Annals of Software Engineering 14, 235–274, 2002
© 2002 Kluwer Academic Publishers. Manufactured in The Netherlands.

The Real-Time Process Algebra (RTPA)

YINGXU WANG wangyx@enel.ucalgary.ca
Theoretical and Empirical Software Engineering Research Center (TESERC), Department of Electrical and Computer Engineering, University of Calgary, 2500 University Drive, NW Calgary, AB, Canada T2N 1N4

Abstract. The real-time process algebra (RTPA) is a set of new mathematical notations for formally describing system architectures, and static and dynamic behaviors. It is recognized that the specification of software behaviors is a three-dimensional problem known as: (i) mathematical operations, (ii) event/process timing, and (iii) memory manipulations. Conventional formal methods in software engineering were designed to describe the 1-D (type (i)) or 2-D (types (i) and (iii)) static behaviors of software systems via logic, set and type theories. However, they are inadequate to address the 3-D problems in real-time systems. A new notation system that is capable to describe and specify the 3-D real-time behaviors, the *real-time process algebra* (RTPA), is developed in this paper to meet the fundamental requirements in software engineering.

RTPA is designed as a coherent software engineering notation system and a formal engineering method for addressing the 3-D problems in software system specification, refinement, and implementation, particularly for real-time and embedded systems. In this paper, the RTPA meta-processes, algebraic relations, system architectural notations, and a set of fundamental primary and abstract data types are described. On the basis of the RTPA notations, a system specification method and a refinement scheme of RTPA are developed. Then, a case study on a telephone switching system is provided, which demonstrates the expressive power of RTPA on formal specification of both software system architectures and behaviors. RTPA elicits and models 32 algebraic notations, which are the common core of existing formal methods and modern programming languages. The extremely small set of formal notations has been proven sufficient for modeling and specifying real-time systems, their architecture, and static/dynamic behaviors in real-world software engineering environment.

Keywords: software engineering, descriptive mathematics, formal methods, real-time systems, algebraic specification, 3-D problems, architecture specification, static behaviors, dynamic behaviors

1. Introduction

The history of sciences and engineering shows that new problems require new forms of mathematics. Software engineering is a new discipline. The problems in software engineering require new mathematical means that are expressive and precise in describing and specifying system designs and solutions. Conventional *analytic mathematics* developed for other sciences and engineering disciplines were adopted in software engineering. However, the unsolved fundamental problems inherited in software engineering are indicating that an *expressive mathematical means* for the description and specification of software system architectures, and static and dynamic behaviors is yet to be sought.

Conventional formal methods were based on logic and set theories [Woodcock and Davies 1996; Derrick and Boiten 2001], which were perceived to be suitable for describing static behaviors of software systems. For describing system dynamic behaviors, a variety of algebra-based technologies were proposed since the 1980's [Hoare 1985; Milner 1989; Baeten and Bergstra 1991; Gerber *et al.* 1992; Klusener 1992; Cerone 2000; Dierks 2000; Fecher 2001]. Algebra is a form of mathematics that simplifies difficult problems by using symbols to represent variables, calculus, and their relations. Algebra enables complicated problems to be expressed and investigated in a formal and rigorous process. Hoare [1985], Milner [1989], and others [Corsetti *et al.* 1991; Nicollin and Sifakis 1991; Jeffrey 1992; Vereijken 1995] developed algebraic ways to represent communicating and concurrent systems, known as process algebra. The *process algebra* is a set of formal notations and rules for describing algebraic relations of software processes. Wang and his colleagues found that the existing work on process algebra and their timed variations [Reed and Roscoe 1986; Boucher and Gerth 1987; Schneider 1991; Wang 2001] can be extended to a new form of expressive mathematics: the *Real-Time Process Algebra* (RTPA) [Wang 2002a, b; Wang and King 2000; Wang *et al.* 2000]. 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.

In RTPA a software system is perceived and described mathematically as a set of coherent processes. A *process* in RTPA is a computational operation that transforms a system from a state to another by changing its inputs, outputs, and/or internal variables. A process can be a single meta-process or a complex process building upon the process combinational rules of RTPA known as process relations.

This paper describes the design and applications of RTPA as a comprehensive and expressive mathematical notation system for software system specification and refinement. This paper is organized in 7 sections. Section 2 describes the structure of RTPA and how it addresses fundamental requirements in real-time software system specification and refinement. Section 3 models a set of meta-processes of RTPA as basic system building blocks, and discusses their types, syntaxes, and semantics. Section 4 develops a set of process relations and rules that can be used to form complex processes based on the meta-processes. On the basis of the real-time process theory, the RTPA method for the specification and refinement of system *architectural components* and *operational components* via three-level refinements are described in section 5. Sections 2–5 develop an easy-to-comprehend and easy-to-use formal method – the real-time process algebra. A case study of RTPA on a telephone switching system is presented in section 6, which demonstrates features and the descriptive power of the RTPA notation system and its specification and refinement method.

2. Structure of the RTPA notation system

This section describes the structure of RTPA as a formal notation system and related basic concepts. The fundamental problems in real-time software system specification and refinement are identified, and the RTPA approach to addressing these problems is described.

There are three fundamental categories of *computational operations* in a software system known as: (a) mathematical operations for variable manipulation, (b) timing operations for event manipulation, and (c) space operations for memory manipulation. Therefore, a software engineering problem, in general, is *a three-dimensional (3-D) function* of mathematical operations, time, and memory, i.e.,

$$\text{Software behavior} = f(\text{mathematical-operations}, \text{time}, \text{memory}). \tag{1}$$

Although some systems may require weak timing or non-dynamic memory allocation such as a transaction processing or word processing system, a software system, in general, is a 3-D real-time system.

Conventional formal methods for software engineering are capable of describing the 1-D (mathematical operations) or 2-D (mathematical operations and memory manipulations) behaviors of systems by using predicate logic, temporal logic, set and type theories [Martin-Lof 1975; Woodcock and Davies 1996; Wang *et al.* 2000; Derrick and Boiten 2001]. However, there is a gap for specifying the 3-D behaviors of real-time software systems in a formal approach. Since it is intuitive that a 3-D method has the descriptive power to specify any 2-D or 1-D problem but not vice versa, the development of a 3-D formal method, RTPA, is theoretically and practically fundamental in software engineering.

Definition 1. *Behavior* of a software system is outcomes and effects of computational operations that affect or change the state of a system in a space of input/output events and variables, as well as internal variables and related memory structures.

A software system behaves in a 3-D state space as described by expression (1). Behaviors of software systems can be classified as static and dynamic ones as described below.

Definition 2. The *static behavior* of a software system is a software behavior that can be determined at design and compile time. The *dynamic behavior* of a software system is a software behavior that can be determined at run-time.

In software engineering, basic requirements for describing and specifying a software system can be considered in two categories: *architectural* components and *operational* components. Corresponding to this classification, system specifications can be described in three subsystems as follows:

- system architecture,
- system static behaviors,
- system dynamic behaviors.

It is found that the above categories and subsystems can be described by a set of real-time processes in RTPA [Wang 2002a, b, c]. The concept of process is defined as follows:

Definition 3. A *process* is a basic unit of software system behaviors that represents a transition procedure of a system from one state to another by changing its sets of inputs {I}, outputs {O}, and/or internal variables {V}.

A process can be a single meta-process or a complex process that is built upon meta-processes by using a set of process combination rules – the process relations.

Definitions 1–3 provide a new perception on software systems as a real-time process system. Based on this, RTPA is developed as an expressive software engineering notation system for specifying the 3-D dynamic behaviors of software. The structure of RTPA can be defined as follows:

$$\begin{aligned} \text{RTPA} \mathrel{\hat{=}}\ & \text{Meta-processes} \\ & \| \text{ Process relations} \\ & \| \text{ System architectures} \\ & \| \text{ Primary types} \\ & \| \text{ Abstract data types} \\ & \| \text{ Specification refinement schemes} \end{aligned} \qquad (2)$$

As shown in expression (2), RTPA is a set of coherent mathematical notations and a formal method for specifying software system architectures, static and dynamic behaviors. RTPA can be used to describe both logical and physical models of a system. Therefore, logical views of the architecture of a software system and its operational platform can be described by using the same RTPA notations for the first time. When the system architecture is formally defined, static and dynamic behaviors that perform on the system architectural models, can be specified by a three-level refinement scheme at the system, class, and detailed levels in a top-down approach.

3. Meta-processes of RTPA

Although CSP [Hoare 1985], the timed CSP [Reed and Roscoe 1986; Boucher and Gerth 1987; Schneider 1991], and other process algebra proposals treated any computational operation as a process, RTPA distinguishes the concepts of meta-processes from complex processes and process relations. A meta-process is an elementary process that serves as a basic building block in a software system. Complex processes can be derived from meta-processes according to given process combinatory rules. This section identifies and elicits a set of 16 RTPA meta-processes, which are essential and primary computing operations commonly identified in existing formal methods and modern programming languages.

3.1. Structure of the RTPA meta-processes

The meta-processes of RTPA are elicited from basic computational operations [Cline 1981; Hoare *et al.* 1987; Wilson and Clark 1988; Wang 2002c]. Any complex process is a combination of the meta-processes. There are 16 fundamental meta-processes identified in RTPA including system control, event/time manipulation, memory manipulation, and I/O manipulation processes. Names, syntaxes, and semantics of the RTPA meta-processes are described in table 1. Operational semantics of each meta-process is provided in table 1 for defining the behaviors of the RTPA meta-processes. Detailed definitions of the RTPA meta-processes will be given in section 3.3.

As shown in table 1, 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 [Martin-Lof 1975]. A set of meta-types of RTPA is provided in the following subsection.

3.2. Types of RTPA

The RTPA notation is strongly typed. That is, every operand in RTPA is assigned with a data type labeled as a bold suffix. As shown in table 1, an operand of a meta-process, x**Type**, can be described by two parts: its value x.Value, and its type x.**Type**, i.e.,

$$\begin{aligned} \mathrm{x}\mathbf{Type} &\hat{=} \mathrm{x} : \mathbf{Type} \\ &= \mathrm{x.Value} \\ &\quad \| \ \mathrm{x}.\mathbf{Type} \end{aligned} \tag{3}$$

where **Type** is any valid data type as defined in the RTPA meta-types in table 2.

RTPA predefined 15 meta-types are shown in table 2. The meta-types #2.1 to #2.10 are primary data types. The meta-types date/time (#2.11) are special types for continuous real-time systems, where long-range timing manipulation is needed. The run-time determinable type**RT** (#2.12) is a subset of all the rest meta-types defined in table 2, which is designed to support flexible type specification that is unknown at compile-time, but will be instantiated at run-time. The system architectural type**ST** (#2.13) is a novel and important data type in RTPA that models system architectural components and is going to be described in section 6.1. The event and status types are used to model system event variables @e**S** (#2.14) as a string type, and system status variables Ⓢs**BL** (#2.15) as a Boolean type.

In addition to the meta-types for system modeling, a set of 10 typical and frequently used combinational data objects in system architectural modeling, the abstract data types (ADTs) and their allowable operations, are selected and predefined in RTPA as shown in table 3.

The ADTs, which are developed recursively by using the RTPA notation and meta-types, are a coherent part of the RTPA notation system. Users may use the ADTs and their designed behaviors in system specifications as those of the meta-types. Users of RTPA can directly use and invoke the ADTs and related operations as predefined notations.

Table 1
RTPA meta-processes.

No.	Meta-process	Syntax	Operational semantics
1.1	System	§(SysID**S**)	§(SysID**S**) represents a system, SysID, identified by a string(**S**)

1.2	Assignment	y**Type** := x**Type**	if x.type = y.type
			then x.value ⇒ y.value
			else !(@AssignmentTypeError**S**),
			where run-time determinable type
			Type = {Meta-Types}

1.3	Addressing	ptr**Pˆ** := x**Type**	if ptr.type = x.type
			then x.value ⇒ ptr.value
			else !(@AddressingTypeError**S**),
			where **Type** = {**H**, **Z**, **Pˆ**}

1.4	Input	Port(ptr**Pˆ**)**Type** $\vert>$ x**Type**	if Port(ptr**Pˆ**).type = x.type
			then Port(ptr**Pˆ**).value ⇒ x.value
			else !(@InputTypeError**S**),
			where **Type** = {**B**, **H**}, **Pˆ** = {**H**, **N**, **Z**}

1.5	Output	x**Type** $\vert<$ Port(ptr**Pˆ**)**Type**	if Port(ptr**Pˆ**).type = x.type
			then x.value ⇒ Port(ptr**Pˆ**).value
			else !(@OutputTypeError**S**),
			where **Type** = {**B**, **H**}, **Pˆ** = {**H**, **N**, **Z**}

1.6	Read	Mem(ptr**Pˆ**)**Type** > x**Type**	if Mem(ptr**Pˆ**).type = x.type
			then Mem(ptr**Pˆ**).value ⇒ x.value
			else !(@ReadTypeError**S**),
			where **Type**={**B**, **H**}, **Pˆ** ={**H**, **N**, **Z**}

1.7	Write	x**Type** < Mem(ptr**Pˆ**)**Type**	if Mem(ptr**Pˆ**).type = x.type
			then x.value ⇒ Mem(ptr**Pˆ**).value
			else !(@WriteTypeError**S**),
			where **Type**={**B**, **H**}, **Pˆ** = {**H**, **N**, **Z**}

1.8	Timing	a) @t**hh:mm:ss:ms**
		:= §t**hh:mm:ss:ms**
		b) @t**yy:MM:dd** := §t**yy:MM:dd**
		c) @ t**yy:MM:dd:hh:mm:ss:ms**
		:= §t**yy:MM:dd:hh:mm:ss:ms**
			if @t.type = §t.type
			then §t.value ⇒ @t.value
			else !(@TimingTypeError**S**),
			where **yy** ∈ {00, . . . , 99}, **MM** ∈ {01, . . . , 12},
			dd ∈ {01, . . . , 31}, **hh** ∈ {00, . . . , 23},
			mm, ss ∈ {00, . . . , 59}, **ms** ∈ {000, . . . , 999}

1.9	Duration	@t_n**Z** := §t_n**Z**+ Δn**Z**	if §t_n.type = Δn.type = @t_n.type = **Z** then (§t_n.value + Δn.value) mod MaxValue ⇒ @t_n.value else !(@RelativeTimingTypeError**S**), where MaxValue = the upper bound of the system relative-clock, and the unit of all values is **ms**

Table 1
(Continued.)

No.	Meta-process	Syntax	Operational semantics
1.10	Memory allocation	AllocateObject (ObjectID**S**, NofElements**N**, ElementType**RT**)	n**N** := NofElements**N** $\rightarrow \overset{n}{\underset{i=1}{R}}$ (**new** ObjectID(i**N**) : ElementType**RT**) → Ⓢ ObjectID.Existed**BL** := **T**
1.11	Memory release	ReleaseObject (ObjectID**S**)	**delete** ObjectID**S**// System.Garbage Collection( ) → ObjectID**S** := null → Ⓢ ObjectID.Released**BL** := **T**
1.12	Increase	↑(n**Type**)	if n.value < MaxValue then n.value +1 ⇒ n.value else !(@ValueOutofRange**S**), where **Type** = {**N**, **Z**, **B**, **H**, **P^**}, MaxValue = min{run-time defined upper bound, nature upper bound of **Type**}
1.13	Decrease	↓(n**Type**)	if n.value > 0 then n.value −1 ⇒ n.value else !(@ValueOutofRange**S**), where **Type** = {**N**, **Z**, **B**, **H**, **P^**}.
1.14	Exception detection	!(@e**S**)	↑(ExceptionLogPtr**P^**) → @e**S**⇒ Mem(ExceptionLogPtr**P^**)**S**
1.15	Skip	∅	Exit a current control structure, such as loop, branch, or switch.
1.16	Stop	⊠	System stop

3.3. Description of the RTPA meta-processes

The syntaxes and semantics of the RTPA meta-processes have been summarized in table 1. This subsection provides a set of formal definitions of RTPA meta-processes, which serves as further description of the meta-processes, and their functions and relations.

3.3.1. System

Definition 4. The *system* is a meta-process that acts at the highest level of a process system for dispatching and/or executing a specific process according to system timing or predefined events. A system process is denoted by

$$\S(\text{SysID}\mathbf{S}) \tag{4}$$

where § is the system and (SysID**S**) is a string identity of the system. The operational semantics of system is given in table 1 (#1.1).

Table 2
RTPA meta-types.

No.	Meta-type	Syntax
2.1	Natural number	**N**
2.2	Integer	**Z**
2.3	Real	**R**
2.4	String	**S**
2.5	Boolean	**BL**, **BL** = {**T**, **F**}
2.6	Byte	**B**
2.7	Hexadecimal	**H**
2.8	Pointer	**P**ˆ
2.9	Time	**hh:mm:ss:ms** where **hh** ∈ {00, ..., 23}, **mm, ss** ∈ {00, ..., 59}, **ms** ∈ {000, ..., 999}
2.10	Date	**yy:MM:dd** where **yy** ∈ {00, ..., 99}, **MM** ∈ {01, ..., 12}, **dd** ∈ {01, ..., 31}
2.11	Date/Time	**yyyy:MM:dd: hh:mm:ss:ms** where **yyyy** ∈ {0000, ..., 9999}, **MM** ∈ {01, ..., 12}, **dd** ∈ {01, ..., 31}, **hh** ∈ {00, ..., 23}, **mm**, **ss** ∈ {00, ..., 59}, **ms** ∈ {000, ..., 999}
2.12	Run-time determinable type	**RT**
2.13	System architectural type	**ST**
2.14	Event	@e**S**
2.15	Status	Ⓢs**BL**

3.3.2. Assignment

Definition 5. *Assignment* is a meta-process that transfers x.Value to y.Value, when x.**Type** = y.**Type**. An assignment is denoted by

$$y\textbf{Type} := x\textbf{Type} \tag{5}$$

where **Type** is one of the RTPA meta-types as defined in table 2, and x**Type** can be a constant that matches y.**Type**. The operational semantics of assignment is given in table 1 (#1.2).

3.3.3. Addressing

Definition 6. *Addressing* is a meta-process that assigns x.Value to a pointer ptr**P**ˆ. An addressing is denoted by

$$ptr\textbf{P}\hat{} := x\textbf{Type} \tag{6}$$

where **Type** = {**P**ˆ, **H**, **N**, **Z**}. The operational semantics of addressing is given in table 1 (#1.3).

3.3.4. Input

Definition 7. *Input* is a meta-process that receives data x**Type** from a given system I/O port Port(ptr**P**ˆ), where ptr**P**ˆ is a pointer that identifies the physical address of the port

Table 3
RTPA abstract data types.

No.	ADT	Syntax	Designed behaviors
3.1	Stack	**Stack** : **ST**	**Stack.**{Create, Push, Pop, Clear, EmptyTest, FullTest, Release}
3.2	Record	**Record** : **ST**	**Record.**{Create, fieldUpdate, Update, FieldRetrieve, Retrieve, Release}
3.3	Array	**Array** : **ST**	**Array.**{Create, Enqueue, Serve, Clear, EmptyTest, FullTest, Release}
3.4	Queue (FIFO)	**Queue** : **ST**	**Queue.**{Create, Enqueue, Serve, Clear, EmptTest, FullTest, Release}
3.5	Sequence	**Sequence** : **ST**	**Sequence.**{Create, Retrieve, Append, Clear, EmptyTest, FullTest, Release}
3.6	List	**List** : **ST**	**List.**{Create, FindNext, FindPrior, Findith, FindKey, Retrieve, Update, InsertAfter, InsertBefore, Delete, CurrentPos, FullTest, EmptyTest, SizeTest, Clear, Release}
3.7	Set	**Set** : **ST**	**Set.**{Create, Assign, In, Intersection, Union, Difference, Equal, Subset, Release}
3.8	File (Sequential)	**SeqFile** : **ST**	**SeqFile.**{Create, Reset, Read, Append, Clear, EndTest, Release}
3.9	File (Random)	**RandFile** : **ST**	**RanFile.**{Create, Reset, Read, Write, Clear, EndTest, Release}
3.10	Binary Tree	**BTree** : **ST**	**BTree.**{Create, Traverse, Insert, DeleteSub, Update, Retrieve, Find, Characteristics, EmptyTest, Clear, Release}

interface. An input process is denoted by

$$\text{Port}(\text{ptr}\mathbf{P}\hat{\ })\mathbf{Type} \mid\gtrdot \text{x}\mathbf{Type} \tag{7}$$

where $\mathbf{Type} = \{\mathbf{B}, \mathbf{H}\}$. The operational semantics of input is given in table 1 (#1.4).

3.3.5. Output

Definition 8. *Output* is a meta-process that sends data x**Type** to a given system I/O port Port(ptr**P**ˆ), where ptr**P**ˆ is a pointer that identifies the physical address of the port interface. An *output* process is denoted by

$$\text{x}\mathbf{Type} \mid\lessdot \text{Port}(\text{ptr}\mathbf{P}\hat{\ })\mathbf{Type} \tag{8}$$

where $\mathbf{Type} = \{\mathbf{B}, \mathbf{H}\}$. The operational semantics of *output* is given in table 1 (#1.5).

3.3.6. Read

Definition 9. *Read* is a meta-process that gets data x**Type** from a given memory location Mem(ptr**P**ˆ), where ptr**P**ˆ is a pointer that identifies the physical memory address. A read process is denoted by

$$\text{Mem}(\text{ptr}\mathbf{P}\hat{\ })\mathbf{Type} \gtrdot \text{x}\mathbf{Type} \tag{9}$$

where $\mathbf{Type} = \{\mathbf{B}, \mathbf{H}\}$. The operational semantics of read is given in table 1 (#1.6).

3.3.7. Write

Definition 10. *Write* is a meta-process that puts data $x\textbf{Type}$ to a given memory location Mem(ptr$\textbf{P}$ˆ), where ptr$\textbf{P}$ˆ is a pointer that identifies the physical memory address. A write process is denoted by

$$x\textbf{Type} \lessdot \text{Mem}(\text{ptr}\textbf{P}\hat{}\,)\textbf{Type} \tag{10}$$

where $\textbf{Type} = \{\textbf{B}, \textbf{H}\}$. The operational semantics of write is given in table 1 (#1.7).

3.3.8. Timing

Definition 11. *Timing* is a meta-process that sets the value of a timing variable @t as the absolute time of the current system clock §t. A timing process is denoted by one of the following expressions depending on the need of time range for a system:

$$@t\textbf{hh:mm:ss:ms} := \S t\textbf{hh:mm:ss:ms} \tag{11a}$$

$$@t\textbf{yy:MM:dd} := \S t\textbf{yy:MM:dd} \tag{11b}$$

$$@t\textbf{yy:MM:dd:hh:mm:ss:ms} := \S t\textbf{yy:MM:dd:hh:mm:ss:ms} \tag{11c}$$

where expressions (11a), (11b) and (11c) provide timing ranges from 0 ms to 23 hours, 0 day to 99 years, or 0 ms to 99 years, respectively. The operational semantics of timing is given in table 1 (#1.8).

3.3.9. Duration

Definition 12. *Duration* is a meta-process that sets a relative time $@t_n\textbf{Z}$ as an integer based on the relative system clock $\S t_n\textbf{Z}$ and the given period $\Delta n\textbf{Z}$. A duration process is denoted by

$$@t_n\textbf{Z} := \S t_n\textbf{Z} + \Delta n\textbf{Z} \tag{12}$$

where the unit of all relative timing variables is **ms**. The operational semantics of duration is given in table 1 (#1.9).

3.3.10. Memory allocation

Definition 13. *Memory allocation* is a meta-process that collects a memory block named ObjectID**S** accommodating the number of elements NofElements**N** in type ElementType**RT**. A memory allocation process is denoted by

$$\text{AllocateObject}(\text{ObjectID}\textbf{S}, \text{NofElements}\textbf{N}, \text{ElementType}\textbf{RT}) \tag{13}$$

Memory allocation is a key meta-process for dynamic memory manipulation in RTPA. The operational semantics of memory allocation is given in table 1 (#1.10).

3.3.11. Memory release

Definition 14. *Memory release* is a meta-process that returns a memory block allocated to ObjectID**S**. A memory release process is denoted by

$$\text{ReleaseObject}(\text{ObjectID}\mathbf{S}) \tag{14}$$

The released memory block of ObjectID**S** will then be collected by the system garbage management mechanism provided by an operating system. The operational semantics of memory release is given in table 1 (#1.11).

3.3.12. Increase

Definition 15. *Increase* is a meta-process that adds one to a given variable n**Type**, where **Type** $= \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{H}, \mathbf{P\hat{}}\}$. An increase process is denoted by

$$\uparrow(\text{n}\mathbf{Type}) \tag{15}$$

The operational semantics of increase is given in table 1 (#1.12).

3.3.13. Decrease

Definition 16. *Decrease* is a meta-process that subtracts one from a given variable n**Type**, where **Type** $= \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{H}, \mathbf{P\hat{}}\}$. A decrease process is denoted by

$$\downarrow(\text{n}\mathbf{Type}) \tag{16}$$

The operational semantics of decrease is given in table 1 (#1.13).

3.3.14. Exception detection

Definition 17. *Exception detection* is a meta-process that logs a detected exception event @e**S** at run-time. An exception detection process is denoted by

$$!(@\text{e}\mathbf{S}) \tag{17}$$

The RTPA exceptional detection mechanism is a fundamental process for safety and dependable system specification, which enables system exception detection, handling, or postmortem analysis to be implemented. The operational semantics of exception detection is given in table 1 (#1.14).

3.3.15. Skip

Definition 18. *Skip* is a meta-process that exits a current control structure, such as loop, branch, or switch. A skip process is denoted by

$$\varnothing \tag{18}$$

The operational semantics of skip is given in table 1 (#1.15).

3.3.16. Stop

Definition 19. *Stop* is a meta-process that terminates a system's operation. A stop process is denoted by

$$\boxtimes \tag{19}$$

The operational semantics of stop is given in table 1 (#1.16).

4. Process relations of RTPA

The meta-processes of RTPA developed in section 3 identified a set of essential elements for modeling a software system. It is interesting to realize that there is only a small set of 16 meta-processes in software system modeling. However, via the combination of a number of meta-processes, any architecture and behavior of software systems, particularly the 3-D real-time systems, can be sufficiently described [Higman 1977; Hoare *et al.* 1987; Wilson and Clark 1988; Wang and King 2000]. This section elicits a set of fundamental process relations for building and composing complex processes in the context of real-time software systems.

4.1. Structure of RTPA process relations

The combination rules of meta-processes in RTPA are governed by a set of algebraic process relations as described in table 4 with definitions of their syntaxes and semantics. The rationale of the selection of the 16 process relations is explained below.

The first three process relations in table 4, sequential (#4.1), branch (#4.2, #4.3), and iteration (#4.4–#4.6), have long been identified as the basic control structures (BCSs) of software architectures [Hoare *et al.* 1987; Wilson and Clark 1988]. To represent the modern programming structural concepts, CSP [Hoare 1985] identified the following seven additional process relations: function call (#4.7), recursion (#4.8), parallel (#4.9), concurrency (#4.10), interleave (#4.11), pipeline (#4.12), and jump (#4.16).

RTPA [Wang 2002a, b, c] extends the BCSs and process relations to time-driven dispatch (#4.13), event-driven dispatch (#4.14), and interrupt (#4.15). The 16 process relations (BCSs) are regarded as the foundation of programming and system architectural design, because any complex process can be combinatory implemented by the basic process relations as shown in table 4.

4.2. Description of RTPA process relations

This subsection defines and explains the process relational operations of RTPA for manipulating relationships and combinational rules between meta-processes. The RTPA relational operators, such as sequence, branch, parallel, iteration, interrupt, and recursion, as shown in table 4, provide the rules to form combinatorial processes from meta-processes.

Table 4
RTPA process relations.

No.	Process relation	Syntax	Operational semantics
4.1	Sequence	$\mathrm{P} \rightarrow \mathrm{Q}$	P;
			Q
4.2	Branch	$(?\ \mathrm{exp}\mathbf{BL} = \mathbf{T}) \rightarrow \mathrm{P}$	if exp**BL** $=$ **T**
		$\mid\ ?\sim\ \rightarrow \mathrm{Q}$	then P
		where "$\sim$" means "exp**BL** $=$ **F**"	else Q
4.3	Switch	?exp**Type** $=$	case exp**Type** $=$
		$0 \rightarrow \mathrm{P}_0$	0: P_0
		$\mid 1 \rightarrow \mathrm{P}_1$	1: P_1
		$\mid \ldots$	$\ldots$
		$\mid n-1 \rightarrow \mathrm{P}_{n-1}$	$n-1$: P_{n-1}
		$\mid$ else $\rightarrow \varnothing$	else: exit
		where exp**Type** $= \{\mathbf{N}, \mathbf{Z}, \mathbf{B}, \mathbf{S}\}$	where exp**Type** is a numerical type
4.4	For-do	$\underset{i=1}{\overset{n}{R}}\,(\mathrm{P}(i))$	for $i := 1$ to n
			do $\mathrm{P}(i)$
4.5	Repeat	$\underset{\geqslant 1}{\overset{\mathrm{exp}\mathbf{BL} \neq \mathbf{T}}{R}}\ (\mathrm{P})$	repeat P
			until exp**BL** $\neq$ **T**
4.6	While-do	$\underset{\geqslant 0}{\overset{\mathrm{exp}\mathbf{BL} \neq \mathbf{T}}{R}}\ (\mathrm{P})$	while exp**BL** $=$ **T**
			do P
4.7	Function call	$\mathrm{P} \Lsh \mathrm{F}$	$\mathrm{P}' \rightarrow \mathrm{F} \rightarrow \mathrm{P}''$, where $\mathrm{P} = \mathrm{P}' \cup \mathrm{P}''$
4.8	Recursion	$\mathrm{P} \circlearrowleft \mathrm{P}$	$\mathrm{P}' \rightarrow \mathrm{P} \rightarrow \mathrm{P}''$, where $\mathrm{P} = \mathrm{P}' \cup \mathrm{P}''$
4.9	Parallel	$\mathrm{P} \parallel \mathrm{Q}$	MPSC (multi-processor single clock),
			internal parallel
4.10	Concurrence	$\mathrm{P} \oiint \mathrm{Q}$	MPMC (multi-processor multi-clock),
			external parallel
4.11	Interleave	$\mathrm{P} \mid\mid\mid \mathrm{Q}$	SPSC (single-processor single clock),
			internal virtual parallel
4.12	Pipeline	$\mathrm{P} \gg \mathrm{Q}$	$\mathrm{P} \rightarrow \mathrm{Q}$ and $\{\mathrm{P}_{\mathrm{outputs}}\} \Rightarrow \{\mathrm{Q}_{\mathrm{inputs}}\}$
4.13	Time-driven	@t_i **hh:mm:ss:ms**	@t_1 **hh:mm:ss:ms** $\Lsh \mathrm{P}_1$
	dispatch	$\Lsh \mathrm{P}_i,\ i \in \{1, \ldots, n\}$	$\mid$@t_2**hh:mm:ss:ms** $\Lsh \mathrm{P}_2$
			$\mid \ldots$
			$\mid$@t_n**hh:mm:ss:ms** $\Lsh \mathrm{P}_n$
4.14	Event-driven	@e_i**S** $\Lsh \mathrm{P}_i,\ i \in \{1, \ldots, n\}$	@e_1**S** $\Lsh \mathrm{P}_1$
	dispatch		$\mid$@e_2**S** $\Lsh \mathrm{P}_2$
			$\mid \ldots$
			$\mid$@e_n**S** $\Lsh \mathrm{P}_n$
4.15	Interrupt	$\mathrm{P} \parallel \odot($@e**S**$\nearrow \mathrm{Q} \searrow \odot)$	$\mathrm{P} \parallel$ system interrupt capture;
			if @e**S** captured $=$ **T**
			then (record interrupt point $\odot$ and variables
			$\Lsh \mathrm{Q}$
			$\rightarrow$ recover interrupted variables
			$\rightarrow$ return to the interrupt point $\odot$ and
			continue P)
4.16	Jump	$\mathrm{P} \rightarrow \mathrm{Q}$	$\mathrm{P} \rightarrow$ goto $\mathrm{Q} \rightarrow \mathrm{Q}$

4.2.1. Sequence

Definition 20. *Sequence* is a process relation in which two or more processes are executed one by one. A relational operator, $\rightarrow$, is adopted to denote the sequential relation between processes. Assuming two processes, P and Q, are sequential, their relation can be expressed as follows:

$$\mathrm{P} \rightarrow \mathrm{Q} \tag{20}$$

The operational semantics of the sequence process relation is given in table 4 (#4.1).

4.2.2. Branch

Definition 21. *Branch* is a process relation in which the selection of a process is determined by a conditional expression exp**BL**. A branch (if-then-[else]) process relation can be denoted by

$$\begin{aligned} &(?\,\mathrm{exp}\mathbf{BL} = \mathbf{T}) \rightarrow \mathrm{P} \\ &\quad |\,?\sim \rightarrow \mathrm{Q} \\ &) \end{aligned} \tag{21}$$

where "$\sim$" means "exp**BL** = **F**," or more general, "otherwise." When the *else* branch is optional, expression (21) is equivalent to

$$\begin{aligned} &(?\,\mathrm{exp}\mathbf{BL} = \mathbf{T}) \rightarrow \mathrm{P} \\ &\quad |\,?\sim \rightarrow \varnothing \\ &) \end{aligned}$$

The operational semantics of the branch process relation is given in table 4 (#4.2).

4.2.3. Switch

Definition 22. *Switch* is a process relation in which the branch is determined by a numerical expression exp**Type**. A switch (case) process relation can be denoted by

$$\begin{aligned} &?\,\mathrm{exp}\mathbf{Type} = \\ &\quad 0 \rightarrow \mathrm{P}_0 \\ &\quad |\,1 \rightarrow \mathrm{P}_1 \\ &\quad |\,\ldots \\ &\quad |\,n-1 \rightarrow \mathrm{P}_{n-1} \\ &\quad |\,\mathrm{else} \rightarrow \varnothing \end{aligned} \tag{22}$$

where exp**Type** = {**N**, **Z**, **B**}. The operational semantics of the switch process relation is given in table 4 (#4.3).

4.2.4. For-do

Definition 23. *For-do* is a process relation in which a simple process or a combinatorial process, P(i), is executed repeatedly for n times controlling by an index i, $i \in \{1, \ldots, n\}$. A for-do process relation can be denoted by:

$$\underset{i=1}{\overset{n}{R}} (\mathrm{P}(i)) \tag{23}$$

where the R (big-R) denotes a repeat operation indexed by i, with the lower bound as 1 and upper bound n.

The *big-R notation* is a new mathematical calculus for iteration specification, which has a similar mechanism as that of $\sum_{i=1}^{n} x(i)$. The operational semantics of the for-do process relation is given in table 4 (#4.4).

4.2.5. Repeat

Definition 24. *Repeat* is a process relation in which a simple process or a combinatorial process, P, is executed iteratively for at least one time until the conditional expression exp**BL** is no longer true. A repeat process relation can be denoted by the big-R notation as follows:

$$\underset{\geqslant 1}{\overset{\mathrm{exp}\mathbf{BL}\neq\mathbf{T}}{R}} (\mathrm{P}) \tag{24}$$

where the lower bound of iteration, $\geqslant$1, denotes that P will be repeated at lease one time; the upper bound, exp**BL** $\neq$ **T**, shows the condition to terminate the iteration.

Repeat is a special case of the for-do process relation, where the termination condition of iteration will be determined at run-time by a Boolean conditional expression. The operational semantics of the repeat process relation is given in table 4 (#4.5).

4.2.6. While-do

Definition 25. *While-do* is a process relation in which a simple process or a combinatorial process, P, is executed repeatedly as long as the conditional expression exp**BL** is true. A while-do process relation can be denoted by the big-R notation as follows:

$$\underset{\geqslant 0}{\overset{\mathrm{exp}\,\mathbf{BL}\neq\mathbf{T}}{R}} (\mathrm{P}) \tag{25}$$

where the lower bound, $\geqslant$0, denotes that P may or may not be iterated at run-time if exp**BL** $\neq$ **T** at the beginning. The operational semantics of the while-do process relation is given in table 4 (#4.6).

4.2.7. Function call

Definition 26. *Function call* is a process relation in which a process P calls another process F as a predefined subprocess. A function call process relation can be defined as follows:

$$\mathrm{P} \looparrowright \mathrm{F} \tag{26}$$

In expression (26), the called process F can be regarded as an embedded part of process P. The operational semantics of the function-call process relation is given in table 4 (#4.7).

4.2.8. Recursion

Definition 27. *Recursion* is a process relation in which a process P calls itself. The recursion process relation can be denoted as follows:

$$\mathrm{P} \circlearrowleft \mathrm{P} \tag{27}$$

Recursion processes are frequently used in programming to simplify system structures and to specify neat and provable system functions. It is particularly useful when an infinite or run-time determinable specification has to be clearly expressed.

For example, a simple everlasting clock, *CLOCK*, which does nothing but tick, i.e.,

$$CLOCK \mathrel{\hat{=}} tick \rightarrow tick \rightarrow tick \rightarrow \cdots$$

can be recursively described simply as follows:

$$CLOCK \mathrel{\hat{=}} tick \circlearrowleft CLOCK$$

The operational semantics of the recursion process relation is given in table 4 (#4.8).

4.2.9. Parallel

Definition 28. *Parallel* is a process relation in which two or more processes are executed simultaneously, synchronized by a common system clock. Assuming two processes, P and Q, are synchronous parallel between each other, their parallel relation can be denoted as follows:

$$\mathrm{P} \parallel \mathrm{Q} \tag{28}$$

The parallel process relation is designed to model behaviors of a multi-processor single-clock (MPSC) system as shown in table 4 (#4.9). The operational semantics of parallel may also be extended to denote relations between system architectural concepts that are functionally parallel or equivalent. Details will be shown in section 5.2.

4.2.10. Concurrence

Definition 29. *Concurrence* is a process relation in which two or more processes are executed simultaneously and asynchronously according to separate system clocks, and each such process is executed as a complete task. Assuming two processes, P and Q, are concurrent processes, their concurrent relation can be denoted as follows:

$$\mathrm{P} \oiint \mathrm{Q} \tag{29}$$

The concurrent process relation is designed to model behaviors of a multi-processor multi-clock (MPMC) system as shown in table 4 (#4.10).

4.2.11. Interleave

Definition 30. *Interleave* is a process relation in which two or more processes are executed simultaneously, synchronized by a common system clock, while the execution of each such process would be interrupted by other process(es). Assuming two processes, P and Q, are interleaved processes, the interleave relation can be expressed as follows:

$$\mathrm{P} \mid\mid\mid \mathrm{Q} \tag{30}$$

The interleave process relation is designed to model behaviors of a single-processor single-clock (SPSC) system as shown in table 4 (#4.11).

4.2.12. Pipeline

Definition 31. *Pipeline* is a process relation in which two or more processes are interconnected to each other, and the succeeding process takes the output(s) of the previous process as its input(s). Assuming two processes, P and Q, are pipelined, their pipeline relation can be denoted as follows:

$$\mathrm{P} \gg \mathrm{Q} \tag{31}$$

The operational semantics of the pipeline process relation is given in table 4 (#4.12).

4.2.13. Time-driven dispatch

Definition 32. *Time-driven dispatch* is a process relation in which the ith process P_i is triggered by a predefined system time $@\mathrm{t}_i$**hh:mm:ss:ms**. A time-driven dispatch process relation can be denoted as follows:

$$@\mathrm{t}_i\mathbf{hh{:}mm{:}ss{:}ms} \Lsh \mathrm{P}_i, \quad i \in \{1, \ldots, n\} \tag{32}$$

The operational semantics of the time-driven dispatch process relation is given in table 4 (#4.13).

4.2.14. Event-driven dispatch

Definition 33. *Event-driven dispatch* is a process relation in which the ith process P_i is triggered by a predefined system event $@\mathrm{e}_i\mathbf{S}$. An event-driven dispatch process relation can be denoted as follows:

$$@\mathrm{e}_i\mathbf{S} \Lsh \mathrm{P}_i, \quad i \in \{1, \ldots, n\} \tag{33}$$

The operational semantics of the event-driven dispatch process relation is given in table 4 (#4.14).

4.2.15. Interrupt

Definition 34. *Interrupt* is a process relation in which a running process is temporarily held before termination by another higher priority process, and the interrupted process will be resumed when the high priority process has been completed. Assuming process P is interrupted by process Q on interrupt event @e**S** at interrupt point $\odot$, an interrupt relation can be denoted as follows:

$$\mathrm{P}\| \odot (@\mathrm{e}\mathbf{S} \nearrow \mathrm{Q} \searrow \odot) \tag{34}$$

The interrupt relation describes execution priority and control taking-over between processes. The operational semantics of the interrupt process relation is given in table 4 (#4.15).

4.2.16. Jump

Definition 35. *Jump* is a process relation in which, on the termination of a process P, the system skips the ordinary execution sequence of processes, and invokes a specific given process Q. A skip process relation can be denoted as follows:

$$\mathrm{P} \rightarrowtail \mathrm{Q} \tag{35}$$

The operational semantics of the jump process relation is given in table 4 (#4.16).

5. Specification and refinement of software systems by RTPA

In a well-designed formal method, complicated system specifications should be carried out via a number of systematic refinements in a top-down approach by using a set of coherent notations. On the basis of the RTPA real-time process notations developed in sections 2–4, this section describes the RTPA specification and refinement methods for system architectures, and static and dynamic behaviors via three-level refinements.

5.1. System specification and refinement in RTPA

In RTPA three fundamental aspects of software systems can be described and specified by a coherent set of mathematical notations, i.e.,

$$\begin{aligned}\S(\mathrm{SysID}\mathbf{S}) \triangleq\ & \mathrm{SysID}\mathbf{S}.\mathrm{Architecture}\\ & \|\ \mathrm{SysID}\mathbf{S}.\mathrm{StaticBehaviors}\\ & \|\ \mathrm{SysID}\mathbf{S}.\mathrm{DynamicBehaviors}\end{aligned} \tag{36}$$

The specification of each of the above subsystems, in terms of system architecture, system static behaviors, or system dynamic behaviors, can be implemented by a three-level refinement process at the system, class, and detailed levels as shown in figure 1. Figure 1 provides a strategic scheme of system specification and refinement in RTPA. Figure 1 also shows the defined work products of each specification subsystem at different refinement levels.

In the RTPA specification and refinement scheme, a new concept, the *component logical model* (CLM), is introduced, which is a special architectural component for describing the abstract logical models of system hardware and system control mechanisms. A CLM can be defined as follows:

Refinement $\rightarrow$ $\downarrow$ Specification	**R1. System-Level Specification**	**R2. Class-Level Specification**	**R3. Detailed-Level Specification**
S1. System Architecture	**1.1 System architecture** SysID$\mathbf{S}$.Architecture $\triangleq$ CLM$_1\mathbf{S}$ [n$_1\mathbf{N}$] || CLM$_2\mathbf{S}$ [n$_2\mathbf{N}$] || … || CLM$_k\mathbf{S}$ [n$_k\mathbf{N}$]	**1.2 CLM schemas** CLMSchema $\triangleq$ CLM-ID(i$\mathbf{N}$): (Field$_1$: type$_1$ | constraint$_1$>, Field$_2$: type$_2$ | constraint$_2$>, … Field$_n$: type$_n$ | constraint$_n$>)	**1.3 CLM objects** CLMObject $\triangleq$ CLMSchema$\mathbf{ST}$ || ObjectID$\mathbf{S}$ || {InstanceParameters} || {InitialValues}
S2. Static Behaviors	**2.1 System static behaviors** SysID$\mathbf{S}$.StaticBehaviors $\triangleq$ SysInitial || Process$_1$ || Process$_2$ || … || Process$_n$	**2.2 Process schemas** ProcessSchema $\triangleq$ PN$\mathbf{N}$ // process number || ProcessID$\mathbf{S}$ ({$\mathbf{I}$}; {$\mathbf{O}$}) || {OperatedCLMs} || {RelatedProcesses} || FunctionDescription$\mathbf{S}$	**2.3 Process implementation** ProcessImplementation $\triangleq$ ProcessSchema$\mathbf{ST}$ || ProcessInstID$\mathbf{S}$ || {DetailedProcesses}

S3. Dynamic Behaviors	**3.1 System dynamic behaviors**
	SysID$\mathbf{S}$.DynamicBehaviors $\triangleq$
	|| {Base-level processes}
	|| {High-level processes}
	|| {Low-interrupt-level processes}
	|| {High-interrupt-level processes}
		3.2 Process deployment
		ProcessDeployment $\triangleq$ § $\rightarrow$
		(BaseTimeEvent ↳ {ProcessSet$_1$}
		| HighLevelTimeEvent ↳ {Process set$_2$}
		| LowIntTimeEvent ↳ {Process set$_3$}
		| HighIntTimeEvent ↳ {Process set$_4$}
		)
		$\rightarrow$ §
			3.3 Process dispatch
			ProcessDispatch $\triangleq$ § $\rightarrow$
			(Event$_1$ ↳ {ProcessSet$_1$}
			| Event$_2$ ↳ {ProcessSet$_2$}
			| …
			| Event$_n$ ↳ {ProcessSet$_n$}
			)
			$\rightarrow$ §

Figure 1. The scheme of system specification and refinement by RTPA.

Definition 36. A *component logical model* (CLM) is an 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.

The three refinement steps for system architecture specification (S1 in figure 1) are: system architecture, CLM schemas, and CLM objects. Similarly, the refinement strategy for system static behavior specification (S2) is: system static behaviors, process schemas, and process implementations. System dynamic behaviors (S3) can be specified by: system dynamic behaviors, process deployment, and process dispatch, in a three-level refinement. Detailed explanations and illustrations of the RTPA scheme for system specification and refinement will be given in section 6 via a real-world case study.

5.2. System architecture description by RTPA

There are four types of system meta-architectures known as: *parallel*, *serial*, *pipeline*, and *nested*. Any complicated system architecture can be represented by a combination of these four meta-architectures between components. It is interesting to find that each of the meta-architectures corresponding to a key RTPA process relation as defined in table 4. Therefore, for the first time, not only system behaviors (operations), but also system architectures can be expressed by the same set of formal notations in RTPA.

For example, the left-hand side of figure 3 shows the architecture of a sample system §(SysA$\mathbf{S}$). It can be seen that §(SysA$\mathbf{S}$) consists of serial, parallel, and nested meta-architectures. Therefore, the architecture of §(SysA$\mathbf{S}$) can be formally specified by using RTPA as shown in the right-hand side of figure 3.

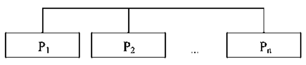
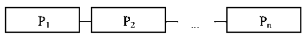
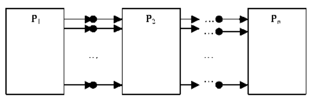
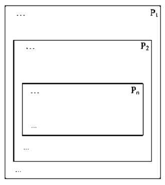
No.	Type of Architecture	Syntax	Examples
1	Parallel	$P \parallel Q$	$\S(\text{ParallelSys}\mathbf{S}) \triangleq P_1 \parallel P_2 \parallel \ldots \parallel P_n$ 
2	Serial	$P \rightarrow Q$	$\S(\text{SerialSys}\mathbf{S}) \triangleq P_1 \rightarrow P_2 \rightarrow \ldots \rightarrow P_n$ 
3	Pipeline	$P \gg Q$	$\S(\text{PipelinedSys}\mathbf{S}) \triangleq P_1 \gg P_2 \gg \ldots \gg P_n$ 
4	Nested	$P \looparrowright Q$	$\S(\text{NestedSys}\mathbf{S}) \triangleq P_1 \looparrowright P_2 \looparrowright \ldots \looparrowright P_n$ 

Figure 2. RTPA meta-architectures.

The formal architectural description of a real-world system example with hardware and software architectures will be demonstrated in section 6.1.

6. A case study: Specification of a telephone switching system by using RTPA

Because a formal software engineering method is designed to solve real-world software system problems, it is found that case studies are essential in both methodology demonstration and evaluation. This section presents a case study on RTPA applications in the formal specification of a telephone switching system (TSS). The functional structure of the TSS system can be divided into four subsystems: call processing, subscriber, route, and signaling as shown in figure 4.

The TSS system consists of 1 call processor and 16 subscribers. There are 5 switching routes and a set of 5 signaling trunks. The call processor uses a number of component logical models (CLMs), such as 16 call records, 16 line scanners, and 16 digits receivers, to control the 16 subscribers.

The RTPA scheme for system specification and refinement has been defined in figure 1. As shown in figure 1, there are three essential subsystems in a system specification: system architecture, static behaviors, and dynamic behaviors. The following sub-

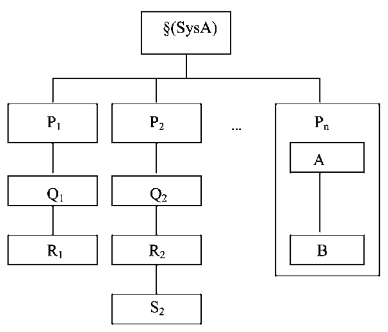
System Architecture	RTPA Expression
 §(SysA) P_1 P_2 ... P_n A Q_1 Q_2 R_1 R_2 B S_2	$\S(\text{SysA}) \triangleq \S \rightarrow (\ (P_1 \rightarrow Q_1 \rightarrow R_1) \ \| \ (P_2 \rightarrow Q_2 \rightarrow R_2 \rightarrow S_2) \ \| \ \ldots \ \| \ (P_n \Lsh (A \rightarrow B)) \ )$

Figure 3. The architecture of a sample system.

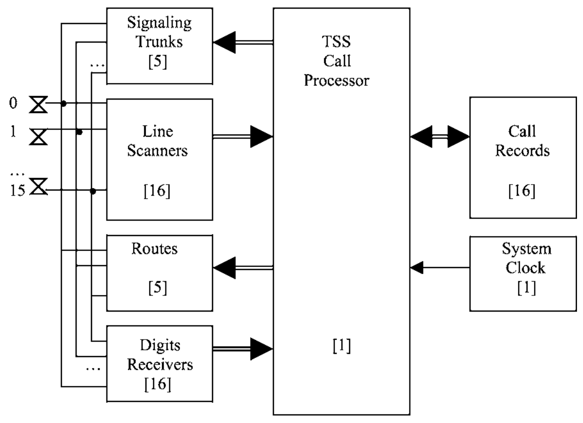


Figure 4. Functional structure of the TSS system.

sections describe the TSS system according to the specification and refinement scheme of RTPA.

6.1. *Specification of the TSS system architecture*

According to expression (36), the top-level specification of the TSS system can be described as follows:

$$\begin{aligned} \S(\text{TSS}) \triangleq\ & \text{TSS.Architecture} \\ & \|\ \text{TSS.StaticBehaviors} \\ & \|\ \text{TSS.DynamicBehaviors} \end{aligned} \tag{37}$$

This subsection describes the specification and refinement of the TSS architectural subsystem. Other subsystems will be developed in sections 6.2 and 6.3.

6.1.1. TSS system architecture

System architecture, at the top level, specifies a list of names of CLMs and their relations. A CLM can be regarded as a predefined class of system hardware or internal control models, which can be inherited or implemented by corresponding CLM objects as specific instants in the succeeding system architectural refinement procedures.

Corresponding to the functional structure of TSS as shown in figure 4, the high-level specification of the architecture of TSS in RTPA is as follows:

$$
\begin{aligned}
\text{TSS.Architecture} \hat{=}\ & \text{CallProcessingSubsys} \\
& \| \ \text{SubscriberSubsys} \\
& \| \ \text{RouteSubsys} \\
& \| \ \text{SignalingSubsys} \\
=\ & (\text{CallProcessor}\mathbf{ST}[1] && \text{// specified by the system static/dynamic} \\
& && \text{// behaviors} \\
& \quad \| \ \text{SysClock}\mathbf{ST}[1] \\
& \quad \| \ \text{CallRecords}\mathbf{ST}[16] \\
& \ ) \\
& \| \ (\text{Sunscribers}\mathbf{ST}[16] && \text{// status represented by the line scanners} \\
& && \text{// and call records} \\
& \quad \| \ \text{LineScanners}\mathbf{ST}[16] \\
& \quad) \\
& \| \ \text{Routes}\mathbf{ST}[5] \\
& \| \ (\text{DigitsReceivers}\mathbf{ST}[16] \\
& \quad \| \ \text{SignalingTrunks}\mathbf{ST}[5] \\
& \quad)
\end{aligned}
\tag{38}
$$

where the number in the square brackets, [n**N**], specifies the required number for a CLM in a system's architectural configuration.

Expression (38) provides the first-step refinement of the architectural specification of the TSS system represented by TSS.Architecture. As a result of the first step refinement, the system's *architectural components* and their relationships are clearly specified. The RTPA method for specifying the system's *operational components* will be described in sections 6.2 and 6.3.

6.1.2. The CLM schemas of TSS

Definition 37. A *CLM schema* is an abstract logical structure of a component logical model in the form of a record that specifies the configuration of a CLM by a set of fields, types, and constraints, as follows:

$$\begin{aligned} \text{CLMSchema}\mathbf{ST} \mathrel{\hat{=}} \text{CLM} - \text{ID}\mathbf{S}(\text{i}\mathbf{N})\text{: } (\\ & \text{Field}_1 : \text{type}_1|\text{constraint}_1>, \\ & \text{Field}_2 : \text{type}_2|\text{constraint}_2>, \\ & \vdots \\ & \text{Field}_n : \text{type}_n|\text{constraint}_n>) \end{aligned} \tag{39}$$

The CLM schemas of the TSS system are further refinements of TSS.Architecture as developed in section 6.1.1. As specified in expression (38), there are 6 CLM schemas in TSS. Therefore, the second step refinement of TSS.Architecture can be carried out as shown in table 5. The RTPA big-R notation is adopted in table 5 to denote the implementation of multiple instances of a CLM schema.

A CLM schema can be treated as the architectural specification of a class, which will be used as a blueprint in further refinement of the CLM objects as an instance in implementing the CLM classes in next step.

6.1.3. The CLM objects of TSS

Definition 38. A *CLM object* in RTPA is a derived instance of a CLM schema and its detailed implementation, i.e.,

$$\begin{aligned} \text{CLMObject}\mathbf{ST} \mathrel{\hat{=}} & \text{CLMSchema}\mathbf{ST} \\ & \| \text{ ObjectID}\mathbf{S} \\ & \| \text{ \{InstanceParameters\}} \\ & \| \text{ \{InitialValues\}} \end{aligned} \tag{40}$$

The CLM objects are results of the final refinement of the specification of the TSS system architecture TSS.Architecture. After the three-step refinement known as system architecture, CLM schemas, and CLM objects, all architectural components, their relations and implementations are obtained systematically.

The following subsections describe the RTPA methodology for detailed system architectural specification – CLM object refinement. The six architectural components of the TSS system, as specified in table 5, are precisely refined and implemented according to expression (40).

Line scanners. The RTPA specification of the architectures of *line scanners* is given in figure 5. Figure 5 shows there are 16 line scanners in TSS that share the same architectural control model of "LineScanners**ST**" as developed in section 6.1.2. At this level of refinement, port addresses are assigned within the range of the specification in the schemas, and initial values of control variables are given that ensure the system enters a valid initial state when it is started.

Digits receivers. The RTPA specification of the architectures of *digits receivers* is given in figure 6. Figure 6 shows there are 16 digits receivers in TSS that share the same architectural control model of "DigitsReceivers**ST**" as developed in section 6.1.2.

Table 5
Specification of the schemas of TSS component logical models (CLMs).

CLM	Schemas of RTPA specification
1. Line scanners	LineScanners**ST** $\hat{=}\ \overset{15}{\underset{i=0}{R}}$ (LineScanner(i**N**): <Status: **N** | Status**N** = {(0, Idle), (1, HookOff), (2, Busy), (3, HookOn), (4, Invalid)}>, <PortAddress: **H** | FF00**H** $\leqslant$ PortAddress**H** $\leqslant$ FF0F**H**>, <ScanInput: **B** | ScanInputB = <xxxx xxxbB>, <CurrentScan: **BL** | **T** = hook-off $\wedge$ **F** = hook-on>, <LastScan: **BL** | **T** = hook-off $\wedge$ **F** = hook-on>)
2. Digits receivers	DigitsReceivers**ST** $\hat{=}\ \overset{15}{\underset{i=0}{R}}$ (DigitsReceiver(i**N**): <Status: **N** | Status**N**= {(0, NoDial), (1, DialStarted), (2, Dialing), (3, DialCompleted)}>, <DigitPort: **H** | FF10**H** $\leqslant$ DigitPort**H** $\leqslant$ FF1F**H**>, <DigitInput: **B** | DigitInput**B** = <xxxx bbbb**B**>, <StatusPort: **H** | FF20**H** $\leqslant$ StatusPort**H** $\leqslant$ FF2F**H**>, <StatusInput: **B** | StatusInput**B** = <xxxx xxxb**B**>, <Digit1: **N** | 0 $\leqslant$ Digit1**N** $\leqslant$ 9 >, <Digit2: **N** | 0 $\leqslant$ Digit2**N** $\leqslant$ 9 >, <NumberOfDigitsReceived: **N** | 1 $\leqslant$ NumberOfDigitsReceived**N** $\leqslant$ 2 >)
3. Routes	Routes**ST** $\hat{=}\ \overset{4}{\underset{i=0}{R}}$ (Route(i**N**): <Status: **BL** | **T** = Busy $\wedge$ **F** = Free>, <CallingNum: **N** | 0 $\leqslant$ CallingNum**N** $\leqslant$ 15>, <CalledNum: **N** | 0 $\leqslant$ CalledNum**N** $\leqslant$ 15>)
4. Signal trunks	SignalTrunks**ST**<SignalTrunkPort: **H** | FF90**H** $\leqslant$ SignalTrunkPort**H** $\leqslant$ FF94**H**>
5. System clock	SysClock**ST** $\hat{=}$ <§t: **N** | 0 $\leqslant$ §t**N** $\leqslant$ 1M> || <CurrentTime: **hh**:**mm**:**ss**:**ms** | 00:00:00:00 $\leqslant$ CurrentTime**hh**:**mm**:**ss**:**ms** $\leqslant$ 23:59:59:99> || <MainClockPort: **B** | MainClockPort**B** = F1**H**>, || <ClockInterval: **N** | ClockInterval**N** = 1**ms**>, || <ClockIntCounter: **N** | 0 $\leqslant$ ClockIntCounter**N** $\leqslant$ 999>
6. Call records	CallRecords**ST** $\hat{=}\ \overset{15}{\underset{i=0}{R}}$ (CallRecord(i**N**): <CallStatus: **BL** | **T** = Active $\wedge$ **F** = Inactive>, <CallProcess: **N** | CallProcess**N** = {(0, Idle), (1, CallOrigination), (2, Dialing), (3, CheckCalledStatus), (4, Connecting), (5, Talking), (6, CallTermination), (7, ExceptionalTermination)}>, <CalledNum: **N** | 0 $\leqslant$ CalledNum**N** $\leqslant$ 15>, <RouteNum: **N** | 0 $\leqslant$ CalledNum**N** $\leqslant$ 4>, <Timer: **N** | 0 $\leqslant$ Timer**N** $\leqslant$ 100**ms**>, <CallingTermination: **BL** | **T** = Yes $\wedge$ **F** = No>, <CalledTermination: **BL** | **T** = Yes $\wedge$ **F** = No>)

LineScannersST $\triangleq \underset{i=0}{\overset{15}{R}}$ (LineScanner(i**N**):

<Status : **N** | Status**N** = {<0, Idle>, <1, HookOff>, <2, Busy>, <3, HookOn>, <4, Invalid>}>,
<PortAddress : **H** | FF00**H** ≤ PortAddress**H** ≤ FF0F**H**>,
<ScanInput : **B** | ScanInputB = <xxxx xxxbB>,
<CurrentScan : **BL** | **T** = hook-off ∧ **F** = hook-on>,
<LastScan : **BL** | **T** = hook-off ∧ **F** = hook-on>
)

= **LineScanner(0):** <Status**N**, PortAddress**H**, ScanInput**B**, CurrentScan**BL**, LastScan**BL**>
:= <0, FF00**H**, 0000 000b**B**, **F**, **F**>

|| **LineScanner(1):** <Status**N**, PortAddress**H**, ScanInput**B**, CurrentScan**BL**, LastScan**BL**>
:= <0, FF01**H**, 0000 000b**B**, **F**, **F**>

|| ...

|| **LineScanner(15):** <Status**N**, PortAddress**H**, ScanInputB, CurrentScan**BL**, LastScan**BL**>
:= <0, FF0F**H**, 0000 000b**B**, **F**, **F**>

Figure 5. Specification of the architecture of line scanners in RTPA.

DigitsReceiversST $\triangleq \underset{i=0}{\overset{15}{R}}$ (DigitsReceiver(i**N**):

<Status : **N** | Status**N** = {(0, NoDial), (1, DialStarted), (2, Dialing), (3, DialCompleted)}>,
<DigitPort : **H** | FF10**H** ≤ DigitPort**H** ≤ FF1F**H**>,
<DigitInput : **B** | DigitInput**B** = <xxxx bbbb**B**>,
<StatusPort : **H** | FF20**H** ≤ StatusPort**H** ≤ FF2F**H**>,
<StatusInput : **B** | StatusInput**B** = <xxxx xxxb**B**>,
<Digit1: **N** | 0 ≤ Digit1**N** ≤ 9>,
<Digit2: **N** | 0 ≤ Digit2**N** ≤ 9>,
<NumberOfDigitsReceived: **N** | 1 ≤ NumberOfDigitsReceived**N** ≤ 2>
)

= **DigitsReceiver(0):** < Status**N** , DigitPort**H**, DigitInput**B**, StatusPort**H** , StatusInput**B**, Digit1**N**, Digit2**N**>
:= <0, FF10**H**, xxxx bbbb**B**, FF20**H**, xxxx xxxb**B**, 0, 0, 0>

|| **DigitsReceiver(1):** < Status**N** , DigitPort**H**, DigitInput**B**, StatusPort**H** , StatusInput**B**, Digit1**N**, Digit2**N**>
:= <0, FF11**H**, xxxx bbbb**B**, FF21**H**, xxxx xxxb**B**, 0, 0, 0>

|| ...

|| **DigitsReceiver(15):** < Status**N** , DigitPort**H**, DigitInput**B**, StatusPort**H** , StatusInput**B**, Digit1**N**, Digit2**N**>
:= <0, FF1F**H**, xxxx bbbb**B**, FF2F**H**, xxxx xxxb**B**, 0, 0, 0>

Figure 6. Specification of the architecture of digital receivers in RTPA.

Routes. The RTPA specification of the architectures of *routes* is given in figure 7. Figure 7 shows there are five switching routes in TSS that share the same architectural control model of "Routes**ST**" as developed in section 6.1.2.

Signaling trunks. The RTPA specification of the architectures of *signal trunks* is given in figure 8. Figure 8 shows that there are five signaling trunks in TSS that share the same architectural control model of "SignalTrunks**ST**" as developed in section 6.1.2.

System clock. The RTPA specification of the architecture of *system clock* is given in figure 9. Figure 9 shows that SysClock**ST** provides both an absolute clock (§t**hh:mm:ss:ms**) and a relative clock (§t_n**N**). The system clock is driven by an external oscillating signal from port MainClockPort**B** = 00F1**H** with an interval of 1 ms.

As defined in expression (11), a long-range absolute SysClock**ST** may be specified, if needed, by §t**yyyy:MM:dd:hh:mm:ss:ms**.

$$\textbf{RoutesST} \triangleq \mathop{R}_{i=0}^{4} (\text{Route}(i\textbf{N}):$$

<Status : **BL** | **T** = Busy ∧ **F** = Free>,
<CallingNum : **N** | 0 ≤ CallingNum**N** ≤ 15>,
<CalledNum : **N** | 0 ≤ CalledNum**N** ≤ 15>
)
= **Route(0):** <Status**BL**, CallingNum**N**, CalledNum**N**> := < **F**, x, x>
|| **Route(1):** <Status**BL**, CallingNum**N**, CalledNum**N**> := < **F**, x, x>
|| **Route(2):** <Status**BL**, CallingNum**N**, CalledNum**N**> := < **F**, x, x>
|| **Route(3):** <Status**BL**, CallingNum**N**, CalledNum**N**> := < **F**, x, x>
|| **Route(4):** <Status**BL**, CallingNum**N**, CalledNum**N**> := < **F**, x, x>

Figure 7. Specification of the architecture of routes in RTPA.

SignalTrunksST ≜ <SignalTrunkPort : **H** | FF90**H** ≤ SignalTrunkPort**H** ≤ FF94**H**>
= <DialTonePort : **H** | DialTonePort**H** = FF90**H**>
|| <BusyTonePort : **H** | BusyTonePort**H** = FF91**H**>
|| <RingingTonePort : **H** | RingingTonePort**H** = FF92**H**>
|| <RingBackTonePort : **H** | RingBackTonePort**H** = FF93**H**>
|| <SpecialTonePort : **H** | SpecialTonePort**H** = FF94**H**>

Figure 8. Specification of the architecture of signaling trunks in RTPA.

SysClockST ≜ <§t_n : **N** | 0 ≤ §t_n**N** ≤ 1M>
|| <§t : **hh:mm:ss:ms** | 00:00:00:00 ≤ §t**hh:mm:ss:ms** ≤ 23:59:59:99>
|| <MainClockPort : **B** | MainClockPort**B** = 00F1H>,
|| <ClockInterval : **N** | ClockInterval**N** = 1**ms**>,
|| <ClockIntCounter : **N** | 0 ≤ ClockIntCounter**N** ≤ 999>

Figure 9. Specification of the architecture of system clock in RTPA.

Call records. The RTPA specification of the architectures of *call records* is given in figure 10. Figure 10 shows there are 16 call records in TSS that share the same architectural control model of "S = CallRecords**ST**" as developed in section 6.1.2.

The system architectural specification developed in this subsection provides a set of abstract object models and clear interfaces between system hardware and software. It is noteworthy that the first five CLMs are logical models of system hardware, while the last one, CallRecords**ST**, is an architectural model of internal system control structures. By reaching this point, the co-design of a real-time system can be separately carried out by hardware and software teams.

It is recognized that system *architecture specification* by the means of CLMs is a fundamental and the most difficult part in software system modeling, while conventional formal methods hardly provide any support for this purpose. From the above examples in this subsection, it can be seen that RTPA provides a set of expressive notations for specifying system architectural structures and control models, including hardware, software, and their interactions. On the basis of the system architecture specification and with the work products of system architectural components (CLMs), specification of the operational components of the TSS system can be carried out directly forward, as shown in the following sections.

```
                 15
CallRecordsST ≙  R  (CallRecord(iN):
                i=0
                     <CallStatus : BL | T = Active ∧ F = Inactive>,
                     <CallProcess : N | CallProcessN = {(0, Idle), (1, CallOrigination), (2, Dialing),
                                        (3, CheckCalledStatus), (4, Connecting), (5, Talking), (6, CallTermination),
                                        (7, ExceptionalTermination)}>,
                     <CalledNum : N | 0 ≤ CalledNumN ≤ 15>,
                     <RouteNum : N  | 0 ≤ CalledNumN ≤ 4>,
                     <Timer : N | 0 ≤ TimerN ≤ 100ms>,
                     <CallingTermination : BL | T = Yes ∧ F = No>,
                     <CalledTermination : BL | T = Yes ∧ F = No>
                     )
 =  CallRecord (0):  <CallStatusBL, CallProcessN, CalledNumN, RouteNumN, TimerN, CallingTerminationBL,
                      CalledTerminationBL := <F, 0, 0, 0, 0, F, F>
  || CallRecord (1): <CallStatusBL, CallProcessN, CalledNumN, RouteNumN, TimerN, CallingTerminationBL,
                      CalledTerminationBL := <F, 0, 0, 0, 0, F, F>
  || ...
  || CallRecord (15): <CallStatusBL, CallProcessN, CalledNumN, RouteNumN, TimerN, CallingTerminationBL,
                      CalledTerminationBL := <F, 0, 0, 0, 0, F, F>
```

Figure 10. Specification of the architecture of call record in RTPA.

6.2. *Specification of system static behaviors*

System static behaviors, as defined in definition 2, are valid operations of system that can be determined at compile-time. This section describes how the TSS static behaviors are specified by three-step refinements: system static behaviors, process schemas, and process implementation, as defined in figure 1.

6.2.1. The TSS static behaviors

System static behaviors describe the high-level configuration of processes of a system and their relations. The TSS static behaviors consist of six processes as specified below:

$$\begin{aligned}\text{TSS.StaticBehaviors} = {} & \text{SysInitial}\\ & \| \text{ SysClock}\\ & \| \text{ LineScanning}\\ & \| \text{ DigitsReceiving}\\ & \| \text{ ConnectDrive}\\ & \| \text{ CallProcessing}\end{aligned} \tag{41}$$

In expression (41), *CallProcessing* is a complex core process in TSS that consists of seven subprocesses as follows:

$$\begin{aligned}\text{TSS.StaticBehaviors.CallProcessing} \mathrel{\hat{=}} {} & \text{CallOrigination}\\ & \| \text{ Dialling}\\ & \| \text{ CheckCalledStatus}\\ & \| \text{ Connecting}\\ & \| \text{ Talking}\end{aligned}$$

$$\begin{aligned} &\parallel \text{CallTermination} \\ &\parallel \text{ExceptionalTermination} \end{aligned} \qquad (42)$$

In the following subsections, the seven parallel call processing subprocesses as specified in expression (42) will be taken as examples to demonstrate the refinement of the TSS static behaviors.

6.2.2. TSS process schemas

As a result of the first-step refinement in the previous subsection, system static behaviors have been described as a set of process names and their relations. The second-step refinement of system static behaviors in RTPA is to specify the schemas of these identified processes as defined in figure 1.

Definition 39. A *process schema* is the structure of a process that identifies the process by a process number PN**N** and a process name ProcessID**S**, lists operated CLMs and relations with other processes, and describes brief functions of the process, as follows:

$$\begin{aligned} \text{ProcessSchema}\mathbf{ST} \hat{=} \ &\text{PN}\mathbf{N} \\ &\parallel \text{ProcessID}\mathbf{S}(\{\mathbf{i}\}; \{\mathbf{o}\}) \\ &\parallel \{\text{OperatedCLMs}\} \\ &\parallel \{\text{RelatedProcesses}\} \\ &\parallel \text{FunctionDescription}\mathbf{S} \end{aligned} \qquad (43)$$

where FunctionDescription**S** is a brief description of major functions of a process, which will be used to guide further refinement of the process.

Following the above generic definition, a set of process schemas is developed as shown in table 6. The process schemas of TSS provide further detailed information on each process functionality, I/O, and its relationships with system architectural components (CLMs) and other processes.

6.2.3. TSS process implementation

The third-step refinement of system static behaviors is to extend the process schemas as specified in section 6.2.2 into detailed processes. This level of specification for system static behaviors is called process implementation.

Definition 40. *Process implementation* is the final-step refinement of static behaviors of a system that extends a process schema to a detailed process by using meta-processes, process relations, and related CLMs provided in RTPA, i.e.,

$$\begin{aligned} \text{ProcessImplementation}\mathbf{ST} \hat{=} \ &\text{ProcessSchemas}\mathbf{ST} \\ &\parallel \text{ProcessInstID}\mathbf{S} \\ &\parallel \{\text{DetailedProcesses}\}) \end{aligned} \qquad (44)$$

Table 6
Specification of the TSS process schemas.

PN	ProcessID**S**({**X**}; {**O**})	Operated CLMs	Related processes	Functional descriptions
1	CallOrigination ({**X**:: LineNum**N**}; {**O**:: CallProcess**N**})	• LineScanners**ST** • CallRecords**ST**	• LineScanning • ConnectDrive	• Find hook-off subscribers from LineScanners**ST** • Record originated calls in CallRecords**ST**
2	Dialing ({**X**:: LineNum**N**}; {**O**:: CallProcess**N**})	• DigitsReceivers**ST** • CallRecords**ST**	• DigitsReceiving • ConnectDrive	• Receive digits from DigitsReceivers**ST** • Record called number in CallRecords**ST**
3	CheckCalledStatus ({**X**:: LineNum**N**}; {**O**:: CallProcess**N**})	• LineScanners**ST** • CallRecords**ST** • Routes**ST**	• LineScanning • ConnectDrive	• Check called status from callRecords**ST** • Find route from Routes**ST** • Send busy tone to calling if called's busy
4	Connecting ({**X**:: LineNum**N**}; {**O**:: CallProcess**N**})	• CallRecords**ST**	• ConnectDrive	• Send RingbBackTone to calling • Send RingingTone to called
5	Talking ({**X**:: LineNum**N**}; {**O**:: CallProcess**N**})	• LineScanners**ST** • CallRecords**ST** • Routes**ST**	• LineScanning • ConnectDrive	• When called answered, connect calling-called using pre-seized routes in CallRecords**ST** • Process calling give-up • Monitor call termination
6	CallTermination ({**X**:: LineNum**N**}; {**O**:: CallProcess**N**})	• LineScanners**ST** • CallRecords**ST** • Routes**ST**	• LineScanning • ConnectDrive	• Process either party termination based on LineScanners**ST** • Release routes according to Routes**ST** • Monitor non-hook-on party in CallRecords**ST**
7	ExceptionalTermination ({**X**:: LineNum**N**}; {**O**:: CallProcess**N**})	• LineScanners**ST** • CallRecords**ST**	• LineScanning • ConnectDrive	• Reset line status in LineScanners**ST**, if monitored party hook-on • If time-out, set line status invalid in LineScanners**ST**

Based on the refined specifications, code can be derived easily and rigorously, and tests of the code can be generated prior to the coding phase. This subsection describes the technology of RTPA for detailed process specification. The seven static behavioral components of the TSS system, as specified in table 6, will be precisely refined, by referring to related specifications of CLMs developed in section 6.1.

CPN1: Call origination process. As defined in expression (42), the TSS call processing processes were divided into seven finite state processes. Each of them is only responsible to a limited and timely continuous operation, in order to guarantee the sys-

```
CallOrigination ({I:: LineNumN}; {O:: CallProcessN}) ≙
{ // CallProcessN := 1
    iN := CallProcessN
      → LineScanner(iN).StatusN := 2                          // Show line busy
      ↳ ConnectDrive (SubscriberLine(iN)N, SignalDialToneN, OnBL)
      → CallRecord(iN).TimerSS := 5                           // Set no dial timer
      → CallRecord(iN).CallStatusBL = T                       // Set call record active
      → CallRecord(iN).CallProcessN := 2                      // To dialing
}
```

Figure 11. Detailed specification of TSS call processing behaviors in RTPA.

```
Dialing ({I:: LineNumN}; {O:: CallProcessN}) ≙
{ // CallProcessN := 2
    iN := CallProcessN
    → (  ( @ DigitsReceiver(iN).StatusN := 0                        // No dial
            → (  ? CallRecord(iN).TimerSS := 0                      // No dial time-out
                  ↳ ConnectDrive (SubscriberLine(iN)N, SignalDialToneN, OffBL)
                  ↳ ConnectDrive (SubscriberLine(iN)N, SignalBusyToneN, OnBL)
                  → CallRecord(iN).TimerN := 10
                  → CallRecord(iN).CallProcessN := 7                // To exceptional termination
          )
        | ( @ DigitsReceiver(iN).StatusN = 1                        // Dial started
            → ( CallRecord(iN).TimerSS := 10                        // Set dial time-out timer
            ↳ ConnectDrive (SubscriberLine(iN)N, SignalDialToneN, OffBL)
            → DigitsReceiver(iN).StatusN := 2
          )
        | ( @ DigitsReceiver(iN).StatusN = 2                        // Dialing
            → (  ? CallRecord(iN).TimerSS := 0                      // Dialing time-out
            ↳ ConnectDrive (SubscriberLine(iN)N, SignalBusyToneN, OnBL)
            → CallRecord(iN).CallingTerminationBL := T)
            → CallRecord(iN).TimerN := 10
            → CallRecord(iN).CallProcessN := 7                      // To exceptional termination
          )
        | ( @ DigitsReceiver(iN).StatusN = 3                        // Dial completed
            → ( CalledNumN := DigitsReceiver (iN).Digit1N * 10 +
                              DigitsReceiver (iN).Digit2N
            → CallRecord(iN).CalledNumN := CalledNumN
            → CallRecord(iN).CallProcessN := 3                      // To check called status
          )
        | ( @ ~                                                     // Otherwise
            → ∅
          )
      )
}
```

Figure 12. Detailed specification of TSS dialing behaviors in RTPA.

tem timing between complicated processes. This is a core technology for implementing multi-thread processes in real-time systems.

Based on the schema developed in section 6.2.2, PN1, the refinement of the *call origination* process can be carried out as shown in figure 11.

CPN2: Dialing process. Based on the schema developed in section 6.2.2, PN2, the refinement of the *dialing* process can be carried out as shown in figure 12.

```
CheckCalledStatus ({I:: LineNumN}; {O:: CallProcessN}) ≙
{ // CallProcessN := 3
   iN := CallProcessN
                  → ( CalledNumN := CallRecord(iN).CalledNumN
                  → ( @ LineScanner(CalledNumN).StatusN = 2 ∨
                        LineScanner(CalledNumN).StatusN = 1 ∨
                        LineScanner(CalledNumN).StatusN = 4          // Busy, hook-off, or invalid
                     → ( CallRecord(iN).TimerSS := 10                // Set busy tone timer
                          ↳ ConnectDrive (SubscriberLine(iN)N, SignalBusyToneN, OnBL)
                          → CallRecord(iN).CallingTerminationBL := T)
                          → CallRecord(iN).TimerN := 10
                          → CallRecord(iN).CallProcessN := 7         // To exceptional termination
                        )
                  | ( @ LineScanner(CalledNumN).StatusN = 0 ∨
                        LineScanner(CalledNumN).StatusN = 3          // Idle or hook-on
                     → LineScanner(CalledNumN).StatusN := 2          // Seize the line
                     → ⓢRouteFoundBL := F                            // To seize a route
                     → $\mathop{R}\limits_{j=0}^{4}$ ( ? Route(iN).StatusBL = T
                                → RouteNumN := jN
                                → ⓢRouteFoundBL := T
                                → ∅
                            )
                     → ( ? ⓢRouteFoundBL := T
                            → CallRecord(iN).RouteNumN := RouteNumN
                            → CallRecord(iN).CallProcessN := 4       // Connecting
                        | ?~
                            ↳ ConnectDrive(SubscriberLine(iN)N, SignalBusyToneN, OnBL)
                            → LineScanner(CalledNumN).StatusN := 0   // Release called line
                            → CallRecord(iN).CallingTerminationBL := T
                            → CallRecord(iN).TimerN := 10
                            → CallRecord(iN).CallProcessN := 7       // To exceptional termination
                        )
                     )
               )
}
```

Figure 13. Detailed specification of TSS check called status behaviors in RTPA.

CPN3: Check call status process. Based on the schema developed in section 6.2.2, PN3, the refinement of the *check call status* process can be carried out as shown in figure 13.

CPN4: Connecting process. Based on the schema developed in section 6.2.2, PN4, the refinement of the *connecting* process can be carried out as shown in figure 14.

CPN5: Talking process. Based on the schema developed in section 6.2.2, PN5, the refinement of the *talking* process can be carried out as shown in figure 15.

CPN6: Call termination process. Based on the schema developed in section 6.2.2, PN6, the refinement of the *call termination* process can be carried out as shown in figure 16.

```
Connecting ({I:: LineNumN}; {O:: CallProcessN}) ≙
{ // CallProcessN := 4
    iN := CallProcessN
    → CalledNumN := CallRecord(iN).CalledNumN
    → RouteNumN := CallRecord(iN).RouteNumN
    ↳ ConnectDrive (SubscriberLine(iN)N, SignalRingBackToneN, OnBL)
    ↳ ConnectDrive (SubscriberLine(CalledNumN)N, SignalRingingToneN, OnBL)
}
```

Figure 14. Detailed specification of TSS connecting behaviors in RTPA.

```
Talking ({I:: LineNumN}; {O:: CallProcessN}) ≙
{ // CallProcessN := 5
    iN := CallProcessN
                    → CalledNumN := CallRecord(iN).CalledNumN
                    → RouteNumN := CallRecord(iN).RouteNumN
                    → ( @ LineScanner(CalledNumN).StatusN = 1 ∨                    // Answered
                          LineScanner(iN).StatusN = 2
                        → ( LineScanner(CalledNumN).Status := 2                   // Show busy
                            ↳ ConnectDrive (SubscriberLine(iN)N, SignalRingBackToneN, OffBL)   // Stop signals
                            ↳ ConnectDrive (SubscriberLine(CalledNumN)N, SignalRingingToneN, OffBL)
                            ↳ ConnectDrive (SubscriberLine(iN)N, RouteNumN, OnBL)              // Connect circuit
                            ↳ ConnectDrive (SubscriberLine(CalledNumN)N, RouteNumN, OnBL)
                            → CallRecord(iN).CallingTerminationBL = T              // Set hook-on monitoring
                            → CallRecord(CalledNumN).CallingTerminationBL = T
                            → CallRecord(iN).CallProcessN = 6                      // To call termination
                          )
                      | @ LineScanner(iN).StatusN = 3 ∨                            // Calling give up before answer
                          LineScanner(iN).StatusN = 1
                        → ( LineScanner(CalledNumN).Status := 0                   // Show idle
                            ↳ ConnectDrive (SubscriberLine(iN)N, SignalRingBackToneN, OffBL)   // Stop signals
                            ↳ ConnectDrive (SubscriberLine(CalledNumN)N, SignalRingingToneN, OffBL)
                            → Route(RouteNumN).Status := F                         // Free seized route
                            → CallRecord(iN).CallStatusBL := F                     // Call gave up
                            → CallRecord(iN).CallProcessN = 0                      // Idle
                          )
                      | @ ~                                                         // Otherwise
                          → ∅
                      )
                  )
}
```

Figure 15. Detailed specification of TSS talking behaviors in RTPA.

CPN7: Exceptional termination process. Based on the schema developed in section 6.2.2, PN7, the refinement of the *exceptional termination* process can be carried out as shown in figure 17.

6.3. Specification of the TSS dynamic behaviors

As described in definition 2, system dynamic behaviors are process relations that can be determined at run-time. According to the RTPA system specification and refinement scheme as shown in figure 1, the work products developed in section 6.2, the specifications of system static behaviors by a set of processes, are only static functional compo-

```
CallTermination ({I:: LineNumN}; {O:: CallProcessN}) ≙
{ // CallProcessN := 6
    iN := CallProcessN
                    → CalledNumN := CallRecord(iN).CalledNumN
                    → RouteNumN := CallRecord(iN).RouteNumN
                    → ( ? CallRecord(iN).CallingTerminationBL = T ∧
                          LineScanner(iN).StatusN = 3                          // Calling hook-on
                          → LineScanner(iN).StatusN = 0                        // Set calling line idle
                          ↳ ConnectDrive (SubscriberLine(iN)N, RouteNumN, OffBL)   // Release route
                          ↳ ConnectDrive (SubscriberLine(CalledNumN)N, RouteNumN, OffBL)
                          ↳ ConnectDrive (SubscriberLine(CalledNumN)N, SignalBusyToneN, OnBL)
                          → Route(RouteNumN).StatusBL := F                     // Free seized route
                          → CallRecord(iN).CallingTerminationBL = F
                          → CallRecord(iN).CallStatusBL := F                   // Call terminated
                          → (  ? CallRecord(iN).CalledTerminationBL = T ∧
                                LineScanner(CalledNumN).StatusN = 3            // Called hook-on
                                → LineScanner(CalledNumN).StatusN = 0          // Set called line idle
                                → CallRecord(iN).CalledTerminationBL = F
                                → CallRecord(iN).CallProcessN = 0              // Set idle
                            | ? ~                                              // Set hook-on monitor for called
                                → CallRecord(iN).CalledTerminationBL = F
                                   → CallRecord(CalledNumN).CallStatusBL := T
                               → CallRecord(CalledNumN).TimerN := 10
                                   → CallRecord(CalledNumN).CallProcessN := 7  // To exceptional termination
                                )
                      )
          )
```

Figure 16. Detailed specification of TSS call termination behaviors in RTPA.

```
ExceptionalTermination ({I:: LineNumN}; {O:: CallProcessN}) ≙
{ // CallProcessN := 7
    iN := CallProcessN
                    → (  @ LineScanner(iN).StatusN = 3                          // Called hook-on
                              → LineScanner(iN).StatusN = 0                     // Set called line idle
                              ↳ ConnectDrive (SubscriberLine(iN)N, SignalBusyToneN, OffBL)
                              → CallRecord(iN).CallStatusBL := F                // Call terminated
                        | @ LineScanner(iN).StatusN = 2 ^ CallRecord(iN).TimerN := 0   // Waiting time out
                              ↳ ConnectDrive (SubscriberLine(iN)N, SignalBusyToneN, OffBL)
                              → LineScanner(iN).StatusN = 4                     // Set to invalid
                              → CallRecord(iN).CallStatusBL := F
                          → CallRecord(iN).CallProcessN := 0
                          )
                 )
        )
}
```

Figure 17. Detailed specification of TSS exceptional termination behaviors in RTPA.

nents of the system. To put the components into a live, coherent, and integrated system, the dynamic behaviors of the system, in terms of the *deployment* and *dispatch* of the static processes yet to be specified.

This subsection describes the TSS system dynamic behaviors via, again, a three-step refinement strategy, as defined in figure 1, i.e., system dynamic behaviors, process deployment, and process dispatch.

6.3.1. System dynamic behaviors of TSS

Definition 41. *Dynamic behaviors* of a system are process relations at run-time, which can be specified by a number of execution priority levels of processes based on their real-time timing requirements.

Generally, system dynamic behaviors, or the timing relationships of all the static processes developed in section 6.2, can be specified at four priority levels as shown below:

$$\begin{aligned} \text{SysIDS.DynamicBehaviors} \hat{=} & \{\text{Base-level processes}\} \\ & \| \{\text{High-level processes}\} \\ & \| \{\text{Low-interrupt-level processes}\} \\ & \| \{\text{High-interrupt-level processes}\} \end{aligned} \tag{45}$$

where the four priority levels in dynamic behavior specification for real-time system can be defined as follows in an increased priority:

Definition 42. A *base-level process* is a process that has no strict execution priority at run-time. All base-level processes of a system are dispatched in the lowest priority when there are no higher level processes scheduled or interrupt events occurred.

Definition 43. A *high-level process* is a process that has some timing requirements for execution priority at run-time. A high-level process may take over the run-time resources of a base-level process in system dispatching.

Definition 44. A *low-interrupt-level process* is an interrupt-event-driven process that has strict execution priority at run-time. A low-interrupt-level process may take over the run-time resources of an ordinary base-level or high-level process in system dispatching.

Definition 45. A *high-interrupt-level process* is an interrupt-event-driven process that has extremely strict execution priority at run-time. A high-interrupt-level process may take over the run-time resources of all other type processes in system dispatching.

According to expression (45) and definitions 42–45, the dynamic behaviors of TSS, at the high-level refinement, can be specified as follows:

$$\begin{aligned} \text{TSS.DynamicBahaviors}\mathbf{ST} \hat{=} & \{\text{Base-level processes}\} \\ & \| \{\text{High-level processes}\} \\ & \quad \text{// There is no high-level process in TSS} \\ & \| \{\text{Low-interrupt-level processes}\} \\ & \| \{\text{High-interrupt-level processes}\} \end{aligned}$$

$$
\begin{array}{lll}
= (\text{SystemInitial} & & // \text{ Base level} \\
\quad \| \;(\text{CallOrigination} & & // \text{ The 7 call processing processes} \\
\qquad \mid \text{Dialing} & & \\
\qquad \mid \text{CheckCalledStatus} & & \\
\qquad \mid \text{Connecting} & & \\
\qquad \mid \text{Talking} & & \\
\qquad \mid \text{CallTermination} & & \\
\qquad \mid \text{ExceptionalTermination} & & \\
\quad) & & \\
\quad \| \;\text{LineScanning} & & // \text{ Low-int level} \\
\quad \| \;(\text{SysClock} & & // \text{ High-int level} \\
\quad \| \;\text{DigitsReceiving} & & \\
\quad) & &
\end{array}
\tag{46}
$$

6.3.2. Process deployment in TSS

Definition 46. *Process deployment* is detailed dynamic process relations at run-time, which refines system dynamic behaviors by specifying precise *time-driven relations* between the system clock, system interrupt sources, and processes at different priority levels as follows:

$$
\begin{array}{ll}
\text{ProcessDeployment}\mathbf{ST} \mathrel{\hat{=}} \S \rightarrow & \\
& (\text{BaseTimeEvent}\llcorner\!\!\!\rightarrow \{\text{ProcessSet}_1\} \\
& \| \;\text{HighLevelTimeEvent}\llcorner\!\!\!\rightarrow \{\text{ProcessSet}_2\} \\
& \| \;\text{LowIntTimeEvent}\llcorner\!\!\!\rightarrow \{\text{ProcessSet}_3\} \\
& \| \;\text{HighIntTimeEvent}\llcorner\!\!\!\rightarrow \{\text{ProcessSet}_4\} \\
&) \\
& \rightarrow \S
\end{array}
\tag{47}
$$

According to the time-driven process relations in process deployment, the TSS dynamic behaviors specified in expression (46) can be further refined as shown in figure 18, where precise timing relationships between different priority levels are specified.

Figure 18 shows, according to the system timing priority at run-time, the TSS system deploys the *LineScanning* process into the 100 ms low-level interrupt processes, and *SysClock* and *DigitsReceiving* into the 10 ms high-level interrupt processes. The rest processes are deployed into the base-level services because there is no strict timing constraint in operation.

6.3.3. Process dispatch in TSS

Definition 47. *Process dispatch* is detailed dynamic process relations at run-time, which refines system dynamic behaviors by specifying *event-driven relations* of the system as

```
TSS.ProcessDeployment ≙
{
// Base level processes
   @SystemInitial
   ↳ ( SysInitial
            @SysShutDownS=T
       ↳          R          CallProcessing
                 ≥1
       → ⊠
     )
|| // High-interrupt-level processes
   ⊙ @SysClock10msInt
     ↗ (SysClock
         ↳ DigitsReceiving
       )
     ↘ ⊙
|| // Low-interrupt-level processes
   ⊙ @SysClock100msInt
     ↗ LineScanning
     ↘ ⊙
}
```

Figure 18. Specification of TSS process deployment.

follows:

$$\begin{aligned}\text{ProcessDispatch}\mathbf{ST} \hat{=} \ \S \rightarrow \ & \\ & (\text{Event}_1 \ \Lsh \{\text{ProcessSet}_1\} \\ & \mid \text{Event}_2 \ \Lsh \{\text{ProcessSet}_2\} \\ & \mid \ldots \\ & \mid \text{Event}_n \ \Lsh \{\text{ProcessSet}_n\} \\ &) \\ & \rightarrow \S \end{aligned} \tag{48}$$

Process dispatch specifies event-driven relations of a system. According to expression (48), the specification of TSS process dispatch is developed in figure 19.

As specified in expression (46) and figure 19, the *CallProcessing* process is a combined process with seven state-transition processes for controlling a call from origination to termination. Since TSS is operating at the millisecond level, while a telephone call may last for a considerably long period, the system cannot serve and wait for the completion of a transition for a specific call for all the time. Therefore, switching functions for an individual call are divided into seven states, corresponding to the seven dispatching processes as shown in figure 19.

This section described a real-world case study on a relative complicated real-time software system according to the RTPA specification and refinement method and scheme as defined in figure 1. The TSS architecture, and static and dynamic behaviors are for-

```
CallProcessing ≙
{
 nN := 15
  → $\overset{n}{\underset{i=0}{R}}$ ( ? ⓢCallRecord.CallStatusBL = T                           // A calling subscriber
          → LineNumN := iN
          → (  @ CallRecord(iN).CallProcessN = 0                            // Idle
                 → ∅
             | @ CallRecord(iN).CallProcessN = 1                            // Call origination
                 ↳ CallOrigination ({I:: LineNumN}; {O:: CallProcessN})
             | @ CallRecord(iN).CallProcessN = 2                            // Dialing
                 ↳ Dialling ({I:: LineNumN}; {O:: CallProcessN})
             | @ CallRecord(iN).CallProcessN = 3                            // Check called status
                 ↳ CheckCalledStatus ({I:: LineNumN}; {O:: CallProcessN})
             | @ CallRecord(iN).CallProcessN = 4                            // Connecting
                 ↳ Connecting ({I:: LineNumN}; {O:: CallProcessN})
             | @ CallRecord(iN).CallProcessN = 5                            // Talking
                 ↳ Talking ({I:: LineNumN}; {O:: CallProcessN})
             | @ CallRecord(iN).CallProcessN = 6                            // Call termination
                 ↳ CallTermination ({I:: LineNumN}; {O:: CallProcessN})
             | @ CallRecord(iN).CallProcessN = 7                            // Exceptional termination
                 ↳ ExceptionalTermination ({I:: LineNumN}; {O:: CallProcessN})
            )
       )
}
```

Figure 19. Specification of TSS process dispatch.

mally specified by a set of three-level refinements. The final-level of the TSS specifications, as recorded in figures 5–19, provide a set of detailed and precise design blueprints for code implementation, test, and verification. This case study demonstrated that RTPA is a practical formal engineering method for real-time system specification and refinement based on a single set of formal notations.

7. Conclusions

The phenomenon that the software engineering community is still facing almost the same problems as we dealt with 30 years ago indicates the inadequacy of the analytic mathematical means used in software engineering. Conventional formal methods do not distinguish system static and dynamic behaviors, lack descriptive power on system architectural specifications, and focus on system logical states manipulation rather than precise system functional behaviors. Therefore, seeking a new form of expressive mathematics that is suitable for the 3-D real-time system specification problems is fundamentally important. The *real-time process algebra* (RTPA) is one of the efforts towards the development of an essentially small set of formal notations with reasonably expressive power for real-time system specification and refinement.

RTPA has been developed as an algebra-based, expressive, and easy-to-comprehend notation system, and a practical specification and refinement method for real-time system description and specification. This paper has described the design and applications of RTPA as a comprehensive mathematical notation system. The structure of the RTPA notations and the method of RTPA specification and refinement have been presented. Sufficient sets of 16 meta-processes and 16 process relations have been elicited from comparative analyses of formal methods and modern programming languages, and empirical system specifications. A stepwise specification and refinement method has been developed for describing both system architectural and operational components. A case study of RTPA on a real-world problem, the TSS telephone switching system, has been presented to demonstrate features and the descriptive power of the RTPA notation system and the specification and refinement method. This paper has shown that a real-time system, including its architecture, and static and dynamic behaviors, can be essentially and sufficiently described by the coherent set of RTPA notations.

A number of case studies of RTPA specification have been carried out, such as a lift dispatching system (LDS) [Wang and Foinjong 2002], a digital switching system (DSS), an automated teller machine (ATM), a telephone switching system (TSS), and a set of abstract data types (ADTs). RTPA has also been used to specify algorithms and software process models such as CMM. Transformation between an RTPA specification and an OO programming language, e.g., Java, has been investigated [Wang and Wu 2002], which will lead to the development of an RTPA-based code generation tool as a long-term goal of this work. Experiences have shown that the RTPA notation system has the following advantages:

- easy to learn and acquisition,
- easy to comprehend,
- suitable for specifying the 3-D real-time system behaviors,
- suitable for specifying both architectural and operational components in a system,
- expressive for both system architectures and behaviors,
- expressive for real-time events and timing manipulation,
- strongly typed,
- built-in exceptional detection mechanisms for safety-critical applications.

Gaining from the features of RTPA as the smallest set of formal notations, and its stepwise method for system specification and refinement, ordinary software engineers have been able to read and comprehend an RTPA specification by themselves within a few days, and to use it as a descriptive tool for new system specifications through a one-week training [Wang and Wu 2002]. The application results encouragingly demonstrated that RTPA is a powerful and practical software engineering notation system for both academics and practitioners in software engineering.

Acknowledgements

This work is supported by the fund of the Natural Sciences and Engineering Research Council of Canada (NSERC). It also related to the author's work in the IEEE Software Notation Planning Group (SNPG) towards the development of a standard software engineering notation system. The author would like to acknowledge the support of NSERC and IEEE SNPG. The author would like to thank the anonymous referees and Profs. O. Balci and A. Bryant for their valuable comments that improved the presentation and quality of this paper.

References

Baeten, J.C.M. and J.A. Bergstra (1991), "Real Time Process Algebra," In *Formal Aspects of Computing*, Vol. 3, pp. 142–188.

Boucher, A. and R. Gerth (1987), "A Timed Model for Extended Communicating Sequential Processes," In *Proceedings of ICALP'87*, Lecture Notes in Computer Science, Vol. 267, Springer.

Cerone A. (2000), "Process Algebra Versus Axiomatic Specification of a Real-Time Protocol", Lecture Notes in Computer Science, Vol. 1816, Springer, Berlin, pp. 57–67.

Cline, B. (1981), *Microprogramming Concepts and Techniques*, Petrcelli, New York.

Corsetti, E., A. Montanari, and E. Ratto (1991), "Dealing with Different Time Granularities in Formal Specifications of Real-Time Systems," *The Journal of Real-Time Systems 3*, 2, June, 191–215.

Derrick, J. and E. Boiten (2001), *Refinement in Z and Object-Z: Foundations and Advanced Applications*, Springer-Verlag, London.

Dierks, H. (2000), "A Process Algebra for Real-Time Programs," Lecture Notes in Computer Science, Vol. 1783, Springer, Berlin, pp. 66–76.

Fecher, H. (2001), "A Real-Time Process Algebra with Open Intervals and Maximal Progress," *Nordic Journal of Computing 8*, 3, 346–360.

Gerber, R., E.L. Gunter, and I. Lee (1992), "Implementing a Real-Time Process Algebra," In M. Archer, J.J. Joyce, K.N. Levitt, and P.J. Windley, Eds., *Proceedings of the International Workshop on the Theorem Proving System and Its Applications*, August, IEEE Computer Society Press, Los Alamitos, CA, pp. 144–154.

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

Hoare, C.A.R. (1985), *Communicating Sequential Processes*, Prentice-Hall.

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

Jeffrey, A. (1992), "Translating Timed Process Algebra into Prioritized Process Algebra," In *Proceedings of the 2nd International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, J. Vytopil, Ed., Lecture Notes in Computer Science, Vol. 571, Springer-Verlag, Nijmegen, The Netherlands, pp. 493–506.

Klusener, A.S. (1992), "Abstraction in Real Time Process Algebra," In *Proceedings of Real-Time: Theory in Practice*, J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, Eds., Lecture Notes in Computer Science, Springer, Berlin, pp. 325–352.

Martin-Lof, P. (1975), "An Intuitionist Theory of Types: Predicative Part," In *Logic Colloquium 1973*, H. Rose and J.C. Shepherdson, Eds., North-Holland.

Milner, R. (1989), *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ.

Nicollin, X. and J. Sifakis (1991), "An Overview and Synthesis on Timed Process Algebras," In *Proceedings of the 3rd International Computer Aided Verification Conference*, pp. 376–398.

Reed, G.M. and A.W. Roscoe (1986), "A Timed Model for Communicating Sequential Processes," In *Proceedings of ICALP'86*, Lecture Notes in Computer Science, Vol. 226, Springer, Berlin.

Schneider, S.A. (1991), "An Operational Semantics for Timed CSP," Programming Research Group Technical Report TR-1-91, Oxford University.

Vereijken, J.J. (1995), "A Process Algebra for Hybrid Systems," In *Proceedings of the 2nd European Workshop on Real-Time and Hybrid Systems*, A. Bouajjani and O. Maler, Eds., Grenoble, France, June.

Wang, Y. (2001), “Formal Description of the UML Architecture and Extendibility,” *The International Journal of the Object 6*, 4, 469–488.

Wang, Y. (2002a), “A New Math for Software Engineering – The Real-Time Process Algebra (RTPA),” In *Proceedings of the 2nd ASERC Workshop on Quantitative and Soft Computing Based Software Engineering (QSSE’02)*, April, Banff, AB, Canada.

Wang, Y. (2002b), “A New Approach to Real-Time System Specification,” In *Proceedings of the 2002 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE’02),* Winnipeg, MB, Canada, May.

Wang, Y. (2002c), “Description of Static and Dynamic Behaviors of Software Components by the Real-Time Process Algebra (RTPA),” In *Component-Based Software Engineering*, F. Barbier, Ed., Kluwer Academic, UK.

Wang, Y. and N.C. Foinjong (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),* Winnipeg, MB, Canada, May.

Wang, Y. and G. King (2000), *Software Engineering Processes: Principles and Applications*, CRC Press, 752 pp.

Wang, Y., H. Sjostrom, K.-L. Lundback, G.N. Sauer, L.-B. Fredriksson, H. Edler, O. Bridal, A. Lindbom, and J. Hedberg (2000), “Distributed System Dependability Description and Comprehension,” Technical Report D10.3 of PALBUS on Reliable/Distributed/Real-Time Control Buses, The Swedish National Testing and Research Institute (SP), pp. 1–81.

Wang, Y. and W. Wu (2002), “Case Studies on Translation of RTPA Specifications into Java Programs,” In *Proceedings of the 2002 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE’02),* Winnipeg, MB, Canada, May.

Wilson, L.B. and R.G. Clark (1988), *Comparative Programming Languages*, Addison-Wesley, Wokingham, England.

Woodcock, J. and J. Davies (1996), *Using Z: Specification, Refinement, and Proof*, Prentice Hall International, London.