The Formal Design Model of a Real-Time Operating System (RTOS+): Static and Dynamic Behaviors

Yingxu Wang, University of Calgary, Canada
Guangping Zeng, University of Science and Technology, China and University of California, Berkeley, USA
Cyprian F. Ngolah, Sentinel Trending & Diagnostics Ltd., Canada
Philip C.-Y. Sheu, Wuhan University, China and University of California, Irvine, USA
C. Philip Choy, University of Calgary, Canada
Yousheng Tian, University of Calgary, Canada

ABSTRACT

A real-time operating system (RTOS) provides a platform for the design and implementation of a wide range of applications in real-time systems, embedded systems, and mission-critical systems. This paper presents a formal design model for a general RTOS known as RTOS+ that enables a specific target RTOS to be rigorously and efficiently derived in real-world applications. The methodology of a denotational mathematics, Real-Time Process Algebra (RTPA), is described for formally modeling and refining architectures, static behaviors, and dynamic behaviors of RTOS+. The conceptual model of the RTOS+ system is introduced as the initial requirements for the system. The architectural model of RTOS+ is created using RTPA architectural modeling methodologies and refined by a set of Unified Data Models (UDMs). The static behaviors of RTOS+ are specified and refined by a set of Unified Process Models (UPMs). The dynamic behaviors of the RTOS+ system are specified and refined by the real-time process scheduler and system dispatcher. This work is presented in two papers in serial due to its excessive length. The static and dynamic behavioral models of RTOS+ is described in this paper; while the conceptual and architectural models of RTOS+ has been published in IJSSCI 2(2).

Keywords: Design Frameworks, Formal Design Models, Real-Time Operating System, Software Engineering, System Architecture Specification, Unified Data Models, Unified Process Models

DOI: 10.4018/jssci.2010070106
INTRODUCTION

An operating system is a set of integrated system software that organizes, manages, and controls the resources and computing power of a computer or a computer network. It also provides users a logical interface for accessing the physical machine in order to run applications. A general-purpose operating system may be perceived as an agent between the hardware and resources of a computer and the applications and users. An operating system may be divided into three subsystems known as those of the kernel or system management, the resource management, and the process management (Dijkstra, 1968; Brinch-Hansen, 1971; Liu & Layland, 1973; Peterson & Silberschatz, 1985; Anderson et al., 1989; McDermid, 1991; Deitel & Kogan, 1992; Tanenbaum, 1994; Viscarola & Mason, 2001; Silberschatz et al., 2003; Wang, 2004, 2007). The kernel of an operating system is a set of central components for computing, including CPU scheduling and process management. The resource management subsystem is a set of supporting functions for various system resources and user interfaces. The process management subsystem is a set of transition manipulation mechanisms for processes and threads interacting with the system kernel and resources.

A real-time operating system (RTOS) is an operating system that supports and guarantees timely responses to external and internal events of real-time systems (Laplace, 1977; Sha et al., 1990; ISO/IEC, 1996; Ford, 1997; Bollella et al., 2002; Kreuzinger et al., 2002; Ngolah, Wang, & Tan, 2004). An RTOS monitors, responds to, and controls an external environment, which is connected to the computer system through sensors, actuators, or other input-output (I/O) devices. In a real-time system in general and an RTOS in particular, the correctness of system behaviors depends not only on the logical results of computation but also on the time point at which the results are obtained. Real-time systems can be divided into hard and soft real-time systems. In the former, a failure to meet timing constraints will be of serious consequences, while in the latter, a timing failure may not significantly affect the functioning of the system.

A great variety of RTOS’s have been developed in the last decades (Lewis & Berg, 1998; Labrosse, 1999; Yodaiken, 1999; Riv & Harbour, 2001; Lamie, 2008; ETTX, 2009). The existing RTOS’s are characterized as target-machine-specific, implementation-dependent, and not formally modeled. Therefore, they are usually not portable as a generic real-time operating system to be seamlessly incorporated into real-time or embedded system implementations. Problems often faced by RTOS’s are CPU and tasks scheduling, time/event management, and resource management. Efficient and precise methodologies and notations for describing solutions to these problems are critical in RTOS design and implementation. Generally, RTOS’s require multitasking, process threads, and a sufficient number of interrupt levels to deal with the random interrupt mechanisms in real-time systems. In modern RTOS’s, multitasking is a technique used for enabling multiple tasks to share a single processor. A thread is an individual execution of a process in order to handle concurrent tasks. In addition, an interrupt is a request of an external device or internal process that causes the operating system to suspend the execution of a current low priority task, serve the interrupt, and hand control back to the interrupted process.

This paper develops a comprehensive design paradigm of the formal real-time operating system (RTOS+) in a top-down approach on the basis of the RTPA methodology. The conceptual model, architectural model, and the static/dynamic behavioral models of RTOS+ are systematically presented. In the remainder of this paper, the conceptual model of RTOS+ is described as the initial requirements for the system. The architectural model of RTOS+ is created based on the conceptual model using the RTPA architectural modeling methodologies and refined by a set of unified data models (UDMs). Then, the static behaviors of RTOS+ are specified and refined by a set of unified process models (UPMs). The dynamic behaviors of RTOS+ are specified and refined by process
priority allocation, process scheduling, and system dispatching models. Due to its excessive length and complexity, this paper presents the second part of RTOS+ on its static and dynamic behavioral models following in serial with the conceptual and architectural models of RTOS+ published in IJSSCI 2(2) (Wang et al., 2010c).

THE STATIC BEHAVIORAL MODELS OF THE RTOS+ SYSTEM

According to the RTPA methodology, a static behavior is an encapsulated function of a given system that can be determined before run-time (Wang, 2007). On the basis of the system architecture specifications and with the UDMs of system architectural components developed in the preceding section, the operational components of the given RTOS+ system and their behaviors can be specified as a set of UPMs as behavioral processes operating on the UDMs.

The basic functions of operating systems can be classified as system management, resources management, and processes management. The high-level static behaviors of RTOS+, RTOS+$\$.StaticBehaviorsPC, encompasses three process subsystems such as SystemManagementPC, ResourcesManagementPC, and ProcessesManagementPC in parallel as specified below:

$$\text{RTOS+}\$.StaticBehaviorsPC \triangleq \text{SystemManagementPC} \parallel \text{ResourcesManagementPC} \parallel \text{ProcessesManagementPC} \quad (12)$$

The high level specification of RTOS+$\$.StaticBehaviorsPC as given in Eq. 12 can be further extended and refined as shown in Figure 13 by a set of UPMs. Each schema of the UPMs in Figure 13 models the input data objects (<I:: (..)>), output data objects (<O:: (..)>), and operated UDMs (<UDM:: (..)>) for a specific behavioral process of RTOS+. The UDMs play an important role in system architectural design as global and permanent I/O structures, which usually have a longer life-span than those of the process created or invoked them, particularly in a real-time system.

The following subsections describe how the RTOS+ static behaviors in the three subsystems as specified in Figure 13 are modeled and refined using the denotational mathematical notations and methodologies of RTPA (Wang, 2007, 2008a).

UPMs for System Management in RTOS+

According to the high level specifications of the static behaviors of RTOS+ as given in Figure 13, the system management subsystem of RTOS+ encompasses a set of five behavioral processes such as system initialization, system clock manipulation, system event capture, Device interrupt handling, and the CPU scheduler, i.e.:

$$\text{SystemManagementPC} \triangleq \begin{cases} \text{ SysInitialPC (<I:: (..)>; <O:: (..)>; <UDM:: SCB, PCB, DCB, ECB, MCB, SysClock>)} \\ \text{ SysClockPC (<I:: @1ms⊙; <O:: §tN, §tch:mm:ss:ms>)} \\ \text{ SysEventCapturePC (<I:: (..)>; <O:: RequestQST>)} \\ \text{ DeviceInterruptHandlingPC (<I:: @DeviceInt⊙; <O:: InterruptedQST>)} \\ \text{ CPUSchedulingPC (<I:: ReadyQHST, ReadyQLST>; <O:: CompletedQST, DelayedQST, InterruptedQST, SuspendedQST>)} \end{cases} \quad (13)$$

The detailed UPMs of the system management processes of RTOS+ are refined and elaborated in the following subsections.

A) The System Initialization Process

The system initialization process boots the entire operating system, sets its initial environment, and pre-assigns the initial values of data objects of the system such as variables, constants, as well as architectural (hardware interface) and control (internal) UDMs. Initialization is crucially important for a real-time operating...
system as well as its control logic specified in the behavioral processes.

The system initialization process of RTOS+, SysInitialPC, is modeled in Figure 14, where all system architectural and control UDMs are initialized as specified in the architectural models. It also sets the predetermined initial values of internal control models, system status, and system events. The central function of SysInitialPC is the initialization of the global system control block SCBST, which registers and monitors the system processes, resources, events, and status as well as the current sizes of all dispatching queues.

**B) The System Clock Process**

The system clock process maintains and updates an absolute (calendar) clock and a relative clock for the operating system in order to synchronize all activities, behaviors, events, and states of RTOS+. The system clock process of RTOS+, SysClockPC, is modeled in Figure 15. The source of the system clock is obtained from the 1ms time interrupt signal generated by the system hardware, by which the absolute clock with real-time millisecond, second, minute, and hour, $t_{c} = hh:mm:ss:ms$, are generated and periodically updated. The second clock in RTOS+ is the relative clock, $t_{r} = N$, which is adopted for relative timing and duration manipulations. SysClockPC updates the relative clock $t_{r} = N$ per millisecond, and resets it to zero at midnight every day in order to prevent it from overflow.

In addition to the function of system clock maintaining, SysClockPC also updates all timers created by application processes by decreasing its current value by 1ms if it has not reached the timeout status, i.e., $Timer(iN)ms = 0$. The timeout event of a specific timer will be de-
ected by the system in order to trigger a suitable action in a corresponding application process identified by an ECBST and the event PCB(PNN)ST.TimerStatusBL:= T.

C) The System Event Capture Process

Events capture and handling are important behaviors of a real-time operating system. The event types in RTOS+ can be classified into operational, time, and interrupt events, which are identified by the common event prefix @ and the type suffixes S, TM, and ⊙, respectively, where TM is an abbreviation of time TI = hh:mm:ss:ms, date D = yy:MM:dd, and date-time DT = yyyy:MM:dd:hh:mm:ss:ms. Although a time event @tTM occurs at a given system time point or periodically, the operational event @eS and interrupt event @e⊙ may occur randomly. A special kind of operational events is the exception events, @eS(ex), which is detected by the system exception detection process !(@exS) in order to log a detected exception event at run-time. The RTPA exception detection mechanism provides a fundamental process for safety and dependable system specification, which enables system exception detection, handling, or postmortem analysis to be implemented.

The system event capture process of RTOS+, SysEventCapturePC, is a low interrupt level process that monitors and handles operational events (@eS) generated by a device.
as shown in Figure 16. SysEventCapturePC detects any event occurred externally in order to create a corresponding event control block, ECBST, as modeled in the UDMs of RTOS+. A detected device I/O event will be used to create a new process required by the event. Processing separately as that of external (device) events, however, the time and interrupt events will be handled by system dynamic behaviors (Figure 30) and DeviceInterruptHandlingPC (Figure 17), respectively.

D) The Device Interrupt Handling Process

The interrupt mechanism enables execution priority change and control-taking-over between processes. An interrupt, denoted by $\uparrow\downarrow$, is a parallel process relation in which a running process $P$ is temporarily held before termination by another higher priority process $Q$ triggered by an interrupt event @e* at the interrupt point *. The interrupted process $P$ will then be resumed after process $Q$ has been completed, i.e.:
$P \downarrow Q = P \parallel (@int^* \uparrow Q \downarrow \ast) \quad (14)$

where $\uparrow$ and $\downarrow$ denote an interrupt service and an interrupt return.

Therefore, an interrupt-driven dispatch, denoted by $\downarrow$, is a process operation in which the $i$th process $P_i$ is triggered by a predefined system interrupt $@int^*$, i.e.:

$@int^* \downarrow P_i, i \in \{1, \ldots, n\} \quad (15)$

The device interrupt handling process of RTOS+, $DeviceInterruptHandlingPC$, is a low-level interrupt service process that monitors and handles device interrupt events ($@int^*$) from port interfaces of RTOS+ as shown in Figure 17. $DeviceInterruptHandlingPC$ detects any interrupt requested by a device or system resource in order to create a corresponding event control block from the interrupt in ECBST as modeled in the UDMs of RTOS+. A detected interrupt will be used to dispatch a corresponding interrupt service process after the current base level process is scheduled into the interrupted queue, $InterruptedQST$, by the system process scheduler ($ProcessSchedulerPC$).
As illustrated in Figure 2, once a device interrupt is identified, the system control on the CPU is automatically passed to the interrupt service process, which responds to the interrupt and then returns control to the interrupted task. The interrupted task is then sent back to the ready state after the completion of the interrupt service. An interrupted task may be killed from the system if resources are not enough for servicing a high priority interrupt. RTOS+ allows hierarchical interrupts where a high-level interrupt may take over control from other lower-level ones.

In RTOS+, the periodic time interrupts, @SysClock1msInt and @SysClock10msInt, are the highest level system interrupts in the kernel of RTOS+, which are handled differently as those of the random device and resource interrupts. The periodic time interrupts are detected and handled by the kernel RTOS+§ SysDispatcherPC directly, which will be described in Figure 30.

**E) The CPU Scheduling Process**

CPU scheduling is the most inner kernel process of a real-time operating system to maximize CPU utilization (Brinch-Hansen, 1971). RTOS+ adopts multiprocess and multithread techniques to keep the CPU running different processes or threads of multiple processes on a time-sharing basis with a predetermined scheduling interval. The CPU scheduler selects which process in the ready queue should be run next based on a predefined scheduling algorithm or strategy. The CPU scheduler switches control of the CPU to the process selected for a certain time interval.

---

**Figure 17. The behavior model of the device interrupt handling process**

```plaintext
DeviceInterruptHandling(<<D:: @DeviceIntγ>; <<O:: InterruptedST>;)
    <<UDM:: SCBST, DCSB, ECB, MCBST, CPUST>> PC Δ

{ // Swap out current base level process
    PORTST(IntPortAddress)B > IntVectorB
    DNN := IntVectorB
    ↑(SCBST#InterruptsN)
    ENN := SCBST#InterruptsN
    ECB(ENN).ST.StatusBL := T
    ECB(ENN).ST.TypeN := 3 // Device interrupt
    ECB(ENN).ST.TimeDetectedhh:mm:ss.ms := δhh:mm:ss.ms
    PN := SCBST.CurrentPN
    ECB(ENN).ST.SourceProcN := PN
    ECB(ENN).ST.TargetProcN := 0
    EnqueuePC(<D:: PN>; <O:: ( ); <UDM:: InterruptedST>) // Swap interrupted process
    PCB(PN).ST.TaskStatusN := 7 // Interrupted
    ↑(SCBST#ProcInterruptedN)

// Run interrupt service process for the device request
    DCB(DNN).ST.RecProcN := 0
    DCB(DNN).ST.StatusN := 2 // In use
    DCB(DNN).ST.TimeAllocatedhh:mm:ss.ms := δhh:mm:ss.ms
    PN := DCB(DNN).ST.IntServiceProcN
    PCB(PN).ST.TaskStatusN := 3 // Running
    SCBST.CurrentPN := PN
    SCBST.@RunEndBL := F
    RunPC(<D:: PN>; <O:: SCBST.@RunEndBL>; <UDM:: SCBST, PCB, DCSB, ECB, MCBST, CPUST>)
}
```
In RTOS+, every ten time-intervals (10ms) forms a cycle to run all ready processes in the ReadyQHST (maximum 6) and ReadyQLST (maximum 4) as specified in the design of the system control block SCBST. The CPU scheduler process of RTOS+, CPUSchedulerPC, as shown in Figure 18, dispatches a maximum of ten processes in both ready queues ReadyQHST and ReadyQLST within a low interrupt period 10ms, where each of them will be dispatched for running for 1ms scheduled by the high interrupt period. In the peak CPU load period of RTOS+, 6 tasks from ReadyQHST and 4 from ReadyQLST will be scheduled, respectively, in a series of ten 1ms time intervals. However, during low CPU load period, any combination of high/low tasks may be scheduled including a series of multiple threads of the same process. In case there is a current task in the CPU but no other task in the ReadyQHST and ReadyQLST, the currently scheduled task may continue to run for at least another 1ms interval until the statuses of the ready queues are changed. Except the above two special conditions, the current process will be swapped out of the CPU as a delayed process. Instead, a new qualified process in the front of ReadyQHST or ReadyQLST will be scheduled into CPU for running.

CPUSchedulerST is triggered by @SysClock1msInt* per 1ms. As modeled in Figure 18, when there is no currently dispatched task in the CPU as well as in either ready queues, ReadyQHST or ReadyQLST, the scheduler does nothing but exits the current cycle of CPU scheduling.

**UPMS FOR RESOURCES MANAGEMENT IN RTOS+**

According to the high level specifications of the static behaviors of RTOS+ as given in Figure 13, the resources management subsystem of RTOS+ encompasses the behavioral processes of memory management, device (port) management, and file management, i.e.:

\[
\text{ResourcesManagementPC} \triangleq (\text{AllocateMemoryPC}(<\mathbf{i}:: \text{PN}>; <\mathbf{o}:: \text{MemID, &MemAllocatedBL}>; <\text{UDM}:: \text{SCBST, PCBST, ECBST, RCBST, MCBST}>)) \notag \]
\[
\text{ReleaseMemory}(<\mathbf{i}:: \text{MemID}>; <\mathbf{o}:: \&\text{MemReleased}>; <\text{UDM}:: \text{SCBST, PCBST, ECBST, MCBST}>)) \notag \]
\[
\text{GetDevicePC}(<\mathbf{i}:: \text{DeviceType, PN}>; <\mathbf{o}:: \&\text{DeviceAllocated}>; <\text{UDM}:: \text{SCBST, DCBST, PCBST}>)) \notag \]
\[
\text{ReleaseDevicePC}(<\mathbf{i}:: \text{DN}>; <\mathbf{o}:: ( )>; <\text{UDM}:: \text{SCBST, DCBST}>)) \notag \]
\[
\text{FileManagerPC}(<\mathbf{i}:: \text{FileID}>; <\mathbf{o}:: ( )>; <\text{UDM}:: \text{Files}>)) \notag \]

(16)

The detailed UPMs of the resource management processes of RTOS+ are refined and elaborated in the following subsections.

**A) The Memory Management Process**

Memory management is one of the key functions of operating systems because memory is both the working space and storage of data and files. RTOS+ adopts a flexible segmentation method for system memory management with variable sizes of memory blocks for different events and processes. The memory management process of RTOS+, MemoryManagementPC, encompasses the AllocateMemoryPC and ReleaseMemoryPC processes, as shown in Figure 19. MemoryManagementPC supports dynamic and absolute physical memory manipulations that are necessary for real-time operating systems and real-time applications.

The process AllocateMemoryPC deals with dynamic memory allocations in RTOS+. When a request for memory from a process is identified by the system, AllocateMemoryPC searches for a suitable sized memory block that is free. When a suitable memory block is identified by a given memory block number (MBN), AllocateMemoryPC sets its status as in-use. Then, the associated process requiring the memory will be registered in the memory control block. Inversely, the number of the allo-
cated memory block, MemIDN = MBNN, will be registered to the requiring process. If there is no free and suitable memory block available in the system, a feedback status, &MemAllocatedBL:= F, will be generated along with a system warning for the exception condition.

The process ReleaseMemoryPC deals with dynamic memory release requests of the system when a process terminates and the memory allocated to it is no longer required. ReleaseMemoryPC operates on the memory control block when it is invoked. It gets the memory block number associated to the process that requests to release the memory. Then, it sets the status of the given memory block to free

---

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
and disconnects the link between the memory block and the previously occupied process.

**B) The Device Management Process**

Devices as well as their I/O ports are directly manipulated by the processes DeviceManagementPC in RTOS+ as shown in Figure 20. DeviceManagementPC encompasses both GetDevicePC and ReleaseDevicePC processes. Additional plug-in device drivers may also be incorporated via the same interface mechanisms by the adding device and deleting device processes.

The process GetDevicePC handles device allocations in RTOS+. When a request for a device from a process is identified by the system, GetDevicePC searches for a suitable type of device that is free. When a suitable device is found, GetDevicePC sets its status as seized. Then, the associate process required the device is registered in the device control block. If there is no free and/or suitable device available in the system, a feedback status, &DeviceAllocatedBL := F, will be generated.

The process ReleaseDevicePC handles device release requests for the system when a process terminates the use of a device. ReleaseDevicePC operates on the device control block when it is invoked. It first gets the device number (DNN) to be released associated to the process. Then, it sets the status of the given device in DCBST as free and disconnects the link between the device and the previously associated process.

**C) The File Management Process**

On the basis of the architectural model of the file system FilesST as given in Figure 10, the behaviors of the file management subsystem of RTOS+, FileManagementPC, are modeled by a set of 14 behavioral processes in the categories of file system administrations and file manipulations as shown in Figure 21. Detailed RTPA models of the file management processes have been provided in (Wang, 2007).

**UPMS FOR PROCESS MANAGEMENT IN RTOS+**

A process is a basic unit of system function that represents an execution of a program on a computer under the support of an operating system. A process can be a system process or a user process. The former executes system code, and the latter runs an application. Processes may be executed sequentially or concurrently depending on the type of operating systems.

A thread is an important concept of process management in operating systems (Anderson et al., 1989; Lewis & Berg, 1998). A thread is a basic unit of CPU utilization, or a flow of control within a process, supported by a PCBST, a program counter, a set of registers, and a stack. Conventional operating systems are single thread systems. Multithreaded systems enable a process to control a number of execution threads. The benefits of multithreaded operating system are responsiveness, resource sharing, implementation efficiency, and utilization of multiprocessor architectures of modern computers.

According to the high level specifications of the static behaviors of RTOS+ as given in Figure 13, the process management subsystem of RTOS+ encompasses a set of nine task handling processes as specified in Figure 22 through 29 corresponding to the states of behavioral processes for the entire task lifecycle of process manipulating as shown in Figure 2, i.e.:

```plaintext
ProcessesManagementPC ( \begin{align*} 
\text{ProcessRequestPC} &: \langle i \rangle \rightarrow \text{TaskPriority}, \text{MemoryReqB}, \text{DeviceReqH}, \text{RequestQST} >; \\
\text{ProcessCreationPC} &: \langle i \rangle \rightarrow \text{RequestQST} >; \\
\text{ProcessRunningPC} &= \text{CPUSchedulerPC}; \\
\text{ProcessCompletedPC} &: \langle i \rangle \rightarrow \text{CompletedQST} >; \\
\text{ProcessInterruptedPC} &: \langle i \rangle \rightarrow \text{InterruptQST} >; \\
\end{align*} 
) 
```

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
The following subsections formally describe the refined UPMs of process management in RTOS+ such as those of process request, creation, ready, running, completion, interrupted, delayed, suspended, and killed. The relationship and interactions of the nine task handling processes will be modeled in the dynamic behaviors of RTOS+.

A) The Request Process

The request process of RTOS+, ProcessRequestPC, is modeled as shown in Figure 22, which operates on the UDMs of SCBST, PCBST, and RequestQST as specified in the architectures of RTOS+. ProcessRequestPC requires RTOS+
to establish a process for a user application. It creates a new PCB in PCBST with information such as TaskPriorityN, MemoryReqB, and DeviceIDS in order to specify the priority of the target process and to allocate required memory and device(s). Then, it queues the process request into RequestQST for further processing by ProcessSchedulerPC.

B) The Creation Process

The creation process of RTOS+, ProcessCreationPC, is modeled as shown in Figure 23, which operates on the UDMs of SCBST, PCBST, DCBST, ECBST, MCBST, RequestST, WaitingQST, and KilledQST as specified in the architectures of RTOS+. For each user request in applications, ProcessCreationPC allocates both required memory and device for the process. Then, it completes the creation process by queuing the process into WaitingQST. For a process under creation, if there is not enough memory or the required device is busy, it will be sent into the killed queue KilledQST. In this case, a proper system exception log and related warning information will be generated.

C) The Ready Process

The ready process of RTOS+, ProcessReadyPC, is modeled as shown in Figure 24, which operates on the UDMs of SCBST, PCBST, DCBST, ECBST, MCBST, WaitingQST, ReadyQHST, and ReadyQLST as specified in the architectures of RTOS+. ProcessReadyPC first fetches the front process in the waiting queue, WaitingQST, if it is not empty. Then, the process will be
dispatched to ReadyQHST or ReadyQLST depending on the specified priority of the task.

The capacity of ReadyQHST is designed for six high-priority processes and that of ReadyQLST for four low priority processes, which fit the maximum dispatching cycle of 10ms for ten 1ms executing intervals for each of these ready processes. Therefore, when either target ready queue is full, the process will be re-queued back to WaitingQST for future dispatching. A process on the scheduling states of creation, interrupted, and delayed can be sched-
uled into the ready queues as soon as the queues are not full. However, a suspended process may do so only when its required resource/device has become available.

D) The Running Process

It is noteworthy that the process State 3, ProcessRunningPC, in the state transition diagram as given in Figure 2 is not dispatched at the base level as other state transition processes. Due to its central role in the kernel of RTOS+, ProcessRunningPC is executed at the high periodic interrupt level per 1ms known as the process CPUSchedulerPC as specified in Figure 18.

E) The Completed Process

The completed process of RTOS+, ProcessCompletedPC, is modeled as shown in Figure 25, which operates on the UDMs of SCBST, PCBST, DCBST, ECBST, MCBST, and CompletedQST as specified in the architectures of RTOS+. ProcessCompletedPC releases the front process in the CompletedQST if the queue is not empty. It first fetches the process number...
PNN by which the specific PCB in PCBST is found. Then, the occupied memory, device, and event control blocks associated with the process are released. As a result, the task status of PNN is set as 10 to indicate a free process block in PCBST for new tasks. In addition, the system control numbers of completed processes #Proc-CompletedN is updated in SCBST.

**F) The Interrupted Process**

The interrupted process of RTOS+, ProcessInterruptedPC, is modeled as shown in Figure 26, which operates on the UDMs of SCBST, PCBST, DCBST, ECBST, MCBST, CPUST, ReadyQHST, ReadyQLST, InterruptedQST, and KilledST as specified in the architectures.
of RTOS+. ProcessInterruptedPC handles the process in front of InterruptedQST if it is not empty. The interrupted process caused by a device interrupt is re-enqueued into ReadyQHST or ReadyQLST according to its task priority. During the re-schedule of an interrupted process to the ready queues, if the target queue is full, the process will be returned to InterruptedQST if it is not full as well; Otherwise, the interrupted process will be killed due to lack of a resource. In this case, a proper system exception log and related warning information will be generated.

G) The Delayed Process

The delayed process of RTOS+, ProcessDelayedPC, is modeled as shown in Figure 27, which operates on the UDMs of SCBST, PCBST, DCBST, ECBST, MCBST, ReadyQHST, ReadyQLST, and DelayedQST as specified in the architectures of RTOS+. ProcessDelayedPC handles the first process in front of DelayedQST if it is not empty. A high priority delayed process will be moved into ReadyQHST, while a low priority delayed process will be moved into ReadyQLST, if the queues are not full limited by six high priority processes and four low priority ones, respectively. When there is no available ready queue, the delayed process is re-queued back into DelayedQST when it is not full; Otherwise, the delayed process will be killed due to lack of a resource. In this case, a proper system exception log and related warning information will be generated.

H) The Suspended Process

The suspended process of RTOS+, ProcessSuspendedPC, is modeled as shown in Figure 28, which operates on the UDMs of SCBST, PCBST, DCBST, ECBST, MCBST, ReadyQHST, ReadyQLST, SuspendedQST, and KilledST as specified in the architectures of RTOS+. ProcessSuspendedPC handles the first processes in front of SuspendedQST if it is not empty. Because the reason for a process’ suspension was the non-availability of a certain device required by the process, ProcessSuspendedPC checks if the device identified by DNN in DCBST has become free or not. If the current status of the device remains unchanged, the suspended process is returned into SuspendedQST; Otherwise, the suspended process can be re-enqueued into ReadyQHST or ReadyQLST according to its task priority. During the re-schedule of a process to the ready queues, if the target queue is full, the target process will be returned to SuspendedQST when it is not full as well; Otherwise, the suspended process will be killed due to lack of a resource. In this case, a proper system exception log and related warning information will be generated.
I) The Killed Process

The killed process of RTOS+, ProcessKilledPC, is modeled as shown in Figure 29, which operates on the UDMs of SCBST, PCBST, DCBST, ECBST, MCBST, and KilledQST as specified in the architectures of RTOS+. ProcessKilledPC releases the front process in the KilledQST if the queue is not empty. It fetches the process number PNN in order to find the specific PCB in PCBST. Then, the occupied memory, device, and event control blocks associated with the
process are released. As a result, the task status of PNN is set as 10 to indicate a free process block in PCBST for new tasks. In addition, the system control number of killed processes #ProcKilledN is updated in SCBST.

THE DYNAMIC BEHAVIORAL MODELS OF THE RTOS+ SYSTEM

Dynamic behaviors of a system are run-time process deployment and dispatching mechanisms
Figure 28. The behavior model of the suspended process

```
ProcessSuspended(<l:: SuspendedQST>; <q:: ReadyQHST, ReadyQLST>;
   <UDM:: SCBST, PCBST, DCBST, ECBST, MCBST, KilledQST>>)

   { SuspendedQST.EmptyB = F
     → ServePC(<l:: SuspendedQST>; <q:: PNN>; <UDM:: PCBST>)
     → Down(PCB(PNN).DeviceReqN)
     → DNN := PCBST(PNN).DeviceReqN
     → ( DCBST(DNN).StatusN = 0
         → EnqueuePC(<l:: SuspendedQST>; <q:: ( ); <UDM:: PCBST>)
         → PCB(PNN).TaskStatusN := 7 // Return SuspendedQ
         → ~ (SCBST.#ProcSuspendedN)
     | ~
     → DCBST(DNN).StatusN = 1 // Seize the device
     → ~ PCBST(PNN).TaskPriorityN = 1 // High priority task
     → ( ReadyQHST.FullB = F
         → EnqueuePC(<l:: PNN>; <q:: ( ); <UDM:: ReadyQHST>)
         → PCB(PNN).TaskStatusN := 2 // Ready
         → ~ (SCBST.#ProcReadyH)
     | ~
     → ( SuspendedQST.FullB = F // Return SuspendedQ due to ReadyQH is full
         → EnqueuePC(<l:: PNN>; <q:: ( ); <UDM:: SuspendedQST>)
         → PCB(PNN).TaskStatusN := 7 // Suspended
         → ~ (SCBST.#ProcSuspendedN)
     | ~
     → ! (Process re-schedule for, PNN, failed due to lack of resources.)
     → EnqueuePC(<l:: PNN>; <q:: ( ); <UDM:: KilledQST>)
     → PCB(PNN).TaskStatusN := 8 // Killed
     → ~ (SCBST.#ProcKilledN)
     )
   )

   )

   { ~
     // Low priority task
     → ( ReadyQLST.FullB = F // Less than 4 low priority proc ready
     → EnqueuePC(<l:: PNN>; <q:: ( ); <UDM:: ReadyQLST>)
     → PCB(PNN).TaskStatusN := 2 // Ready
     → ~ (SCBST.#ProcReadyL)
     | ~
     → ( SuspendedQST.FullB = F // Return SuspendedQ due to ReadyQL is full
         → EnqueuePC(<l:: PNN>; <q:: ( ); <UDM:: SuspendedQST>)
         → PCB(PNN).TaskStatusN := 7 // Suspended
         → ~ (SCBST.#ProcSuspendedN)
     | ~
     → ! (Process re-schedule for, PNN, failed due to lack of resources.)
     → EnqueuePC(<l:: PNN>; <q:: ( ); <UDM:: KilledQST>)
     → PCB(PNN).TaskStatusN := 8 // Killed
     → ~ (SCBST.#ProcKilledN)
     )
   )

)
based on the static behaviors modeled in UPMs. The dynamic behaviors of RTOS+, as modeled by RTOS+§.DynamicBehaviorsPC, integrate and interact the static behavioral processes at run-time. With the UPMs developed in the preceding section as a set of static behavioral processes of RTOS+, this section models and elaborates the dynamic behaviors of the RTOS+ system at run-time via the dynamic processes of system dispatcher and process scheduler as follows:

$$\text{RTOS+§.DynamicBehaviorsPC} \triangleq \text{SysDispatcherPC} \mid \text{ProcessSchedulerPC} \quad (18)$$

where the former describes the time- and interrupt-driven mechanisms of the system, and the latter specifies the event-driven mechanism of RTOS+.

**THE SYSTEM DISPATCHER OF RTOS+**

The system dispatcher of RTOS+, RTOS+§.SysDispatcherPC, handles time- and interrupt-driven behaviors of the real-time operating system at the system level. SysDispatcherPC encompasses 5-level processes, as shown in Figure 30, known as the tasks at the base, high-periodic-interrupt, low-periodic-interrupt, device interrupt, and user request interrupt levels, as well as their interactions. The system dispatcher of RTOS+ runs ProcessSchedulerPC (Figure 31) continuously at the base level, after the system is initialized, unless the system is requested to shut down. However, four kinds of interrupts may occur during the running of the base level processes, which are @SysClock-1msInt⊙, @SysClock10msInt⊙, @DeviceInt⊙, and @UserRequestInt⊙ in descending priority. Each interrupt redirects the regular routine processes to more urgent real-time requests such as: a) System clock update (Figure 15) and CPU scheduling (Figure 18) at the high-periodic-interrupt level; b) System event capture (Figure 16) at the low-periodic-interrupt level; c) Device interrupt handling (Figure 17) at the device interrupt level; and d) User’s application interrupt handling (Figure 22) at the user request interrupt level.

In RTOS+, four categories of sources, known as user applications, external events, device I/O interrupts, and system interrupts, may be identified for requesting the creation of a process. An application process is initiated by ProcessRequestPC via a user’s system call, @UserRequestInt⊙, with request information

---

*Figure 29. The behavior model of the killed process*

```
ProcessKilled(<I:: KilledQST>; <O:: ()>;
<UDM:: SCBST, PCBST, ECBST, RCBST, MembST, PORTST, CPUST>)PC ≅

{ KillQST.EmptyBL = F
  -> ServePC(<I:: KilledQST>; <O:: PNG>; <UDM:: PCBST>)
  -> ↓(SCBST,#ProcKilledN)
  → MemIDN := PCBST(PGN).MemoryReqN
  -> ReleaseMemoryPC(<I:: MemIDN>; <O:: MemReleasedBL>;
      <UDM:: SCBST, PCBST, ECBST, RCBST, MCBST>)
  → DNN := PCBST(PGN).DeviceReqN
  -> ReleaseDevicePC(<I:: DNN>; <O:: ()>;<UDM:: SCBST, DCBST>)
  → ECBST(EventReqN).StatusBL := F
  → ECBST(EventReqN).SourceProcN := 0
  → ECBST(EventReqN).TargetProcN := 0
  → PCBST(PGN).TaskStatusN := 10
}
```

---

*Figure 29. The behavior model of the killed process*
in specific PCBST. An external event process is initiated by SysEventCapturePC via @SysClock10msInt⊙ interrupt with request information in specific PCBST. A device I/O interrupt is directly served by a predesigned process, DeviceInterruptHandlingPC, via a corresponding call @DeviceInt⊙ with request information in specific PCBST, where the device interrupt service process is application specific dependent on the function of a specific device. A system interrupt is directly handled by CPUSchedulerPC (Figure 18) and SysClockPC driven by the periodic interrupt @SysClock1msInt⊙.

### THE PROCESS SCHEDULER OF RTOS+

Corresponding to the conceptual state transition diagram as given in Figure 2, the process scheduler of RTOS+ is modeled as shown in Figure 31. An event-driven mechanism is adopted based on the status of a specific process. Because all task transition interfaces of RTOS+ are buffered in different queues according to the FIFO mechanism as shown in Figure 1, in each scheduling cycle, ProcessSchedulerPC handles only one of the processes in each task state, i.e.,

![Process Scheduler Diagram](image)

**Figure 30. The dynamic process dispatching model of RTOS+**

<table>
<thead>
<tr>
<th>RTOS+§.SysDispatcherPC</th>
<th>$\rightarrow$ RTOS+§</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ // Base level tasks</td>
<td></td>
</tr>
<tr>
<td>@SystemInitial§</td>
<td>$\rightarrow$ (SysInitialPC</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$ ProcessSchedulerPC</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$ R</td>
</tr>
<tr>
<td></td>
<td>$\rightarrow$ @SysShutDownBL</td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>$\vdash$ // High-periodic-interrupt level tasks</td>
<td></td>
</tr>
<tr>
<td>@SysClock10msInt⊙</td>
<td>$\vdash$ (SysClockPC</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ CPUSchedulerPC</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ @SysClock10msInt⊙ := off</td>
</tr>
<tr>
<td>$\vdash$</td>
<td></td>
</tr>
<tr>
<td>$\vdash$ // Low-periodic-interrupt level tasks</td>
<td></td>
</tr>
<tr>
<td>@SysClock10msInt⊙</td>
<td>$\vdash$ (SysEventCapturePC</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ @SysClock10msInt⊙ := off</td>
</tr>
<tr>
<td>$\vdash$</td>
<td></td>
</tr>
<tr>
<td>$\vdash$ // Device-interrupt level tasks</td>
<td></td>
</tr>
<tr>
<td>@DeviceInt⊙</td>
<td>$\vdash$ (DeviceInterruptHandlingPC</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ @DeviceInt⊙ := off</td>
</tr>
<tr>
<td>$\vdash$</td>
<td></td>
</tr>
<tr>
<td>$\vdash$ // User-interrupt level tasks</td>
<td></td>
</tr>
<tr>
<td>@UserRequestInt⊙</td>
<td>$\vdash$ (ProcessRequestPC</td>
</tr>
<tr>
<td></td>
<td>$\vdash$ @UserRequestInt⊙ := off</td>
</tr>
<tr>
<td>$\vdash$</td>
<td></td>
</tr>
</tbody>
</table>

$\rightarrow$ RTOS+§
from PCB(PNN)ST.TaskStatusN = 1 through 8 corresponding to the creation through killed processes. Each scheduled process is obtained from the front of a specific queue by using the support process ServePC for queue operations. Then, it returns to the base level control of the system and re-enters the system outer cycles to repeat the same routine processes.

It is noteworthy that some routine processes are not handled by ProcessSchedulerPC in process scheduling, because of their special priority allocation and timing requirements, such as CPUSchedulerPC and all periodic interrupt processes. These exceptions are indicated in Figure 31 by the skip operator (→ ∅) of RTPA.

As shown in Figure 30 and Figure 31, RTOS+ schedules all real-time tasks in the system by the most inner-loop 1ms periodic CPU scheduling, the 10ms periodic process scheduling, and the base level routine processes. SysDispatcherPC schedules the execution of each ready process for a 1ms interval within the 10ms outer loop. This dispatching mechanism guarantees any high priority process will have a chance to be executed for at least once in ten 1ms intervals. Therefore, the entire system behaviors are embodied by the CPUSchedulerPC in the kernel, the ProcessSchedulerPC in the middle, and the SysDispatcherPC on the top of RTOS+.

CONCLUSION

The design of real-time operating systems has been recognized as a comprehensive and complex system design paradigm in computing, software engineering, and information system design. This paper has demonstrated that the RTOS+ system, including its architecture, static behaviors, and dynamic behaviors, can be essentially and sufficiently described by RTPA. On the basis of the formal specifications of RTOS+ by the coherent set of UDMs and UPMs in RTPA, the architectural and behavioral consistency and run-time integrity of an RTOS have been significantly enhanced. It has been identified that RTOS+ can be applied not only as a formal design paradigm of RTOS’s, but also a support framework for a wide range of applications in design and implementation of real-time and embedded systems. The RTOS+ system model may have also provided a test bench for the expressive power and modeling
capability of existing formal methods in software engineering.

With a stepwise specification and refinement methodology for describing both system architectural and operational components, the formal model of the RTOS+ system has provided a foundation for implementation of a derived real-time operating system in multiple programming languages and on different operating platforms. It has also improved the controllability, reliability, maintainability, and quality of the design and implementation in real-time software engineering. The formal models of RTOS+ have been adopted as part of the supporting environment for the implementation of the RTPA-based software code generator (RTPA-CG) (Wang et al., 2010b; Ngolah & Wang, 2009). On the basis of the formal and rigorous models of the RTOS+ system, code can be automatically generated by RTPA-CG or be manually transferred from the formal models.

A series of formal design models of real-world and real-time applications in RTPA have been developed using RTPA notations and methodologies (Wang, 2002, 2007, 2008a, 2008b, 2008c, 2009a; Wang & Huang, 2008) in the formal design engineering approach, such as the telephone switching system (Wang, 2009b), the lift dispatching system (Wang et al., 2009), the automated teller machine (ATM) (Wang et al., 2010a), the real-time operating system (RTOS+), the air traffic control system (to be reported), the railway dispatching system (to be reported), and the intelligent traffic lights control system (to be reported). Further studies have demonstrated that RTPA is not only useful as a generic notation and methodology for software engineering, but also good at modeling human cognitive processes in cognitive informatics and computational intelligence as reported in (Wang, 2008d, 2009a; Wang & Ruhe, 2007; Wang & Chiew, 2010).

ACKNOWLEDGMENT

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

REFERENCES


**Yingxu Wang** is professor of cognitive informatics and software engineering, President of International Institute of Cognitive Informatics and Cognitive Computing (IICICC), 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 JTC1 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 Transactions on System, Man, and Cybernetics (Part A), Associate Editor of ASP Journal on Advanced Mathematics and Applications, and Editor-in-Chief of CRC Book Series in Software Engineering. He is the initiator of a number of cutting-edge research fields such as cognitive informatics, denotational mathematics (such as concept algebra, system algebra, real-time process algebra (RTPA), granular algebra, visual semantic algebra, and inference algebra), abstract intelligence, theoretical software engineering, and built-in tests. He has published over 110 peer reviewed journal papers, 200+ peer reviewed full conference papers, and 12 books in cognitive informatics, software engineering, and computational intelligence. He is the recipient of dozens of leadership, research achievement, best paper, and teaching awards in the last 30 years.

**Guangping Zeng** is a professor of computer science and Director of School of Information Engineering at the University of Science and Technology, Beijing (USTB), China. He received a PhD and MSc in computer science from USTB in 2005 and 1995, respectively, and a BSc in Petroleum Engineering from the Chinese Petrol University in 1984. He was a postdoctoral fellow in EECS at University of California, Berkeley during 2008 to 2009. He was the former head of Department of Computer Science and Technology at USTB in 2003-2005. He is a senior member and a steering committee member of the Chinese Association of AI. He has published over 70 peer-reviewed papers and two books. His research interests are in artificial intelligence, intelligent software, soft-man, virtual robots, embedded networks, and Internet-based computing.

**Cyprian F. Ngolah** received a PhD in Software Engineering from the University of Calgary, Canada in 2006, an MSc in Control Engineering from the University of Bradford, England in
1989, and a BSc in Mathematics and Computer Science from the University of Essex, England in 1988. He taught several computer science and software engineering courses at both the graduate and undergraduate levels for thirteen years at University of Buea, Cameroon. He is currently a senior software engineer in the Research and Development Department of Sentinel Trending & Diagnostics Ltd, Calgary, carrying out research on the development of a neural network for machine condition monitoring and predictive maintenance using vibration analysis. His main research interests are in real-time process algebra and its applications, tool support for formal specification languages, real-time operating systems, formal methods in software engineering and real-time software systems, and artificial neural networks.

Phillip C-Y. Sheu is currently a professor of Computer Engineering, Information and Computer Science, and Biomedical Engineering at the University of California, Irvine. He received his Ph.D. and M.S. degrees from the University of California at Berkeley in Electrical Engineering and Computer Science in 1986 and 1982, respectively, and his B.S. degree from National Taiwan University in Electrical Engineering in 1978. Between 1982 and 1986, he also worked as a computer scientist at Systems Control Technology, Inc., Palo Alto, CA., where he designed and implemented aircraft control systems, and he worked as a product planning engineer at Advanced Micro Devices Inc., Sunnyvale, CA, where he designed and integrated CAD systems. From 1986 to 1988, he was an assistant professor at School of Electrical Engineering, Purdue University. From 1989 to 1993, he was an associate professor of Electrical and Computer Engineering at Rutgers University. He has published two books: (1) Intelligent Robotic Planning Systems and (2) Software Engineering and Environment - An Object-Oriented Perspective, and more than 100 papers in object-relational data and knowledge engineering and their applications, and biomedical computations. He is currently active in research related to complex biological systems, knowledge-based medicine, semantic software engineering, proactive web technologies, and large real-time knowledge systems for defense and homeland security. His current research projects are sponsored by the National Science Foundation, National Institute of Health, and Department of Defense. Dr. Sheu is a Fellow of IEEE.

Philip Choy is a PhD candidate in Software Engineering with the Theoretical and Empirical Software Engineering Research Center (TESERC) at the University of Calgary. He received his BSc (Hon.) in Computer and Information Science from Queen's University, Kingston., Canada in 1985. He is a member of the IEEE and has served on the Canadian Board of the IEEE for over a decade. He has also held a number of executive positions with the IEEE Southern Alberta Section (SAS) including the Chairmanship during 2007 and 2008. He has over two decades of experience in the IT, software, and telecommunications industries, where he has held various executive positions. His research interests are in software engineering, cognitive informatics, and real-time systems as well as their industrial applications.

Yousheng Tian is a PhD candidate in cognitive computing and software engineering with the International Center for Cognitive Informatics and Cognitive Computing (ICCICC) as well as Theoretical and Empirical Software Engineering Center (TESERC) in Dept. of Electrical and Computer Engineering at the University of Calgary, Canada. He received a PhD from Xian Jiantong University, China, in Computer Science in 2002 and was a Post Doctoral Fellow at ICCICC during 2006 to 2007. His research interests are in cognitive informatics, cognitive computing, software engineering, machine learning, and denotational mathematics.