The Formal Design Models of a Set of Abstract Data Types (ADTs)

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

ABSTRACT

Type theories are fundamental for underpinning data object modeling and system architectural design in computing and software engineering. Abstract Data Types (ADTs) are a set of highly generic and rigorously modeled data structures in type theory. ADTs also play a key role in Object-Oriented (OO) technologies for software system design and implementation. This paper presents a formal modeling methodology for ADTs using the Real-Time Process Algebra (RTPA), which allows both architectural and behavioral models of ADTs and complex data objects. Formal architectures, static behaviors, and dynamic behaviors of a set of ADTs are comparatively studied. The architectural models of the ADTs are created using RTPA architectural modeling methodologies known as the Unified Data Models (UDMs). The static behaviors of the ADTs are specified and refined by a set of Unified Process Models (UPMs) of RTPA. The dynamic behaviors of the ADTs are modeled by process dispatching technologies of RTPA. This work has been applied in a number of real-time and non-real-time system designs such as a Real-Time Operating System (RTOS+), a Cognitive Learning Engine (CLE), and the automatic code generator based on RTPA.

Keywords: Abstract Data Types (ADT), Design Frameworks, Formal Design Models, Software Engineering, System Architecture Specification, Unified Data Models (UDM), Unified Process Models (UPM)

INTRODUCTION

Computational operations can be classified into the categories of data object, behavior, and resource modeling and manipulations. Based on this view, programs are perceived as a coordination of the data objects and behaviors in computing. Data object modeling is a process to creatively extract and abstractly represent a real-world problem by data models based on the constraints of given computing resources.

Using types to model real-world entities can be traced back to the mathematical thought
of Bertrand Russell (Russell, 1903) and Georg Cantor in 1932 (Lipschutz & Lipson, 1997). A type is a category of variables that share a common property such as the kind of data, domain, and allowable operations. Types are an important logical property shared by data objects in programming (Cardelli & Wegner, 1985; Mitchell, 1990). Although data in their most primitive form is a string of bits, types are found expressively convenient for data representation at the logical level in programming. Type theory can be used to prevent computational operations on incompatible operands, to help software engineers to avoid obvious and not so obvious pitfalls, and to improve regularity and orthogonality in programming language design.

**Definition 1.** A data type, shortly a type, is a set in which all member data objects share a common logical property or attribute.

The mathematical foundation of types is set theory. The maximum range of values that a variable can assume is a type, and a type is associated with a set of predefined or allowable operations. Methodologies of types and their properties have been defined in Real-Time Process Algebra (RTPA) (Wang, 2002, 2008a, 2008b, 2008c), where 17 primitive types in computing and software engineering have been elicited (Wang, 2007). A type can be classified as either primitive or derived (complex) types. The former is the most elementary types that cannot further be divided into more simple ones; the latter is a compound form of multiple primitive types based on certain type rules. Most primitive types are provided by programming languages; while most user defined types are derived ones.

A type system specifies data object modeling and manipulation rules of a programming language, as that of a grammar system that specifies program syntaxes and composing rules of the language. Therefore, the generic complex types can be modeled by abstract data types (Guttag, 1977; Broy et al., 1984), which are a logical model of a complex and/or user defined data type with a set of predefined operations.

**Definition 2.** An Abstract Data Type (ADT) is an abstract model of data objects with a formal encapsulation of the logical architecture and valid operations of the data object.

An ADT encapsulates a data structure and presents the user with an interface through which data can be accessed. It exports a type, a set of valid operations, and any axioms and preconditions that define the application domain of the ADT. ADTs extend type construction techniques by encapsulating both data structures and functional behaviors. The interface and implementation of an ADT can be separated in design and implementation. Based on the models of ADTs as generic data structures, concrete data objects can be derived in computing.

A number of ADTs have been identified in computing and system modeling such as stack, queue, sequence, record, array, list, tree, file, and graph (Wang, 2007). A summary of the ten typical ADTs is provided in Table 1 where the structures and behaviors of the ADTs are described. ADTs possess the following properties: (i) An extension of type constructions by integrating both data structures and functional behaviors; (ii) A hybrid data object modeling technique that encapsulates both user defined data structures (types) and allowable operations on them; (iii) The interface and implementation of an ADT are separated. Detailed implementation of the ADT is hidden to applications that invoke the ADT and its predefined operations.

There are a number of approaches to the specification of ADTs. Mathematically, the main approaches are of logical and algebraic, as well as their combinations. Although each of these approaches has its advantages, there are gaps when applying them to solve real-world problems. The logical approach is good at specifying high-level static behaviors of ADTs in forms of their preconditions and post-conditions of operations. For instance, a specification of queue as an ADT in predicate logic is shown in Figure 1 (Stubbs & Webre, 1985). However, the logic-based approach is not suit-
### Table 1. Summary of typical ADTs in software system modeling

<table>
<thead>
<tr>
<th>No.</th>
<th>ADT</th>
<th>Structure</th>
<th>Behaviors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stack</td>
<td>StackST</td>
<td>{Create, Push, Pop, Clear, EmptyTest, FullTest, Release}</td>
</tr>
<tr>
<td>2</td>
<td>Queue</td>
<td>QueueST</td>
<td>{Create, Enqueue, Serve, Clear, EmptTest, FullTest, Release}</td>
</tr>
<tr>
<td>3</td>
<td>Sequence</td>
<td>SequenceST</td>
<td>{Create, Append, FindElement, Retrieve, Clear, EmptyTest, FullTest, Release}</td>
</tr>
<tr>
<td>4</td>
<td>Record</td>
<td>RecordST</td>
<td>{Create, Initialization, UpdateField, UpdateRecord, RetrieveField, RetrieveRecord, Release}</td>
</tr>
<tr>
<td>5</td>
<td>Array</td>
<td>ArrayST</td>
<td>{Create, Update, Retrieve, Initialization, Clear, Release}</td>
</tr>
<tr>
<td>6</td>
<td>List</td>
<td>ListST</td>
<td>{Create, FindNext, FindPrior, FindLast, FindKey, Retrieve, Update, InsertAfter, InsertBefore, Delete, CurrentPos, FullTest, EmptyTest, GetSize, Clear, Release}</td>
</tr>
<tr>
<td>7</td>
<td>Binary Tree</td>
<td>B-TreeST</td>
<td>{Create, Traverse, Insert, DeleteSub, Update, Retrieve, Find, Characteristics, EmptyTest, Clear, Release}</td>
</tr>
<tr>
<td>8</td>
<td>File (Sequential)</td>
<td>SeqFileST</td>
<td>SeqFileST $\cup {\text{Read, Append, EmptyTest, FullTest}}$</td>
</tr>
<tr>
<td>9</td>
<td>File (Random)</td>
<td>RandFileST</td>
<td>{Create, Open, Reset, Save, Close, Delete} $\cup {\text{ReadRec, InsertRec, UpdateRec, DeleteRec, EmptyTest, FullTest}}$</td>
</tr>
<tr>
<td>10</td>
<td>Diagraph</td>
<td>DigrapgST</td>
<td>{Create, InsertNode, InsertEdge, DeleteNode, DeleteEdge, Retrieve, Update, FindNode, FindEdge, CurrentNode, CurrentEdge, Release}</td>
</tr>
</tbody>
</table>

**Figure 1. A logical model of the ADT queue (adapted from Stubbs & Webre, 1985)**

<table>
<thead>
<tr>
<th>Elements:</th>
<th>c, f, g, ... of type stdelement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structure:</td>
<td>$Q = {&lt;c, t_0&gt;, &lt;f, t_1&gt;, &lt;g, t_2&gt;, \ldots}$, where $t_i$ is the time of insertion of $&lt;x, t_j&gt;$ into $Q$ if $x \not\in Q$ and $t_i &lt; t_j$</td>
</tr>
<tr>
<td>Domain:</td>
<td>$0 \leq #Q \leq \text{maxsize}$</td>
</tr>
<tr>
<td>Operations:</td>
<td></td>
</tr>
<tr>
<td>enqueue($c$: stdelement)</td>
<td>$\exists Q \land #Q \neq \text{maxsize}$</td>
</tr>
<tr>
<td>post:</td>
<td>$Q = Q' \cup {&lt;c, t_0&gt;}$</td>
</tr>
<tr>
<td>serve($var$: stdelement)</td>
<td>$\exists Q \land Q \not\leftarrow {}$</td>
</tr>
<tr>
<td>post:</td>
<td>$Q = Q' - {&lt;c, t_0&gt;}(\forall x, y \in Q, t_x \leq t_y)$</td>
</tr>
<tr>
<td>empty: boolean</td>
<td>$\exists Q$</td>
</tr>
<tr>
<td>pre:</td>
<td>$\exists Q$</td>
</tr>
<tr>
<td>post:</td>
<td>$\text{empty} = (#Q = 0)$</td>
</tr>
<tr>
<td>full: boolean</td>
<td>$\exists Q$</td>
</tr>
<tr>
<td>pre:</td>
<td>$\exists Q$</td>
</tr>
<tr>
<td>post:</td>
<td>$\text{full} = (#Q = \text{maxsize})$</td>
</tr>
<tr>
<td>clear:</td>
<td>$\exists Q$</td>
</tr>
<tr>
<td>create:</td>
<td>true</td>
</tr>
<tr>
<td>post:</td>
<td>$\exists Q \land Q = {}$</td>
</tr>
</tbody>
</table>
able for behavioral refinement and the modeling of dynamic behaviors of ADTs.

Another approach to ADT specification is the algebraic methods, which treat an ADT as an algebraic structure of sorts and operations on the sorts (McDermid, 1991). An algebraic model of an ADT, stack, is illustrated in Figure 2 (Louden, 1993). The advantage of algebraic models of ADTs is its abstraction and elegance. However, it is often too abstract for system implementers and users, especially for dealing with some non-trivial ADTs.

In order to enhance the algebraic methods and reduce their complexity, RTPA is introduced (Wang, 2007). The RTPA methodology provides an explicit and easy-to-use algebraic approach for ADT and system modeling. RTPA reveals two fundamental technologies for ADT system modeling and refinement known as the unified data model and the unified process model (Wang, 2007).

**Definition 3.** A **Unified Data Model (UDM)** is a generic architectural model of a software system as well as its hardware components, interfaces, and internal control structures, which can be rigorously modeled and refined in denotational mathematics as a tuple, i.e.:

\[
UDDM = \left( \bigwedge \mathbb{R}^{s \in S, e \in S_j} \forall e \in S_j, p(e) > \right)
\]

**Definition 4.** The **Unified Process Model (UPM)** of a program \( \phi \) is a composition of a finite set of \( k \) processes according to the time-, event-, and interrupt-based process dispatching rules, i.e.:

\[
\phi = \bigwedge \mathbb{R}^{k=1}_m \left( \bigwedge \mathbb{R}^{i=1}_{n-1} \left( \bigwedge \mathbb{R}^{j=1}_{i+1} \right) \right)
\]

This paper presents a formal algebraic modeling methodology for ADTs using a denotational mathematics, RTPA, which allows both architectural and behavioral models of ADTs and complex data objects to be rigorously designed and implemented in a top-down approach. The methodology of a denotational mathematics based on RTPA is introduced to formally model and refine architectures, static behaviors, and dynamic behaviors of a set of ADTs. Stacks, queues, sequences, and

---

**Figure 2. An algebraic model of the ADT stack (adapted from Louden, 1993)**

<table>
<thead>
<tr>
<th>ADT Stack (element : ( RT )) : ( ST )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>operations:</strong></td>
</tr>
<tr>
<td>create: ( \rightarrow ) stack</td>
</tr>
<tr>
<td>push: ( stack \times element \rightarrow stack )</td>
</tr>
<tr>
<td>pop: ( stack \rightarrow element )</td>
</tr>
<tr>
<td>empty: ( stack \rightarrow Boolean )</td>
</tr>
<tr>
<td><strong>variables:</strong></td>
</tr>
<tr>
<td>s: stack;</td>
</tr>
<tr>
<td>e: element</td>
</tr>
<tr>
<td><strong>axioms:</strong></td>
</tr>
<tr>
<td>create(s) = s</td>
</tr>
<tr>
<td>push(s, e) = es</td>
</tr>
<tr>
<td>pop(create(s)) = error</td>
</tr>
<tr>
<td>pop(push(s, e)) = e</td>
</tr>
<tr>
<td>empty(create(s)) = T</td>
</tr>
<tr>
<td>empty(push(s, e)) = F</td>
</tr>
</tbody>
</table>
records are chosen to comparatively elaborate the proposed RTPA-based ADT modeling methodologies. The architectural models of ATDs are created using the RTPA methodology known as UDMs. The static behaviors of ADTs are specified and refined by the RTPA methodology known as UPMs. The dynamic behaviors of ADTs are modeled by the RTPA methodology known as the event-driven process dispatching models.

THE FORMAL MODEL OF ADT1 – STACKS

A stack is a typical data structure for modeling the Last-In-First-Out (LIFO) mechanism of an ADT with a set of elements in the same type. The conceptual model of stacks and its key control variables are introduced in this section. Based on it, formal models of the stack ADT in terms of its architectural and static/dynamic behavioral models are rigorously developed in RTPA.

The Conceptual Model of Stacks

The stack as a common data structure is described as shown in Figure 3. The protocol of stacks is LIFO, which implies that the I/O operations of stacks must be on its top or most recent elements as identified by the pointer of the current position. In the stack model, StackST, each ElementRT shares the same type where RT represents the run-time type in RTPA. Two of the key control variables of the stack are SizeN denoting the maximum capacity of the stack and CurrentPosP denoting the current top position of the stack. The addresses of elements grow from the bottom up with the relative base address at 0H.

The top-level ADT model of the stack, StackST, encompassing its architecture, static behaviors, and dynamic behaviors, can be specified in RTPA as follows:

\[
\text{ADT}_S.\text{StackST} \triangleq \text{StackST.} \text{ArchitectureST} \\
\quad \quad \text{| StackST.} \text{StaticBehaviorsPC} \\
\quad \quad \text{| StackST.} \text{DynamicBehaviorsPC}
\]

According to the RTPA methodology for system modeling, specification, and refinement (Wang, 2007, 2008a), the following subsections will refine the top level framework of StackST into detailed architectural models (UDMs) and behavioral models (UPMs).

The Architectural Model of Stacks

The architecture of StackST can be rigorously modeled using the UDM technology of RTPA, which is a predefined class of system hardware or internal control models that can be inherited or implemented by corresponding UDM objects as specific instances in the succeeding architectural refinement for the system. The UDM model of the stack ADT, StackST. ArchitectureST, as shown in Figure 4 provides a generic architectural model for any concrete stack in applications with three key fields, i.e., ElementRT, SizeN, and CurrentPosP, where the constraints of each field are given in the right-hand side of the vertical bar. Supplement to the key architectural attributes, there is a set of status fields in the stack, which models the current operational status of the stack in Boolean type such as CreatedBL, PushedBL, PoppedBL, ClearedBL, EmptyBL, FullBL, and ReleasedBL. It is noteworthy that the type of the data elements in an abstract stack is specified in the run-time type RT, i.e., RT ∈ {S, N, R, ST, …}, for design flexibility. However, the data elements in a concrete stack must be in the same type once it is chosen at run-time for a specific implementation of an instance of the generic abstract stack.

The Static Behavioral Model of Stacks

A static behavior is an encapsulated function of a given system that can be determined
before run-time (Wang, 2007). On the basis of the UDM model of Stack\textsubscript{ST} developed in the preceding subsection, the behaviors of the stack can be modeled as a set of UPMs operating on the UDMs and related input variables. The high-level behavioral model of the stack ADT is modeled by Stack\textsubscript{ST}.StaticBehaviors\textsubscript{PC} as shown in Eq. 4. It can be further refined by a set of UPMs for each of the behavioral processes. The schemas of the UPMs in Eq. 4 model the input data objects (<\textit{I}:: (…)>), output data objects (<\textit{O}:: (…)>), and operated UDMs (<\textit{UDM}:: (…)>) for each specific process of Stack\textsubscript{ST}. 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 a real-time system.

Stack\textsubscript{ST}.StaticBehaviors\textsubscript{PC} \triangleq

- \texttt{CreateStack\textsubscript{PC}(<\textit{I}:: StackIDS, SizeN, ElementRT>; <\textit{O}:: StackIDST.CratedBL}; <\textit{UDM}:: StackIDST>)
- \texttt{Push\textsubscript{PC}(<\textit{I}:: StackIDS, ElementRT>; <\textit{O}:: StackIDST.PushedBL}; <\textit{UDM}:: StackIDST>)
- \texttt{Pop\textsubscript{PC}(<\textit{I}:: StackIDS>; <\textit{O}:: StackIDST.PoppedBL, ElementRT}; <\textit{UDM}:: StackIDST>)
- \texttt{Clean\textsubscript{PC}(<\textit{I}:: StackIDS>; <\textit{O}:: StackIDST.ClearedBL}; <\textit{UDM}:: StackIDST>)
- \texttt{EmptyTest (<\textit{I}:: StackIDS>; <\textit{O}:: StackIDST.EmptyBL}; <\textit{UDM}:: StackIDST>)
- \texttt{FullTest (<\textit{I}:: StackIDS>; <\textit{O}:: StackIDST.FullBL}; <\textit{UDM}:: StackIDST>)
- \texttt{ReleaseStack\textsubscript{PC}(<\textit{I}:: StackIDS}; <\textit{O}:: StackIDST. ReleasedBL}; <\textit{UDM}:: StackIDST>)

The following subsections describe how each of the seven behavioral processes of ADT Stack\textsubscript{ST} as specified in Eq. 4 are modeled and refined using the denotational mathematical notations and methodologies of RTPA.

\textbf{a) The Process of Stack Creation}

The stack creation process of Stack\textsubscript{ST}, CreateStack\textsubscript{PC}, is formally modeled as shown in Figure 5, which establishes a new stack in system memory and links it to a specified logical ID. The input arguments of the process are the given name of the stack as well as its size and the type of element. The output result and status are the creation of the stack as well as the settings of related initial values in the UDM of StackIDST.Architecture\textsubscript{ST}. In order to create a physical stack and link it to the logical name of StackIDST, the process calls a system support process, AllocateObject\textsubscript{PC}, for dynamic memory manipulation for ADTs as illustrated in Figure 12. When the given stack has already existed or cannot be established due to memory

---

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
availability, CreateStack\textsubscript{PC} results in a specific error message and sets StackID\textsubscript{ST}.Created\textsubscript{BL} = F.

\textbf{b) The Process of Stack Push}

The push process of Stack\textsubscript{ST}, Push\textsubscript{PC}, is formally modeled as shown in Figure 6, which puts a given element onto the top of the stack. The input arguments of the process are the target stack ID and the given element. The output is the status of the push operation. Push\textsubscript{PC} writes the given element onto the top of the current stack contents, after checking that the stack is not full and the CurrentPos\textsubscript{P} has been updated. When the given stack has already been full or does not exist, Push\textsubscript{PC} generates a specific error message and sets StackID\textsubscript{ST}.Pushed\textsubscript{BL} = F.

\textbf{c) The Process of Stack Pop}

The pop process of Stack\textsubscript{ST}, Pop\textsubscript{PC}, is formally modeled as shown in Figure 7, which elicits the top element of the stack. The input argument of the process is the target stack ID. Its outputs are the top element obtained and the status of the pop operation. When the target stack is not empty, Pop\textsubscript{PC} does not only read the current top element of the stack, but also remove it from the stack. As a result, Pop\textsubscript{PC} reduces the current pointer by one after the pop operation. When the given stack has already been empty or does not exist, Pop\textsubscript{PC} generates a specific error message and sets StackID\textsubscript{ST}.Popped\textsubscript{BL} = F.

\textbf{d) The Process of Clear Stack}

The clear process of Stack\textsubscript{ST}, Clear\textsubscript{PC}, is formally modeled as shown in Figure 8, which logically sets all elements of the stack as empty. The input argument of the process is the given stack ID. Its output is the status of the clear operation. Clear\textsubscript{PC} only logically sets the pointer StackID\textsubscript{ST}.CurrentPos\textsubscript{P} = 0 in order to denote the stack has been cleaned, rather than to remove all data elements of the stack. It is noteworthy that Clear\textsubscript{PC} is different from Release\textsubscript{PC}, where the former logically declares the given stack as empty, while the latter physically removes the target stack as well as its contents from the memory. If the given stack does not exist, Clear\textsubscript{PC} generates a specific error message.

\textbf{e) The Process of Stack Empty Test}

The empty test process of Stack\textsubscript{ST}, EmptyTest\textsubscript{PC}, is formally modeled as shown in Figure 9, which detects whether a given stack is empty. The input argument of the process is the target stack ID. Its output is the status of the stack as being empty or not. The status of an empty stack is characterized by StackID\textsubscript{ST}.CurrentPos\textsubscript{P} = 0. Therefore, EmptyTest\textsubscript{PC} verifies if the logical top pointer StackID\textsubscript{ST}.CurrentPos\textsubscript{P} points at the relative base address of the stack in order to determine whether the stack is empty or not.
When the target stack does not exist, EmptyTestPC generates a specific error message.

f) The Process of Stack Full Test

The full test process of StackST, FullTestPC, is formally modeled as shown in Figure 10, which detects whether a given stack is full. The input argument of the process is the target stack ID. Its output is the status of the stack as being full or not. The status of a full stack is characterized by StackIDST.CurrentPosP = StackIDST.SizeN. Therefore, FullTestPC verifies if the logical top pointer StackIDST.CurrentPosP is equal to the physical size of the stack StackIDST.SizeN in order to determine whether the stack is full or not. When the given stack does not exist, FullTestPC generates a specific error message.

g) The Process of Stack Release

The stack release process, ReleaseStackPC, is formally modeled as shown in Figure 11, which physically removes a given stack and related memory with the contents. The input argument of the process is the given stack ID. Its output is the status of the release operation. ReleaseStackPC disconnects the physical stack and its logical name. Then, the memory space of the physical stack is returned to the system by calling a system support process, ReleaseObjectPC, for dynamic memory manipulation for ADTs as illustrated in Figure 13. If the given stack does not exist logically or physically, ReleaseStackPC produces a specific error message and sets StackIDST.ReleasedBL = F.
h) Associate Processes of Dynamic Memory Manipulations for ADTs

There are two support processes, AllocateObjectPC and ReleaseObjectPC, for dynamic memory allocation and the management of ADT creation and release manipulations. The AllocateObjectPC process as shown in Fig. 12 finds out a suitable block of available memory for the required ObjectST, whose size is determined by:

|Elements| N × Byte(ElementTypeRT)| N| bytes.

AllocateObjectPC repeats requests a unit of suitable bytes for each element of ObjectST. Then, it links the logical ID of the object to the

---

**Figure 6. The UPM model of the stack push process**

```
PushPC(<I:: StackIDST, ElementRT>; <O:: StackIDST, PushedBL>; <UDM:: StackIDST>) △
{
  → ( ✶ StackIDST.CreatedBL = T
       → ( ✶ StackIDST.CurrentPosP < StackIDST.SizeH
            → ↑ (StackIDST.CurrentPosP)
            → ElementRT < StackIDST(CurrentPosP)ST.ElementRT
            → StackIDST.PushedBL := T
            | ✶~
                → StackIDST.PushedBL := F
                → StackIDST.FullBL := T
                → ! (“StackIST is full.”)
           )
       | ✶~
           → StackIDST.PushedBL := F
           → ! (“StackIST does not exist.”)
    )
}
```

---

**Figure 7. The UPM model of the stack pop process**

```
PopPC(<I:: StackIDST>; <O:: StackIDST, PoppedBL, ElementRT>; <UDM:: StackIDST>) △
{
  → ( ✶ StackIDST.CreatedBL = T
       → ( ✶ StackIDST.CurrentPosP > 0
            → StackIDST(CurrentPosP)ST.ElementRT > ElementRT
            → ↓ (StackIDST.CurrentPosP)
            → StackIDST.PoppedBL := T
            | ✶~
                → StackIDST.PoppedBL := F
                → StackIDST.EmptyBL := T
                → ! (“StackIST is empty.”)
           )
       | ✶~
           → StackIDST.PoppedBL := F
           → ! (“StackIST does not exist.”)
    )
}
```
Figure 8. The UPM model of the stack clear process

\[
\text{ClearPC}(<\text{I}: \text{StackIDS}>, <\text{O}: \text{StackIDST.ClearedBL}>; <\text{UDM}: \text{StackIDST}>) \triangleq \\
\{ \\
\rightarrow ( \begin{aligned} &\text{StackIDST.CreatedBL} = \text{T} \\
&\rightarrow \text{StackIDST.CurrentPosP} := 0 \\
&\rightarrow \text{StackIDST.ClearedBL} := \text{T} \\
&\rightarrow \text{StackIDST.EmptyBL} := \text{T} \\
\end{aligned} \\
|\begin{aligned} &\text{StackIDST.ClearedBL} := \text{F} \\
&\rightarrow ! (\text{‘StackIDST’ does not exist.”}) \\
\end{aligned} \\
\} \\
\}
\]

Figure 9. The UPM model of the stack empty test process

\[
\text{EmptyTestPC}(<\text{I}: \text{StackIDS}>, <\text{O}: \text{StackIDST.EmptyBL}>, <\text{UDM}: \text{StackIDST}>) \triangleq \\
\{ \\
\rightarrow ( \begin{aligned} &\text{StackIDST.CreatedBL} = \text{T} \\
&\rightarrow ( \begin{aligned} &\text{StackIDST.CurrentPosP} = 0 \\
&\rightarrow \text{StackIDST.EmptyBL} := \text{T} \\
&\rightarrow \text{StackIDST.FullBL} := \text{F} \\
\end{aligned} \\
|\begin{aligned} &\text{StackIDST.EmptyBL} := \text{F} \\
\end{aligned} \\
\} \\
|\begin{aligned} &\text{StackIDST.EmptyBL} := \text{F} \\
&\rightarrow ! (\text{‘StackIDST’ does not exist.”}) \\
\end{aligned} \\
\} \\
\}
\]

Figure 10. The UPM model of the stack full test process

\[
\text{FullTestPC}(<\text{I}: \text{StackIDS}>, <\text{O}: \text{StackIDST.FullBL}>, <\text{UDM}: \text{StackIDST}>) \triangleq \\
\{ \\
\rightarrow ( \begin{aligned} &\text{StackIDST.CreatedBL} = \text{T} \\
&\rightarrow ( \begin{aligned} &\text{StackIDST.CurrentPosP} = \text{StackIDST.SizeN} \\
&\rightarrow \text{StackIDST.FullBL} := \text{T} \\
&\rightarrow \text{StackIDST.EmptyBL} := \text{F} \\
\end{aligned} \\
|\begin{aligned} &\text{StackIDST.FullBL} := \text{F} \\
\end{aligned} \\
\} \\
|\begin{aligned} &\text{StackIDST.FullBL} := \text{F} \\
&\rightarrow ! (\text{‘StackIDST’ does not exist.”}) \\
\end{aligned} \\
\} \\
\}
\]
Figure 11. The UPM model of the release stack process

```
ReleaseStackPC(<I:: StackIDS>; <O:: StackIDST. ReleasedBL>; <UDM:: StackIDST>) \triangleq
{
  \rightarrow ( \triangleright StackIDST. CreatedBL = T
    \rightarrow ObjectIDS := StackIDS
    \rightarrow ReleaseObjectPC(<I:: ObjectIDS>; <O:: ObjectReleasedBL>; <UDM:: MEMST>)
  )
  \rightarrow ( \triangleright ObjectReleasedBL = T
    \rightarrow StackIDS := MEM(PhysicalStackST)
    \rightarrow StackIDST. ReleasedBL := T
  )
  \rightarrow StackIDST. ReleasedBL := F
  \rightarrow ! (Target memory for ‘StackIDST’ is not found.)
  )
  \rightarrow StackIDST. DeletedBL := F
  \rightarrow ! (‘StackIDST’ does not existed.)
}
```

Figure 12. The UPM model of the allocate object process

```
AllocateObjectPC(<I:: ObjectIDS, #ElementsN, ElementTypeBT>; <O:: ObjectIDST. ExistedBL, MEMST>; <UDM:: ObjectIDST, MemoryST>) \triangleq
{
  \rightarrow R New (ObjectIDST(ST) : ElementTypeBT)
  \rightarrow 1 \setminus N
  \rightarrow ( \triangleright ObjectAllocatedBL = T
    \rightarrow ObjectIDS := MEM(ObjectIDSTST)
    \rightarrow ObjectIDST. ExistedBL := T
  )
  \rightarrow ObjectIDST. ExistedBL := F
}
```

Figure 13. The UPM model of the release object process

```
ReleaseObjectPC(<I:: ObjectIDS>; <O:: ObjectIDST. ReleasedBL>; <UDM:: ObjectIDST, MemoryST>) \triangleq
{
  \rightarrow ObjectIDS := MEM(ObjectIDSTST)
  \rightarrow ReleasePC(<I:: MEM(ObjectIDST); <O:: MemoryReleasedBL>; <UDM:: MemoryST>)
  \rightarrow ObjectIDS := \perp
  \rightarrow ( \triangleright MemoryReleasedBL = T
    \rightarrow ObjectIDST. ReleasedBL := T
  )
  \rightarrow ObjectIDST. ReleasedBL := F
}
```
allocated memory block. If memory allocation is failed, AllocateObject\textsubscript{PC} feeds back an error message ObjectID\textsubscript{ST}.Existed\textsubscript{BL} = F.

The ReleaseObject\textsubscript{PC} process is a support process as shown in Figure 13, which is invoked by an ADT release process such as ReleaseStack\textsubscript{PC}. The ReleaseObject\textsubscript{PC} process identifies an associate memory block of a given ObjectID\textsubscript{S} and disconnects the object from the memory. After the release operation, the object is set to be undefined, i.e., ObjectID\textsubscript{ST} := ⊥. If memory release is failed, ReleaseObject\textsubscript{PC} feeds back an error message ObjectID\textsubscript{ST}.Released\textsubscript{BL} = F.

The Dynamic Behavior Model of Stacks

Dynamic behaviors of a system are run-time process deployment and dispatching mechanisms based on the static behaviors. Because system static behaviors are a set of component processes of the system, to put the static processes into an interacting system at run-time, the dynamic behaviors of the system in terms of process dispatching are yet to be specified. With the work products, Stack\textsubscript{ST}.StaticBehaviors\textsubscript{PC}, developed in the preceding section as a set of static behavioral processes, this subsection describes the dynamic behaviors of Stack\textsubscript{ST} at run-time using the RTPA process dispatching methodology. Stack\textsubscript{ST}.DynamicBehaviors\textsubscript{PC} as shown in Figure 14 models the event-driven behaviors of Stack\textsubscript{ST}, which establishes the relations between system events and the stack behavioral processes. The event-driven dispatching mechanisms also put Stack\textsubscript{ST} into the context of applications.

Figures 3 through 14 describe a typical ADT model, Stack\textsubscript{ST}, in a coherent design and using an unified formal notation. With the RTPA specification and refinement methodology, the mechanisms, architectures, and behaviors of Stack\textsubscript{ST} are rigorously and precisely modeled.

THE FORMAL MODEL OF ADT2 – QUEUES

A queue is a typical data structure for modeling the First-In-First-Out (FIFO) mechanism of an ADT with a set of elements in the same type. The conceptual model of queues and its key control variables are introduced in this section. Based on it, formal models of the queue ADT in terms of its architectural and static/dynamic behavioral models are rigorously developed in RTPA.

The Conceptual Model of Queues

The queue as a common data structure is described as shown in Figure 15. The protocol of queues is FIFO, which implies that the I/O operations of queues must be at both its tail and head, respectively. In the queue model, Queue\textsubscript{ST}, each element\textsubscript{RT} share the same type \textsubscript{RT}. Two of the key control variables of the queue are SizeOfQueue\textsubscript{N} denoting the maximum capacity of the queue and Current-Pos\textsubscript{P} denoting the current tail position of the queue. The address of an element is growing from head to tail with the relative base address reserved at 0\textsubscript{H}. The head or the front element of the queue, Element\textsubscript{1RT}, is always located at address 1\textsubscript{H} based on the relative base address of the queue Base\textsubscript{N}.

The top-level ADT model of queue, Queue\textsubscript{ST}, encompassing its architecture, static behaviors, and dynamic behaviors, can be specified in RTPA as follows:

\[
\text{ADT}§.\text{Queue}^{\text{ST}} \triangleq \text{Queue}^{\text{ST}}.\text{Architecture}^{\text{ST}} \quad || \quad \text{Queue}^{\text{ST}}.\text{StaticBehaviors}^{\text{PC}} \\
\quad || \quad \text{Queue}^{\text{ST}}.\text{DynamicBehaviors}^{\text{PC}}
\]

(5)

According to the RTPA methodology for system modeling, specification, and refinement (Wang, 2007, 2008a), the following subsections will refine the top level framework of Queue\textsubscript{ST} into detailed architectural models (UDMs) and behavioral models (UPMs).

The Architectural Model of Queues

The architecture of Queue\textsubscript{ST} can be rigorously modeled using the UDM technology of RTPA. The UDM model of the queue ADT, Queue\textsubscript{ST}.Architecture\textsubscript{ST}, as shown in Figure 16 provides
a generic architectural model for any concrete queue in applications with three key fields, i.e., \( \text{Element}^{\text{RT}} \), \( \text{Size}^{\text{N}} \), and \( \text{CurrentPos}^{\text{P}} \), where the constraints of each field are given in the right-hand side of the vertical bar. Supplement to the key architectural attributes, there is a set of status fields in the queue, which models the current operational status of the queue in Boolean type such as \( \text{Created}^{\text{BL}} \), \( \text{Enqueued}^{\text{BL}} \), \( \text{Served}^{\text{BL}} \), \( \text{Cleared}^{\text{BL}} \), \( \text{Empty}^{\text{BL}} \), \( \text{Full}^{\text{BL}} \), and \( \text{Released}^{\text{BL}} \).

**The Behavioral Model of Queues**

On the basis of the UDM model of Queue\( ^{\text{ST}} \) developed in the preceding subsection, the behaviors of the queue can be modeled as a set of UPMs operating on the UDMs and related input variables. The high level behavioral model of the queue ADT is modeled by Queue\( ^{\text{ST}} \).StaticBehaviors\( ^{\text{PC}} \) as shown in Eq. 6. The schemas of the queue can be further refined by a set of UPMs for each of the behavioral processes.
A set of seven behavioral processes such as create, enqueue, serve, clear, empty test, full test, and release is designed in Queue\_ST. StaticBehaviors\_PC. The following subsections describe how the static behaviors of Queue\_ST as specified in Eq. 6 are modeled and refined using the denotational mathematical notations and methodologies of RTPA. Because of the similarity of the ADT manipulation processes between Queue\_ST and Stack\_ST as described in Eq. 4, only two key processes, Enqueue\_PC and Serve\_PC, of Queue\_ST will be formally modeled in the following subsections.

a) The Process of Enqueue

The enqueue process of Queue\_ST, Enqueue\_PC, is formally modeled as shown in Figure 17, which puts a given element at the tail of the queue. The input arguments of the process are the target queue ID and a given element. Its output is the status of the enqueue operation. Enqueue\_PC checks if the given queue is not full and shifts CurrentPos\_P to the next position of the current tail, before the given element is appended into the queue. As a consequence, CurrentPos\_P of the queue is increased by one. When the given queue has already been full or does not exist, Enqueue\_PC generates a specific error message and sets QueueID\_ST.Enqueue\_BL = F.

b) The Process of Serve Queue

The serve process of Queue\_ST, Serve\_PC, is formally modeled as shown in Figure 18, which obtains the element in the front of the queue. Its outputs are the front element of the queue and the status of the serve operation. The serve operation may be carried out when the given queue is not empty. Once the front element is elicited, Serve\_PC has to update the queue by shifting all other elements that follow the currently removed front element by one place towards the front of the queue. As a consequence, CurrentPos\_P of the queue is decreased by one. When the given queue has already been empty or does not exist, Serve\_PC generates a specific error message and sets QueueID\_ST.Served\_BL = F.

Contrasting the behavioral models of Queue\_ST.Behaviors\_PC in RTPA as developed in this section and that of predicate logic as shown in Figure 1, advances of the RTPA method and notations are well demonstrated. Among them, the most important merit is that an ADT model in RTPA can be seamlessly transformed into code in any programming language when the formal models are designed and refined systematically.

The Dynamic Behavior Model of Queues

On the basis of the work products, Queue\_ST. StaticBehaviors\_PC, developed in the preceding subsection as a set of static behavioral processes, this subsection describes the dynamic behaviors of Queue\_ST at run-time using the RTPA process dispatching methodology. Queue\_ST. DynamicBehaviors\_PC as shown in Figure 19 models the event-driven behaviors of Queue\_ST, which establishes the relations between system events and the stack behavioral processes. The event-driven dispatching mechanisms also put Queue\_ST into the context of applications.

Figures 15 through 19 describe a typical ADT model, Queue\_ST, in a coherent design and using a unified formal notation. With the RTPA specification and refinement methodology, the mechanisms, architectures, and behaviors of Queue\_ST are rigorously and precisely modeled.

THE FORMAL MODEL OF ADT3 – SEQUENCES

A sequence is a typical data structure for modeling an ADT with a series of elements in the same type where their order information is important. The conceptual model of sequences and its key control variables are introduced in this section. Based on it, formal models of the sequence ADT in terms of its architectural and static/dynamic behavioral models are rigorously developed in RTPA.
The Conceptual Model of Sequences

The sequence as a common data structure is described as shown in Figure 20. The protocol of sequence is FIFO, which implies that the I/O operations of sequence must be at its tail and head, respectively. In the sequence model, SequenceST, each ElementRT share the same type RT. Two of the key control variables of the sequence are SizeN denoting the maximum capacity of the sequence and CurrentPosP denoting the current tail position of the sequence. The addresses of elements grow from the head to the tail with the relative base address reserved at 0H. The head or the front element of the sequence, ElementRT, is always located at address 1H based on the relative base address of the sequence BaseN.

The top-level ADT model of sequence, SequenceST, encompassing its architecture, static behaviors, and dynamic behaviors, can be modeled in RTPA as follows:
ADT§.SequenceST ⊆ SequenceST.

ArchitectureST
|| SequenceST. StaticBehaviorsPC
|| SequenceST. DynamicBehaviorsPC

(7)

According to the RTPA methodology for system modeling, specification, and refinement (Wang, 2007, 2008a), the following subsections will refine the top level framework of SequenceST into detailed architectural models (UDMs) and behavioral models (UPMs).

The Architectural Model of Sequences

The architecture of SequenceST can be rigorously modeled using the UDM technology of RTPA. The UDM model of the sequence ADT, SequenceST. ArchitectureST, as shown in Figure 21 provides a generic architectural model for any concrete sequence in applications with three key fields, i.e., ElementRT, SizeN, and CurrentPositionP, where the constraints of each field are given in the right-hand side of the vertical bar. Supplement to the key architectural attributes, there is a set of status fields in the sequence, which models the current operational status of the sequence in Boolean type such as CreatedBL, AppendedBL, FirstFoundBL, LastFoundBL, RetrievedBL, ClearedBL, EmptyBL, FullBL, and ReleasedBL.

The Behavioral Model of Sequences

On the basis of the UDM model of SequenceST developed in the preceding subsection, the behaviors of the sequence can be modeled as a set of UPMs operating on the UDMs and related input variables. The high level behavioral model of the sequence ADT is modeled by SequenceST. StaticBehaviorsPC as shown in Eq. 8. The schemas of the sequence can be further refined by a set of UPMs for each of the behavioral processes.

A set of eight behavioral processes such as create, append, find element, retrieve, clear, empty test, full test, and release is designed in SequenceST. StaticBehaviorsPC. The following subsections describe how the static behaviors of SequenceST as specified in Eq. 8 are modeled and refined using the denotational mathematical notations and methodologies of RTPA. Because of the similarity of ADT manipulation processes between SequenceST and StackST as described in Eq. 4, only key processes, AppendPC, FindElementPC, and RetrievePC, of SequenceST will be formally modeled in the following subsections.

SequenceST. StaticBehaviorsPC ⊆
( CreateSequencePC(<I: SequenceIDS, SizeN, ElementRT>;
<O: SequenceIDST.CrattedBL>; <UDM:: SequenceIDST>)
| AppendPC(<I: SequenceIDS, ElementRT>; <O: SequenceIDST.AppendedBL>; <UDM:: SequenceIDST>)
| FindElementPC(<I: SequenceIDS, ElementRT>; <O: SequenceIDST.ElementFoundBL, ElementPositionP>; <UDM:: SequenceIDST>)
| RetrievePC(<I: SequenceIDS, iN>; <O: SequenceIDST.RetrievedBL, ElementRT>; <UDM:: SequenceIDST>)
| ClearPC(<I: SequenceIDS>; <O: SequenceIDST.ClearedBL>; <UDM:: SequenceIDST>)
| EmptyTest (<I: SequenceIDS>; <O: SequenceIDST.EmptyBL>; <UDM:: SequenceIDST>)
| FullTest (<I: SequenceIDS>; <O: SequenceIDST.FullBL]; <UDM:: SequenceIDST>)
| ReleaseSequencePC(<I: SequenceIDS>; <O: SequenceIDST.ReleasedBL>; < :: SequenceIDST>) )

(8)

a) The Process of Sequence Append

The append process of SequenceST, AppendPC, is formally modeled as shown in Figure 22,
Figure 18. The UPM model of the serve queue process

\[
\text{Serve}_{PC}(\langle I: \text{QueueID}_{S}; < O: \text{QueueID}_{ST}.\text{Served}_{BL}, \text{Element}_{RT}; < \text{UDM}: \text{QueueID}_{ST} \rangle) \triangleq \\
\quad \rightarrow ( \checkmark \ \text{QueueID}_{ST}.\text{Created}_{BL} = \text{T} \\
\quad \quad \rightarrow ( \checkmark \ \text{QueueID}_{ST}.\text{CurrentPos}_{P} > 0 \\
\quad \quad \quad \rightarrow \text{QueueID}(1)_{ST}.\text{Element}_{RT} > \text{Element}_{RT} \\
\quad \quad \quad \quad \rightarrow \mathbb{R}_{n=2} \text{QueueID}((n)_{ST}.\text{Element}_{RT} > \text{QueueID}((n-1)_{ST}.\text{Element}_{RT}} \\
\quad \quad \quad \rightarrow \downarrow (\text{QueueID}_{ST}.\text{CurrentPos}_{P}) \\
\quad \quad \quad \rightarrow \text{QueueID}_{ST}.\text{Served}_{BL} := \text{T} \\
\quad \quad \quad \rightarrow \checkmark \quad (\text{QueueID}_{ST} \text{is empty.}) \\
\quad \quad \rightarrow \text{QueueID}_{ST}.\text{Served}_{BL} := \text{F} \\
\quad \rightarrow \text{QueueID}_{ST}.\text{Empty}_{BL} := \text{T} \\
\quad \rightarrow \downarrow (\text{QueueID}_{ST} \text{does not exist.}) \\
\quad ) \\
\]

Figure 19. The dynamic behavioral model of the queue ADT

\[
\text{Queue}_{ST}.\text{DynamicBehaviors}_{PC} \triangleq \\
\quad \rightarrow ( \quad @\text{CreateQueue}_{S} \leftrightarrow \text{CreateQueue}_{PC}(\langle I: \text{QueueID}_{S}; \text{Size}_{N}, \text{Element}_{RT}; < O: \text{QueueID}_{ST}.\text{Created}_{BL}; \text{UDM}: \text{QueueID}_{ST} \rangle) \\
\quad \quad @\text{Enqueue}_{S} \leftrightarrow \text{Enqueue}_{PC}(\langle I: \text{QueueID}_{S}, \text{Element}_{RT}; < O: \text{QueueID}_{ST}.\text{Enqueued}_{BL}; \text{UDM}: \text{QueueID}_{ST} \rangle) \\
\quad \quad @\text{Serve}_{S} \leftrightarrow \text{Serve}_{PC}(\langle I: \text{QueueID}_{S}; < O: \text{QueueID}_{ST}.\text{Served}_{BL}, \text{Element}_{RT}; \text{UDM}: \text{QueueID}_{ST} \rangle) \\
\quad \quad @\text{EmptyTest}_{S} \leftrightarrow \text{EmptyTest}_{PC}(\langle I: \text{QueueID}_{S}; < O: \text{QueueID}_{ST}.\text{Empty}_{BL}; \text{UDM}: \text{QueueID}_{ST} \rangle) \\
\quad \quad @\text{FullTest}_{S} \leftrightarrow \text{FullTest}_{PC}(\langle I: \text{QueueID}_{S}; < O: \text{QueueID}_{ST}.\text{Full}_{BL}; \text{UDM}: \text{QueueID}_{ST} \rangle) \\
\quad \quad @\text{ReleaseQueue}_{S} \leftrightarrow \text{ReleaseQueue}_{PC}(\langle I: \text{QueueID}_{S}; < O: \text{QueueID}_{ST}.\text{Released}_{BL}; \text{UDM}: \text{QueueID}_{ST} \rangle) \\
\quad ) \rightarrow \$ \\
\]

Figure 20. The conceptual model of sequences
which puts an element at the current end of the sequence. The input arguments of the process are the target sequence ID and the given element. Its output is the status of the append operation. AppendPC writes the given element at the end of the sequence after checking that the sequence is not full and CurrentPosP is shifted to the address of the next element of the current tail. When the given sequence has already been full or does not exist, AppendPC generates a specific error message and sets SequenceIDST.AppendedBL = F.

b) The Process of Find Element in a Sequence

The find element process of SequenceST, FindElementPC, is formally modeled as shown in Figure 23, which finds out the position of a given element in the sequence. The input arguments of the process are the target sequence ID and the given element. Its outputs are the position of the element in the sequence, if found, and the status of the search operation. FindElementPC is a read-only operation that reports the position of the given element in a sequence, but
does not change the structure and contents of the sequence. When the given sequence does not exist or is empty, FindElement<PC> generates a specific error message and sets SequenceID<ST>.ElementFound<BL> = F.

c) The Process of Sequence Retrieve

The retrieve process of Sequence<ST>, Retrieve<PC>, is formally modeled as shown in Figure 24, which reads the rth element in the sequence. The input arguments of the process are the target sequence ID and a given position in the sequence. Its outputs are the element at the given position, if any, and the status of the retrieve operation. Retrieve<PC> is a read-only operation to find out the content at a certain position of the sequence, after conforming that the sequence is not empty. When the given sequence is empty or does not exist, Retrieve<PC> generates a specific error message and sets SequenceID<ST>.Retrieved<BL> = F.

The Dynamic Behavior Model of Sequences

On the basis of the work products, Sequence<ST>.StaticBehaviors<PC>, developed in the preceding subsection as a set of static behavioral processes, this subsection describes the dynamic behaviors of Sequence<ST> at run-time using the RTPA process dispatching methodology. Sequence<ST>.DynamicBehaviors<PC> as shown in Figure 25 models the event-driven behaviors of Sequence<ST>, which establishes the relations between system events and the stack behavioral processes. The event-driven dispatching mechanisms also put Sequence<ST> into the context of applications.

Figures 20 through 25 describe a typical ADT model, Sequence<ST>, in a coherent design and using a unified formal notation. With the RTPA specification and refinement methodology, the mechanisms, architectures, and behaviors of Sequence<ST> are rigorously and precisely modeled.

Figure 23. The UPM model of the find element process

FindElement<PC>(<I:: SequenceID<ST>, Element<RT>>; <O:: SequenceID<ST>.ElementFound<BL>, ElementPosition<P>; <UDM:: SequenceID<ST>>) A
{  
    → ( v SequenceID<ST>.Created<BL> = T ∧ SequenceID<ST>.CurrentPos<P ≠ 0
        ∧ SizeOfSequence<IN> )
        → R
        i<IN> = 1
        ( v SequenceID<ST>.Element(<IN>)<RT> = Element<RT>
            → ElementPosition<P := i<IN>
            → SequenceID<ST>.ElementFound<BL> := T
            → ⊙
            | v~
                → SequenceID<ST>.ElementFound<BL> := F
        )
        | v~
        → SequenceID<ST>.ElementFound<BL> := F
        → !(“‘SequenceID<ST>‘ does not exist or is empty.”)
    )
}
THE FORMAL MODEL OF ADT4 – RECORDS

A record is a typical data structure for modeling an ADT with a set of fields configured in different types. The conceptual model of record and its key control variables are introduced in this section. Based on it, formal models of the record ADT in terms of its architectural and static/dynamic behavioral models are rigorously developed in RTPA. It is recognized that record is one of the most powerful and widely used ADTs and data structures, because it allows flexible field structures and data types. The mathematical model of a record is a tuple, which forms a fundamental architectural modeling means in ADT and software system modeling.

The Conceptual Model of Records

The record as a common data structure is described as shown in Figure 26. The protocol of record is the direct accessibility to its fields, which implies that the I/O operations of records may be directly (randomly) conducted in any field of the record. In the record model, RecordST, the element in each field, Field(i)RT, is allowed in different types. Two of the key control variables of the record ADT are MaxFieldsN and #FieldsN. The former denotes the maximum allowable fields of the record; and the latter denotes the current position of the last field in the record. The addresses and the order of all fields are parallel and arbitrary. The constraints on each field of RecordST may be explicitly specified in the architectural model of the record.

The top-level ADT model of the record, RecordST, encompassing its architecture, static behaviors, and dynamic behaviors, can be modeled in RTPA as follows:

\[
\text{ADT} \triangleq \text{RecordST} \triangleq \text{RecordST}.\text{ArchitectureST} \\
\quad \quad \quad \equiv \text{RecordST}.\text{StaticBehaviorsPC} \\
\quad \quad \quad \equiv \text{RecordST}.\text{DynamicBehaviorsPC}
\]

(9)

According to the RTPA methodology for system modeling, specification, and refinement (Wang, 2007, 2008a), the following subsections will refine the top level framework of RecordST into detailed architectural models (UDMs) and behavioral models (UPMs).

The Architectural Model of Records

The architecture of RecordST can be rigorously modeled using the UDM technology of RTPA. The UDM model of the record ADT, RecordST. ArchitectureST, as shown in Figure 27 provides a generic architectural model for any concrete record in applications with two key parameters, i.e., #FieldsN and FiledType(i)RT, where the constraints on each field are given in the right-hand side of the vertical bar. Supplement to the key architectural attributes, there is a set of status fields in the record, which models the current operational status of the record in Boolean type such as CreatedBL, InitializedBL, FieldRetrievedBL, RecordRetrievedBL, FieldUpdatedBL, RecordUpdatedBL, ClearedBL, EmptyBL, FullBL, and ReleasedBL.

The Behavioral Model of Records

On the basis of the UDM model of RecordST developed in the preceding subsection, the behaviors of the record can be modeled as a set of UPMs operating on the UDMs and related input variables. The high level behavioral model of the record ADT is modeled by RecordST. StaticBehaviorsPC as shown in Eq. 10. The schemas of the record can be further refined by a set of UPMs for each of the behavioral processes.

RecordST. StaticBehaviorsPC \triangleq
\begin{align*}
\{ & \text{CreateRecordPC}(<I::\text{RecordIDS}, \\
& #\text{FieldsN}, \sum_{i=1}^{#\text{FieldsN}} \text{Field}(i)RT>; \\
& <O::\text{RecordIDST}.\text{CreatedBL}>; <\text{UDM}::\text{RecordST}>)
\end{align*}
Figure 24. The UPM model of the sequence retrieve process

\[
\text{Retrieve}_\text{PC}(<\text{S}>, \text{in}_r >; <\text{O}>: \text{SequenceID}_\text{ST}.\text{Retrieved}_\text{BL}; <\text{UDM}>: \text{SequenceID}_\text{ST}) \triangleq \\
\{ \\
\quad \rightarrow ( \\
\quad \quad \Rightarrow \text{SequenceID}_\text{ST}.\text{Created}_\text{BL} = \text{T} \land \text{SequenceID}_\text{ST}.\text{CurrentPosP} \neq 0 \\
\quad \quad \quad \rightarrow ( \\
\quad \quad \quad \quad \Rightarrow 0 < \text{iN} \leq \text{SequenceID}_\text{ST}.\text{CurrentPosP} \\
\quad \quad \quad \quad \quad \rightarrow \text{SequenceID}_\text{ST}.\text{Element}(\text{iN})_\text{RT} \neq \text{Element}_\text{RT} \\
\quad \quad \quad \quad \quad \rightarrow \text{SequenceID}_\text{ST}.\text{Retrieved}_\text{BL} := \text{T} \\
\quad \quad \quad \quad \quad \mid \Rightarrow \\
\quad \quad \quad \quad \quad \rightarrow \text{SequenceID}_\text{ST}.\text{Retrieved}_\text{BL} := \text{F} \\
\quad \quad \quad \quad \quad \mid \Rightarrow \ (“\text{SequenceID}_\text{ST}’ \text{ does not exist or is empty. “}) \\
\quad \quad \mid \Rightarrow \\
\quad \quad \rightarrow \text{SequenceID}_\text{ST}.\text{Retrieved}_\text{BL} := \text{F} \\
\quad \quad \rightarrow \ (“\text{SequenceID}_\text{ST}’ \text{ does not exist or is empty. “}) \\
\} 
\]

Figure 25. The dynamic behavioral model of the sequence ADT

\[
\text{Sequence}_\text{ST}.\text{DynamicBehaviors}_\text{PC} \triangleq \\
\{ \text{§} \rightarrow \}
\]

\[
| \Rightarrow \text{CreateSequence}_\text{PC} <\text{S}>, \text{Size}_\text{N}, \text{Element}_\text{RT}>; <\text{O}>: \text{SequenceID}_\text{ST}.\text{Created}_\text{BL}; <\text{UDM}>: \text{SequenceID}_\text{ST}) \\
| \Rightarrow \text{Append}_\text{PC} <\text{S}>, \text{Element}_\text{RT}; <\text{O}>: \text{SequenceID}_\text{ST}.\text{Appended}_\text{BL}; <\text{UDM}>: \text{SequenceID}_\text{ST} \\
| \Rightarrow \text{FindElement}_\text{PC} <\text{S}>, \text{Element}_\text{RT}; <\text{O}>: \text{SequenceID}_\text{ST}.\text{ElementFound}_\text{BL}, \text{ElementPositionP}; <\text{UDM}>: \text{SequenceID}_\text{ST} \\
| \Rightarrow \text{Retrieve}_\text{PC} <\text{S}>, \text{Retrieved}_\text{BL}, \text{Element}_\text{RT}; <\text{O}>: \text{SequenceID}_\text{ST}.\text{Retrieved}_\text{BL}; <\text{UDM}>: \text{SequenceID}_\text{ST} \\
| \Rightarrow \text{Clear}_\text{PC} <\text{S}>, <\text{O}>: \text{SequenceID}_\text{ST}.\text{Cleared}_\text{BL}; <\text{UDM}>: \text{SequenceID}_\text{ST} \\
| \Rightarrow \text{EmptyTests}_\text{PC} <\text{S}>, <\text{O}>: \text{SequenceID}_\text{ST}.\text{Empty}_\text{BL}; <\text{UDM}>: \text{SequenceID}_\text{ST} \\
| \Rightarrow \text{FullTests}_\text{PC} <\text{S}>, <\text{O}>: \text{SequenceID}_\text{ST}.\text{Full}_\text{BL}; <\text{UDM}>: \text{SequenceID}_\text{ST} \\
| \Rightarrow \text{ReleaseSequence}_\text{PC} <\text{S}>, <\text{O}>: \text{SequenceID}_\text{ST}.\text{Released}_\text{BL}; <\text{UDM}>: \text{SequenceID}_\text{ST} \\
\}
\]

Figure 26. The conceptual model of records
**Figure 27. The UDM model of the record ADT**

\[
\text{RecordST. ArchitectureST} \triangleq \text{RecordIDST} :: \\
\langle \# \text{Fields} : N \mid 1 \leq \# \text{FieldsN} \leq \text{MaxFieldsN} \rangle, \\
\# \text{FieldsN} - 1 \\
R_{\text{N}=0} <\text{FieldType}(iN) : \text{RT} \mid \text{FieldType}(iN)RT \in \{S, N, R, ST, \ldots\}> \\
\langle \text{Created} : BL \mid \text{CreatedBL} = \{(T, Yes), (F, No)} \rangle, \\
\langle \text{Initialized} : BL \mid \text{InitializedBL} = \{(T, Yes), (F, No)} \rangle, \\
\langle \text{FieldRetrieved} : BL \mid \text{FieldRetrievedBL} = \{(T, Yes), (F, No)} \rangle, \\
\langle \text{RecordRetrieved} : BL \mid \text{RecordRetrievedBL} = \{(T, Yes), (F, No)} \rangle, \\
\langle \text{FieldUpdated} : BL \mid \text{FieldUpdatedBL} = \{(T, Yes), (F, No)} \rangle, \\
\langle \text{RecordUpdated} : BL \mid \text{RecordUpdatedBL} = \{(T, Yes), (F, No)} \rangle, \\
\langle \text{Released} : BL \mid \text{ReleasedBL} = \{(T, Yes), (F, No)} \rangle \\
\rangle
\]

**Figure 28. The UPM model of the record initialization process**

\[
\text{InitializeRecordPC}(<\text{I} : \text{RecordIDST}, \text{N} ; \text{O} : \text{RecordIDST} \cdot \text{InitializeBL} ; \text{<UDM} : \text{RecordST}>) \triangleq \\
\{} \\
\rightarrow ( \vee \text{RecordIDST} \cdot \text{Created} \cdot \text{CreatedBL} = T \land \text{RecordIDST} \cdot \# \text{FieldsN} \neq 0 \\
\# \text{FieldsN} - 1 \\
R_{\text{N}=0} \text{RecordIDST} \cdot \text{Field}(iN)RT := \text{InitialValueRT} \\
\rightarrow \text{RecordIDST} \cdot \text{InitializedBL} := T \\
| \neg \\
\rightarrow \text{RecordIDST} \cdot \text{InitializedBL} := F \\
\rightarrow ! ("\text{RecordIDST} does not exist or is empty."))
\}
\]

**Figure 29. The UPM model of the retrieve field process**

\[
\text{RetrieveFieldPC}(<\text{I} : \text{RecordIDST}, \text{N} ; \text{O} : \text{RecordIDST} \cdot \text{FieldRetrievedBL} ; \text{Field}(iN)RT ; \text{<UDM} : \text{RecordST}) \triangleq \\
\{} \\
\rightarrow ( \star \text{RecordIDST} \cdot \text{Created} \cdot \text{CreatedBL} = T \land \text{RecordIDST} \cdot \# \text{FieldsN} \neq 0 \\
\rightarrow ( ( \star 0 \leq iN \leq \text{RecordIDST} \cdot \# \text{FieldsN} \\
\rightarrow \text{RecordIDST} \cdot \text{Field}(iN)RT > \text{Field}(iN)RT \\
\rightarrow \text{RecordIDST} \cdot \text{FieldRetrievedBL} := T \\
| \neg \\
\rightarrow \text{RecordIDST} \cdot \text{RetrievedBL} := F \\
\rightarrow ! ("\text{Field index is out of range."})
\}) \\
| \neg \\
\rightarrow \text{RecordIDST} \cdot \text{RetrievedBL} := F \\
\rightarrow ! (" \text{RecordIDST} does not exist or is empty. ")
\}
\]
Figure 30. The UPM model of the record retrieve process

![Figure 30. The UPM model of the record retrieve process](image)

Figure 31. The UPM model of the record field update process

```
UpdateFieldPG(<l:: RecordIDST>, Field(iN)RT); <o:: RecordIDST, FieldUpdatedBL>; <UDM:: RecordST>) =
{
    ( RecordIDST.CreatedBL = T \& RecordIDST.##FieldsN = 0
        → ( 0 ≤ iN ≤ RecordIDST.##FieldsN
            → RecordIDST.Field(iN)RT := Field(iN)RT
            → RecordIDST.FieldUpdatedBL := T
            )
        )
        → RecordIDST.CreatedBL := F
        → ! ("Field index is out of range.")
    )
    → RecordIDST.CreatedBL := F
    → ! ("RecordIDST does not exist or is empty.")
}
```

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
The input arguments of the process are the Target record ID and the given initial value. Its output is the status of the initialization operation. InitializeRecordPC puts the initial value into each element of the record, after checking that the record exists and is not empty. When the given record is empty or does not exist, InitializeRecordPC generates a specific error message and sets RecordIDST. InitializedBL = F.

b) The Process of Record Field Retrieve

The record field retrieve process of RecordST, RetrieveFieldPC, is formally modeled as shown in Figure 29, which reads the contents of a given field of the record. The input arguments of the process are the target record ID and the number of the expected field. Its outputs are the contents of the given field in the record and the status of the retrieve operation. RetrieveFieldPC is a read-only operation to report the contents of a specific field in the record. Before the retrieve is conducted, the given number of the field must be validated and the record must not be empty. When the given record is empty or does not exist, RetrieveFieldPC generates a specific error message and sets RecordIDST. FieldRetrievedBL = F.

c) The Process of Record Retrieve

The record retrieve process of RecordST, RetrieveRecordPC, is formally modeled as shown in Figure 30, which reads the contents of all fields from the given record. The input argument of the process is the target record ID. Its outputs are the contents of each field of the record and the status of the retrieve operation. RetrieveRecordPC is a read-only operation to repetitively report the contents of all fields in the record. When the given record is empty or does not exist, RetrieveRecordPC generates a specific error message and sets RecordIDST. RecordRetrievedBL = F.

d) The Process of Update Record Field

The update record field process of RecordST, UpdateFieldPC, is formally modeled as shown in
Figure 31, which writes a given value onto the specific field of the record. The input arguments of the process are the target record ID and the data for the given field. Its output is the status of the field update operation. UpdateField\textsubscript{PC} is an inverse operation of RetrieveField\textsubscript{PC}. Before the operation is carried out, the number of the field must be validated and the record must be nonempty. When the given record is empty or does not exist, UpdateField\textsubscript{PC} generates a specific error message and sets StackID\textsubscript{ST}. FieldUpdated\textsubscript{BL} = F; so also when the field index is out of scope.

e) The Process of Record Update

The update record process of Record\textsubscript{ST}, UpdateRecord\textsubscript{PC}, is formally modeled as shown in Figure 32, which writes a set of values onto each field of the record. The input arguments of the process are the target record ID, number of fields, and the data for each given field. Its output is the status of the record update operation. UpdateRecord\textsubscript{PC} is an inverse operation of RetrieveRecord\textsubscript{PC}. Before the operation is carried out, the number of the filed must match with that of the record. When the given record is empty or does not exist, UpdateRecord\textsubscript{PC} generates a specific error message and sets StackID\textsubscript{ST}.RecordUpdated\textsubscript{BL} = F; so also when the total number of fields does not match that of the record.

The Dynamic Behavior Model of Record

On the basis of the work products, Record\textsubscript{ST}. StaticBehaviors\textsubscript{PC}, developed in the preceding subsection as a set of static behavioral processes, this subsection describes the dynamic behaviors of RecordRecord\textsubscript{ST} at run-time using the RTPA process dispatching methodology. Record\textsubscript{ST}. DynamicBehaviors\textsubscript{PC} as shown in Figure 33 models the event-driven behaviors of Record\textsubscript{ST}, which establishes the relations between system events and the stack behavioral processes. The event-driven dispatching mechanisms also put Record\textsubscript{ST} into the context of applications.

Figures 26 through 33 describe a typical ADT model, Record\textsubscript{ST}, in a coherent design and using a unified formal notation. With the RTPA specification and refinement methodology, the mechanisms, architectures, and behaviors of Record\textsubscript{ST} are rigorously and precisely modeled. The practical formal engineering methodology of RTPA for system modeling and specification provides a coherent notation system and systematic approach for large-scale software and hybrid system design and implementation. A series of formal design models of real-world and real-time applications in RTPA have been developed using RTPA notations and methodologies (Wang, 2002, 2007, 2008a, 2008b, 2009b; Wang & Huang, 2008) 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 et al., 2010b), the Real-Time Operating System (RTOS+) (Wang et al., 2010c, 2010d), and the Air Traffic Control System (ATCS, to be reported). Further studies have demonstrated that RTPA is not only useful as a generic notation and methodology for software engineering, but also good at modeling human cognitive processes in cognitive informatics and computational intelligence as reported in (Wang, 2008d, 2009a; Wang & Ruhe, 2007; Wang & Chiew, 2010).

CONCLUSION

Abstract Data Types (ADTs) have been recognized as an important set of rigorously modelled complex data structures with pre-specified behaviors. This paper has introduced the formal RTPA modeling methodology for ADT architectures and behavioral specification and refinement in a top-down approach. The architectures, static behaviors, and dynamic behaviors of a set of typical ADTs such as stack, queue, sequence, and record, have been comparatively studied. Two generic methodologies of RTPA known as the Unified Data Models (UDMs) for system architectural modeling and the Unified Process
Figure 32. The UPM model of the record update process

\[
\text{UpdateRecord} \overset{\text{PC}}{\rightarrow} \text{RecordID} \text{S}, \#\text{Fields}_{N}, \begin{array}{c}
\text{R} \\
\#N-1
\end{array} \text{FieldID}(i)R; \\
\begin{array}{c}
\text{O} \\
\#N-1
\end{array} \text{RecordID}ST.\text{RecordUpdatedBL} >; < \text{UDM} :: \text{RecordST} > \Rightarrow
\]

\{

\begin{array}{c}
\text{RecordID}ST.\text{CreatedBL} = T \land \text{RecordID}ST.\#\text{Fields}_{N} \neq 0 \\
\end{array}

\rightarrow \begin{array}{c}
\text{RecordID}ST.\#\text{Fields}_{N} = \text{RecordID}ST.\#\text{Field}_{N} \\
\#
\end{array}

\begin{array}{c}
\text{R} \\
\#N-1
\end{array} \\
\text{RecordID}ST.\text{Field}(i)RT := \text{Field}(i)RT \\
\text{RecordID}ST.\text{RecordUpdatedBL} := T
\}

\begin{array}{c}
\text{RecordID}ST. \text{RecordUpdatedBL} := F \\
\end{array}

\rightarrow \begin{array}{c}
\text{Input field number does not match ‘RecordIDST’.”} \\
\end{array}

\}

\begin{array}{c}
\text{RecordID}ST. \text{RecordUpdatedBL} := F \\
\end{array}

\rightarrow \begin{array}{c}
\text{‘RecordIDST’ does not exist or is empty.”} \\
\end{array}

\}

Figure 33. The dynamic behavioral model of the record ADT

\[
\text{RecordST. DynamicBehaviors} \overset{\text{PC}}{\rightarrow} \text{RecordID} \text{S}, \#\text{Fields}_{N}, \begin{array}{c}
\text{R} \\
\#N-1
\end{array} \text{FieldID}(i)R; \\
\begin{array}{c}
\text{O} \\
\#N-1
\end{array} \text{RecordID}ST.\text{RecordUpdatedBL} >; < \text{UDM} :: \text{RecordST} > \Rightarrow
\]

\{

\begin{array}{c}
\text{CreateRecord} \overset{\text{PC}}{\rightarrow} \text{RecordID} \text{S}, \#\text{Fields}_{N}, \begin{array}{c}
\text{R} \\
\#N-1
\end{array} \text{FieldID}(i)R; \\
\begin{array}{c}
\text{O} \\
\#N-1
\end{array} \text{RecordID}ST.\text{RecordUpdatedBL} >; < \text{UDM} :: \text{RecordST} > \Rightarrow
\}

\begin{array}{c}
\text{RecordID}ST.\text{RecordUpdatedBL} >; < \text{UDM} :: \text{RecordST} > \Rightarrow
\}

\begin{array}{c}
\text{UpdateField} \overset{\text{PC}}{\rightarrow} \text{RecordID} \text{S}, \#\text{Fields}_{N}, \begin{array}{c}
\text{R} \\
\#N-1
\end{array} \text{FieldID}(i)R; \\
\begin{array}{c}
\text{O} \\
\#N-1
\end{array} \text{RecordID}ST.\text{FieldUpdatedBL} >; < \text{UDM} :: \text{RecordST} > \Rightarrow
\}

\begin{array}{c}
\text{RecordID}ST.\text{FieldUpdatedBL} >; < \text{UDM} :: \text{RecordST} > \Rightarrow
\}

\begin{array}{c}
\text{ReleaseQueue} \overset{\text{PC}}{\rightarrow} \text{RecordID} \text{S}, \#\text{Fields}_{N}, \begin{array}{c}
\text{R} \\
\#N-1
\end{array} \text{FieldID}(i)R; \\
\begin{array}{c}
\text{O} \\
\#N-1
\end{array} \text{RecordID}ST.\text{ReleasedBL} >; < \text{UDM} :: \text{RecordST} > \Rightarrow
\}

\begin{array}{c}
\text{RecordID}ST.\text{ReleasedBL} >; < \text{UDM} :: \text{RecordST} > \Rightarrow
\}

\}

\}
Models (UPMs) for behavioral modeling have been elaborated. On the basis of the formal and rigorous models of the ADT system, code can be automatically generated or be manually transferred from the formal models.

The RTPA models of more complex ADTs such as universal arrays, lists, binary trees, files, and digraphs will be reported in related future work in the Series of Formal Software Design Models, Patterns, and Frameworks in IJSSCI. This work have been applied in a number of real-time and nonreal-time system designs and modeling such as a Real-Time Operating System (RTOS+), an Air Traffic Control System (ATCS), as well as the developments of the ADT library of an RTPA support tool and the anonymous automatic code generator (RTPA-CG) (Wang et al., 2010a; Ngolah & Wang, 2009) based on RTPA.

ACKNOWLEDGMENT

The authors would like to acknowledge both the Natural Science and Engineering Council of Canada (NSERC) and the International Institute for Software Technology at the United Nations University (IIST/UNU) for their partial support to this work. We would like to thank the reviewers for their valuable comments and suggestions.

REFERENCES


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

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.

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

Phillip C-Y. Sheu is currently a professor of Computer Engineering, Information and Computer Science, and Biomedical Engineering at the University of California, Irvine. He received his Ph.D. and M.S. degrees from the University of California at Berkeley in Electrical Engineering and Computer Science in 1986 and 1982, respectively, and his B.S. degree from National Taiwan University in Electrical Engineering in 1978. Between 1982 and 1986, he also worked as a computer scientist at Systems Control Technology, Inc., Palo Alto, CA., where he designed and implemented aircraft 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.