INTRODUCTION

A real-time system is characterized by event-/time-/interrupt-driven behaviours that are constrained by both its logic correctness and timing correctness. Although nonreal-time transaction processing systems may only consider the logical correctness, real-time systems have to put emphases on dynamic timing constraints with system control logics. A Lift Dispatching System (LDS) is a typical real-time control system characterized by its high degree of complexity,
intricate interactions with hardware devices and users, and necessary requirements for domain knowledge (Hayes, 1985; McDermid, 1991; Chenais & Weinberger, 1992; Liu, 2000; Wang, 2002, 2007; Ngolah et al., 2004). All these factors warrant an LDS system as a complex but ideal design paradigm in large-scale software system design in general and in real-time system modeling in particular.

The lift scheduling problem has been studied as a real-time system in Chenais and Weinberger (1992) and Hamdi et al. (1995) to be NP-complete, because for \( n \) lifts, if there are \( p \) requests, then there would be up to \( np \) possible dispatching strategies. Further, the problem is dynamic, i.e., during executing a given dispatching plan, new requests presented inside the cabins of lifts and from the floors may often interrupt and change the current dispatching strategy. The request scheduler therefore must be able to find a suitable dispatching mechanism in order to ensure there is no request to wait for a long period before being served.

There is no systematical and detailed repository and formal documentation of design knowledge and modeling prototypes of an LDS system nor a formal model of it in denotational mathematics and formal notation systems (Wang, 2008d). This article presents the formal design, specification, and modeling of the LDS system using a denotational mathematics known as Real-Time Process Algebra (RTPA) (Wang, 2002, 2003, 2007, 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, 2007, 2008a, 2008c). According to the RTPA methodology for system modeling and refinement, a software system can be specified as a set of architectural and operational components as well as their interactions. The former is modeled by Unified Data Models (UDMs, also known as the component logical model (CLM)) (Wang, 2007), 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 is modeled by static and dynamic processes using the Unified Process Models (UPMs) (Hoare, 1978, 1985; Bjorner & Jones, 1982; Corsetti & Ratto, 1991; Wang, 2007, 2008a; Wang & King, 2000; Wang & Ngolah, 2002).

This article develops a formal design model of the LDS system in a top-down approach on the basis of the RTPA methodology. In the remainder of this article, the conceptual model of the LDS system is described as the initial requirements of the system. The architectural model of the LDS system is created based on the conceptual model using the RTPA architectural modeling methodologies and refined by a set of UDMs. Then, the static behaviors of the LDS system are specified and refined by a set of processes (UPMs). The dynamic behaviors of the LDS system are specified and refined by process priority allocation, process deployment, and process dispatching models. With the formal and rigorous models of the LDS system, code can be automatically generated by the RTPA Code Generator (RTPA-CG) (Wang, 2007), or be seamlessly transferred into program code manually. The formal models of LDS 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.

**THE CONCEPTUAL MODEL OF THE LDS SYSTEM**

The Lift Dispatching System (LDS) is a real-time computer controlled system for multiple lifts in a building with multiple floors. In the conceptual model of the LDS system, as given in Figure 1, there are three lifts serving six floors. The LDS system encompasses three lifts, a controller implemented by a processor, a set of control interfaces, and a set of 30 buttons. On each floor of the building, there are three buttons corresponding to each lift in both directions, except that there are only upward buttons on floor 1 and downward buttons on floor 6. On each floor there
is also an indicator of the current level of each lifts. In addition to the external equipments, the cabin of each lift consists of a bell and a set of internal buttons to control the door’s open/close and to enter the expected numbers of internal requests.

Once an external button is pressed, its built-in lamp is lit up to indicate that the request is received by the system. If a button is pressed for multiple times after a request for that button has already been pended, there is no further effect. The system also automatically set the rest buttons on the same floor with the same direction as pressed. Lights of the same group of interlinked buttons will go off when a lift arrives to serve the request. A parked lift as dispatched for serving a request at a certain floor will automatically open and then close the door of its cabin within a predesigned period except it is forced to be opened or closed by internal buttons with a higher priority.

The design constraints of the lift system and its dispatching algorithm can be informally described as follows:

1. The lift that responds to a request does so within minimum time and minimum energy consumption.
2. Only one lift should respond to a given request on a certain floor.
3. If there is no request pending, a lift should be parked on the current floor where it reached in its last dispatched destination.
4. The door of a lift remains closed except the conditions specified in item (5) is met.
5. The door of a lift keeps open when reached the floors where a current request has been identified or the lift is in the initial state on floor 1.
6. A lift does not stop to serve a request if it is moving in the opposite direction of the pending request in a given dispatching cycle.

All design constraints and requirements for the LDS system as stated above will be rigorously specified in the formal LDS design models in the following sections, particularly the UPM of LiftDispatching\text{PC} and LiftServing\text{PC}.

**Figure 1. Conceptual model of the LDS system**

{Figure showing the conceptual model of the LDS system with buttons, control interfaces, the controller, and lifts.}
The top level framework of the LDS system can be modeled by a set of architecture, static behaviors, and dynamic behaviors using RTPA (Wang, 2002, 2008a) as follows:

\[ \text{LDS} \triangleq \text{LDS}.\text{Architecture} \ || \ \text{LDS}.\text{StaticBehaviors} \ || \ \text{LDS}.\text{DynamicBehaviors} \]

where || indicates that these three subsystems related in parallel, and $\text{}$, $\text{ST}$, and $\text{PC}$ 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 LDS into detailed architectural models (UDMs) and behavioral models (UPMs).

**THE ARCHITECTURAL MODEL OF THE LDS SYSTEM**

The architecture of a hybrid hardware/software system and/or a real-time system is a system framework that represents the overall structure, components, processes, and their interrelationships and interactions. The following subsections specify the architecture of LDS, LDS$\text{.Architecture} \text{ST}$, by a high-level architectural model based on its conceptual model as provided in Figure 1. Each of its architectural components will be refined as a UDM (also known as Component Logical Model (CLM)) (Wang, 2002a, 2007).

**The Architectural Framework of LDS**

System architectures, at the top level, specify a list of identifiers 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 architectural refinement for the system.

Corresponding to the conceptual model of LDS as shown in Figure 1, the high-level specification of the architecture of LDS, LDS$\text{.Architecture} \text{ST}$, is given in Figure 2 in RTPA. LDS$\text{.Architecture} \text{ST}$ encompasses parallel structures of Lifts$\text{ST}$, Buttons$\text{ST}$, SysClock$\text{ST}$, and Controller$\text{ST}$, as well as a set of events @Events$\text{S}$ and a set of statuses &Status$\text{BL}$. The controller of LDS, Controller$\text{ST}$, is a subsystem of internal control structures that may be further refined by a set of UDMs such as the RequestEvenRecord$\text{ST}$, LiftStatusRecord$\text{ST}$, LiftDispatchList$\text{ST}$, and ServiceQueues$\text{ST}$, where the numbers in angel brackets indicate the configuration of how many data objects that share the same UDM.

The events of LDS are predefined global control variables of the system, as given in Eq. 2, which represent an external stimulus to a system or the occurring of an internal change of status such as an action of users, an updating of the environment, and a change of the value of a control variable. Types of general events, @Events$\text{S}$, that may trigger a behavior in a system can be classified into operational (@e$\text{S}$), time (@t$\text{TM}$), and interrupt (@int$\text{Ω}$) events, where @ is the event prefix, and $\text{S}$, $\text{TM}$, and $\text{Ω}$ the type suffixes of string, time, and interrupt, respectively, i.e.:
A status denoted by \( φ_{\text{BL}} \) is an abstract model of system state in Boolean type such as an operation result and an internal condition. The LDS status as a predefined global control variable is as follows:

\[
@\text{Events}_S \triangleq @\text{SystemInitial}_S \\
| @t_{\text{TM}} = \text{nth:mm:ss} \\
| @\text{SysClock100msInt}
\]

\[\text{(2)}\]

A UDM is a generic structural type defined in RTPA (Wang, 2002a, 2007). Mathematically, the UDM is an \( n \)-tuple to model a system architectural component such as a hardware interface, an internal logical model, and/or a common control structure of a system. UDMs are a powerful modeling means in system architectural modeling, which can be used for unifying user defined complex data objects in system modeling, which represent the abstraction and formal representation of domain knowledge and structural information.
As modeled in Figure 2, the LDS system encompasses seven UDMs for modeling the system hardware interfaces and internal control structures as follows:

\[
\text{LDS.UDMs} \triangleq \text{HardwareInterfaceCLMs} \sqcup \text{InternalControlStructures} = (\langle \text{Lifts: ST} | [3]\rangle \\
| \langle \text{Buttons: ST} | [30]\rangle \\
| \langle \text{SysClock: ST} | [1]\rangle \\
| \langle \text{RequestEvenRecord: ST} | [30]\rangle \\
| \langle \text{LiftStatusRecord: ST} | [3]\rangle \\
| \langle \text{LiftDispatchList: ST} | [3]\rangle \\
| \langle \text{ServiceQueues: ST} | [4]\rangle \\
\)
\]

(4)

where the Lifts\textit{ST}, Buttons\textit{ST}, and SysClock\textit{ST} are hardware interface UDMs, while the RequestEvenRecord\textit{ST}, LiftStatusRecord\textit{ST}, LiftDispatchList\textit{ST}, and ServiceQueues\textit{ST} are internal control UDMs.

The configuration of the UDMs in LDS is indicated by the numbers in the angle brackets in Eq. 4, where each number shows how many similar devices or internal control structures are equipped that share the same UDM schema. For example, there are 3 lifts, 30 buttons, 4 service queues, and 3 lift dispatching list in the LDS system. Each of the seven type system UDMs in LDS is designed and modeled in the following subsections in the two categories of system hardware and internal control structures.

**The UDM Structures of the LDS Hardware System**

The hardware system of LDS and their interfaces are modeled by a set of UDMs such as Lifts\textit{ST}, Buttons\textit{ST}, and SysClock\textit{ST} UDMs. Each of the three system UDMs in LDS is designed and modeled in the following subsections.

**a. The Lifts**

The UDM model of lifts, Lifts\textit{ST}, and its three derived objects Lift(iN)\textit{ST}, 1 ≤ iN ≤ 3, are modeled as shown in Figure 3. Lifts\textit{ST} encompasses 18 fields known as the engine drive ports and control signals (UpDrivePort\textit{H}, UpDriveOutput\textit{B}, DownDrivePort\textit{H}, DownDriveOutput\textit{B}; StopDrivePort\textit{H}, StopDriveOutput\textit{B}), door drive ports and control signals (DoorOpenPort\textit{H}, DoorOpenOutput\textit{B}; DoorClosePort\textit{H}, DoorCloseOutput\textit{B}; DoorBellPort\textit{H}, DoorBellOutput\textit{B}), and I/O devices (IndicatorPort\textit{H}, IndicatorOutput\textit{B}; CurrentLevelPort\textit{H}, CurrentLevelInput\textit{B}; DestScanPort\textit{H}, DestScanInput\textit{B}). The port PORT\textit{ST} = PORT\textit{ST}(PortAddress\textit{H})\textit{B}, as well as memory MEM\textit{ST}, is a generic UDM model in RTPA that describes the architectural structure of system I/O ports identified by a linear space of byte-type data identified by a hexadecimal port address. Typical operations on ports are input and output, i.e. (Wang, 2007, 2008a):

\[
\text{PORT} \textit{ST}(\text{PortAddress}\textit{H})\textit{B} \triangleright \text{PortInput}\textit{B} \\
\text{PortOutput}\textit{B} \triangleright \text{PORT} \textit{ST}(\text{PortAddress}\textit{H})\textit{B}
\]

(5)
Each field in \( \text{LiftsST} \) is modeled by an RTPA type where its constraints, if any, are provided following the vertical bar. For instance, the constraint for the field of current level input in byte type is \( 1 \leq \text{CurrentLevelInputB} \leq 6 \).

The three derived concrete lift models,
share the same structure as specified by the abstract schema LiftST. The concrete objects obtain their refined physical or logical parameters such as port addresses and initial values of the I/O signals.

b. The Buttons

The buttons of LDS, ButtonsST, are external keys installed on each floor for receiving service requests. As shown in the conceptual model of LDS in Figure 1, there are three buttons for each direction on each floor. Therefore, there are totally 30 button objects, Key(iNST), 1 ≤ i ≤ 30, that need to be modeled in LDS, which share a common UDM, ButtonsST, as shown in Figure 4.

The schema of ButtonsST models four fields with certain design constraints such as the PortAddress (the physical interface of the key), KeyInput (the key status information obtained from the key port...
where only the three most least significant bits are effective), DirectionBL (the direction of request the key represents: T denotes upward and F downward), KeyPositionN (the floor number of a key represents).

The 30 derived concrete key models,

\[ \text{Key}(iN)ST, \]

share the same structure as specified by the abstract schema ButtonsST. The concrete objects obtain their refined physical or logical parameters such as port addresses and initial values of floors, directions, and the pattern of input signals of key status.

c) 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 LDS, SysClockST, is designed as given in Figure 5. SysClockST provides an absolute (calendar) clock CurrentTimehh:mm:ss as the logical time reference of the entire system and a relative clock §tN as a generic counter. The InterruptCounterN is adopted to transfer the basic timing ticks at 100ms interval into the second signal. The real-time system clock is updated by the process SysClockPC, which will be described in the following section on system static behaviors.

The UDM Structures of the LDS Controller

The internal control system of LDS is modeled by a set of UDMs such as the RequestEventRecordST, LiftStatusRecordST, LiftDispatchListST, and ServiceQueuesST. Each of the four internal control UDMs in LDS is designed and modeled in the following subsections.

a. The Request Event Record

A request event record, RequestEventRecordST, as shown in Figure 6 is an internal control structure that monitors and registers the current and last statuses of each of the 30 keys based on periodical scans conducted by the process RequestScanningPG as shown in Figure 12. Once a key press is recognized, the flag RequestEventRecordST.RequestIdentifiedBL will be set to true in order to avoid redundant processing of possible repeated requests on the same key.

Each request event record, RequestRecord(iN)ST, 1 ≤ iN ≤ 30 in Figure 6, is designed corresponding to a specific button in LDS, Key(iN)ST, which share the same structure of the common abstract schema as specified in RequestEventRecordST.

Figure 5. The schema and detailed UDM model of system clock
Figure 6. The schema and detailed UDM model of request event record

<table>
<thead>
<tr>
<th>Request Event Record ST ⊳ RequestRecord(3)ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>= RequestRecord(1)ST : (Current Status BL, Last Status BL, Request Identified BL) = (F, F, F)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Figure 7. The schema and detailed UDM model of lift status record

<table>
<thead>
<tr>
<th>Lift Status Record ST ⊳ LiftRecord(3)ST</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>= LiftRecord(1)ST : (Lift Status BL, Current Direction BL, Current Level N, Current Destination N,</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Copyright © 2009, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
b. The Lift Status Record

The lift status record, LiftStatusRecord_{ST}, is an internal structure that maintains a dynamic status of each elevator as shown in Figure 7. LiftStatusRecord_{ST} records the real-time information of an elevator such as the operating status, current direction, current level, current destination, and last destination.

As modeled in Figure 7, the LDS system initializes all lifts in the idle state at floor 1 toward the upward direction. The initial current and last destinations are arbitrarily set on floor 1. By checking the information updated in LiftStatusRecord_{ST}, the lift dispatching and serving processes can obtain real-time status of a lift under control.

c. The Lift Dispatch List

The lift dispatching list, LiftDispatchList_{ST}, is an internal structure that models the current identified and dispatched requests for each of the elevators in the LDS system. In the lift dispatching list, all requests on the same floor with the same direction will be treated as an identical service request event. Therefore, there are maximum six dispatching stops in a single way operation where certain floors will be set as Level(i)_{BL} = T by the system dispatching process.

The dynamic status of the dispatching lists is maintained in real-time. When the request on a dispatched floor i_{N} is served, Level(i)_{BL} will be reset to F. An indicator, #WaitingServices_{N}, is adopted to show the current total number of requests in all floors of a single way operation. #WaitingServices_{N} will be synchronously updated during operation when Level(i)_{BL} is set or reset.

d. The Service Queues

Four service queues are adopted in LDS for modeling the new and waiting requests identified in the LDS system in two directions. The architecture of the service queues is a first-come-first-serve (FCFS) structure as shown in Figure 9. The instances of the ServiceQueues_{ST} are the UpRequestQueue_{ST}, DownRequestQueue_{ST}, UpWaitingQueue_{ST}, and DownWaitingQueue_{ST},

**Figure 8. The schema and detailed UDM model of lift dispatch list**
where the pair of request queues contain newly identified requests and the pair of waiting queues contain requests that cannot be served in a previous dispatching cycle.

The major operational processes of the FCFS ServiceQueues\
\(ST\) are \textit{enqueue} and \textit{serve}, which will be described in Eq. 7. The former is a process that appends an element at the end of the queue and tests if the queue is full. The latter is a process that fetches the front element of the queue, tests if the queue is empty, and shifts the remainder elements toward the front of the queue by one place. There are also internal manipulations for the queues, such as creation, memory allocation, and release.

The system architectural models specified in this section provide 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 \textit{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 or UDMs, specification of the operational components of the LDS system as behavioral processes can be carried out directly as elaborated in the following sections.

**THE STATIC BEHAVIOR MODELS OF THE LDS 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 specifications and with the UDMs of system architectural components developed in preceding section, the operational components of the given LDS system and their behaviors can be specified as a set of UPMs as behavioral processes operating on the UDMs.

The static behaviors of the LDS, LDS\(\#\).\textit{StaticBehaviors}\(\underline{PC}\), can be described through operations on its architectural models. LDS\(\#\).\textit{StaticBehaviors}\(\underline{PC}\) encompasses the processes of SysInitial\(\underline{PC}\), SysClock\(\underline{PC}\), RequestScanning\(\underline{PC}\), RequestProcessing\(\underline{PC}\), LiftDispatching\(\underline{PC}\), and LiftServing\(\underline{PC}\) in parallel as specified below:
The following subsections describe how the LDS static behaviors as specified in Eq. 6 are modeled and refined using the denotational mathematical notations and methodologies of RTPA in term of a set of UPMs.

**The System Initialization Process**

System initialization is a common 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. Initialization is crucially important for a real-time system as well as its control logic specified in the functional processes. The system initialization process of LDS, SysInitialPC, is modeled in Figure 10, where all system architectural and control UDMs are initialized as specified in their UDMs. Then, the system clock and timing interrupt are set to their initial logical or calendar values.

The initialization process sets the system control models such as the buttons, the lifts, and the system clock into given initial statuses. It also initializes the control models such as the request event records, the lift status records, and the lift dispatch lists. In the UDMs initializations, all initial values as specified in the UDM objects as given in the preceding section will be implemented in order to set the system into a correct initial state.

**The System Clock Process**

The system clock process is a 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 LDS, SysClockPC, is modeled in Figure 11. The source of the system clock is obtained from the 100ms interrupt clock signal generated by system hardware, by which the absolute clock with real-time second, minute, and hour, SysClockST.CurrentTime hh:mm:ss, 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. The relative clock is reset to zero at midnight each day in order to prevent it from overflow.

![Figure 10. The behavior model of the system initialization process](image-url)
The Request Scanning Process

The request scanning process of LDS, RequestScanning_{PC}, as shown in Figure 12 periodically scans the thirty buttons per 100ms in order to detect if any key is pressed on every floor. The sample period is set as 100ms based on the factor that a human key-press action is between 200ms to 500ms. If a key-press is detected, the status record flag corresponding to the button is set to true in the RequestRecord_{ST}, so that this key-press will be identified if it is a new or a repeated request by the following request processing process.

According to the conceptual model of LDS in Figure 1, the request scanning process handles the buttons in 10 groups. Each group consists of 3 buttons on the same floor and with the same direction. When any key is pressed in a group, all keys in the group will be treated as pressed.

The request scanning process also periodically monitors the current level of a moving lift from the its level sensor input. Based on the input per 100ms, the current level fields in the UDMs
of the lift and lift record, LiftRecord(iN)ST.CurrentLevelB and Lift(iN)ST.CurrentLevelInputB, are updated at real time.

The Request Processing Process

The request processing process of LDS, RequestProcessingPC, as shown in Figure 13 is another periodical process that identifies new requests and masks repeated requests on the same floor toward the same direction. This mechanism is adopted to ensure that the LDS system may only recognize multiple requests once at a certain floor for a certain direction during a given operation cycle.

The main algorithm of the request processing process is to detect if a reported key-press is a new request according to the algorithm as follows:

\[
\text{NewRequestBL} \triangleq \text{RequestRecord(iN)ST.CurrentStatusBL} = \text{T} \land \text{RequestRecord(iN)ST.LastStatusBL} = \text{F}
\]  

(7)

That is, a new request is identified by two periodical scans per 100ms where the last scan detected false (the key was unpressed) and the current scan detects true (the key is pressed). Once a new request is identified, the request processing process marks all other two buttons related to it on the same floor toward the same direction as pressed. Then, the recognized service request is registered into one of the corresponding ServiceQueuesST by the process Queue(iN)ST.EnqueuePC as specified in Eq. 8.
Predefined operations on ServiceQueues<sub>ST</sub>, encompassing the UpRequestQueue<sub>ST</sub>, nocomalling DownRequestQueue<sub>ST</sub>, UpWaitingQueue<sub>ST</sub>, and DownWaitingQueue<sub>ST</sub>, can be specified by a set of process models as follows:

\[
\text{ServiceQueues}_{\text{ST}}.\text{StaticBehaviors}_{\text{PC}} \triangleq \\
\begin{align*}
&\text{Create}_{\text{PC}}(<1:: (\text{QueueID$s$})>; <O:: ((\text{QueueIDExistedBL})> ; \\
&\text{UDM:: (QueueID$ST$)}) >) \\
&\mid \text{Enqueue}_{\text{PC}}(<I:: (\text{QueueID$s$, RequestFloorNum}$N$}); <O:: (\text{QueueIDFullBL}, \text{QueueIDEnqueueSucceedBL})> ; \\
&\text{UDM:: (QueueID$ST$)}) >) \\
&\mid \text{Serve}_{\text{PC}}(<I:: (\text{QueueID}$S$); <O:: (\text{RequestFloorNum}$N$, \text{QueueIDEmptyBL}, \text{QueueIDServeSucceedBL})> ; \\
&\text{UDM:: (QueueID$ST$)}) >) \\
&\mid \text{Clear}_{\text{PC}}(<I:: (\text{QueueID}$S$); <O:: (\text{QueueIDEmptyBL})> ; \\
&\text{UDM:: (QueueID$ST$)}) >) \\
&\mid \text{EmptyTest}_{\text{PC}}(<I:: (\text{QueueID}$S$); <O:: (\text{QueueIDEmptyBL})> ; \\
&\text{UDM:: (QueueID$ST$)}) >) \\
&\mid \text{FullTest}_{\text{PC}}(<I:: (\text{QueueID}$S$); <O:: (&\text{QueueIDFullBL})> ; \\
&\text{UDM:: (QueueID$ST$)}) >) \\
&\mid \text{Release}_{\text{PC}}(<I:: (\text{QueueID}$S$); <O:: (&\text{QueueIDExistedBL})> ; \\
&\text{UDM:: (QueueID$ST$)}) >) 
\end{align*}
\]

where the Enqueue<sub>PC</sub> and Serve<sub>PC</sub> processes are invoked in the RequestProcessing<sub>PC</sub> and LiftDispatching<sub>PC</sub> processes, respectively. Details of these queue behavioral models may be referred to (Wang, 2007).

### The Lift Dispatching Processes

The lift dispatching process is modeled by two similar subprocesses: a) New request dispatching, LiftDispatching NewRequest<sub>PC</sub>, as shown in Figure 14; and b) Waiting request dispatching, LiftDispatching WaitingRequest<sub>PC</sub>, as shown in Figure 15. Requests in waiting are given a higher priority in the dispatching algorithm in order to prevent a request from waiting for too many dispatching cycles in rush hours.

A key concept of the LDS design is the dispatching cycle that is a single directional serve in which the dispatched lift moves from an idle state or a U-turn state to the top/bottom floor or to a floor where it has completed all dispatched serves as listed in the current dispatching list. The operations of each lift in the LDS system are therefore divided into a series of dispatching cycles.

The lift dispatch process finds out a new or existing request when any of the UpRequestQueue<sub>ST</sub>, DownRequestQueue<sub>ST</sub>, UpWaitingQueue<sub>ST</sub>, or DownWaitingQueue<sub>ST</sub> is not empty. Then, it searches through the lift status record, LiftStatusRecord<sub>ST</sub>, in order to find one or multiple candidate lift(s) that either move in the same direction as the given request or are idle. If the result is positive, the distances from the requested floor to all candidate lifts are calculated, which results in the determination of an optimal dispatching with the shortest distance; otherwise the given request is sent to either UpWaitingQueue<sub>ST</sub> or DownWaitingQueue<sub>ST</sub>. It is
Figure 14. The behavior model of the lift dispatching process for new requests

```
LiftDispatching_NewRequest (L): (O): (U): (UDM): (RequestRecordsST, UpRequestQueueST, DownRequestQueueST, UpWaitingQueueST, DownWaitingQueueST, KeyST, LiftRecordsST, DispatchListST) -> PG  
expL <= UpRequestQueueST.EmptyL  // Process upward requests
R^1_{expL; f}
  (\rightarrow UpRequestQueueST.ServeFG(\@, RequestFloorNumH, \@UpRequestQueueEmptyBL)
     \@UpRequestQueueServeSucceedBL, UpRequestQueueST)  // Fetch first request and 
     \rightarrow MinDistanceH = 6  // update queue empty status
     \rightarrow LiftFoundBL = F
   )
   \rightarrow \{  
   LifRecord(k,HST, LiftStatusBL) = F \lor  // Idle
     LifRecord(k,HST, CurrentDirectionBL) = T  // Move up 
     LifRecord(k,HST, CurrentLevelH \leq RequestFloorNumH) // Lower/equal to request floor
     \rightarrow Distance(k,HST, RequestFloorNumH) = LiftRecord(k,HST, CurrentLevelH)
     \rightarrow Distance(k,HST, MinDistanceH) = MinDistanceH
     \rightarrow MinDistanceH = Distance(k,HST)
     \rightarrow AvailableLiftH = kH
     \rightarrow LiftFoundBL = T
   \}
   \rightarrow \{ \@LiftFoundBL = T
     \rightarrow DispatchListAvailableLiftHST, Level(RequestFloorNumHBL) = T
     \rightarrow LiftDispatchListAvailableLiftHST, WaitingServiceHST
     \rightarrow LiftFoundBL = F  // No lift available
     \rightarrow DownWaitingQueueST.ServeFG(\@, RequestFloorNumH, \@DownWaitingQueueEmptyBL
     \@DownWaitingQueueServeSucceedBL, DownWaitingQueueST)
     \rightarrow DownWaitingQueueST.EmptyBL = F
   \)
   \rightarrow \{ \@LiftFoundBL = T
     \rightarrow DispatchListAvailableLiftHST, Level(RequestFloorNumHBL) = T
     \rightarrow LiftDispatchListAvailableLiftHST, WaitingServiceHST
     \rightarrow LiftFoundBL = F  // No lift available
     \rightarrow DownWaitingQueueST.ServeFG(\@, RequestFloorNumH, \@DownWaitingQueueEmptyBL
     \@DownWaitingQueueServeSucceedBL, DownWaitingQueueST)
     \rightarrow DownWaitingQueueST.EmptyBL = F
   \}
   \rightarrow expBL = T
\)
```

Figure 15. The behavior model of the lift dispatching process for waiting requests
noteworthy that any lift moving in the opposite direction is ruled out as a possible candidate in a given dispatching cycle.

a.  The Lift Dispatching Process for New Requests

The behavior model of the lift dispatching process for new requests, LiftDispatching_NewRequestPC, is shown in Figure 14. The lift dispatching process allocates a suitable lift to a certain request or multiple requests according to the dispatching decision-making criteria as given in the design Constraints 1 – 6 in the conceptual model of the LDS system. The dispatching algorithm can be formally specified as follows:

\[
\text{OptimalDispatchStrategy_{BL}} \triangleq \min (\text{Distance}(k^N) | 0 \leq \text{Distance}(k^N) \leq 6) \land \\
(\text{LiftRecord}(k^N).\text{LiftStatus}_{BL} = F \lor \\
\text{Key}(i^N).\text{Direction}_{BL} = \text{LiftRecord}(k^N).\text{CurrentDirection}_{BL})
\]

(9)

where \( k \) is the number of lifts, \( i \) the number of the key under serving, and \( \min \) a predefined function to find the minimum of distance.

The above algorithm guarantees both a fast and energy-efficient serve without turning any lift in its current operating direction before it complete designated serves in a given dispatching cycle. Once the \( i \)th optimal lift is found for a new request according to the dispatching algorithm, it will be registered into the UDM DispatchingList\((i^N)_{ST}\) for an immediate serve; Otherwise, it will be put into the UpWaitingQueue\(_{ST} \) or DownWaitingQueue\(_{ST} \) for a priority serve in the next dispatching cycle of the LDS system. In the latter case, the dispatching distance becomes negative, such as a request at a floor lower than all current upward serving lifts or higher than all downward serving lifts. These waiting requests will be served in the next dispatching cycle from either Floors 1 to 6 or Floors 6 to 1.

b.  The Lift Dispatching Process for Waiting Requests

The behavior model of the lift dispatching process for waiting requests, LiftDispatching_WaitingRequestPC, is shown in Figure 15. A waiting request is one that could not be served in a previous operating cycle due to a busy state of the system or any dispatching constraint. The dispatching strategy for serving waiting request is similar to those of new requests, except that it checks UpWaitingQueue\(_{ST} \) and DownWaitingQueue\(_{ST} \) in order to put any existing request into the dispatching process.

The Lift Serving Processes

The lift serving process encompasses two similar subprocesses: a) Upward request serve, LiftServing_UpwardPC, as shown in Figure 16; and b) Downward request serve, LiftServing_DownwardPC, as shown in Figure 17. These two processes directly control all equipments of each lift such as the engine, door, indicator, bell, level sensor, and internal request buttons as illustrated in Figure 1.
Figure 16. The behavior model of the lift serving process for upward requests

\[
\begin{align*}
&\text{LiftServingUpward}(\text{In}; \text{Out}) := \text{LiftRecord}(\text{ST}; \text{CurrentLevel}) \Rightarrow \text{Display current level} \\
&\quad \Rightarrow \text{LiftRecord}(\text{ST}; \text{IndicatorOutput}) \iff \text{PORT}(\text{Lift}(\text{ST}; \text{IndicatorPort})) \\
&\quad \Rightarrow \text{DispatchList}(\text{ST}; \#\text{WaitingService}) > 0 \land \text{LiftRecord}(\text{ST}; \text{CurrentDirection}) = \text{T} \\
&\quad \Rightarrow \text{LiftRecord}(\text{ST}; \text{CurrentLevel}) \Rightarrow \text{Serve dispatched at the floor} \\
&\quad \Rightarrow \text{DispatchList}(\text{ST}; \#\text{WaitingService}) \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{UpDrivePort})) \iff \text{Release up-drive signal} \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{StopDrivePort})) \iff \text{Stop at a dispatched floor} \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{DoorClosePort})) \iff \text{Release close-door signal} \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{DoorOpenPort})) \iff \text{Open doors} \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{DoorBellPort})) \iff \text{Sound bell} \\
&\quad \Rightarrow \text{TimeOut} = \text{F} \\
&\quad \Rightarrow \text{hhmm:ss} = \text{hhmm:ss} + 8 \\
&\quad \Rightarrow \text{TimeOut} = \text{T} \iff \text{Delay 8s based on system clock} \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{DoorBellPort})) \iff \text{Stop bell} \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{DoorOpenPort})) \iff \text{Release open-door signal} \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{DoorClosePort})) \iff \text{Close doors} \\
&\quad \Rightarrow \text{Key}(\text{ST}; \text{KeyPosition}) = \text{LiftRecord}(\text{ST}; \text{CurrentLevel}) \land \\
&\quad \Rightarrow \text{Key}(\text{ST}; \text{Direction}) = \text{LiftRecord}(\text{ST}; \text{CurrentDirection}) \\
&\quad \Rightarrow \text{RequestRecord}(\text{ST}; \text{RequestedId}) \iff \text{Clear requests on the same level} \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{DestScanPort})) \iff \text{Scan new destination level(s)} \\
&\quad \Rightarrow \text{DispatchList}(\text{ST}; \text{CurrentLevel}) \land \text{DispatchList}(\text{ST}; \text{CurrentLevel}) \\
&\quad \Rightarrow \text{DispatchList}(\text{ST}; \text{CurrentLevel}) \Rightarrow \text{Serve dispatched at the floor} \\
&\quad \Rightarrow \text{DispatchList}(\text{ST}; \text{CurrentLevel}) \Rightarrow \text{Serve dispatched at the floor} \\
&\quad \Rightarrow \text{DispatchList}(\text{ST}; \text{CurrentLevel}) \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{StopDrivePort})) \iff \text{Clear stop drive} \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{UpDrivePort})) \iff \text{Move up} \\
&\quad \Rightarrow \text{No service at the floor} \\
&\quad \Rightarrow \text{LiftRecord}(\text{ST}; \text{CurrentLevel}) = 6 \iff \text{Reached a middle floor} \\
&\quad \Rightarrow \text{PORT}(\text{Lift}(\text{ST}; \text{UpDrivePort})) \iff \text{Up drive} \\
&\quad \Rightarrow \text{LiftRecord}(\text{ST}; \text{CurrentDirection}) = \text{F} \\
&\quad \Rightarrow \text{LiftRecord}(\text{ST}; \text{CurrentDirection}) = \text{F} \\
&\quad \Rightarrow \text{Change direction} \\
\end{align*}
\]
a. The Lift Serving Process for Upward Requests

The behavior model of the lift serving process for upward requests, LiftServing_UpwardPC, is shown in Figure 16. In each dispatching cycle, a lift is driven by the lift serve process to serve all requests in its way according to the pre-designated dispatching list, DispatchListST, generated by the dispatching processes of the LDS system. The lift will move toward and stop on each
requested floor as dispatched, light up the floor indicator, ring the bell, open the doors, clear all requests on this floor, and close the doors after a given period (8 second in this design). Then, during the lift moves to the next dispatched floor, it scans the destination information of new customers entered the cabin. The newly identified internal request(s) are used to update the current dispatching list of this lift by updating its DispatchListST. If an internal request is opposite to the current serve direction, it will be registered into the DownWaitingQueueST for serving in the next dispatching cycle.

When a lift completed the serve of all dispatched requests in a dispatching cycle, it parks on the latest reached floor, remains doors open, and waits for a new dispatching list of the next dispatching cycle from the LDS system. If the parked floor is the top or bottom floor, the direction of the lift will be automatically turned.

b. The Lift Serving Process for Downward Requests

The behavior model of the lift serving process for downward requests, LiftServing_DownwardPC, is shown in Figure 17. The downward serving process is similar to that of upward serving, except that the current direction and turning mechanism are inversed, as well as that DownWaitingQueueST is replaced by UpWaitingQueueST.

THE DYNAMIC BEHAVIOR MODEL OF THE LDS SYSTEM

Dynamic behaviors of systems are run-time process deployment and dispatching mechanisms based on the static behaviors modeled in UPMs. Because the 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 priority allocation and deployment are yet to be specified.

With the UPMs developed in the preceding section as a set of static behavioral processes of the LDS system, this section describes the dynamic behaviors of the LDS system at run-time via process priority allocation and process deployments.

LDS 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 systems 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 periodical 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, any system encompasses at least a base level process, particularly for a nonreal-time or transaction processing system.

According to the RTPA system modeling and refinement methodology (Wang, 2007), the first step refinement of the dynamic behaviors of the LDS system on process priority allocation can be specified as shown in Figure 18. It may be observed that all non-periodical processes at run-time, such as SystemInitialPC, LiftDispatching_WaitingRequestPC, LiftDispatching_NewRequestPC, LiftServicing_UpwardPC, and LiftServicing_DownwardPC, are allocated at the base
level, therefore there is no high level processes in the LDS system. However, the processes with strict timing constraints, such as $\text{SysClock}_{PC}$, $\text{RequestScanning}_{PC}$, and $\text{RequestProcessing}_{PC}$, are allocated as periodical interrupt processes.

**LDS Dynamic Process Deployment and Dispatching**

*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 18, the LDS dynamic behaviors can be
further refined with a process deployment model as shown in Figure 19, where precise timing relationships between different priority levels are specified.

The dynamic behaviors of the lift dispatching system can be described by the interactions of parallel categories of processes at the base and interrupt levels, which are triggered by the event @SystemInitialS or @SysClock100msInst. The LDS system repeatedly executes the LiftDispatchingPC and LiftServingPC processes at base level until the event SysShutDownBL = T is captured by the system (§). When the interrupt-level processes occurs per 100ms during run-time of base level processes, the system switches priority scheduling to the interrupt level processes, such as SysClockPC, RequestScanningPC, and RequestProcessingPC. Once completion, the interrupt-level processes hand over control to the system in order to resume the interrupted base-level functions.

The formal design models of the LDS system and their refinements demonstrate a typical system modeling paradigm of the entire architectures, static behaviors, and dynamic behaviors according to the RTPA specification and refinement methodology. The practical formal engineering method of RTPA for system modeling and specification provides a coherent notation system and systematical methodology for large-scale software and hybrid system design and implementation. The final-level refinements of the LDS 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 Lift Dispatching System (LDS), 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 LDS system in a top-down approach. The architectural model of the LDS system has been created using a set of UDMs. The static behaviors of the LDS system have been modeled by a set of UPMs. The dynamic behaviors of the LDS system have been specified by a set of process priority allocation and process deployment models.

The formal model of the LDS system has provided a universal and flexible design of LDS systems. The LDS system allows the design and implementation of the system be easily extended to a complex building with more floors and more elevators by changing the UDM models of LDS. The UPMs of all the behavioral processes of LDS will mainly remain unchanged. Additional functions may be added into the LDS system, such as those of special priority users for maintenance, door sensors for safe closing, and overweight detection/processing.

Based on the rigorous design models and the formal framework of the LDS system, program code can be seamlessly derived. The formal model of LDS 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, 2009; Wang and Ngolah, 2002; Wang and Zhang, 2003; Ngolah et al., 2002, 2004). Since the equivalence between software and human behaviors, RTPA may also be use to describe human dynamic behaviors and mental processes (Wang, 2003, 2008d; Wang and Ruhe, 2007).
ACKNOWLEDGEMENT

The authors would like to acknowledge Natural Science and Engineering Council of Canada (NSERC) and an IIST/UNU fellowship for their partial support to this work. The authors 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 (ICfCI), 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 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.

Cyprian F. Noglah holds a PhD in software engineering from the University of Calgary, Canada in 2006. He obtained a BSc degree in mathematics and computer science from the University of
Essex, England in 1988 and an MSc degree in computer control systems from the University of Bradford, England in 1989. He has taught various computer sciences courses at both the undergraduate and graduate levels. He is currently a lecturer of software engineering at University of Buea, Republic of Cameroon. His main research interests are in real-time process algebra and its applications, tool support for formal specification languages, real-time software systems, formal methods in software engineering, and real-time operating systems.

Hadi Ahmadi is a PhD candidate in Department of Computer Science at University of Calgary. He is now working as a research assistant in iCORE information security (iCIS) lab. His main area of research is information and network security. He is also interested in other topics such as coding, complexity theory, and large-scale software development. He has received an MSc degree in electrical engineering, communications, from Sharif University of Technology.

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 PhD and MS degrees from the University of California at Berkeley in electrical engineering and computer science in 1986 and 1982, respectively, and his BS 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 expert 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.

Shi Ying is a professor in the State Key Laboratory of Software Engineering at Wuhan University, China. He received the bachelor’s degree in computer science in 1986 and the PhD degree in software engineering in 1999 from Wuhan University respectively. His research interests include service-oriented, aspect-oriented and semantic-based software development, software reuse and interoperability. His current research projects include semantic programming language, aspect-oriented software architecture and SOA domain, funded by the National High-Tech Research Development Program, National Natural Science Fund Project, and PhD Degree Foundation Project of the Ministry of Education of China. He has published nearly 90 journal and conference papers.