The Formal Design Model of Doubly-Linked-Circular Lists (DLC-Lists)

Yingxu Wang, University of Calgary, Canada
Cyprian F. Ngolah, Sentinel Trending & Diagnostics Ltd., Calgary, Canada
Xinming Tan, Wuhan University of Technology, China
Phillip C.-Y. Sheu, University of California, Irvine, USA

ABSTRACT

Abstract Data Types (ADTs) are a set of highly generic and rigorously modeled data structures in type theory. Lists as a finite sequence of elements are one of the most fundamental and widely used ADTs in system modeling, which provide a standard encapsulation and access interface for manipulating large-volume information and persistent data. This paper develops a comprehensive design pattern of formal lists using a doubly-linked-circular (DLC) list architecture. A rigorous denotational mathematics, Real-Time Process Algebra (RTPA), is adopted, which allows both architectural and behavioral models of lists to be rigorously designed and implemented in a top-down approach. The architectural models of DLC-Lists are created using RTPA architectural modeling methodologies known as the Unified Data Models (UDMs). The behavioral models of DLC-Lists are specified and refined by a set of Unified Process Models (UPMs) in three categories namely the management operations, traversal operations, and node I/O operations. This work has been applied in a number of real-time and nonreal-time system designs such as a real-time operating system (RTOS+), a file management system (FMS), and the ADT library for an RTPA-based automatic code generation tool.

Keywords: Abstract Data Types, Design Frameworks, Design Reuse, Doubly-Linked Lists, Formal Design Models, System Architecture Specifications, System Behavior Specifications, Unified Data Models, Unified Process Models

1. INTRODUCTION

Data object modeling is a process to creatively extract and abstractly represent a real-world problem by data models. A list is a finite sequence of elements where the order information of its elements is preserved beyond that of sets (Stubbs & Webre, 1985; McDermid, 1991; Bollella...
et al., 2002). Many important data objects can be modeled and implemented by lists such as a set of data with keys, a sentence for natural language parsing, a sequential file, a tree, and a graph (Wiener and Pinson, 2000).

A list can be formally modeled by an Abstract Data Type (ADT) (Guttag, 1977; Broy et al., 1984; Cardelli & Wegner, 1985; Stubbs & Webre, 1985), which is an abstract logical model of a complex and/or user defined data structure with a set of predefined operations. Using types to model real-world entities can be traced back to the mathematical thought of Bertrand Russell in the 1900s (Russell, 1903) and Georg Cantor in 1932 (Lipschutz & Lipson, 1997). A number of ADTs have been identified in computing and system modeling such as stack, queue, sequence, record, array, list, tree, file, and graph (Broy et al., 1984; Mitchell, 1990; McDermid, 1991; Wang, 2007; Wang, Ngolah, Tan, Tian, & Sheu, 2010). ADTs possess the following properties: (1) An extension of type constructions by integrating both data structures and functional behaviors; (2) A hybrid data object modeling technique that encapsulates both user defined data structures and allowable operations on them; (3) The interface and implementation of an ADT are separated where detailed implementation of the ADT is hidden to applications that invoke the ADT and its predefined operations.

In order to formally model the list ADT, an expressive denotational mathematics, Real-Time Process Algebra (RTPA) (Wang, 2002, 2007, 2008a, 2008b, 2008c, 2008d; Wang, Tan, & Ngolah, 2010), is adopted, which allows both architectural and behavioral models of lists to be rigorously designed and implemented in a top-down approach. According to the RTPA methodology for system modeling and refinement, a formal list can be rigorously modeled using two fundamental techniques known as the unified data models and the unified process models (Wang, 2007).

**Definition 1.** A **Unified Data Model** (UDM) is a generic architectural model for a software system, its internal control structures, and its interfaces with hardware components, which can be rigorously modeled and refined as an n-tuple, i.e.:

\[
UDM \triangleq \left( \bigotimes_{i=1}^{n} S_i \bigotimes_{e \in S_i} p_i(e) \right)
\]  

where \( S_i \), \( 1 \leq i \leq n \), is a set and also a type of elements \( e \) that share the property \( p_i \).

**Definition 2.** The **Unified Process Model** (UPM) of a program \( \varphi \) is a composition of a finite set of \( m \) processes according to the time-, event-, and interrupt-based process dispatching rules, \( \otimes_{e_k} \mathcal{P}_k \), i.e.:

\[
UPM \triangleq \varphi
= \bigotimes_{k=1}^{m} \left( \otimes_{e_k} \mathcal{P}_k \right)
= \bigotimes_{k=1}^{m} \left( \otimes_{e_k} \mathcal{P}_k \right) \bigotimes_{i=1}^{n-1} (s_i(k)r_j(k)s_j(k))
\]

where \( s_i \) and \( s_j \) are one of the 17 RTPA meta-processes, \( r_j \) is one of the 17 RTPA algebraic process operations, and \( e_k \) is a general, timing, or interrupt event.

This paper develops a comprehensive design pattern of formal lists using a doubly-linked-circular (DLC) model. In the remainder of this paper, the conceptual models of lists are de-
scribed as the initial requirements for describing the list ADT. The architectural model of lists is created as a set of UDMs based on doubly linked and circular nodes dynamically created in the memory. The static behaviors of lists are modeled and refined by a set of UPMs in three categories namely the management operations, traversal operations, and node I/O operations. The dynamic behaviors of lists are formally specified and refined by process scheduling and system dispatching models with UPMs.

2. THE CONCEPTUAL MODEL OF DLC-LISTS

A list is a typical ADT for manipulating large-volume information and dynamic data with a standard encapsulation and access interface. Lists are characterized by the separation of its logical interface and physical implementation in design and implementations. Lists as a finite sequence of elements are one of the most fundamental and widely used ADTs in system modeling. This section describes the conceptual model of lists. On the basis of the conceptual models as design requirements, a set of formal models of lists will be rigorously developed.

Definition 3. A list \( l \) is a finite sequence with \( n \) elements \( e_1, e_2, \ldots, e_n \) where all elements share a common data type such as a string (\( S \)), natural number (\( N \)), real number (\( R \)), record (the structural type, \( ST \)), or array (\( Array \)), i.e.:

\[
l \triangleq <e_1, e_2, \ldots, e_n>
\]

where \( e_{RT} \in \{S, N, R, ST, Array, \ldots\} \) as defined in RTPA.

Many important data objects and ADTs can be modeled and implemented by lists such as a set of data with keys, a sentence for natural language parsing, and a sequential file. Lists are also the fundamental data model for complex nonlinear structures such as trees and graphs. A sequential program can also be perceived as a finite list of statements (operations).

Various lists may be design and implemented in different forms using an array, a record, a set of doubly-linked records, and a set of doubly-linked-circular records. This paper puts emphases on the doubly-linked-circular record model of lists, because other simple forms of lists may be derived or tailored based on it.

Definition 4. A doubly linked circular list (DLC-List) is a list represented and implemented by a sequence of dynamically allocated nodes with a pair of links toward the prior and next nodes where the head and tail nodes are circularly linked.

It is noteworthy that there are two different but interactive facets in the models of DLC-Lists known as its architectures and behaviors. The conceptual model of the DLC-List is shown in Fig. 1. In the DLC model of lists, two global pointers, Head\( P \) and CurrentPtr\( P \), are provided in the pointer type (\( P \)) that point to the designated first and current node, respectively. The nodes are represented as a set of bidirectionally linked records. Each of which consists of a set of attributes such as the key\( N \) in type of natural number, the data\( RT \) in run-time type for implementation flexibility, as well as structural pointers PriorPtr\( P \), NextPtr\( P \), and Location\( P \). The pointer, Location\( P \), links a node of the DLC-List to its physical address assigned by dynamic memory allocation techniques such as Location\( P := \text{new}(\text{NodeID}\text{S}) \) during node creation.

Based on the architectural model, the behavioral models of DLC-Lists can be designed. The typical functional behaviors of DLC-Lists are identified such as create list, create node, empty
3. THE ARCHITECTURAL MODELS OF DLC-LISTS

On the basis of the conceptual models developed in the preceding section, the top-level ADT model of the DLC-List, List§, can be specified in RTPA as shown in Figure 3. The architectural model specifies that the DLC-List encompasses three parallel subsystems known as its architecture, static behaviors, and dynamic behaviors. According to the RTPA methodology for system modeling, specification, and refinement (Wang, 2007, 2008a), the following subsections will refine the top level framework of List§ into detailed architectural models (UDMs). Then, the static and dynamic behavioural models of DLC-Lists will be developed and refined in Sections 4 and 5.
The UDM model of DLC-Lists, List\textsuperscript{§}.Architecture\textsuperscript{ST}, as shown in Figure 4 provides a generic architectural model for any concrete list in applications with three categories of design attributes known as follows: a) The \textit{architectural} attributes determine the configuration of the list such as its logical size (Size\textsubscript{N}), physical size (MemSize\textsubscript{N}), current number of nodes (#Nodes\textsubscript{N}), the head pointer (Head\textsuperscript{P}), the current pointer (CurrentPtr\textsuperscript{P}), the tail pointer (Tail\textsuperscript{P}), the index number of a node (NodeIndex\textsubscript{N}), and the position for node insertion (Position\textsubscript{N}) such as a position at the current pointer and the prior or next of a given node identified by a key; b) The \textit{node} attributes model a sorted sequence of \(n\) nodes constrained by #Nodes\textsubscript{N} indexed by \textsubscript{i}N equivalent to its Key\textsubscript{N}. There are five fields for each node in the list: Key\textsubscript{N}, Data\textsubscript{RT}, PriorPtr\textsuperscript{P}, NextPtr\textsuperscript{P}, and Location\textsuperscript{P} (the physical location of the node); and c) The \textit{status} attributes are a set of indicators of operational results in Boolean type with a status prefix \& such as \&ListCreated\textsuperscript{BL}, \&NodeCreated\textsuperscript{BL}, \&NodeFound\textsuperscript{BL}, \&NodeRetrieved\textsuperscript{BL}, \&NodeUpdated\textsuperscript{BL}, \&NodeInserted\textsuperscript{BL}, \&NodeDeleted\textsuperscript{BL}, \&Empty\textsuperscript{BL}, \&Full\textsuperscript{BL}, and \&Released\textsuperscript{BL}. Each field of attributes in the UDM of lists is modeled by a primitive type of RTPA (Wang, 2007) and constrained by a certain scope or initial value given in the right-hand side of the vertical bar. It is noteworthy that the type of the data elements in a node is specified in the run-time type \textsubscript{RT}, i.e., \textsubscript{RT} \in \{S, N, R, ST, …\}, for design flexibility. However, the data elements of concrete nodes must be instantiated once it is chosen at run-time for a specific implementation of the generic abstract list.

The architectural model, List\textsuperscript{ST}, can be accessed or invoked in two ways as expressed in Eqs. 4.1 and 4.2. The random access in Eq. 4.1 allows a node of the list to be located by its pointer NodePtr\textsuperscript{P}; While the sequential access in Eq. 4.2 provides an additional approach to enable a node of the list to be located by its index number NodeIndex\textsubscript{N} or the key.

\begin{align*}
\text{TargetNode}^\text{P} & := \text{ListID}(\text{NodePtr}^\text{P}) \text{ST}.\text{Location}^\text{P} \quad (4.1) \\
\text{TargetNode}^\text{P} & := \text{ListID}^\text{ST}.\text{Node}(\text{NodeIndex}^\text{N})\text{ST}.\text{Location}^\text{P} \quad (4.2)
\end{align*}

4. THE BEHAVIORAL PROCESS MODELS OF DLC-LISTS

The static behaviors of the DLC-Lists, List\textsuperscript{§}.StaticBehaviors\textsuperscript{PC}, can be specified by a set of functional operations on its architectural model (UDM), i.e., List\textsuperscript{§}.Architecture\textsuperscript{PC}. On the basis of the UDM models of List\textsuperscript{ST} developed in the preceding subsection, the behaviors of the list ADT can be modeled as a set of UPMs operating on the UDMs of List\textsuperscript{ST} and related input variables. The high-level behavioral model of DLC-Lists is specified by List\textsuperscript{§}.StaticBehaviors\textsuperscript{PC} as shown in Fig. 5. The schemas of the UPMs in Figure 5 model the input data objects \texttt{<I:: (…)>}, output data objects \texttt{<O:: (…)>}, and operated UDMs \texttt{<UDM:: (…)> for each specific process of List\textsuperscript{§}. 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(es) that created and/or invoked them, particularly in real-time and embedded systems.

As informally modeled in Figure 2, the behavioral models of DLC-Lists are specified by a set of 11 UPMs in three categories known as the management operations (such as Create-List\textsuperscript{PC}, CreateNode\textsuperscript{PC}, EmptyTest\textsuperscript{PC}, FullTest\textsuperscript{PC}, Clear\textsuperscript{PC}, and Release\textsuperscript{PC}), traversal operations (such as FindNode\textsuperscript{PC}, InsertNode\textsuperscript{PC}, and DeleteNode\textsuperscript{PC}), and node I/O operations (such as RetrieveNode\textsuperscript{PC} and UpdateNode\textsuperscript{PC}). In the following subsections, each of the static behaviors of DLC-Lists, List\textsuperscript{§}.StaticBehaviors\textsuperscript{PC}, as given in Figure 5 will be further refined by a set of UPMs using the denotational mathematical notations and methodologies of RTPA.
4.1. The Behavioral Model of the List Creation Process

The list creation process, CreateListPC, is formally modeled as shown in Fig. 6, which establishes a new list in system memory or on a disk and links it to a specified logical ID of the list, ListID$. The input arguments of the process are the given name of the list as well as its size (or maximum capacity of nodes) and the type of data in each node. The output result of the process is the status of the list creation operation @ListCreatedBL. The UDM operated by this process is ListST.
Figure 5. The high-level UPM model of DLC-List behaviors

List α . Static Behavior: PC

- CreateList(PC<↓ : ListID, SizeN, DataRT> : <O : @ListCreatedBL> ; <UDM : ListST>)
- CreateNode(PC<↓ : ListID, DataRT> : <O : Node(KeyN, ST, @NodeCreatedBL) ; <UDM : ListST>)
- FindNode(PC<↓ : ListID, KeyN > : <O : NodeIndexN, @NodeFoundBL ; <UDM : ListST>)
- RetrieveNode(PC<↓ : ListID, KeyN > : <O : ListID, Node(NodeIndexN, ST, @NodeRetrievedBL) ; <UDM : ListST>)
- UpdateNode(PC<↓ : ListID, KeyN, DataRT> : <O : @NodeUpdatedBL ; <UDM : ListST>)
- InsertNode(PC<↓ : ListID, KeyN, DataST, PositionN > : <O : @NodeInsertedBL ; <UDM : ListST>)
- DeleteNode(PC<↓ : ListID, KeyN > : <O : @NodeDeletedBL ; <UDM : ListST>)
- EmptyTest(PC<↓ : ListID> : <O : @EmptyBL ; <UDM : ListST>)
- FullTest(PC<↓ : ListID> : <O : @FullBL ; <UDM : ListST>)
- Clear(PC<↓ : ListID> : <O : @ClearedBL ; <UDM : ListST>)
- Release(PC<↓ : ListID> : <O : @ReleasedBL ; <UDM : ListST>)

Figure 6. The UPM model of the list creation process

CreateList(PC<↓ : ListID, SizeN, DataRT> : <O : @ListCreatedBL> ; <UDM : ListST>)

→ (∗ ListIDST, @ListCreatedBL = F
  → ObjectIDS := ListID
  → @ElementsN := 1
  → ElementTypeRT := DataRT
  → AllocateObject(PC<↓ : ObjectIDS, #ElementsN, ElementTypeRT>)
    ;<O : @ObjectAllocatedBL ; <UDM : MEMST>)
→ (∗ @ObjectAllocatedBL = T
  → ListIDST.HeadP := MEM(ObjectIDS)ST  // List initialization
  → ListIDST.CurrentPtrP := ListIDST.HeadP
  → ListIDST.TailP := ListIDST.CurrentPtrP
  → ListIDST.SizeN := SizeN
  → ListIDST. @NodesN := 1
  → ListIDST. Nodes(0)ST.KeyN := 0  // Head node initialization
  → ListIDST. Nodes(0)ST.DataRT := ⊥
  → ListIDST. Nodes(0)ST.PriorPtrP := ListIDST.TailP
  → ListIDST. Nodes(0)ST.NextPtrP := ListIDST.TailP
  → ListIDST. Nodes(0)ST. LocationP := MEM(ObjectIDS)ST
  → ListIDST. EmptyBL := T  // Status initialization
  → ListIDST. @FullBL := F
  → ListIDST. @ListCreatedBL := T
  | ∗ ~
  → ListIDST. @ListCreatedBL := F
  → ! ("ListST memory allocation was failed.")
  )
→ BTTreeIDST, @TreeCreatedBL := F
  → ! ("ListIDST has already been existed.")
)
CreateList\textsubscript{PC} calls a system support process, AllocateObject\textsubscript{PC} (Wang, Ngolah, Tan, Tian, & Sheu, 2010), to physically set up the head node of the list using dynamic memory allocation technology. If a memory block is successfully obtained for the head node of the list, CreateList\textsubscript{PC} initializes the ListID\textsubscript{ST} by the following operations: a) List initialization connects the head pointer and the current pointer to the memory block, and assigns its expected size; b) Head node initialization sets the values of key and location of the head node, and leaves the data and both pointers undefined until the node insertion process; and c) Status initialization sets the initial values of statuses of the list as modeled in List\textsubscript{ST}. Otherwise, it reports an error ListID\textsubscript{ST}.\&Created\textsubscript{BL}:= F with an exception warning of memory allocation failed. In case the given ListID\textsubscript{ST} has already existed, CreateList\textsubscript{PC} results in a specific error message ListID\textsubscript{ST}.\&Created\textsubscript{BL}:= F.

4.2. The Behavioral Model of the Node Creation Process

The node creation process, CreateNode\textsubscript{PC}, is formally modeled as shown in Figure 7, which establishes a new node in system memory or on a disk and links it to a specified logical node, Node\textsubscript{ID}.\textsubscript{S}. The input arguments of the process are the given name of the node and the type of data for dynamic memory allocation. The output results of the process are the node as identified by its key and the status of the node creation operation. CreateNode\textsubscript{PC} calls the system support process, AllocateObject\textsubscript{PC}, to physical set up the node in memory. If a memory block is successfully obtained for the node of the list, CreateNode\textsubscript{PC} links the node to the allocated memory block and initializes the node’s physical location, while leaving the data, key, and both pointers undefined until the node insertion process; Otherwise, it reports an error ListID\textsubscript{ST}.\&NodeCreated\textsubscript{BL}:= F with an exception warning of memory allocation failed.

4.3. The Behavioral Model of the Node Finding Process

The node finding process, FindNode\textsubscript{PC}, is formally modeled as shown in Figure 8, which searches for a specific node based on its given key. The input arguments of the process are the given name of the list and the key as sequential number of the node. The output results of the process are the index of the node and the status of the node searching operation. FindNode\textsubscript{PC} initializes the search result as false before checking if the size of the list is empty. If not, FindNode\textsubscript{PC} iteratively compares the key of each node and the given key in order to find the target node. If it is found, FindNode\textsubscript{PC} returns the NodeIndex\textsubscript{N}, sets the search result as true, and exits the loop. Otherwise, the search result remains false. The result of FindNode\textsubscript{PC} will be used by other list operation processes such as RetrieveNode\textsubscript{PC}, UpdateNode\textsubscript{PC}, and DeleteNode\textsubscript{PC} for identifying the target node.

4.4. The Behavioral Model of the Node Retrieval Process

The node retrieval process, RetrieveNode\textsubscript{PC}, is formally modeled as shown in Figure 9, which reads and shows the contents of a target node in a list identified by the key. The input arguments of the process are the given name of the list and the key of the target node. The output results of the process are the node as identified by its key and the status of the node retrieval operation. RetrieveNode\textsubscript{PC} checks if the target list exists and is not empty before the following operations: a) Search the target node with Key\textsubscript{N} by calling FindNode\textsubscript{PC}; b) If the target node is found, display each field of the target node on the standard monitor, CRT\textsubscript{ST}, of the system supported by the process ShowNode\textsubscript{PC}, and set ListID\textsubscript{ST}.\&NodeRetrieved\textsubscript{BL}:= T; c) Otherwise, report an error ListID\textsubscript{ST}.\&NodeRetrieved\textsubscript{BL}:= F with an exception warning of “node was not found.” In
case the given ListID ST did not exist or was empty, RetrieveNodePC results in a specific error message ListID ST:= F.

4.5. The Behavioral Model of the Node Update Process

The node update process, UpdateNodePC, is formally modeled as shown in Figure 10, which writes new data into the target node in a list identified by the key. The input arguments of the process are the given name of the list, the key of the target node, and the data for updating. The output result of the process is the status of the node update operation. UpdateNodePC checks if the target list exists and is not empty before the following operations: a) Search the target node with KeyN by calling FindNodePC; b) If the target node is found, replace the data field of the

Figure 7. The UPM model of the node creation process

Figure 8. The UPM model of the node finding process
Figure 9. The UPM model of the node retrieval process

RetrieveNodePC(<I: ListIDST, KeyN>; <O: ListIDST, Node(NodeIndexN), ST>, @NodeRetrievedBL; <UDM: ListST>) ≡

\[
\begin{cases}
\text{FindNodePC(<I: ListIDST, KeyN>; <O: NodeIndexN, @NodeFoundBL; <UDM: ListST>)} & \\
\text{ShowNodePC(<I: ListIDST, NodeIndexN); <O: ListIDST, Node(NodeIndexN), ST>; <UDM: ListST}) & \\
\end{cases}
\]

ShowNodePC(<I: ListIDST, KeyN>; <O: ListIDST, Node(NodeIndexN), ST>; <UDM: ListST, CRTST>) ≡

\[
\begin{cases}
\text{ListIDST, Node(NodeIndexN), KeyN} | \text{CRTST} & \\
\text{ListIDST, Node(NodeIndexN), ST, DataRT} | \text{CRTST} & \\
\text{ListIDST, Node(NodeIndexN), ST, PriorPrP} | \text{CRTST} & \\
\text{ListIDST, Node(NodeIndexN), ST, NextPrP} | \text{CRTST} & \\
\text{ListIDST, Node(NodeIndexN), ST, LocationP} | \text{CRTST} & \\
\end{cases}
\]

Figure 10. The UPM model of the node update process

UpdateNodePC(<I: ListIDST, KeyN, DataRT>; <O: @NodeUpdateBL>; <UDM: ListST>) ≡

\[
\begin{cases}
\text{FindNodePC(<I: ListIDST, KeyN); <O: NodeIndexN, @NodeFoundBL; <UDM: ListST>)} & \\
\text{ListIDST, Node(NodeIndexN), ST, DataRT} := \text{DataRT} & \\
\text{ShowNodePC(<I: ListIDST, Node(NodeIndexN), ST); <O: @NodeUpdatedBL; <UDM: ListST}) & \\
\text{ListIDST, Node(NodeIndexN), ST, LocationP} | \text{CRTST} & \\
\end{cases}
\]
target node with given Data\textit{RT}, and set ListID\textit{ST}.\@NodeUpdated\textit{BL}: = T; c) Otherwise, report an error ListID\textit{ST}.\@NodeUpdated\textit{BL}: = F with an exception warning of “node was not found.” In case the given ListID\textit{ST} did not exist or was empty, UpdateNode\textit{PC} results in a specific error message ListID\textit{ST}.\@NodeUpdated\textit{BL}: = F.

4.6. The Behavioral Model of the Node Insertion Process

The \textit{node insertion} process, InsertNode\textit{PC}, is formally modeled as shown in Figure 11, which inserts a newly created node at the current pointer position or a specific position before or after a given node in the list. The input arguments of the process are the list ID, the key of target node, the data, and the position for node insertion. Its output is the status of the insert operation. InsertNode\textit{PC} checks if the target list exists and is not full before the following operations: a) Create the new node to be inserted by calling CreateNode\textit{PC}; b) Depending on the given insertion position, do the following: (1) If the insertion position is at the current pointer (\textit{PositionN} = 1), append the new node into the list at the current position, update the links of both nodes, increase the counter ListID\textit{ST}.\#Nodes\textit{N} by one, assign the key of the new node to be equal to the current number of nodes, and set ListID\textit{ST}.\@NodeInserted\textit{BL}: = T; (2) If the insert position is after the target node identified by Key\textit{N} (\textit{PositionN} = 2), find the target node, insert the new node into the list after the target node, update the links of the new node, the target node, and the original node following the target node, increase the counter ListID\textit{ST}.\#Nodes\textit{N} by one, assign the key of the new node equal to NodeIndex\textit{N} + 1, and set ListID\textit{ST}.\@NodeInserted\textit{BL}: = T; (3) If the insert position is before the target node identified by Key\textit{N} (\textit{PositionN} = 3), find the target node, insert the new node into the list in front of the target node, update the links of the new node, the target node, the original node prior to the target node, increase the counter ListID\textit{ST}.\#Nodes\textit{N} by one, assign the key of the new node equal to NodeIndex\textit{N} - 1, and set ListID\textit{ST}.\@NodeInserted\textit{BL}: = T; c) After the insertion operation, the sequence of affected nodes following the newly inserted node must be updated. This is implemented by increasing the keys of all affected nodes by one. Then, the full status of the list is checked; and d) In cases the target node could not be found, the given node could not be created, or the list did not exist or was full, an exception condition will be set by ListID\textit{ST}.\@NodeInserted\textit{BL}: = F.

4.7. The Behavioral Model of the Node Deletion Process

The \textit{node deletion} process, DeleteNode\textit{PC}, is formally modeled as shown in Figure 12, which removes a specific node at a given position in the list. The input arguments of the process are the list ID and the key of the target node. Its output is the status of the deletion operation. DeleteNode\textit{PC} checks if the target list exists and is not empty before the following operations: a) Find the target node to be deleted by calling FindNode\textit{PC}; b) If the target node is allocated with the return of its NodeIndex\textit{N}, do the following: (1) Determine the pointer of the target node, save both of its prior and next pointers, delete the target node by calling the support process ReleaseObject\textit{PC}, and disconnect the memory block from the target node; (2) Maintain the list by updating the node immediately before and after the deleted node, decrease ListID\textit{ST}.\#Nodes\textit{N} by one, and set ListID\textit{ST}.\@NodeInserted\textit{BL}: = T; (3) After the delete operation, the sequence of affected nodes following the deleted node must be updated. This is implemented by decreasing the keys of all affected nodes by one. Then, the empty status of the list is checked; and d) In cases the given node could not be found as well as the list did not exist or was empty, an exception condition will be set by ListID\textit{ST}.\@NodeDeleted\textit{BL}: = F.
Figure 11. The UPM model of the node insertion process

```
InsertNodePC(<1:: ListIDST, KeyH, DataST, PositionH>; <O:: @NodeInsertedBL>; <UDM:: ListST>) ⊆
  {→ (◆ ListIDST.@CreatedBL = T, ListIDST.@FullBL = T
      → CreateNodePC(<E:: NodeIDST, KeyH, DataST, PositionH>; <O:: NodeST, @NodeCreatedBL>; <UDM:: ListST>))
     → (◆ ListIDST.@NodeCreatedBL = T
          → (◆ PositionH = 1 // Insert a node at current position
               → NodeST.PriorP := ListIDST.CurrentP
               → NodeST.NextP := Nil
               → ListIDST.CurrentP := NodeST.LocationP
               → ListIDST.CurrentP := NodeST.LocationP
               → ListIDST.#Nodes := 1
               → ListIDST.KeyH := Nil
               → ListIDST.@NodeInsertedBL := T)
          → (◆ PositionH = 2 // Insert a node after a given node
               → FindNodePC(<1:: ListIDST, KeyH>; <O:: NodeIndexH, @NodeFoundBL>; <UDM:: ListST>)
                  → (◆ ListIDST.@NodeFoundBL = T
                    → TargetNodeP := ListIDST.Node(NodeIndexH, ST.LocationP
                    → NodeST.PriorP := TargetNodeP
                    → NodeST.NextP := NodeST.LocationP
                    → NodeST.KeyH := NodeIndexH + 1
                    → NodeST.LocationP := TargetNodeST.NextP
                    → ListIDST.#Nodes := 1
                    → ListIDST.KeyH := Nil
                    → ListIDST.@NodeInsertedBL := T)
            )
          )
          )
          )
     )
 )
 )
 )
→ (◆ ListIDST.@NodeInsertedBL := F
     → ! ('The target node was not found in ListIDST.')
     )
 )
 )
 )
 )
→ (◆ PositionH = 3 // Insert a node before a given node
     → ListIDST.@NodeInsertedBL := F
     → ! ('The target node was not found in ListIDST.')
     )
 )
 )
 )
→ (◆ ListIDST.@NodeInsertedBL := F
     → ! ('The target node was not found in ListIDST.')
     )
 )
 )
→ (◆ ListIDST.@NodeInsertedBL := F
     → ! ('The given node was not created due to memory limitation.')
     )
 )
 )
→ (◆ ListIDST.@NodeInsertedBL := F
     → ! ('ListIDST was full or not existed.')
     )
 )
```

4.8. The Behavioral Model of the List Empty Test Process

The empty test process, EmptyTest\textsubscript{PC}, is formally modeled as shown in Figure 13, which detects whether a given list is empty. The input argument of the process is the target list ID. Its output is the status of the list as being empty or not. The status of an empty list is characterized by ListID\textsubscript{ST}.\#Nodes\textsubscript{N} = 1 where only the head node exists. Therefore, EmptyTest\textsubscript{PC} verifies if the number of nodes in the list is one in order to determine whether it is empty. When the given list did not exist, EmptyTest\textsubscript{PC} generates a specific error message ListID\textsubscript{ST}.\@Empty\textsubscript{BL} := F.

4.9. The Behavioral Model of the List Full Test Process

The full test process, FullTest\textsubscript{PC}, is formally modeled as shown in Figure 14, which detects whether a given list is full. The input argument of the process is the target list ID. Its output is the status of the list as being full or not. The status of a full list is characterized by ListID\textsubscript{ST}.\#Nodes\textsubscript{N} = ListID\textsubscript{ST}.Size\textsubscript{N}. Therefore, FullTest\textsubscript{PC} verifies if the number of nodes has reached the defined capacity of the list in order to determine whether it is full. When the given list did not exist, FullTest\textsubscript{PC} generates a specific error message ListID\textsubscript{ST}.\@Full\textsubscript{BL} := F.
4.10. The Behavioral Model of the List Clear Process

The list clear process, ClearPC, is formally modeled as shown in Figure 15, which not only logically sets the given list as empty, but also physically releases all existing nodes in memory. The input argument of the process is the given list ID. Its output is the status of the clear operation. ClearPC is equivalent to the sequential operations to first release the given list and then to recreate it by calling ReleaseListPC and CreateListPC, respectively. In case the given list did not exist, ClearPC generates a specific error message ListIDST..SeleniumBL = F.

4.11. The Behavioral Model of the List Release Process

The list release process, ReleasePC, is formally modeled as shown in Figure 16, which physically removes a given list and releases the memory occupied. The input argument of the process is the given list ID. Its output is the status of the release operation. ReleasePC frees and returns the
memory block of each node to the system by calling a system support process, ReleaseObjectPC (Wang, Ngolah, Tan, Tian, & Sheu, 2010) for dynamic memory manipulation. It then disconnects the logical name of the list and its physical entity in memory. If the given list has not been created, ReleasePC produces a specific error message ListIDST.®ReleasedBL = F.

5. THE DYNAMIC BEHAVIORAL MODEL OF DLC-LISTS

According to the RTPA methodology, dynamic behaviors of a system are run-time process deployment and dispatching mechanisms based on the static behaviors modeled in UPMs, which is

---

Copyright © 2011, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
Figure 17. The UPM model of the dynamic behaviors of the DLC-List

particularly useful when the List§ is a component or embedded part of a large software system. The dynamic behaviors of List§ integrate and dispatch the static behavioral processes of lists as modeled in List§.StaticBehaviorsPC. Based on the UPMs developed in the preceding sections, this section models the dynamic behaviors of the DLC-List at run-time via the dynamic processes of system dispatching elaborated by the event-driven mechanisms of the system.

The dynamic behaviors of DLC-Lists, List§.DynamicBehaviorsPC, are shown in Figure 17, which serve as an interface of the system to external users who can invoke any pre-defined process in the list ADT. The dynamic behaviors of List§ at run-time can be formally specified using the RTPA process dispatching methodology. List§.DynamicBehaviorsPC as given in Fig. 17 establishes a set of top-level run-time relations between the 11 pairs of events and processes such as (@CreateListS ↔ CreateListPC), (@CreateNodeS ↔ CreateNodePC), (@FindNodeS ↔ FindNodePC), (@RetrieveNodeS ↔ RetrieveNodePC), (@UpdateNodeS ↔ UpdateNodePC), (@InsertNodeS ↔ InsertNodePC), (@DeleteNodeS ↔ DeleteNodePC), (@EmptyTestS ↔ EmptyTestPC), (@FullTestS ↔ FullTestPC), and (@ReleaseS ↔ ReleasePC). Any exceptional event that is not specified as a valid one in the system will be ignored by the skip operator (→ ∅). The event-driven dispatching mechanism also puts List§ into the context of a specific application.

The practical formal engineering methodology of RTPA for system modeling and specification provides a coherent notation system and a systematical approach for large-scale software and hybrid system design and implementation. A series of formal design models and frameworks of real-world and real-time applications in RTPA have been developed using RTPA notations and methodologies (Wang, 2002, 2007, 2008a; Wang & Huang, 2008; Wang, Tan, & Ngolah, 2010) in the formal design engineering approach, such as the telephone switching system (TSS) (Wang, 2009b), the lift dispatching system (LDS) (Wang et al., 2009), the automated teller machine (ATM) (Wang, Zhang, Sheu, Li, & Guo, 2010), the real-time operating system (RTOS+) (Wang et al., 2010a, 2010b), the autonomic code generator (RTPA-CG) (Wang, Tan, & Ngolah, 2010), the ADTs (Wang, Ngolah, Tan, Tian, & Sheu, 2010), the file management system (FMS) (Wang et al., 2011), and the air traffic control system (to be reported). Further studies have demonstrated that RTPA is not only elegant and practically useful as a generic notation and hierarchical methodology for software engineering, but also good at modeling human cognitive

6. CONCLUSION

Lists are one of the most fundamental and widely used ADTs in system modeling. However, there was a lack of a formal and complete model for lists. This paper has developed a comprehensive design pattern of formal lists using a doubly-linked-circular (DLC) model of lists. The conceptual model, architectural model, and static/dynamic behavioral models of lists have been systematically presented. The generic UDM and the 12 UPMs of lists have provided a set of rigorous architectural and behavioral models of formal lists based on them any concrete lists can be derived and implemented. An expressive and elegant denotational mathematics, Real-Time Process Algebra (RTPA), has been adopted to rigorously design and refine both architectural and behavioral models of lists and their manipulations in a top-down approach. This work has been applied in a number of real-time and nonreal-time system designs such as a real-time operating system (RTOS+), a file management system (FMS), and the ADT library for an RTPA-based automatic code generation tool.

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 anonymous reviewers for their valuable comments and suggestions.

REFERENCES


Copyright © 2011, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.


teller machine (ATM). *International Journal of Software Science and Computational Intelligence, 2*(1),
102–131. doi:10.4018/jssci.2010101907

Cambridge University Press.

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 and a BSc in Electrical
Engineering from Shanghai Tiedao University. 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 (Part 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 intel-
ligence, denotational mathematics, theoretical software engineering, coordinative work orga-
nization theory, cognitive complexity of software, and built-in tests. He has published over 118
peer reviewed journal papers, 200+ 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 35 years.

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.
Xinming Tan is a professor in the School of Computer Science and Technology at Wuhan University of Technology, China. He is the head of the Department of Computer Science. He received a BSc and an MSc in Computer Science at Wuhan University of Technology, and a PhD in Software Engineering at University of Calgary, Canada in 2007. His major research interests are in formal methods, real-time systems, and cognitive informatics.

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 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.