time manipulation processes are constrained by the following laws of software.

Law 21. The law of *system clock* states that a software system, in particular a real-time system, needs to maintain two clocks at the system level known as the *absolute clock* $\S t\mathbf{TM}$ and the *relative clock* $\S \tau\mathbf{N}$, as follows:

$$\text{a) } \S t\mathbf{TM} \triangleq [0:1:1:0:0:0:0\mathbf{yyyy:MM:dd:hh:mm:ss:ms}, \quad (28a) \\ 9999:12:31:23:59:59:999\mathbf{yyyy:MM:dd:hh:mm:ss:ms}]$$

where a subset of the date/time range \mathbf{TM} may be implemented for nonreal-time or noncontinuous systems.

$$\text{b) } \S \tau\mathbf{N} \triangleq [0\mathbf{ms}, 1 \times 10^8\mathbf{ms}] \quad (28b)$$

where the range of the relative clock is determined by $\S \tau\mathbf{N} = 24\mathbf{hh} \times 60\mathbf{mm} \times 60\mathbf{ss} \times 1000\mathbf{ms} = 8.64 \times 10^7\mathbf{ms}$. The relative clock $\S \tau\mathbf{N}$ may be reset to zero at midnight every day in system modeling.

Law 22. The law of *absolute timing* states that the absolute system clock $\S t\mathbf{TM}$ can be used to set, $\underline{\underline{\@}}$, a timing event or the execution of a process $\@t\mathbf{TM}$ at a precise point of calendar time, i.e.:

$$\@t\mathbf{TM} \underline{\underline{\@}} \S t\mathbf{TM} \quad (29)$$

where $\mathbf{TM} = \mathbf{yy:MM:dd | hh:mm:ss:ms | yyyy:MM:dd:hh:mm:ss:ms}$.

Law 23. The law of *relative timing* states that the relative clock $\S \tau\mathbf{N}$ can be used to set, $\underline{\underline{\@}}$, a timing event or the execution of a process $\@t\mathbf{N}$ at a certain relative time point, i.e.:

$$\@t\mathbf{N} \underline{\underline{\@}} \S \tau\mathbf{ms} \quad (30)$$

where $\S \tau\mathbf{N} \in [0, 1 \times 10^8\mathbf{ms}]$.

Law 24. The law of *duration timing* states that an absolute timing event $@t\mathbf{TM}$ on the system clock $\S t\mathbf{TM}$, or a relative timing event $@t\mathbf{N}$ on the system clock $\S \mathbf{N}$, can be set, \triangleq , with and a given duration $\Delta n\mathbf{TM}$ or $\Delta n\mathbf{ms}$ for a timing event or the execution of a process, i.e.:

$$@t\mathbf{TM} \triangleq \S t\mathbf{TM} + \Delta n\mathbf{TM} \quad (31a)$$

or

$$@t\mathbf{N} \triangleq \S \mathbf{N} + \Delta n\mathbf{ms} \quad (31b)$$

3.6 Laws of Skip

Skip is special meta-process in programming that has no semantic effect on the current process P^k at a given embedded layer k in a program, such as a branch, loop, or function. However, it redirects the system to jump to execute an upper-layer process P^{k-1} in the embedded hierarchy. Therefore, skip is also known as *exit* or *break* in programming. The skip processes are constrained by the following laws of software.

Law 25. The law of *jump equivalence* of skip processes states that a skip does nothing functionally but adjusts the internal control to jump to a predefined process outside the current executing layer, i.e.:

$$P \rightarrow \otimes \rightarrow Q = P \curvearrowright Q \quad (32)$$

where the *jump* process operation will be explained in Section 4.2.

Law 26. The law of *skip absorption* states that skips can be replaced by jump process relations in a branch, parallel, concurrent, or pipeline process structure, i.e.:

$$\begin{aligned} ((P \rightarrow \otimes) \mathbb{R} (Q \rightarrow \otimes)) \rightarrow S &= (P \curvearrowright S) \mathbb{R} (Q \curvearrowright S) \\ &= (P \mathbb{R} Q) \curvearrowright S \end{aligned} \quad (33)$$

where $\mathbb{R} = \{!, \parallel, \textcircled{\parallel}, \gg\}$.

Law 27. The law of *skew skip absorption* states that a skip can be replaced by a jump process in a skew branch, parallel, concurrent, or pipeline process structure, i.e.:

$$((P \rightarrow T) \mathbb{R} (Q \rightarrow \otimes)) \rightarrow S = (P \rightarrow T \rightarrow S) \mathbb{R} (Q \curvearrowright S) \quad (34)$$

where $\mathbb{R} = \{!, \parallel, \textcircled{\parallel}, \gg\}$.

According to Table 4, the top-level meta-process, *system*, obeys a set of architectural representation laws, which will be described in Section 4, such as Law 33 for serial architecture representation, Law 46 for recurrent CLM representation, Law 56 for nested architecture representation, Law 63 for parallel architecture representation, and Law 77 for coupled architecture representation.

4 Laws of Algebraic Operations of Software Behaviors

The relational operations of process behaviors defined in RTPA provides a set of 17 process composing rules for constructing complex processes and manipulating

advanced computing behaviors on the basis of the RTPA meta-processes. A comprehensive set of algebraic laws for relational process operations will be established in this section.

Let P, Q, S be meta or complex processes $P, Q, S \in \mathfrak{A}$, and \mathbb{R}, \mathbb{R}' be different relational operators, $\mathbb{R}, \mathbb{R}' \in \mathfrak{R}$, then a set of algebraic laws of software process compositions can be elicited as defined in Table 5, where the simplest process can be a single event.

Table 5 provides a set of generic algebraic laws of software process relations and compositional rules. Observing the table it can be seen that the 12 relational laws, $\mathcal{L}1 - \mathcal{L}12$, can be classified into six pairs, i.e., associative/dissociative, reflexive/irreflexive, symmetric/antisymmetric, transitive/intransitive, distributive/ nondistributive, and elicitive/nonelicitive. It is noteworthy that each pair of laws is exclusive, i.e., a specific process relational operation $\mathbb{R} \in \mathfrak{R}$ only obeys one of the laws in a certain pair as shown in Table 5.

Table 5. Generic Algebraic Laws of Software Processes

No.	Law	Description
Law $\mathcal{L}1$	Associative	$\mathbb{R}_1 \circ (\mathbb{R}_2 \circ \mathbb{R}_3) \Rightarrow (\mathbb{R}_1 \circ \mathbb{R}_2) \circ \mathbb{R}_3$
Law $\mathcal{L}2$	Dissociative	$\mathbb{R}_1 \circ (\mathbb{R}_2 \circ \mathbb{R}_3) \nRightarrow (\mathbb{R}_1 \circ \mathbb{R}_2) \circ \mathbb{R}_3$
Law $\mathcal{L}3$	Reflexive	$P \mathbb{R} Q \Rightarrow P = Q$
Law $\mathcal{L}4$	Irreflexive	$P \mathbb{R} Q \Rightarrow P \neq Q$
Law $\mathcal{L}5$	Symmetric	$P \mathbb{R} Q \Rightarrow Q \mathbb{R} P$
Law $\mathcal{L}6$	Asymmetric	$P \mathbb{R} Q \nRightarrow Q \mathbb{R} P$
Law $\mathcal{L}7$	Transitive	$P \mathbb{R} S \wedge S \mathbb{R} Q \Rightarrow P \mathbb{R} Q$
Law $\mathcal{L}8$	Intransitive	$P \mathbb{R} S \wedge S \mathbb{R} Q \nRightarrow P \mathbb{R} Q$
Law $\mathcal{L}9$	Distributive	$P \mathbb{R} (Q \mathbb{R}' S) \Rightarrow (P \mathbb{R} Q) \mathbb{R}' (P \mathbb{R} S)$
Law $\mathcal{L}10$	Nondistributive	$P \mathbb{R} (Q \mathbb{R}' S) \nRightarrow (P \mathbb{R} Q) \mathbb{R}' (P \mathbb{R} S)$
Law $\mathcal{L}11$	Elicitive	$(P \mathbb{R} Q) \mathbb{R}' (P \mathbb{R} S) \Rightarrow P \mathbb{R} (Q \mathbb{R}' S)$
Law $\mathcal{L}12$	Nonelicitive	$(P \mathbb{R} Q) \mathbb{R}' (P \mathbb{R} S) \nRightarrow P \mathbb{R} (Q \mathbb{R}' S)$

A mapping of the 12 generic laws into each of the relational operators is summarized in Table 6. The laws and properties may be used to enhance the understanding of the mechanisms and behaviors of the fundamental processes in software engineering and computational intelligence. In addition to the generic laws for software processes and behaviors, there are special laws for most of the relational operations as identified in the right-most column of Table 6, which will be described individually in the corresponding laws.

The following subsections describe the algebraic laws for each of the 17 relational process operations $\mathbb{R} \in \mathfrak{R}$, plus any additional special law that a particular process operation must obey.

4.1 Laws of Sequential Processes

The sequential operation of processes is associative, reflective, distributive, and elicitive. However, it is asymmetric and intransitive. The sequential process operations are constrained by the following laws of software.

Table 6. Algebraic Laws of Relational Process Operations

No.	Process relation	Notation	Generic algebraic laws												Special algebraic laws
			\mathcal{L}_1	\mathcal{L}_2	\mathcal{L}_3	\mathcal{L}_4	\mathcal{L}_5	\mathcal{L}_6	\mathcal{L}_7	\mathcal{L}_8	\mathcal{L}_9	\mathcal{L}_{10}	\mathcal{L}_{11}	\mathcal{L}_{12}	
1	Sequence	\rightarrow	✓		✓			✓		✓	✓		✓		L32, L33
2	Jump	\curvearrowright	✓		✓			✓		✓	✓		✓		-
3	Branch	$ $		✓		✓		✓		✓	✓		✓		L41
4	Switch	$ \dots $		✓		✓		✓		✓		✓	✓		L43
5	While	R^*	-	-	-	-	-	-	-	-	-	-	-	-	L44
6	Repeat	R^+	-	-	-	-	-	-	-	-	-	-	-	-	L44
7	For-do	R^i	-	-	-	-	-	-	-	-	-	-	-	-	L44, L45, L46
8	Recursion	\circlearrowleft	-	-	-	-	-	-	-	-	-	-	-	-	L47, L48, L49
9	Function call	\mapsto	✓			✓		✓		✓	✓		✓		L56
10	Parallel	\parallel	✓			✓	✓		✓			✓	✓		L62, L63
11	Concurrency	$\text{\textcircled{=}}$	✓			✓	✓		✓			✓	✓		L69
12	Interleave	$\text{\textcircled{ }}$	✓			✓	✓		✓			✓	✓		-
13	Pipeline	\gg	✓			✓		✓		✓		✓	✓		L76, L77
14	Interrupt	\Leftarrow		✓		✓		✓		✓		✓		✓	L78, L79, L80
15	Time-dispatch	\Leftarrow_t		✓		✓	✓			✓		✓	✓		L84, L85
16	Even-dispatch	\Leftarrow_e		✓		✓	✓			✓		✓	✓		L89, L90
17	Interrupt-dispatch	\Leftarrow_i		✓		✓	✓			✓		✓	✓		L94, L95

a) *Associativity of Sequential Processes*

Law 28. The law of *associativity* of sequential processes states that a list of sequential processes can be arbitrarily associated whenever their original order of sequence is preserved, i.e.:

$$P \rightarrow Q \rightarrow S = (P \rightarrow Q) \rightarrow S = P \rightarrow (Q \rightarrow S) \quad (35)$$

Law 28, sequential associativity, is the foundation of system modularization, decomposition, and integration in software engineering.

b) *Reflective of Sequential Processes*

Law 29. The law of *reflectivity* of sequential processes states that a list of sequential processes is reflective, *iff* they are identical, i.e.:

$$P \rightarrow Q = Q \rightarrow P \Rightarrow P = Q \quad (36)$$

c) *Distributivity of Sequential Processes*

Law 30. The law of *distributivity* of sequential processes states that a linear process S can be distributed into a pair of disjunctive conditional branch processes $P \mid Q$, i.e.:

$$S \rightarrow (P \mid Q) = (S \rightarrow P) \mid (S \rightarrow Q) \quad (37a)$$

or

$$(P \mid Q) \rightarrow S = (P \rightarrow S) \mid (Q \rightarrow S) \quad (37b)$$

d) *Elicitivity of Common Statements*

Law 31. The law of *elicitivity of common statements* states that common processes in a pair of disjunctive conditional branch processes can be elicited as shared processes, i.e.:

$$(S \rightarrow P) \mid (S \rightarrow Q) = S \rightarrow (P \mid Q) \quad (38a)$$

or

$$(P \rightarrow S) \mid (Q \rightarrow S) = (P \mid Q) \rightarrow S \quad (38b)$$

Law 31 is an inverse statement of Law 30, which is frequently used to elicit the common component from a branch structure.

In addition, sequential processes also obey the following special laws.

e) *Procedural Representation of Compound Statements*

Law 32. The law of *procedural representation of compound statements* states that an arbitrary subset of sequential processes P, Q, S can be associated into a procedure or function F , i.e.:

$$\begin{aligned} P \rightarrow (Q_1 \rightarrow Q_2 \rightarrow \dots \rightarrow Q_n) \rightarrow S = \\ P \rightarrow F \rightarrow S, F = (Q_1 \rightarrow Q_2 \rightarrow \dots \rightarrow Q_n) \end{aligned} \quad (39)$$

where F is the *procedure* elicited from the list of processes.

Law 32 is an extension of Law 28, sequential associativity, which provides a powerful means to derive structural programs that implements the principles of encapsulation, abstraction, and information hiding in software engineering.

f) Representation of Serial Architectures

Law 33. The law of *serial architecture representation* states that the sequential processes can be used as an abstract representation of recurring serial architectures (SAs) in system modeling, i.e.:

$$SAST \triangleq (P_1 \rightarrow P_2 \rightarrow \dots \rightarrow P_n) \quad (40)$$

where each process P_i , $1 \leq i \leq n$, denotes a component in the system.

Law 33 shows that system architecture modeling share the same properties as those of behavioral processes. Both of them can be modeled by processes and their relational operations.

4.2 Laws of Jump Processes

The jump operation of processes is associative, reflective, distributive, and elicitive, but it is asymmetric and intransitive. The jump process operations are constrained by the following laws of software.

a) Associativity of Jump Processes

Law 34. The law of *associativity* of jump processes states that multiple jumps between a list of sequential processes can be arbitrarily associated whenever their original order of sequence is preserved, i.e.:

$$P \curvearrowright Q \curvearrowright S = (P \curvearrowright Q) \curvearrowright S = P \curvearrowright (Q \curvearrowright S) \quad (41)$$

b) Reflective of Jump Processes

Law 35. The law of *reflectivity* of jump processes states that the jump operation between two processes is reflective, *iff* they are identical, i.e.:

$$P \curvearrowright Q = Q \curvearrowright P \Rightarrow P = Q \quad (42)$$

c) Distributivity of Jump Processes

Law 36. The law of *distributivity* of jump processes states that a process S can be distributed into a pair of disjunctive conditional branch processes $P \mid Q$ by the jump operation, i.e.:

$$S \curvearrowright (P \mid Q) = (S \curvearrowright P) \mid (S \curvearrowright Q) \quad (43a)$$

or

$$(P \mid Q) \curvearrowright S = (P \curvearrowright S) \mid (Q \curvearrowright S) \quad (43b)$$

d) Elicitivity of Jump Processes

Law 37. The law of *elicitivity of jump statements* states that a common process in a pair of branch processes can be elicited as a shared process, i.e.:

$$(S \curvearrowright P) | (S \curvearrowright Q) = S \curvearrowright (P | Q) \quad (44a)$$

or

$$(P \curvearrowright S) | (Q \curvearrowright S) = (P | Q) \curvearrowright S \quad (44b)$$

The elicitivity of jump operations is an inversed operation of jump distributivity as given in Law 36.

4.3 Laws of Branch Processes

The branch operation of processes is distributive and elicitive, but it is dissociative, irreflexive, asymmetric, and intransitive. The branch process operations are constrained by the following laws of software.

a) Distributive of Branch Processes

Law 38. The law of *distributivity of branch* states that a process S can be distributed into a pair of disjunctive conditional branch processes $P | Q$ by a relational operation \mathbb{R} , i.e.:

$$S \mathbb{R} (P | Q) = (S \mathbb{R} P) | (S \mathbb{R} Q) \quad (45a)$$

or

$$(P | Q) \mathbb{R} S = (P \mathbb{R} S) | (Q \mathbb{R} S) \quad (45b)$$

where $\mathbb{R} = \{\parallel, \text{ff}, \curvearrowright, \gg, \parallel\}$.

b) Elicitivity of Invariant Process from Branch Structures

Law 39. The law of *elicitivity of sequential processes* states that a common sequential process S within a disjunctive conditional process can be elicited and executed outside the conditional construct, i.e.:

$$\begin{aligned} & \blacklozenge \text{exp} \mathbf{BL} = \mathbf{T} \\ & \quad \rightarrow S \\ & \quad \rightarrow P \\ & | \blacklozenge \sim \\ & \quad \rightarrow S \\ & \quad \rightarrow Q \\ & = S \rightarrow (\blacklozenge \text{exp} \mathbf{BL} = \mathbf{T} \\ & \quad \rightarrow P \\ & \quad | \blacklozenge \sim \\ & \quad \rightarrow Q \\ & \quad) \end{aligned} \quad (46a)$$

where $\text{exp} \mathbf{BL}$ is a Boolean expression, and $\blacklozenge \sim$ denotes otherwise.

Law 39 can be similarly expressed in other form as follows:

$$\begin{aligned}
 & \diamond \exp \mathbf{BL} = \mathbf{T} \\
 & \quad \rightarrow P \\
 & \quad \rightarrow S \\
 & | \diamond \sim \\
 & \quad \rightarrow Q \\
 & \quad \rightarrow S \\
 & = (\diamond \exp \mathbf{BL} = \mathbf{T} \\
 & \quad \rightarrow P \\
 & \quad | \diamond \sim \\
 & \quad \quad \rightarrow Q \\
 & \quad) \\
 & \quad \rightarrow S
 \end{aligned} \tag{46b}$$

where S is independent from the Boolean expression $\exp \mathbf{BL}$ or the execution of S will not affect the Boolean value of $\exp \mathbf{BL}$.

Law 39, sequential elicitivity, is the theoretical foundation of common function or object elicitation, improvement of programming efficiency, and well structured programming in software engineering.

c) Skew Symmetry of Branch Processes

Law 40. The law of *skew symmetry of branch* states that a branch or conditional choice is commutative on the true and false branches, i.e.:

$$\begin{aligned}
 & \diamond \exp \mathbf{BL} = \mathbf{T} \\
 & \quad \rightarrow P \\
 & | \diamond \sim \\
 & \quad \rightarrow Q \\
 & = \diamond \exp \mathbf{BL} = \mathbf{F} \\
 & \quad \rightarrow Q \\
 & | \diamond \sim \\
 & \quad \rightarrow P
 \end{aligned} \tag{47}$$

d) Embedded Branch Processes

Law 41. The law of *embedded branch* states that multiple branches can be nested on the else branches in order to form a multi-layer branch structure, i.e.:

$$\begin{aligned}
 & \diamond \exp_0 \mathbf{BL} = \mathbf{T} \\
 & \quad \rightarrow P_0 \\
 & | \diamond \sim \\
 & \quad \rightarrow \diamond \exp_1 \mathbf{BL} = \mathbf{T} \\
 & \quad \quad \rightarrow P_1 \\
 & \quad | \diamond \sim \\
 & \quad \quad \dots \\
 & \quad \quad \rightarrow \diamond \exp_n \mathbf{BL} = \mathbf{T} \\
 & \quad \quad \quad \rightarrow P_n \\
 & \quad \quad | \diamond \sim \\
 & \quad \quad \rightarrow \otimes \\
 & = \diamond \exp \mathbf{N} = 0 \rightarrow P_0 \\
 & \quad | 1 \rightarrow P_1 \\
 & \quad | \dots \\
 & \quad | n \rightarrow P_n \\
 & \quad | \sim \rightarrow \otimes
 \end{aligned} \tag{48}$$

4.4 Laws of Switch Processes

The switch operation of processes is elicitive, but it is dissociative, irreflexive, asymmetric, intransitive, and nondistributive. Therefore, Law 38 for branch structures can be extended to the switch structure as follows.

a) Elicitivity of Sequential Processes

Law 42. The law of *elicitivity* of sequential processes states that a common sequential process Q within a switch process can be elicited and executed outside the switch construct, i.e.:

$$\begin{aligned}
 & \diamond \exp \mathbf{N} = 0 \rightarrow (P_0 \rightarrow Q) \\
 & \quad | 1 \rightarrow (P_1 \rightarrow Q) \\
 & \quad | \dots \\
 & \quad | n \rightarrow (P_n \rightarrow Q) \\
 & \quad | \sim \rightarrow Q \\
 & = \diamond \exp \mathbf{N} = 0 \rightarrow P_0 \\
 & \quad | 1 \rightarrow P_1 \\
 & \quad | \dots \\
 & \quad | n \rightarrow P_n \\
 & \quad | \sim \rightarrow \emptyset \\
 & \rightarrow Q
 \end{aligned} \tag{49a}$$

or

$$\begin{aligned}
& \diamond \exp \mathbf{N} = 0 \rightarrow (Q \rightarrow P_0) \\
& \quad | 1 \rightarrow (Q \rightarrow P_1) \\
& \quad | \dots \\
& \quad | n \rightarrow (Q \rightarrow P_n) \\
& \quad | \sim \rightarrow Q \\
& = Q \rightarrow (\diamond \exp \mathbf{N} = 0 \rightarrow P_0 \\
& \quad | 1 \rightarrow P_1 \\
& \quad | \dots \\
& \quad | n \rightarrow P_n \\
& \quad | \sim \rightarrow \emptyset) \\
& \quad)
\end{aligned} \tag{49b}$$

The sequential elicitivity is the theoretical foundation of common function or object elicitation, improvement of programming efficiency, and well structured programming in software engineering.

b) Equivalent Embedded Branch Processes

Law 43. The law of *equivalent embedded branch* states that the switch operation is equivalent to the multiple embedded structures in computing, i.e.:

$$\begin{aligned}
& \diamond \exp \mathbf{N} = 0 \rightarrow P_0 \\
& \quad | 1 \rightarrow P_1 \\
& \quad | \dots \\
& \quad | n \rightarrow P_n \\
& \quad | \sim \rightarrow \otimes \\
& = \diamond \exp_0 \mathbf{BL} = \mathbf{T} \\
& \quad \rightarrow P_0 \\
& \quad | \diamond \sim \\
& \quad \rightarrow \diamond \exp_1 \mathbf{BL} = \mathbf{T} \\
& \quad \quad \rightarrow P_1 \\
& \quad | \diamond \sim \\
& \quad \dots \\
& \quad \rightarrow \diamond \exp_n \mathbf{BL} = \mathbf{T} \\
& \quad \quad \rightarrow P_n \\
& \quad | \diamond \sim \\
& \quad \rightarrow \otimes
\end{aligned} \tag{50}$$

4.5 Laws of Iterative Processes

Although the iterative operations of processes do not obey any of the generic algebraic laws as given in Table 5, they are constrained by the following special laws of software.

a) *Equivalence between Different Forms of Iterations*

Law 44. The law of *equivalence between different forms of iterations* states that all forms of iterative constructs, such as while-do R^* , repeat-do R^+ , and for-do R^i , are equivalent, i.e.:

$$a) R^*P = \underset{\text{exp } \mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{R}} P \quad (51a)$$

$$b) \begin{aligned} R^+P &= P \rightarrow R^*P \\ &= P \rightarrow \underset{\text{exp } \mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{R}} P \end{aligned} \quad (51b)$$

$$c) \begin{aligned} R^iP(i\mathbf{N}) &= \prod_{i\mathbf{N}=1}^n P(i\mathbf{N}) \\ &= (i\mathbf{N} := 1 \\ &\rightarrow \text{exp } \mathbf{BL}=(i\mathbf{N} \leq n\mathbf{N}) \\ &\rightarrow \underset{\text{exp } \mathbf{BL}=\mathbf{T}}{\overset{\mathbf{F}}{R}} P(i\mathbf{N}) \\ &\rightarrow \uparrow (i\mathbf{N}) \\ &.) \end{aligned} \quad (51c)$$

where R^* , R^+ , and R^i are known as the big-R notation of iterative behaviors and operations in software engineering [32], [39], [42].

b) *Cumulativeness of Iterations*

Law 45. The law of *cumulativeness of iterations* states that two of sequential iterations of identical process P can be concatenated, i.e.:

$$\underset{i\mathbf{N}=0}^{n-1} R P(i\mathbf{N}) \rightarrow \underset{i\mathbf{N}=n}^{n+m-1} R P(i\mathbf{N}) = \underset{i\mathbf{N}=0}^{n+m-1} R P(i\mathbf{N}) \quad (52)$$

c) *Recurrent Denoting of Logic Architectures*

Law 46. The law of *recurrent Component Logical Model (CLMs) representation* [32], [39], [42] states that the iterative process relations can be used as an abstract model to denote repetitive architectural patterns in computing, i.e.:

$$CLMST \triangleq \mathop{R}_{i\mathbb{N}=0}^{n-1} CLM[i\mathbb{N}] \quad (53)$$

Law 46 can be illustrated by the following example.

Example 1. The port architecture of a computer, PA , with 1024 ports in various types \mathbb{T}_i , can be denoted as follows:

$$PAST \triangleq \mathop{R}_{i\mathbb{N}=0}^{1023} PORT[i\mathbb{N}]\mathbb{T}_i \quad (54)$$

where $\mathbb{T}_i \in \{\mathbf{B}, \mathbf{H}, \mathbf{N}, \mathbf{Z}, \mathbf{R}, \mathbf{S}\} \subset \mathfrak{T}$.

4.6 Laws of Recursive Processes

The recursive operation of processes is constrained by the following special laws of software.

a) Two-Phase Recursions

Law 47. The law of *two-phase recursion* states that a recursion is carried out by a series of deductive embedding processes (denoted by \circlearrowleft) and then followed by an inversed series of inductive de-embedding processes (denoted by \circlearrowright), i.e.:

$$\begin{aligned} \mathop{R}_{i\mathbb{N}=n\mathbb{N}}^0 P^{i\mathbb{N}} \circlearrowleft P^{i\mathbb{N}-1} &\rightarrow \mathop{R}_{i\mathbb{N}=0}^{n\mathbb{N}} P^{i\mathbb{N}} \circlearrowright P^{i\mathbb{N}+1} \\ &= P^n \circlearrowleft P^{n-1} \circlearrowleft \dots \circlearrowleft P^1 \circlearrowleft P^0 \circlearrowright P^1 \circlearrowright \dots \circlearrowright P^{n-1} \circlearrowright P^n \end{aligned} \quad (55)$$

where in the first phase of *embedding*, a given layer of nested process is deduced to a lower layer till it is embodied to a known value P^0 . In the second phase of *de-embedding*, the value of a higher layer process is deduced by the lower layer starting from the base layer P^0 , where its value has already been known at the end of the preceding phase.

b) Terminable Recursions

Law 48. The law of *terminable recursion* states that a recursive function is terminable or non circular, *iff*: (a) A base value P^0 exists for certain arguments for which the function does not refer to itself; and (b) In each recursion, the argument of the function must be closer to the base value, i.e.:

$$\mathop{R}_{i\mathbb{N}=n\mathbb{N}}^0 P^{i\mathbb{N}} \circlearrowleft P^{i\mathbb{N}-1} \quad (56)$$

c) Equivalence between Recursions and Iterations

Law 49. The law of *equivalence between recursive and iteration* states that a recursive structure can always be represented by an equivalent iterative structure, i.e.:

$$\begin{aligned}
 \overset{0}{R} P^{\mathbf{n}} \circ P^{\mathbf{n}-1} &= \overset{0}{R} (\diamond i\mathbf{n} > 0 \\
 &\quad \rightarrow P^{\mathbf{n}} := P^{\mathbf{n}-1} \\
 &\quad | \diamond \sim \\
 &\quad \rightarrow P^0 \\
 &\quad)
 \end{aligned} \tag{57}$$

where a process P at layer i of embedment, P^i , calls itself at an inner layer $i-1$, P^{i-1} , $0 \leq i \leq n$, and n is the *depth of recursion* or embedment that is determined by an explicitly specified conditional expression $\text{exp}\mathbf{BL} = \mathbf{T}$ inside the body of P .

4.7 Laws of Function Calls

The function call operation of processes is associative, distributive, and elicitive, but it is irreflexive, asymmetric, and intransitive. The function-call process operations are constrained by the following laws of software.

a) Function Elicitation from Recurring Patterns

It is a good practice in programming if the common portion of program R in both paths of branch structures can be elicited as a shared process or a list of sequential statements. The law of elicitivity is in line with the principle of information hiding [24].

Law 50. Recurring patterns, processes, algorithms, and methods of classes in programming can be elicited and predefined as a procedure or function.

Some typical recurring algorithms and processes are provided below, such as the *unit increment/decrement* function and the *modular* function.

Example 2. A pair of unit increment and decrement functions can be introduced to simplify frequently used expressions in programming.

$$\begin{aligned}
 x\mathbb{T} &:= x\mathbb{T} + 1 \\
 &\triangleq \uparrow(x\mathbb{T}), \mathbb{T} \in \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}\} \subseteq \mathfrak{T}
 \end{aligned} \tag{58a}$$

$$\begin{aligned}
 x\mathbb{T} &:= x\mathbb{T} - 1 \\
 &\triangleq \downarrow(x\mathbb{T}), \mathbb{T} \in \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{H}, \mathbf{P}, \mathbf{TI}, \mathbf{D}, \mathbf{DT}\} \subseteq \mathfrak{T}
 \end{aligned} \tag{58b}$$

where the definitions of the specific types in the set of abstract type \mathbb{T} may be referred to [32], [39].

Example 3. Let $x\mathbf{Z}$ be an arbitrary integer, and $M\mathbf{N}$ be a positive integer, a modular function *mod* yields the integer remainder $r\mathbf{N}$, i.e.:

$$\begin{aligned}
 x\mathbf{Z} \bmod M\mathbf{N} &= r\mathbf{N}, \quad 0 \leq r\mathbf{N} < M\mathbf{N} \\
 &= \begin{cases} x\mathbf{Z} - kM\mathbf{N}, & x\mathbf{Z} > 0 \\ kM\mathbf{N} - x\mathbf{Z}, & x\mathbf{Z} < 0 \end{cases}
 \end{aligned} \tag{59}$$

For instances, according to Eq. 59, $18 \bmod 12 = 18 - 1 \bullet 12 = 6$, and $-26 \bmod 7 = 4 \bullet 7 - 26 = 2$.

Programmers may define their own functions as a basic system construction mechanism in computing.

b) Associativity of Function Calls

Law 51. The law of *associativity of function calls* states that a sequence of linear procedure calls can be arbitrarily associated or grouped, i.e.:

$$P \rightsquigarrow Q \rightsquigarrow S = (P \rightsquigarrow Q) \rightsquigarrow S = P \rightsquigarrow (Q \rightsquigarrow S) \quad (60)$$

c) Distributivity of Function Calls

Law 52. The law of *distributivity of function calls* states that a sequential process S can be distributed into a pair of disjunctive conditional processes $P \mid Q$, i.e.:

$$(P \mid Q) \rightsquigarrow S = (P \rightsquigarrow S) \mid (Q \rightsquigarrow S) \quad (61)$$

d) Function Elicitation from Branch Structures

Law 53. The law of *function elicitation from branch structures* states that a common pattern S in both branches of a conditional structure can be elicited and separately encapsulated as a predefined defined procedure, i.e.:

$$\begin{aligned} & \blacklozenge \text{exp} \mathbf{BL} = \mathbf{T} \\ & \quad \rightarrow P \\ & \quad \rightsquigarrow S \\ & \mid \blacklozenge \sim \\ & \quad \rightarrow Q \\ & \quad \rightsquigarrow S \\ & = (\blacklozenge \text{exp} \mathbf{BL} = \mathbf{T} \\ & \quad \rightarrow P \\ & \quad \mid \blacklozenge \sim \\ & \quad \rightarrow Q \\ & \quad) \\ & \quad \rightsquigarrow S \end{aligned} \quad (62)$$

e) Function Elicitation from Switch Structures

Law 54. The law of *function elicitation from switch structures* states that a common pattern S in both branches of a conditional structure can be elicited and separately encapsulated as a predefined procedure, i.e.:

$$\begin{aligned}
 & \blacklozenge \text{exp} \mathbf{N} = 0 \rightarrow (P_0 \rightarrow S) \\
 & \quad | 1 \rightarrow (P_1 \rightarrow S) \\
 & \quad | \dots \\
 & \quad | n \rightarrow (P_n \rightarrow S) \\
 & \quad | \sim \rightarrow S \\
 & = \blacklozenge \text{exp} \mathbf{N} = 0 \rightarrow P_0 \\
 & \quad | 1 \rightarrow P_1 \\
 & \quad | \dots \\
 & \quad | n \rightarrow P_n \\
 & \quad | \sim \rightarrow \emptyset \\
 & \rightarrow S
 \end{aligned} \tag{63}$$

f) *Function Elicitation from Parallel Structures*

Law 55. The law of *function elicitation from parallel structures* states that a common pattern S in both sides of a parallel structure can be elicited and separately encapsulated as a predefined procedure, i.e.:

$$(P \rightarrow S) \parallel (Q \rightarrow S) = (P \parallel Q) \rightarrow S \tag{64}$$

The laws of procedure elicitation from various constructs and process relations are the theoretical foundation of common function or object elicitation, improvement of programming efficiency, and structured programming in software engineering.

g) *Representation of Embedded Architectures*

Law 56. The law of *nested architecture representation* states that the function-call processes are an abstract representation of recurring nested architectures, *NAs*, of systems, i.e.:

$$NAST \triangleq P_1 \rightarrow P_2 \rightarrow \dots \rightarrow P_n \tag{65}$$

where each process P_i , $1 \leq i \leq n$, denotes a nested component in the system.

4.8 Laws of Parallel Processes

The parallel operation of processes is associative, symmetric, transitive, and elicitive, but it is irreflexive and nondistributive. The parallel process operations are constrained by the following laws of software, assuming that all processes belong to and are synchronized in the same system in the parallel structure.

a) *Associativity of Parallel Processes*

Law 57. The law of *associativity of parallel processes* states that a list of parallel processes can be arbitrarily associated or grouped, i.e.:

$$P \parallel Q \parallel S = (P \parallel Q) \parallel S = P \parallel (Q \parallel S) = P \parallel (S \parallel Q) \tag{66}$$

b) Symmetry of Parallel Processes

Law 58. The law of *symmetry of parallel processes* states that parallel process relations are commutative, i.e.:

$$P \parallel Q = Q \parallel P \quad (67)$$

c) Transitivity of Parallel Processes

Law 59. The law of *transitivity of parallel processes* states that parallel process relations are transitive between each other, i.e.:

$$(P \parallel Q) \parallel (Q \parallel S) = P \parallel Q \parallel S \quad (68)$$

d) Elicitivity of Parallel Processes

Law 60. The law of *elicitivity of parallel processes* states that the common process in two groups of parallel processes can be elicited, i.e.:

$$(P \parallel S) \rightarrow (Q \parallel S) = (P \rightarrow Q) \parallel S = S \parallel (P \rightarrow Q) \quad (69)$$

e) Elicitivity of Event

Law 61. The law of *elicitivity of event* in parallel processes states that a common event which triggers different parallel processes is extractive, i.e.:

$$@e\mathbf{s} \mapsto P \parallel @e\mathbf{s} \mapsto Q = @e\mathbf{s} \mapsto (P \parallel Q) \quad (70)$$

f) Idempotency of Identical Parallel Processes

Law 62. The law of *idempotency of identical parallel processes* states that parallel process relation between the same process is idempotent, i.e.:

$$P \parallel P = P \quad (71)$$

g) Representation of Parallel Architectures

Law 63. The law of *parallel architecture representation* states that the parallel processes can be used as an abstract model of recurring parallel architectures, *PAs*, of systems, i.e.:

$$PAST \triangleq P_1 \parallel P_2 \parallel \dots \parallel P_n \quad (72)$$

where each process P_i , $1 \leq i \leq n$, denotes a parallel component in the system.

4.9 Laws of Concurrent Processes

The concurrent operation of processes is associative, symmetric, transitive, and elicitive, but it is irreflexive and nondistributive. The concurrent process operations are constrained by the following laws of software. The differences between concurrent and parallel processes are that the former are implemented and executed on separated machines or they are asynchronized in a distributed environment.

a) Associativity of Concurrent Processes

Law 64. The law of *associativity of concurrent processes* states that a list of concurrent processes can be arbitrarily associated or grouped, i.e.:

$$P \parallel Q \parallel S = (P \parallel Q) \parallel S = P \parallel (Q \parallel S) = (P \parallel S) \parallel Q \quad (73)$$

b) Symmetry of Concurrent Processes

Law 65. The law of *symmetry of concurrent processes* states that concurrent process relations are commutative, i.e.:

$$P \parallel Q = Q \parallel P \quad (74)$$

c) Transitivity of Concurrent Processes

Law 66. The law of *transitivity of concurrent processes* states that concurrent process relations are transitive between each other, i.e.:

$$(P \parallel Q) \parallel (Q \parallel S) = P \parallel Q \parallel S \quad (75)$$

d) Elicitivity of Concurrent Processes

Law 67. The law of *elicitivity of concurrent processes* states that the common process in two groups of concurrent processes can be elicited, i.e.:

$$(P \parallel S) \rightarrow (Q \parallel S) = (P \rightarrow Q) \parallel S = S \parallel (P \rightarrow Q) \quad (76)$$

e) Elicitivity of Event

Law 68. The law of *elicitivity of event* in concurrent processes states that a common event which triggers different concurrent processes is extractive, i.e.:

$$@e \mathbf{s} \mapsto P \parallel @e \mathbf{s} \mapsto Q = @e \mathbf{s} \mapsto (P \parallel Q) \quad (77)$$

f) Idempotency of Identical Concurrent Processes

Law 69. The law of *idempotency of identical concurrent processes* states that concurrent process relation between the same process is idempotent, i.e.:

$$P \parallel P = P \quad (78)$$

4.10 Laws of Interleave Processes

The interleave operation of processes is associative, symmetric, transitive, and elicitive, but it is irreflexive and nondistributive. The interleave process operations are constrained by the following laws of software.

a) Associativity of Interleave Processes

Law 70. The law of *associativity of interleave processes* states that a list of interleaved processes can be arbitrarily associated or grouped, i.e.:

$$P \parallel Q \parallel S = (P \parallel Q) \parallel S = P \parallel (Q \parallel S) = P \parallel (S \parallel Q) \quad (79)$$

b) Symmetry of Interleave Processes

Law 71. The law of *symmetry of interleave processes* states that interleaved process relations are commutative, i.e.:

$$P \parallel Q = Q \parallel P \quad (80)$$

c) Transitivity of Interleave Processes

Law 72. The law of *transitivity of interleave processes* states that interleaved process relations are transitive between each other, i.e.:

$$(P \parallel Q) \parallel (Q \parallel S) = P \parallel Q \parallel S \quad (81)$$

d) Elicitivity of Interleave Processes

Law 73. The law of *elicitivity of interleave processes* states that the common process in two groups of interleaved processes can be elicited, i.e.:

$$(P \parallel S) \rightarrow (Q \parallel S) = (P \rightarrow Q) \parallel S = S \parallel (P \rightarrow Q) \quad (82)$$

4.1.1 Laws of Pipeline Processes

The pipeline operation of processes is associative and elicitive, but it is irreflexive, asymmetric, intransitive, and nondistributive. The pipeline process operations are constrained by the following laws of software.

a) Associativity of Pipeline Processes

Law 74. The law of *associativity of pipeline processes* states that a list of interlinked processes can be arbitrarily associated or grouped, i.e.:

$$P \gg Q \gg S = (P \gg Q) \gg S = P \gg (Q \gg S) = P \gg (S \gg Q) \quad (83)$$

b) Elicitivity of Pipeline Processes

Law 75. The law of *elicitivity of interleave processes* states that the common process in two groups of interleaved processes can be elicited, i.e.:

$$(S \rightarrow P) \gg (S \rightarrow Q) = S \rightarrow (P \gg Q) \quad (84)$$

or

$$(P \rightarrow S) \gg (Q \rightarrow S) = (P \gg Q) \rightarrow S \quad (85)$$

c) Pairwise Coupling of Pipeline Processes

Law 76. The law of *pairwise coupling of pipeline processes* states that each of the outputs of process P , O_{P_i} , $1 \leq i \leq n$, is connected to the counterpart process Q 's input, I_{Q_i} , i.e.:

$$\prod_{i=1}^n (O_{P_i} = I_{Q_i}) \quad (86)$$

where O_P and I_Q are the outputs and inputs of processes P and Q , and $\#O_P = \#I_Q$.

d) Representation of Coupled Architectures

Law 77. The law of *coupled architecture representation* states that the pipeline processes can be used as an abstract model of recurring pairwise coupled architectures, CAs, of systems, i.e.:

$$\mathbf{CAST} \triangleq P_1 \gg P_2 \gg \dots \gg P_n \quad (87)$$

where each process P_i , $1 \leq i \leq n$, denotes a component in the system.

It is noteworthy that Laws 33, 46, 56, 63, and 77 provide a set of five laws for basic system architectures in system modeling known as the serial, recurrent (CLM), nested, parallel, and coupled structures, respectively. These laws also indicate that not only system behaviors but also system architectures can be modeled by the RTPA process relational operations [32], [39], [42].

4.1.2 Laws of Interrupt Processes

The interrupt operation of processes is dissociative, irreflexive, asymmetric, intransitive, nondistributive, and nonelictive. However, interrupt process operations are constrained by the following special laws.

a) Parallel Mechanism of Interrupt Processes

Law 78. The law of *parallel mechanism of interrupt processes* states that an interrupt service process Q is parallel to the main process P , which is triggered by the i th interrupt event $@int_i \odot$, i.e.:

$$P \not\prec Q \triangleq P \parallel @int_i \odot \not\triangleright Q_i \not\searrow \odot \quad (88)$$

where $\not\triangleright$ and $\not\searrow$ denote an interrupt service and an interrupt return, respectively.

b) Hierarchy of Interrupt Priorities

Law 79. The law of *hierarchy of interrupt priorities* states that multiple interrupt resources interacting with a computing system can be configured at different levels of priorities l , i.e.:

$$\prod_{l=1}^{nM} int_l \hookrightarrow Q_l \quad (89)$$

c) Maximum Duration of Interrupt Services

Law 80. The law of *the maximum duration of interrupt service* states that the duration of an interrupt process in $P \not\prec Q$ should not exceed the basic time slice of system dispatching $\S t_d$, i.e.:

$$t_{int} = t_Q < \S t_d \quad (90)$$

4.1.3 Laws of Time Dispatch Processes

The time dispatch operation of processes at the system level is symmetric and elicitive, but it is dissociative, irreflexive, intransitive, and nondistributive. The time-dispatch process operations are constrained by the following laws of software.

a) Elicitivity of Timing Event

Law 81. The law of *elicitivity of timing events* in system dispatch states that a common timing event which triggers two different dispatches can be elicited and the two relational processes can be joined, i.e.:

$$(@ t \mathbf{TM} \hookrightarrow P) \mathbb{R} (@ t \mathbf{TM} \hookrightarrow Q) = @ t \mathbf{TM} \hookrightarrow (P \mathbb{R} Q) \quad (91)$$

where $\mathbb{R} \in \{\rightarrow, \parallel, \overset{\circ}{\parallel}, \lll, \ggg\}$.

b) Elicitivity of Process in Timing Dispatches

Law 82. The law of *elicitivity of timing dispatch processes* states that a common process S can be elicited from a pair of time-driven dispatch processes, i.e.:

$$\begin{aligned} S \rightarrow (@ t_1 \mathbf{TM} \hookrightarrow P) \mid S \rightarrow (@ t_2 \mathbf{TM} \hookrightarrow Q) \\ = S \rightarrow (@ t_1 \mathbf{TM} \hookrightarrow P \mid @ t_2 \mathbf{TM} \hookrightarrow Q) \end{aligned} \quad (92)$$

$$\begin{aligned} (@ t_1 \mathbf{TM} \hookrightarrow P) \rightarrow S \mid (@ t_2 \mathbf{TM} \hookrightarrow Q) \rightarrow S \\ = (@ t_1 \mathbf{TM} \hookrightarrow P \mid @ t_2 \mathbf{TM} \hookrightarrow Q) \rightarrow S \end{aligned} \quad (93)$$

c) Symmetry of Timing Dispatches

Law 83. The law of *symmetry of timing dispatching processes* states that a time-driven dispatch is symmetric or commutative, i.e.:

$$@ t_1 \mathbf{TM} \hookrightarrow P \mid @ t_2 \mathbf{TM} \hookrightarrow Q = @ t_2 \mathbf{TM} \hookrightarrow Q \mid @ t_1 \mathbf{TM} \hookrightarrow P \quad (94)$$

d) Skew Symmetry of Timing Dispatches

Law 84. The law of *skew symmetric of timing dispatch* states that a time-driven dispatch is symmetric or commutative on a pair of complemented events, i.e.:

$$@ t \mathbf{TM} \hookrightarrow P \mid @ \bar{t} \mathbf{TM} \hookrightarrow Q = @ \bar{t} \mathbf{TM} \hookrightarrow Q \mid @ t \mathbf{TM} \hookrightarrow P \quad (95)$$

f) Idempotency of Timing Dispatches **Law 85.** The law of *idempotency of timing dispatch* states that a time-driven dispatch can be omitted if it is unconditional, i.e.:

$$@ t \mathbf{TM} \hookrightarrow P \mid @ \bar{t} \mathbf{TM} \hookrightarrow P = P \quad (96)$$

4.1.4 Laws of Event Dispatch Processes

The event dispatch operation of processes at the system level is symmetric and elicitive, but it is dissociative, ireflexive, intransitive, and nondistributive. The event-dispatch process operations are constrained by the following laws of software.

(a) Elicitivity of Operating Event

Law 86. The law of *elicitivity of operating events* in system dispatch states that a common event which triggers two different dispatches can be elicited and the two relational processes can be joined, i.e.:

$$(@ e\mathbf{S} \hookrightarrow P) \mathbb{R} (@ e\mathbf{S} \hookrightarrow Q) = @ e\mathbf{S} \hookrightarrow (P\mathbb{R}Q) \quad (97)$$

where $\mathbb{R} \in \{\rightarrow, \parallel, \text{\textcircled{+}}, \text{\textcircled{||}}, \gg\}$.

b) Elicitivity of Process in Event Dispatches

Law 87. The law of *elicitivity of event dispatch* states that a common process S can be elicited from a pair of event dispatch processes, i.e.:

$$\begin{aligned} S &\rightarrow (@ e_1\mathbf{S} \hookrightarrow P) \mid S \rightarrow (@ e_2\mathbf{S} \hookrightarrow Q) \\ &= S \rightarrow (@ e_1\mathbf{S} \hookrightarrow P \mid @ e_2\mathbf{S} \hookrightarrow Q) \end{aligned} \quad (98)$$

$$\begin{aligned} (@ e_1\mathbf{S} \hookrightarrow P) &\rightarrow S \mid (@ e_2\mathbf{S} \hookrightarrow Q) \rightarrow S \\ &= (@ e_1\mathbf{S} \hookrightarrow P \mid @ e_2\mathbf{S} \hookrightarrow Q) \rightarrow S \end{aligned} \quad (99)$$

c) Symmetry of Event Dispatches

Law 88. The law of *symmetry of event dispatching processes* states that an event-driven dispatch is symmetric or commutative, i.e.:

$$@ e_1\mathbf{S} \hookrightarrow P \mid @ e_2\mathbf{S} \hookrightarrow Q = @ e_2\mathbf{S} \hookrightarrow Q \mid @ e_1\mathbf{S} \hookrightarrow P \quad (100)$$

d) Skew Symmetry of Event Dispatches

Law 89. The law of *skew symmetric of event dispatch* states that an event-driven dispatch is symmetric or commutative on a pair of complemented events, i.e.:

$$@ e\mathbf{S} \hookrightarrow P \mid @ \bar{e}\mathbf{S} \hookrightarrow Q = @ \bar{e}\mathbf{S} \hookrightarrow Q \mid @ e\mathbf{S} \hookrightarrow P \quad (101)$$

f) Idempotency of Event Dispatches

Law 90. The law of *idempotency of event dispatch* states that an event-driven dispatch can be omitted if it is unconditional, i.e.:

$$@ e\mathbf{S} \hookrightarrow P \mid @ \bar{e}\mathbf{S} \hookrightarrow P = P \quad (102)$$

4.1.5 Laws of Interrupt Dispatch Processes

The interrupt dispatch operation of processes at the system level is symmetric and elicitive, but it is dissociative, irreflexive, intransitive, and nondistributive. The interrupt-dispatch process operations are constrained by the following laws of software.

a) Elicitivity of Interrupt Event

Law 91. The law of *elicitivity of interrupt events* in system dispatch states that a common interrupt which triggers two different dispatches can be elicited and the two relational processes can be joined, i.e.:

$$(@ int\textcircled{+} \hookrightarrow P) \mathbb{R} (@ int\textcircled{+} \hookrightarrow Q) = @ int\textcircled{+} \hookrightarrow (P\mathbb{R}Q) \quad (103)$$

where $\mathbb{R} \in \{\rightarrow, \parallel, \text{\textcircled{+}}, \text{\textcircled{||}}, \gg\}$.

b) Elicitivity of Process in Interrupt Dispatches

Law 92. The law of *elicitivity of interrupt dispatch processes* states that a common process S can be elicited from a pair of interrupt-driven dispatch processes, i.e.:

$$\begin{aligned} S &\rightarrow (@ int_1 \odot \hookrightarrow P) | S \rightarrow (@ int_2 \odot \hookrightarrow Q) \\ &= S \rightarrow (@ int_1 \odot \hookrightarrow P | @ int_2 \odot \hookrightarrow Q) \end{aligned} \quad (104a)$$

$$\begin{aligned} (@ int_1 \odot \hookrightarrow P) \rightarrow S | (@ int_2 \odot \hookrightarrow Q) \rightarrow S \\ = (@ int_1 \odot \hookrightarrow P | @ int_2 \odot \hookrightarrow Q) \rightarrow S \end{aligned} \quad (104b)$$

c) Symmetry of Interrupt Dispatches

Law 93. The law of *symmetry of interrupt dispatching processes* states that an interrupt-driven dispatch is symmetric or commutative, i.e.:

$$@ int_1 \odot \hookrightarrow P | @ int_2 \odot \hookrightarrow Q = @ int_2 \odot \hookrightarrow Q | @ int_1 \odot \hookrightarrow P \quad (105)$$

d) Skew Symmetry of Interrupt Dispatches

Law 94. The law of *skew symmetric of interrupt dispatch* states that an interrupt-driven dispatch is symmetric or commutative on a pair of complemented events, i.e.:

$$@ int \odot \hookrightarrow P | @ \overline{int} \odot \hookrightarrow Q = @ \overline{int} \odot \hookrightarrow Q | @ int \odot \hookrightarrow P \quad (106)$$

e) Idempotency of Interrupt Dispatches

Law 95. The law of *idempotency of interrupt dispatch* states that an interrupt-driven dispatch can be omitted if it is unconditional, i.e.:

$$@ int \odot \hookrightarrow P | @ \overline{int} \odot \hookrightarrow P = P \quad (107)$$

5 Conclusions

The exploration on the nature of software and its fundamental behaviors constrained by the laws of mathematics, cognitive informatics, system science, and formal linguistics are a profound effort in computing and software engineering. This paper has presented the mathematical laws of software and fundamental computing behaviors on the basis of the generic mathematical model of programs and RTPA. A comprehensive set of 95 algebraic laws in the categories of meta-processes, process relations, and system compositions has been systematically established, which lays a theoretical foundation for analyzing and modeling software behaviors and software system architectures. Supplementary to the algebraic laws of processes and process relations as presented in this paper, the cognitive informatics laws [33], [34], [36], [40], system laws [39], formal linguistic and semantic laws [6], [36], [39], [43] of software may be referred to given literature.

The applications of the mathematical laws of software and the generic mathematical model of programs have provided new perspectives on software engineering foundations and practices. An RTPA type checker and an RTPA code generator have been

implemented based on the algebraic laws, which automatically generates code in C++ or Java based on a formal system model in RTPA [29], [30]. A number of real-world software systems have been formally modeled in RTPA based on the algebraic laws, such as the telephone switching system [35], [39], the lift dispatching system [49], the real-time operating system [50], and the ATM system [51]. The mathematical laws of software and RTPA are not only useful for rigorously modeling and manipulating software systems, but also widely applied in human cognitive process modeling and computational intelligence [35], [39].

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

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, New York. Addison-Wesley Publication Co, Reading (1985)
2. Baeten, J.C.M., Bergstra, J.A.: *Real Time Process Algebra. Formal Aspects of Computing* 3, 142–188 (1991)
3. Boole, G.: *The Laws of Thought*, Prometheus Books, NY (1854) (reprint, 2003)
4. Boucher, A., Gerth, R.: A Timed Model for Extended Communicating Sequential Processes. In: Ottmann, T. (ed.) *ICALP 1987. LNCS*, vol. 267. Springer, Heidelberg (1987)
5. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys* 17(4), 471–522 (1985)
6. Chomsky, N.: Three Models for the Description of Languages. *I.R.E. Transactions on Information Theory* 2(3), 113–124 (1956)
7. Chomsky, N.: On Certain Formal Properties of Grammars. *Information and Control* 2, 137–167 (1959)
8. Dierks, H.: A Process Algebra for Real-Time Programs. In: Maibaum, T.S.E. (ed.) *ETAPS 2000 and FASE 2000. LNCS*, vol. 1783, pp. 66–76. Springer, Heidelberg (2000)
9. Dijkstra, E.W.: *A Discipline of Programming*. Prentice Hall, Englewood Cliffs (1976)
10. Fecher, H.: A Real-Time Process Algebra with Open Intervals and Maximal Progress. *Nordic Journal of Computing* 8(3), 346–360 (2001)
11. Goguen, J.A., Thatcher, J.W., Wagner, E.G., Wright, J.B.: Initial Algebra Semantics and Continuous Algebras. *Journal of the ACM* 24(1), 59–68 (1977)
12. Higman, B.: *A Comparative Study of Programming Languages*, 2nd edn. MacDonal (1977)
13. Hoare, C.A.R.: An Axiomatic Basis for Computer Programming. *Communications of the ACM* 12(10), 576–580 (1969)
14. Hoare, C.A.R.: Communicating Sequential Processes. *Communications of the ACM* 21(8), 666–677 (1978)
15. Hoare, C.A.R.: *Communicating Sequential Processes*. Prentice-Hall International, London (1985)
16. Hoare, C.A.R., Hayes, I.J., He, J., Morgan, C.C., Roscoe, A.W., Sanders, J.W., Sorensen, I.H., Spivey, J.M., Sufrin, B.A.: *Laws of Programming*. *Communications of the ACM* 30(8), 672–686 (1987)

17. Klusener, A.S.: Abstraction in Real Time Process Algebra. In: de Bakker, J.W., Huizing, C., de Roever, W.P., Rozenberg, G. (eds.) Proc. Real-Time: Theory in Practice, pp. 325–352. Springer, Berlin (1992)
18. Loudon, K.C.: Programming Languages: Principles and Practice. PWS-Kent Publishing Co., Boston (1993)
19. Martin-Lof, P.: An Intuitionistic Theory of Types: Predicative Part. In: Rose, H., Shepherdson, J.C. (eds.) Logic Colloquium 1973. North-Holland, Amsterdam (1975)
20. Milner, R.: A Calculus of Communicating Systems. LNCS, vol. 92. Springer, Heidelberg (1980)
21. McDermid, J.A. (ed.): Software Engineer's Reference Book. Butterworth-Heinemann Ltd., Oxford (1991)
22. Mitchell, J.C.: Type systems for programming languages. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, pp. 365–458. North Holland, Amsterdam (1990)
23. Nicollin, X., Sifakis, J.: An Overview and Synthesis on Timed Process Algebras. In: Proc. 3rd International Computer Aided Verification Conference, pp. 376–398 (1991)
24. Parnas, D.L., Clements, P.C.: A Rational Design Process: How and Why to Fake It. IEEE Trans. on Software Engineering 12(2), 251–257 (1986)
25. Reed, G.M., Roscoe, A.W.: A Timed model for Communicating Sequential Processes. In: Kott, L. (ed.) ICALP 1986. LNCS, vol. 226. Springer, Heidelberg (1986)
26. Scott, D.S., Strachey, C.: Towards a Mathematical Semantics for Computer Languages, Programming Research Group Technical Report PRG-1-6, Oxford University (1971)
27. Schneider, S.A.: An Operational Semantics for Timed CSP, Programming Research Group Technical Report TR-1-91, Oxford University (1991)
28. Stubbs, D.F., Webre, N.W.: Data Structures with Abstract Data Types and Pascal. Brooks/Cole Publishing Co., Monterey (1985)
29. Tan, X., Wang, Y., Ngolah, C.F.: A Novel Type Checker for Software System Specifications in RTPA. In: Proc. 17th Canadian Conference on Electrical and Computer Engineering (CCECE 2004), Niagara Falls, ON, Canada, pp. 1549–1552. IEEE CS Press, Los Alamitos (2004)
30. Tan, X., Wang, Y., Ngolah, C.F.: Design and Implementation of an Automatic RTPA Code Generator. In: Proc. 19th Canadian Conference on Electrical and Computer Engineering (CCECE 2006), Ottawa, ON, Canada, pp. 1605–1608 (May 2006)
31. Tarski, A.: The Semantic Conception of Truth. Philosophic Phenomenological Research 4, 13–47 (1944)
32. Wang, Y.: The Real-Time Process Algebra (RTPA). Annals of Software Engineering: A International Journal 14, 235–274 (2002)
33. Wang, Y.: On Cognitive Informatics (Keynote Speech). In: Proc. 1st IEEE International Conference on Cognitive Informatics (ICCI 2002), Calgary, Canada, pp. 34–42. IEEE CS Press, Los Alamitos (2002)
34. Wang, Y.: On Cognitive Informatics. Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy, USA 4(3), 151–167 (2003)
35. Wang, Y.: Using Process Algebra to Describe Human and Software System Behaviors. Brain and Mind 4(2), 199–213 (2003)
36. Wang, Y.: On the Informatics Laws and Deductive Semantics of Software. IEEE Transactions on Systems, Man, and Cybernetics (C) 36(2), 161–171 (2006)
37. Wang, Y.: Keynote: Cognitive Informatics - Towards the Future Generation Computers that Think and Feel. In: Proc. 5th IEEE International Conference on Cognitive Informatics (ICCI 2006), Beijing, China, pp. 3–7. IEEE CS Press, Los Alamitos (2006)

38. Wang, Y.: Cognitive Informatics and Contemporary Mathematics for Knowledge Representation and Manipulation (Invited Plenary Talk). In: Wang, G.-Y., Peters, J.F., Skowron, A., Yao, Y. (eds.) RSKT 2006. LNCS (LNAI), vol. 4062, pp. 69–78. Springer, Heidelberg (2006)
39. Wang, Y.: Software Engineering Foundations: A Software Science Perspective. CRC Series in Software Engineering, vol. II. Auerbach Publications, Boca Raton (2007)
40. Wang, Y.: The Theoretical Framework of Cognitive Informatics. *International Journal of Cognitive Informatics and Natural Intelligence* 1(1), 1–27 (2007)
41. Wang, Y.: Keynote: On Theoretical Foundations of Software Engineering and Denotational Mathematics. In: Proc. 5th Asian Workshop on Foundations of Software, Xiamen, China, pp. 99–102 (2007)
42. Wang, Y.: On the Big-R Notation for Describing Iterative and Recursive Behaviors. *International Journal of Cognitive Informatics and Natural Intelligence* 2(1), 17–28 (2008)
43. Wang, Y.: Deductive Semantics of RTPA. *International Journal of Cognitive Informatics and Natural Intelligence* 2(2), 95–121 (2008)
44. Wang, Y.: On Concept Algebra: A Denotational Mathematical Structure for Knowledge and Software Modeling. *International Journal of Cognitive Informatics and Natural Intelligence* 2(2), 1–19 (2008)
45. Wang, Y.: On System Algebra: A Denotational Mathematical Structure for Abstract System modeling. *International Journal of Cognitive Informatics and Natural Intelligence* 2(2), 20–42 (2008)
46. Wang, Y.: RTPA: A Denotational Mathematics for Manipulating Intelligent and Computational Behaviors. *International Journal of Cognitive Informatics and Natural Intelligence* 2(2), 44–62 (2008)
47. Wang, Y.: On Laws of Work Organization in Human Cooperation. *International Journal of Cognitive Informatics and Natural Intelligence* 1(2), 1–15 (2008)
48. Wang, Y.: In: On Contemporary Denotational Mathematics for Computational Intelligence, *Transactions of Computational Science*, 2(2), LNCS, vol. 5050. Springer, Heidelberg (August 2008)
49. Wang, Y., Noglah, C.F.: Formal Specification of a Real-Time Lift Dispatching System. In: Proc. 2002 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE 2002), Winnipeg, Manitoba, Canada, pp. 669–674 (May 2002)
50. Wang, Y., Noglah, C.F.: Formal Description of Real-Time Operating Systems using RTPA. In: Proc. 2003 Canadian Conference on Electrical and Computer Engineering (CCECE 2003), Montreal, Canada, pp. 1247–1250. IEEE CS Press, Los Alamitos (2003)
51. Wang, Y., Zhang, Y.: Formal Description of an ATM System by RTPA. In: Proc. 16th Canadian Conference on Electrical and Computer Engineering (CCECE 2003), Montreal, Canada, pp. 1255–1258. IEEE CS Press, Los Alamitos (2003)
52. Wilson, L.B., Clark, R.G.: *Comparative Programming Language*, Wokingham, UK. Addison-Wesley Publishing Co, Reading (1988)
53. Woodcock, J., Davies, J.: *Using Z: Specification, Refinement, and Proof*. Prentice Hall International, London (1996)