ABSTRACT

A typical real-time system, the Telephone Switching System (TSS), is a highly complicated system in design and implementation. This paper presents the formal design, specification, and modeling of the TSS system using a denotational mathematics known as Real-Time Process Algebra (RTPA). The conceptual model of the TSS system is introduced as the initial requirements for the system. Then, the architectural model of the TSS system is created using the RTPA architectural modeling methodologies and refined by a set of Unified Data Models (UDMs). The static behaviors of the TSS system are specified and refined by a set of Unified Process Models (UPMs) such as call processing and support processes. The dynamic behaviors of the TSS system are specified and refined by process priority allocation, process deployment, and process dispatching models. Based on the formal design models of the TSS system, code can be automatically generated using the RTPA Code Generator (RTPA-CG), or be seamlessly transformed into programs by programmers. The formal model of TSS may not only serve as a formal design paradigm of real-time software systems, but also a test bench of the expressive power and modeling capability of exiting formal methods in software engineering. [Article copies are available for purchase from InfoSci-on-Demand.com]

Keywords: Architectural Modeling; Behavioral Modeling; Code Generation; Design Paradigms; Formal Design Models; Processes; Real-Time Systems; System Dispatching; System Frameworks; System Refinement; Unified Data Model; Unified Process Model

INTRODUCTION

Telephone Switching Systems (TSS’s) are one of the typical real-time and mission-critical systems, as those of air-traffic control and banking systems, characterized by their high degree of complexity, intricate interactions with hardware devices and users, and necessary requirement for domain knowledge (Labrosse, 1999; Liu, 2000; McDermid, 1991; Ngolah et al., 2004; Wang, 2007a). All these factors warrant a TSS system as a complex but ideal design paradigm in real-world large-scale software system design in general and in real-time system modeling in particular.

There is no systematical and detailed repository and formal documentation of design knowledge and modeling prototypes of a TSS system nor a formal model of it in denotational mathematics and formal notation systems. This
This article presents the formal design, specification, and modeling of a TSS system using a denotational mathematics known as Real-Time Process Algebra (RTPA) (Wang, 2002, 2008a, 2008b). RTPA introduces only 17 meta-processes and 17 process relations to describe software system architectures and behaviors with a stepwise refinement methodology (Wang, 2007a, 2008a, 2008c, 2008d). According to the RTPA methodology for system modeling and refinement, a software system can be specified as a set of architectural components and operational components. The former are modeled by the Unified Data Models (UDMs, also known as Component Logical Models CLMs), which is an abstract model of the system hardware interface, an internal logic model of hardware, and/or a control structure of a system. The latter are modeled by static and dynamic processes in terms of the Unified Process Models (UPMs) (Hoare, 1978; Milner, 1980; Hoare et al., 1987; Baeten and Bergstra, 1991; Corsetti and Ratto, 1991; Vereijken, 1995; Dierks, 2000; Wang, 2007a, 2008a; Wang and King, 2000).

This article develops a formal design model of the TSS system in a top-down approach on the basis of the RTPA methodology. In the remainder of this article, the conceptual model of the TSS system is described as the initial requirements of the system. The architectural model of the TSS system is created based on the conceptual model using the RTPA architectural modeling methodologies and refined by a set of CLMs. Then, the static behaviors of the TSS system are specified and refined by a set of processes. The dynamic behaviors of the TSS system are specified and refined by process priority allocation, process deployment, and process dispatching models. With the formal and rigorous models of the TSS system, code can be automatically generated by the RTPA Code Generator (RTPA-CG) (Wang, 2007a, Wang et al., 2009), or be seamlessly transferred into programs manually. The formal model of TSS may not only serve as a formal design paradigm of real-time software systems, but also a test bench of the expressive power and modeling capability of existing formal methods in software engineering.

THE CONCEPTUAL MODEL OF THE TSS SYSTEM

A Telephone Switching System (TSS) is a complex real-time system (Thompson, 2000; Wang, 2007a). The functional structure of the TSS system can be described by a conceptual model as illustrated in Figure 1, which consists of four subsystems known as the call processing, subscribers, routes, and signaling subsystems.

The configuration of the TSS system encompasses 1 call processor and 16 subscribers. There are 5 internal switching routes and a set of 5 signaling trunks providing the dial, busy, ringing, ring-back, and special tones. The call processor modeled by a set of functional processes operates on the line scanners, call records, digits receivers, signaling trunks, system clock, and routes in order to implement a coherent program-controlled switching functions.

The framework of the TSS system, encompassing its architecture, static behaviors, and dynamic behaviors, can be specified using RTPA as follows (Wang, 2002, 2008a):

\[
\begin{align*}
\text{§(TSS)} & \triangleq \text{TSS}.\text{Architecture}^{\text{ST}} \\
& \quad || \text{TSS}.\text{StaticBehaviors}^{\text{PC}} \\
& \quad || \text{TSS}.\text{DynamicBehaviors}^{\text{PC}}
\end{align*}
\]

(1)

where || indicates that these three subsystems related in parallel, and §, ST, and PC are type suffixes of system, system structure, and process, respectively.

According to the RTPA methodology for system modeling, specification, and refinement (Wang, 2008a, 2008b), the top level model of any system may be specified in a similar structure as given in Eq. 1. The following sections will extend and refine the top level framework of the TSS§ into detailed architectural models and behavioral models.
THE ARCHITECTURAL MODEL OF THE TSS SYSTEM

The architecture of a software system or a hybrid hardware/software system is a system framework that represents the overall structure, components, processes, and their interrelationships and interactions. The following subsections specify the architecture of TSS, TSS Architecture, by a high-level architecture model based on its conceptual model as developed in Figure 1. Then, each of its architectural components will be refined as a corresponding UDM.

The Architectural Framework of TSS

System architectures, at the top level, specify a list of names of UDMs and their relations. A UDM may be regarded as a predefined class of system hardware or internal control models, which can be inherited or implemented by corresponding UDM objects as specific instances in the succeeding system architectural refinement procedures.

Corresponding to the conceptual model of TSS as shown in Figure 1, the high-level specification of the architecture of TSS, TSS Architecture, is given in Figure 2 using RTPA. The high-level architectural model TSS Architecture encompasses parallel structures of CallProcessingSubsys, SubscriberSubsys, RouteSubsys, and SignalingSubsys, as well as a set of events @Events and a set of statuses @Status. The call processing subsystem CallProcessingSubsys is further refined by a set of UDMs such as a CallProcessor, a SysClock, and 16 CallRecords. Similarly, the subscriber subsystem SubscriberSubsys is refined by 16 Subscribers and 16 LineScanners; and the signaling subsystem is refined by 16 Digits Receivers and 5 Signaling Trunks.

The configuration of the UDMs in TSS is indicated by the numbers in the angle brackets, where each number shows how many similar devices or internal control structures are equipped that share the same UDM schema.

The UDM Structures of TSS

The UDMs of a specific system represent the abstraction and formal representation of domain knowledge and structural inform-
As modeled in Figure 2, the TSS system encompasses 6 kinds of UDMs for modeling system hardware interfaces and internal control structures as follows:

\[
\text{TSS}. \text{UDMs} \triangleq \begin{array}{l}
\text{CallProcessingSubsys} \\
\text{SubscriberSubsys} \\
\text{RouteSubsys} \\
\text{SignalingSubsys} \\
\text{<@Events : S>} \\
\text{<@Status : BL>}
\end{array}
\]

where the LineScannersST, DigitsReceiversST, SignalTrunksST, and SysClockST are hardware interface UDMs, while the RoutesST and CallRecordsST are internal control UDMs.

Each of the six-type system UDMs in TSS is designed and modeled in the following subsections.

**a) The Line Scanners**

A line scanner is an interface device of a telephone switching system that connects a subscriber or a pair of telephone lines to the switching system. Each telephone or subscriber is assigned a line scanner in a switching system. The schema of the line scanners, LineScannersST, is designed as given in Figure 3, where all the 16 line scanners share the same structure.

LineScannersST encompasses five fields known as the StatusN, PortAddressH, ScanInputB, CurrentScanBL, and LastScanBL. In the LineScannersST UDM, the StatusN denotes the operating states of the line scanner with the type of natural number; The PortAddressH denotes the designated addresses of a set of port interfaces in hexadecimal type; The ScanInputB denotes the information inputted in byte type from scanning the lines as specified by the PortAddressH; and the CurrentScanBL and the LastScanBL denote the logical line scan status in Boolean type. For each scan period, the following operations are conducted: LastScanBL
Figure 3. The UDM schema of line scanners

\[
\text{LineScanner}_{ST} \triangleq \bigwedge_{i=0}^{15} \text{LineScanner}(i)_{ST}:
\]

\[
\begin{align*}
\text{PORT}_{ST} &\colon \langle \text{PortAddress} : H | \text{FF00} \leq \text{PortAddress} \leq \text{FF0F} \rangle, \\
\text{ScanInput}_{B} &\colon \langle \text{ScanInput} : B | \text{ScanInput} = \langle \text{x} x x x x x x x B \rangle \rangle, \\
\text{Status}_{N} &\colon \{0, \text{Idle} \}, \{1, \text{HookOn} \}, \{2, \text{HookOff} \}, \{3, \text{Busy} \}, \{4, \text{Seized} \}, \{5, \text{Invalid} \} \rangle, \\
\text{CurrentScan}_{BL} &\colon \langle T = \text{HookOff} \wedge F = \text{HookOn} \rangle, \\
\text{LastScan}_{BL} &\colon \langle T = \text{HookOff} \wedge F = \text{HookOn} \rangle
\end{align*}
\]

= \text{CurrentScan}_{BL} \text{and \ CurrentScan}_{BL} = (\text{ScanInput}_{B}.b_{0})_{BL}, \ i.e., (\text{ScanInput}_{B}.b_{0})_{BL} := T \text{\ when \ } \text{ScanInput}_{B}.b_{0} = 1, \ \text{otherwise \ (ScanInput}_{B}.b_{0})_{BL} = F.

Each field in the UDM, LineScanner_{ST}, is declared by an RTPA type where its constraint, if any, is provided following the vertical bar. For instance, the constraints for the field Status_{N} is 0 \leq \text{Status}_{N} \leq 5, where the logical meaning of the numbers represents idle, hook-on, hook-off, busy, seized, and invalid, respectively.

The 16 refined concrete UDM objects of the line scanners can be derived as shown in Figure 4 on the basis of the abstract schema as given in Figure 3. Each concrete line scanner LineScanner(i)_{ST} in Figure 4 obtains its refined physical or logical parameters.

b) The Digits Receivers

A digital receiver is a UDM model of the TSS system that specifies the functional requirements and mechanisms for receiving, retaining, and representing a series of numbers sent via a pair of telephone lines by a subscriber. Digital receivers monitor subscriber line close/open signals and transfer the serial pulses into digits in order to represent the numbers that the subscriber dialed. The schema of the digital receivers, DigitsReceivers_{ST}, is designed as given in Figure 5, refined by the 16 concrete UDM objects of DigitsReceiver(i)_{ST}.

A digit receiver may work at one of the four statuses known as no-dial, dial started, dialing, and dial completed, which are accessed by the call processor at a designated status port. During the dialing process, dial pulses are received by the call processor from the given digit port. After processing, the pulses are transferred into decimal digits and stored in Digit1_{N} and Digit2_{N} assuming that the given TSS using two numbers. However, it may be flexibly re-specified when it is needed for a larger system.

c) The Signaling Trunks

A signaling trunk of a switching system is a device that generates and sends a specific signal to a specific subscriber line. The signaling trunks can be classified into those of dial tone, busy tone, ringing tone, ring back tone, and special tone. The schema and detailed structures of the signaling trunk UDM, SignalTrunks_{ST}, is designed as given in Figure 6. Each concrete UDM objects of SignalTrunk(i)_{ST} is a logical model of the physical trunk that specifies the port address of a tone for signaling distributions.

d) The System Clock

A system clock is a typical real-time device for event timing, process duration manipulation, and system synchronization. The UDM model of the system clock of TSS, SysClock_{ST}, is designed as given in Figure 7. SysClock_{ST} provides an absolute (calendar) clock CurrentTime_{hh:mm:ss:ms} as the logical time reference of the entire system and a relative clock \$_{N}$ as a generic counter. The InterruptCounter_{N} is adopted to transfer the basic timing pace at 1ms into 1 second signals.
e) The Switching Routes

A route of a switching system is an internal circuit or digital channel that connects a pair of subscriber lines together for conversation by a certain instruction. The required number of routes in a switching system is always far smaller than half of the number of subscribers, because not all subscribers are in use for all the time according to teletraffic theories. The schema of routes, Routes_{ST}, is designed as given in Figure 8, refined by the five concrete UDM objects of Routes(i_{N})_{ST}. Routes_{ST} is a logical model of the physical routes that specifies the parameters of the status of a route and which pair of subscribers known as the calling and called parties are connected when it is occupied.

f) The Call Records

A call record is an internal logical structure in a switching system that is uniquely created for and associated to a call in order to retain detail.
information in its entire lifecycle. The schema of the call records, CallRecordsST, is designed as given in Figure 9, refined by the 16 concrete UDM objects of CallRecord(i)ST.

The CallRecord(i)ST in TSS models the internal control structure of calls with the information such as the CallStatusN, CallProcessN, CalledNumN, RouteNumN, TimerN, CallingTerminationBL, and CalledTerminationBL. The CallProcessN is a set of call process numbers defined as \{0, Idle\}, \{1, CallOrigination\}, \{2, Dialing\}, \{3, CheckCalledStatus\}, \{4, Connecting\}, \{5, Talking\}, \{6, CallTermination\}, \{7, ExceptionalTermination\}. Each call record is initialized as shown in UDM objects in Figure 9.

The system architectural specification developed in this subsection provides a set of abstract object models and clear interfaces between system hardware and software. By reaching this point, the co-design of a real-time system can be separately carried out by separated
hardware and software teams. It is recognized that system architecture specification by the means of UDMs 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 (UDMs), specification of the operational
components of the TSS system can be carried out directly forward, as shown in the following sections.
THE STATIC BEHAVIOR MODELS OF THE TSS SYSTEM

A static behavior is a component-level function of a given system that can be determined before run-time. On the basis of the system architecture specification and with the work products of system architectural components developed in preceding section, the operational components of the given TSS system and their behaviors can be specified as a set of behavioral processes operating on the UDMs.

The TSS static behaviors, TSS$^S$.StaticBehaviors$^{PC}$, encompass two subsystems known as the SysSupportProcesses$^{PC}$ and CallProcessingProcesses$^{PC}$ in parallel as specified below:

\[
TSS^S$.StaticBehaviors^{PC} \triangleq \\
| \text{SysSupportProcesses}^{PC} | [4] \\
| \text{CallProcessingProcesses}^{PC} | [7]
\]

where the former consists of 4 system support processes and the latter consists of 7 call processing processes.

The following subsections describe how the TSS static behaviors as specified in Eq. 3 are modeled and refined using the denotational mathematical notations and methodologies of RTPA.

Modeling System Support Processes of the TSS Static Behaviors

The static behaviors of the support processes subsystem in TSS as specified in Eq. 3 can be further refined in the following model with four processes:

\[
TSS^S$.StaticBehaviors^{PC}.SysSupportProcesses^{PC} \triangleq \\
| \text{SysInitial}^{PC} \\
| \text{SysClock}^{PC} \\
| \text{LineScanning}^{PC} \\
| \text{DigitsReceiving}^{PC}
\]

where each of the system support processes will be formally modeled and described in RTPA in the following subsections.

a) The System Initialization Process

System initialization is a common support process of a real-time system that boots the system, sets its initial environment, and preassigns the initial values of data objects of the system such as variables, constants, as well as architectural (hardware interface) and control (internal) UDMs. The system initialization process of TSS, SysInitial$^{PC}$, is modeled in Figure 10, where all system architectural and control UDMs are initialized. Then, the system clock and timing interrupt are set to their initial logical or calendar values.

Figure 10. The behavior model of the system initialization process

\[
\text{SysInitial}^{PC} \triangleq \\
\{ \text{Initial SystemModels}^{ST} \\
\rightarrow \text{Initial ControlModels}^{ST} \\
\rightarrow \text{SysClock}^{ST}.\$tN := 0 \\
\rightarrow \text{SysClock}^{ST}.\text{CurrentTime}^{hh:mm:ss} := hh:mm:ss \\
\rightarrow \text{SysClock}^{ST}.\text{InterruptCounter}^{N} := 0 \\
\} 
\]
**b) The System Clock Process**

The **system clock** process is a typical support process of a real-time system that maintains and updates an absolute (calendar) clock and a relative clock for the system. The system clock process of TSS, **SysClockPC**, is modeled in Figure 11. The source of the system clock is obtained from the 1ms interrupt clock signal generated by system hardware, by which the absolute clock with real-time millisecond, second, minute, and hour, **SysClockST.CurrentTime hh:mm:ss:ms**, are generated and periodically updated. The second clock in a real-time system is the relative clock, **SysClockST.§tN**, which is usually adopted for relative timing and duration manipulations. Both system clocks are reset to zero at midnight each day.

**SysClockPC** is also responsible to update all timers set in other processes by reducing its current value by one until its time out, i.e., **CallRecord(iNST.CallStatusBL = T ∧ CallRecord(iNST.TimerSS ≠ 0)**. Any time-out event will be captured by the system immediately after it reaches 0.

**c) The Line Scanning Process**

**Line scanning** is a special real-time support process that monitors the line statuses of all subscribers periodically and transfers them into logical states in terms of *idle, hook-on, hook-off, busy, seized, and invalid*. The line scanning process of TSS, **LineScanningPC**, is modeled in Figure 12 based on the UDM of LinscannersST. The latest status of a line is inputted into **LineScanner(iNST.CurrentScanBL** from **LineScanner(iNST.ScanInputB**, after **LineScanner(iNST.LastScanBL** is saved. Then the four basic operating statuses of the line can be logically determined as given in Table 1.

In Figure 12, the algorithm for line status detection in the process of line scanning can be expressed in Table 1, where the four possible line status known as *idle, hook-on, hook-off, and busy*, are determined by the periodical current and last scan inputs, i.e., **LineScanner(iNST.CurrentScanBL ∧ LineScanner(iNST.LastScanBL**. The fourth status is set by the system when a called line is preseized for connection. The fifth status is set by the system when a given line is malfunction or out of service.
Figure 12. The behavior model of the line scanning process

Table 1. Algorithm of line status determination in TSS

<table>
<thead>
<tr>
<th>No.</th>
<th>LineScanner(iN).ST.CurrentScanBL</th>
<th>LineScanner(iN).ST.LastScanBL</th>
<th>LineScanner(iN).ST.StatusN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>F</td>
<td>F</td>
<td>0 – Idle</td>
</tr>
<tr>
<td>2</td>
<td>F</td>
<td>T</td>
<td>1 – Hook-on</td>
</tr>
<tr>
<td>3</td>
<td>T</td>
<td>F</td>
<td>2 – Hook-off</td>
</tr>
<tr>
<td>4</td>
<td>T</td>
<td>T</td>
<td>3 – Busy</td>
</tr>
<tr>
<td>5</td>
<td>Set by the system</td>
<td></td>
<td>4 – Seized</td>
</tr>
<tr>
<td>6</td>
<td>Set by the system</td>
<td></td>
<td>5 – Invalid</td>
</tr>
</tbody>
</table>

d) The Digits Receiving Process

Digital receiving is a special real-time support process that receives the called subscriber number sent by the calling subscriber in high frequency periodical interrupt cycles in order to meet its timing constraints. The digital receiving process of TSS, DigitalReceivingPC, is modeled in Figure 13.

This process only checks lines that its PNN = 2 that has progressed into the dialing process. Then, the dial status of a line DigitsReceiver(iN).StatusInputB is checked from the input of the DigitsReceiver(iN).Sta-
Figure 13. The behavior model of the digit receiving process

```
DigitsReceivingPC ≡
{ nN := 15
  → R
       ( CallRecord(iN)ST.CallProcessN = 2 // Dialing
         → PORT(DigitsReceiver(iN)ST.StatusPortH B |> DigitsReceiver(iN)ST.StatusInputB
         → ( CallRecord(iN)ST.StatusInputB = 0000 0001B // A number valid
             → DigitsReceiver(iN)ST.StatusN := 1 // Dial started
             → PORT(DigitsReceiver(iN)ST.DigitPortH B |> DigitsReceiver(iN)ST.DigitInputB
             → ( CallRecord(iN)ST.#DigitsReceivedH = 0
          - DigitsReceiver(iN)ST.Digit1N := DigitsReceiver(iN)ST.DigitInputB
          - DigitsReceiver(iN)ST.StatusN := 2 // First digit received
          - ↑ (DigitsReceiver(iN)ST.#DigitsReceivedH := 0
             - ~
          - DigitsReceiver(iN)ST.Digit2N := DigitsReceiver(iN)ST.DigitInputB
          - DigitsReceiver(iN)ST.StatusN := 3 // All digits received
          - DigitsReceiver(iN)ST.#DigitsReceivedH := 0
       ) )
  )
}
```

If the status is valid, the dial pulse on the line DigitsReceiver(iN).DigitInputB is inputted from DigitsReceiver(iN).DigitPortH. According to the architectural model of the DigitsReceiversST designed in Figure 5, the dial status DigitsReceiver(iN).StatusN will be set to dial stated (1), first digit received (2), and all digits received (3) dependent on the progress of the dial process of a particular line.

### Modeling Call-Processing Processes of the TSS Static Behaviors

The static behaviors of the TSS call processing subsystem, as modeled in Eq. 3, can be further refined in the following model with seven processes:

\[
\text{TSSStaticBehaviorsPC.CallProcessingProcessesPC} \triangleq \\
\text{CallOriginationPC} \\
\text{|| DiallingPC} \\
\text{|| CheckCalledStatusPC} \\
\text{|| ConnectingPC} \\
\text{|| TalkingPC} \\
\text{|| CallTerminationPC} \\
\text{|| ExceptionalTerminationPC}
\]

where each of the call processing processes will be formally modeled and described in RTPA in the following subsections.

The configuration of processes of the TSS system and a set of process schemas are designed as shown in Table 2, which refine the high level model of TSS static behaviors as given in Eq. 5. The process schemas of TSS provide further detailed information on each process’ functionality, I/O, and its relationships with
Table 2. Specification of the TSS process schemas

<table>
<thead>
<tr>
<th>PN</th>
<th>ProcessIDPC ([I]; {O})</th>
<th>Operated CLMST</th>
<th>Related Processes</th>
<th>Functional Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CallOriginationPC</td>
<td>LineScannersST</td>
<td>LineScanningPC</td>
<td>Find hook-off subscribers from LineScannersST</td>
</tr>
<tr>
<td></td>
<td>([I:: LineNumN];</td>
<td>CallRecordsST</td>
<td>ConnectDrivePC</td>
<td>Record originated calls in CallRecordsST</td>
</tr>
<tr>
<td></td>
<td>{O:: CallProcessN}]</td>
<td></td>
<td>SysClockPC</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>DialingPC</td>
<td>DigitsReceiversST</td>
<td>DigitsReceiving-PC</td>
<td>Receive digits from DigitsReceiversST</td>
</tr>
<tr>
<td></td>
<td>([I:: LineNumN];</td>
<td>CallRecordsST</td>
<td>ConnectDrivePC</td>
<td>Record called number in CallRecordsST</td>
</tr>
<tr>
<td></td>
<td>{O:: CallProcessN})</td>
<td></td>
<td>SysClockPC</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CheckCalledStatusPC</td>
<td>LineScannersST</td>
<td>LineScanningPC</td>
<td>Check called status from callRecordsST</td>
</tr>
<tr>
<td></td>
<td>([I:: LineNumN];</td>
<td>CallRecordsST</td>
<td>ConnectDrivePC</td>
<td>Find route from RoutesST</td>
</tr>
<tr>
<td></td>
<td>{O:: CallProcessN})</td>
<td></td>
<td>SysClockPC</td>
<td>Send busy tone to calling if called’s busy</td>
</tr>
<tr>
<td>4</td>
<td>ConnectingPC</td>
<td>CallRecordsST</td>
<td>ConnectDrivePC</td>
<td>Send RingBackTone to calling</td>
</tr>
<tr>
<td></td>
<td>([I:: LineNumN];</td>
<td></td>
<td>SysClockPC</td>
<td>Send RingingTone to called</td>
</tr>
<tr>
<td></td>
<td>{O:: CallProcessN})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>TalkingPC</td>
<td>LineScannersST</td>
<td>LineScanningPC</td>
<td>Process either party termination based on LineScannersST</td>
</tr>
<tr>
<td></td>
<td>([I:: LineNumN];</td>
<td>CallRecordsST</td>
<td>ConnectDrivePC</td>
<td>Release routes according to RoutesST</td>
</tr>
<tr>
<td></td>
<td>{O:: CallProcessN})</td>
<td></td>
<td>SysClockPC</td>
<td>Monitor non-hook-on party in CallRecordsST</td>
</tr>
<tr>
<td>6</td>
<td>CallTerminationPC</td>
<td>LineScannersST</td>
<td>LineScanningPC</td>
<td>Reset line status in LineScannersST, if monitored party hook-on</td>
</tr>
<tr>
<td></td>
<td>([I:: LineNumN];</td>
<td>CallRecordsST</td>
<td>ConnectDrivePC</td>
<td>If time-out, set line status invalid in LineScannersST</td>
</tr>
<tr>
<td></td>
<td>{O:: CallProcessN})</td>
<td></td>
<td>SysClockPC</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Exceptional-TerminationPC</td>
<td>LineScannersST</td>
<td>LineScanningPC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>([I:: LineNumN];</td>
<td>CallRecordsST</td>
<td>ConnectDrivePC</td>
<td></td>
</tr>
<tr>
<td></td>
<td>{O:: CallProcessN})</td>
<td></td>
<td>SysClockPC</td>
<td></td>
</tr>
</tbody>
</table>

system architectural components (UDMs) and other processes.

a) The Process of Call Origination

Call origination is the first call processing process that identifies new call requests of subscribers and creates associated internal control structures for each new call. The call origination process of TSS, CallOriginationPC, is modeled in Figure 14, where related system support processes and operated UDMs are cross referenced.

The CallOriginationPC process finds hook-off subscribers from LineScannersST and registers newly originated calls in CallRecordsST. The dial tone is sent to the subscriber which originated a new call by invoking the
Figure 14. The behavioral model of the call origination process

```
CallOriginationPC (<I :: LineNumN>; <O :: CallProcessN>; <UDMs :: LineScannersST, CallRecordsST>) △

{ // PNN = 1
  iN := LineNumN
  → LineScanner(iN)ST.StatusN := 3 // Show line is busy
  → ConnectDrivePC (SubscriberLine(iN)N, DialToneN, OnBL)
  → CallRecord(iN)ST.TimerSS := 5 // Set no dial timer
  → CallRecord(iN)ST.CallStatusBL := T // Set call record active
  → CallRecord(iN)ST.CallProcessN := 2 // To dialing
}
```

Figure 15. The behavioral model of the dialing process

```
DialingPC (<I :: LineNumN>; <O :: CallProcessN>; <UDMs :: DigitsReceiverST, CallRecordsST>) △

{ // PNN = 2
  iN := LineNumN
  → ( ☹ DigitsReceiver(iN)ST.StatusN = 0 // No dial
       → ( ☹ CallRecord(iN)ST.TimerSS = 0 // No dial time-out
            → ConnectDrivePC (SubscriberLine(iN)N, DialToneN, OffBL)
            → ConnectDrivePC (SubscriberLine(iN)N, BusyToneN, OnBL)
            → CallRecord(iN)ST.TimerSS := 10
            → CallRecord(iN)ST.CallProcessN := 7 // To exceptional termination
       )
       | ☹ DigitsReceiver(iN)ST.StatusN = 1 // Dial started
       → ( ☹ CallRecord(iN)ST.TimerSS = 10 // Set dial time-out timer
            → ConnectDrivePC (SubscriberLine(iN)N, DialToneN, OffBL)
            → DigitalScanner(iN)ST.StatusN := 2
       )
       | ☹ DigitsReceiver(iN)ST.StatusN = 2 // Dialing
       → ( ☹ CallRecord(iN)ST.TimerSS = 0 // Dialing time-out
            → ConnectDrivePC (SubscriberLine(iN)N, BusyToneN, OnBL)
            → CallRecord(iN)ST.CallingTerminationBL := T
            → CallRecord(iN)ST.TimerSS := 10
            → CallRecord(iN)ST.CallProcessN := 7 // To exceptional termination
       )
       | ☹ DigitsReceiver(iN)ST.StatusN = 3 // Dial completed
       → CalledNumN := DigitalScanner(iN)ST.Digit1N * 10 +
         DigitalScanner(iN)ST.Digit2N
       → CallRecord(iN)ST.CalledNum := CalledNumN
       → CallRecord(iN)ST.CallProcessN := 3 // To check called status
       | ☹ ~ // Otherwise
       → ∅
  )
}
```
predesigned process ConnectDrivePC. A no-dial timer, CallRecord(iN)ST.TimerSS, is set for 5 seconds in order to monitor if the line that hears the dial tone will act to dial the called number within the given period of time. Any time out event will be detected in the next process. Then, it transfers the current call process number (PN = 1) CallRecord(iN)ST.CallProcessN, from PN = 1 (Call origination) to PN = 2 (Dialing).

b) The Process of Dialing

Dialing is the second call processing process that receives digits dialed by the calling subscriber on a specific line and registers them in the associated call record. The dialing process of TSS, DialingPC, is modeled in Figure 15, where related system support processes and operated UDMs are cross referenced.

The DialingPC process checks the status of dialing in the UDM DigitsReceiver(iN)ST as detected by the system. There are four status in the phase of dialing in TSS such as no dial, dial started (first digit has been received), dialing (in progress), and dial completed (all expected digits have been received) as modeled in DigitsReceiver(iN)ST.StatusN. Based upon each status, particular actions in term of predesignated processes will be invoked as given in Figure 14. If the no dial timer, CallRecord(iN)ST.TimerSS, goes off during the dialing process, the system will immediately send the busy tone to the subscriber and trigger a special process that transfers the current process to PN = 7 (Exceptional termination); Otherwise, the process goes further to seek an available internal route for connecting the pair of lines after pre-seize the called line by marking it as busy. When a free route is found, the process transfers to the next state PN = 4 (Connecting). However, if there is no free route available in the system, the busy tone will be sent to the calling subscriber, the seized called line status will be released, and the process transfers to PN = 7 (Exceptional termination).

c) The Process of Check Called Status

Check called status is the third call processing process that looks into the current status of a given called subscriber, finds an available internal switching route between the calling and called parties, and sends busy tone to calling subscriber when called is busy or no route is free. The check called status process of TSS, CheckCalledStatusPC, is modeled in Figure 16, where related system support processes and operated UDMs are cross referenced.

The CheckCalledStatusPC process tests the status of the called subscriber line in order to get through the requested call. If the called line is unavailable (i.e., it is busy, just hooked-off for a new call, or invalid), the busy tone will be sent to the calling subscriber, and the call is transferred into PN = 7 (Exceptional termination); Otherwise, the process goes further to seek an available internal route for connecting the pair of lines after pre-seize the called line by marking it as busy. When a free route is found, the process transfers to the next state PN = 4 (Connecting). However, if there is no free route available in the system, the busy tone will be sent to the calling subscriber, the seized called line status will be released, and the process transfers to PN = 7 (Exceptional termination).

d) The Process of Connecting

Connecting is the fourth call processing process that informs the called subscriber with the ringing tone, and at the same time, sends the ringing tone to the calling subscriber that is waiting for the answer of the call. The connecting process of TSS, ConnectingPC, is modeled in Figure 17, where related system support processes and operated UDMs are cross referenced.

The ConnectingPC process retrieves necessary information for a call such as the numbers of calling and called lines, as well as the switching route preallocated in preceding process. Then, the ringing tone and ring-back tone are sent to the called and calling subscriber lines, respectively, before the process transfers to PN = 5 (Talking).
e) The Process of Talking

Talking is the fifth call processing process that physically connects both parties using pre-seized route in the dialing process when the called subscriber answered, and monitors terminations by either party. The talking process of TSS, TalkingPC, is modeled in Figure 18, where related system support processes and operated UDMs are cross referenced.
The TalkingPC process first detects if the called line answers the call. If the called subscriber hooks off to answer the ring while the calling party is hearing the ring back tone, the called line will be seized and marked as busy to avoid any cross connection by other calls. At the same time, signals to both parties are stopped and a physical route between the calling and called lines is then connected via the pre-seized switching route in order to make conversation.
This results in a successful switching sequence and the system enters the process $P_{NN} = 6$ (Call termination).

A possible exceptional condition in this process is that, during waiting for answer, the calling subscriber may give up before the called line hook off. This will trigger the release of the called line and the occupied route, and the cancel of both signals to the calling and called parties. Then, the system transfers to $P_{NN} = 0$ (Ready for call origination).

It is noteworthy that the system do nothing when it enters this process if the called party has not answer the ringing signal and the calling party has remained in the waiting status by hearing the ring back tone. In this case, there is no state transition, i.e., the system remains in $P_{NN} = 5$ (Talking) until next round processing.

**f) The Process of Call Termination**

Call termination is the final normal call processing process that handles call ending of either party, releases the occupied route, and immediately sends the busy tone to the party that has not hook-on. The call termination process of TSS, $CallTerminationPC$, is modeled in Figure 19, where related system support processes and operated UDMs are cross referenced.

The $CallTerminationPC$ process handles the end of a call by monitoring the status of $CallingTerminationBL$ in the $CallRecordST$ of.
the calling line known as *calling-party control* of call termination. When a call termination is detected on the calling line, the subscriber and the route are immediately disconnected and released. Then, the status of the called line in the conversation is checked. If the called line has already hooked-on, it is set as free, the line status is transferred to \( \text{PN}_N = 0 \) (Ready for new call origination), and the engaged call record is set to terminated. However, if the called line remains unterminated when it is hearing the busy tone, it will be transferred to \( \text{PN}_N = 7 \) (Exceptional termination) after the 10 second monitoring timer is set.

It is noteworthy that, when a calling party controlled billing system is included in the TSS system, the starting point of billing is when the called line answers, i.e., \( \text{LineScanner(CalledNum}_N \text{ST}.\text{Status}_N = 2 \) in the TalkingPC process. However, the ending point of billing is triggered by the event \( \text{CallRecord(CallingNum}_N \text{ST.\text{CallStatus}}_BL = T \land \text{LineScanner(CallingNum}_N \text{ST.\text{Status}_N} = 1 \), where the former indicates that the given line's termination status is under monitoring and the letter denotes that the line has just hooked on. In addition, the *called-party controlled* or *both-party controlled* billing techniques may be adopted in a similar approach.

g) The Process of Exceptional Termination

*Exceptional termination* is the seventh call processing process that handles all possible exceptional events and conditions in any previous call processing process by sending the busy tone to a given subscriber line. The exceptional termination process of TSS, \( \text{ExceptionalTerminationPC} \), is modeled in Figure 20, where related system support processes and operated UDMs are cross referenced.

The \( \text{ExceptionalTerminationPC} \) process handles special situations in any previous call processing processes when any party does not hook-on after time out in call termination during receiving the busy tone. In the case the status of the line under monitoring for exceptional termination is detected to be hooked–on, the

![Figure 20. The behavioral model of the external termination process](image-url)
line will be released and the busy tone will be stopped. However, in the case of no termination after the timer is out, the line will no longer be sending the busy tone, but its status in the LineScanner ST will be set as invalid until the system administrator turns it back to normal services.

In the design of the TSS system, the complex call processing process, CallProcessing-Processes PC, is divided into seven finite state processes in which each of them is only handle a limited and timely continuous operation. This is a typical real-time technique that guarantees rigid system timing for complicated real-time multi-threads dispatching. Further details of real-time system dispatching will be described in dynamic behaviors modeling of the TSS system. Based on the refined specifications and denotational mathematical models, code can be derived easily and rigorously, and tests of the code can be generated prior to the coding phase.

THE DYNAMIC BEHAVIOR MODEL OF THE TSS SYSTEM

Dynamic behaviors of a system are run-time process deployment and dispatching mechanisms based on the static behaviors. Because system static behaviors are a set of component processes of the system, to put the static processes into a live and interacting system at run-time, the dynamic behaviors of the system in terms of process deployment and dispatches are yet to be specified.

With the work products developed in the preceding section as a set of static behavioral processes of the TSS system, this section describes the dynamic behaviors of TSS at run-time using a three-step refinement strategy via process priority allocation, process deployments, and process dispatches.

TSS Process Priority Allocation

The process priority allocation of system dynamic behaviors is the executing and timing requirements of all static processes at run-time. In general, process priorities can be specified at 4 levels for real-time and nonreal-time system in an increasing priority known as: L1: base level processes, L2: high level processes, L3: low interrupt level processes, and L4: high interrupt level processes. The L1 and L2 processes are system dynamic behaviors that are executable in normal sequential manner. However, the L3 and L4 processes are executable in cyclic manner triggered by certain system timing interrupts. It is noteworthy that some of the priority levels may be omitted in modeling a particular system, except the base level processes. That is, all systems encompass at least a base level process, particularly a nonreal-time or transaction system.

According to the RTPA system modeling and refinement methodology (Wang, 2007a), the first step refinement of the dynamic behaviors of the TSS system on process priority allocation

![Figure 21. Process priority allocation of TSS dynamic behaviors](image-url)
can be specified as shown in Figure 21. It may be observed that all transactional processes at run-time, such as SystemInitialPC and the seven call processing processes, are allocated at the base level, therefore there is no high level processes in the TSS system. However, the processes with strict timing constraints, such as LineScanningPC, SysClockPC, and DigitalReceivingPC, are allocated as low or high level interrupt processes dependent on their timing priorities and executing frequencies.

**TSS Dynamic Process Deployment**

*Process deployment* is a dynamic behavioral model of systems at run-time, which refines the timing relations and interactions among the system, system clock, system interrupts, and all processes at different priority levels. Process deployment is a refined model of process priority allocation for time-driven behaviors of a system. On the basis of the process priority allocation model as developed in previous subsection in Figure 21, the TSS dynamic behaviors can be further refined with a process deployment model as shown in Figure 22, where precise timing relationships between different priority levels are specified.

In Figure 22, § represents the system at top level where all external, timing and interrupt events, SystemInitialS, SysClock1msInt, and @SysClock100msInt, are captured. The big-R notation indicates that, after the TSS system is initialized, the seven call processing processes collectively represented by CallProcessingPC are repetitively executing at the base level until SysShutdownBL = T. The base level operations may be interrupted when a cyclic timing interrupt, such as SysClock1msInt and SysClock100msInt, is captured by the system, then one or a set of predesignated interrupt level processes will be invoked. At the completion of an execution of any interrupt process, the system will return to the interrupt point of the base level process where it was interrupted.

**TSS Dynamic Process Dispatching**

*Process dispatch* is a dynamic behavioral model of systems at run-time, which refines relations between system events and processes. Dynamic process dispatch specifies event-driven behaviors of a system. In the TSS system, the iterative call processing process, CallProcessingPC, is a complex process that can be further refined in a system process dispatching framework as shown in Figure 23.

The TSS process dispatching model specifies that the system iteratively handles each of the 16 subscriber requests for call processing when the i-th CallRecord(i)ST.CallStatusBL = T. Then, the system adopts a switch structure to handle one of the seven possible line status represented by the values of CallRecord(i)ST.CallProcessN. Based on the value of the current call process number, a preallocated process will be dispatched except CallRecord(i)ST.CallProcessN = 0.

As specified in Figure 23, the CallProcessingPC process is a complex process with seven state-transition processes for controlling a call
from origination to termination. Because the TSS system 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, the switching functions for an individual call are divided into seven coherent states, corresponding to the seven dispatching processes as modeled in Figure 23.

The practical formal engineering method of RTPA for system modeling and specification provides a coherent notation system and systematic methodology for large-scale software and hybrid system design and implementation. The formal design models and their refinements demonstrate a typical system modeling paradigm of the entire architectures, static behaviors, and dynamic behaviors of the TSS system according to the RTPA specification and refinement methodology. The final-level refinements of the TSS specifications provide a set of detailed and precise design blueprints for seamless code generation, system implementation, tests, and verifications.

**CONCLUSION**

This article has demonstrated that a complex real-time Telephone Switching System (TSS), including its architecture, static behaviors, and dynamic behaviors, can be formally and efficiently described by RTPA. On the basis of the RTPA methodologies, this article has systematically developed a formal design model...
of the TSS system in a top-down approach. The architectural model of the TSS system has been created using a set of UDMs. The static behaviors of the TSS system have been modeled by a set of call processing processes. The dynamic behaviors of the TSS system have been specified and refined by a set of process priority allocation, process deployment, and dispatching models.

Based on the rigorous design models and the formal framework of the TSS system, program code can be seamlessly derived. The formal model of TSS may not only serve as a formal design paradigm of real-time software systems, but also a test bench of the expressive power and modeling capability of exiting formal methods in software engineering. Related real-world case studies on formal system modeling and refinement in RTPA may be referred to (Wang and Ngolah, 2003; Wang and Zhang, 2003; Wang et al., 2009; Tan et al., 2004; Ngolah et al., 2004). Since the equivalence between software and human behaviors, RTPA may also be use to describe human dynamic behaviors and mental processes (Wang, 2003, 2007b; Wang and Ruhe, 2007).

ACKNOWLEDGMENT

The author would like to acknowledge Natural Science and Engineering Council of Canada (NSERC) for its partial support to this work. The author would like to thank the anonymous reviewers for their invaluable comments that have greatly improved the latest version of this article.

REFERENCES


Yingxu Wang is professor of cognitive informatics and software engineering, Director of International Center for Cognitive Informatics (ICCI), and Director of Theoretical and Empirical Software Engineering Research Center (TESERC) at the University of Calgary. He is a Fellow of WIF, a P.Eng of Canada, a Senior Member of IEEE and ACM, and a member of ISO/IEC/TC/1 and the Canadian Advisory Committee (CAC) for ISO. He received a PhD in Software Engineering from The Nottingham Trent University, UK, in 1997, and a BSc in Electrical Engineering from Shanghai Tiedao University in 1983. He has industrial experience since 1972 and has been a full professor since 1994. He was a visiting professor in the Computing Laboratory at Oxford University in 1995, Dept. of Computer Science at Stanford University in 2008, and the Berkeley Initiative in Soft Computing (BISC) Lab at University of California, Berkeley in 2008, respectively. He is the founder and steering committee chair of the annual IEEE International Conference on Cognitive Informatics (ICCI). He is founding editor-in-chief of International Journal of Cognitive Informatics and Natural Intelligence (IJCINI), founding editor-in-chief of International Journal of Software Science and Computational Intelligence (IJSSCI), associate editor of IEEE Trans on System, Man, and Cybernetics (A), and editor-in-chief of CRC Book Series in Software Engineering. He is the initiator of a number of cutting-edge research fields and/or subject areas such as cognitive informatics, cognitive computing, abstract intelligence, denotational mathematics, theoretical software engineering, coordinative work organization theory, cognitive complexity of software, and built-in tests. He has published over 105 peer reviewed journal papers, 193 peer reviewed conference papers, and 12 books in cognitive informatics, software engineering, and computational intelligence. He is the recipient of dozens international awards on academic leadership, outstanding contribution, research achievement, best paper, and teaching in the last 30 years.