Design and Implementation of an Autonomic Code Generator Based on RTPA

Yingxu Wang, University of Calgary, Canada
Xinming Tan, Wuhan University of Technology, China
Cyprian F. Ngolah, Sentinel Trending & Diagnostics Ltd., Calgary, Canada

ABSTRACT

Real-Time Process Algebra (RTPA) is a denotational mathematics for the algebraic modeling and manipulations of software system architectures and behaviors by the Unified Data Models (UDMs) and Unified Process Models (UPMs). On the basis of the RTPA specification and refinement methodologies, automatic software code generation is enabled toward improving software development productivity. This paper examines designing and developing the RTPA-based software code generator (RTPA-CG) that transfers system models in RTPA architectures and behaviors into C++ or Java. A two-phrase strategy has been employed in the design of the code generator. The first phrase analyzes the lexical, syntactical, and type specifications of a software system modeled in RTPA, which results in a set of abstract syntax trees (ASTs). The second phrase translates the ASTs into C++ or Java based on predesigned mapping strategies and code generation rules. The toolkit of RTPA code generator encompasses an RTPA lexer, parser, type-checker, and a code builder. Experimental results show that system models in RTPA can be rigorously processed and corresponding C++/Java code can be automatically generated using the toolkit. The code generated is executable and effective under the support of an RTPA run-time library.

Keywords: Denotational Mathematics, Design Frameworks, Formal Methods, Real-Time Process Algebra, System Modeling, Unified Data Models, Unified Process Models

INTRODUCTION

Automatic code generation on the basis of rigorous system models and formal specifications is one of the central objectives of software engineering, which leads to the improvement of software development productivity, efficiency, and quality (McDermid, 1991; Michael & Butler, 1996; Wang, 2007). However, automatic code generation is an intricate and difficult endeavor in software engineering. Many efforts were reported on simplified system specifications and simple target languages (RAISE, 1995; Univan & Chris, 2000). It is a great challenge to design and implement a comprehensive code generator that covers the entire range of
software modeling needs including real-time device drives and machine-level event/time/interrupt-driven mechanisms.

The process metaphor for software system modeling, specification, and implementation was initially proposed by Hoare and others (Hoare, 1978; Milner, 1980) that perceived a software system as the composition of a set of interacting processes.

Various algebraic approaches to describe the behaviors of communicating and concurrent systems were developed known as process algebra (Hoare, 1978, 1985; Milner, 1980), which provide a set of formal notations and rules for describing algebraic relations of software behaviors. A set of denotational mathematics was recently developed (Wang, 2008a) for rigorously modeling both architectures and behaviors of software systems. Denotational mathematics is a category of expressive mathematical structures that deal with high-level mathematical entities beyond numbers and sets, such as abstract objects, complex relations, perceptual information, abstract concepts, knowledge, intelligent behaviors, behavioral processes, and systems (Wang, 2009a). The paradigms of de-notational mathematics are such as concept algebra (Wang, 2008b), Real-Time Process Algebra (RTPA) (Wang, 2008c), and system algebra (Wang et al., 2008).

RTPA is a form of denotational mathematics for formally modeling and describing architectures and behaviors of software systems (Wang, 2002, 2008c, 2008d). Based on the RTPA methodology and models, software code may be seamlessly generated as shown in Fig. 1. In the scheme of RTPA-based code generation, the RTPA architectural model for a system is used to generate the structural framework and global/local variables of classes or objects; while the RTPA behavioral model is then transferred into object methods in a target programming language.

This paper develops a methodology and a tool that facilitate automatic C++ and Java code generation based on formal system models specified and refined in RTPA. First, RTPA is briefly introduced as a denotational mathematical methodology for software architecture and behavior modeling and manipulations. Second, the RTPA-based software code generator (RTPA-CG) is described on its design strategies, structures, and support environments. Third, the implementation of RTPA-CG is presented with its lexer, parser, type checker, and run-time library, as well as their integration. Last, applications of RTPA-CG are reported with case studies and experimental results.

Figure 1. Program code generation based on RTPA
RTPA: A DENOTATIONAL MATHEMATICS FOR SOFTWARE SYSTEM MODELING

On the basis of the process metaphor of software systems, abstract processes can be rigorously treated as a long chain of embedded relations (Wang, 2007) beyond sets and functions. RTPA is designed as a coherent algebraic system for software system modeling, specification, refinement, and implementation. RTPA encompasses a set of meta-processes and relational process operations, which can be used to describe both logical and physical models of software and intelligent systems. Logic views of system architectures and their physical platforms can be described using the same set of notations. When system architecture is formally modeled, the static and dynamic behaviors performed on the architectural model can be specified by a three-level refinement scheme at the system, class, and object levels in a top-down approach.

Definition 1. RTPA is a denotational mathematical structure for algebraically modeling and manipulating system behavioral processes and their data objects and architectures by a triple, i.e.:

\[
\text{RTPA} \triangleq (\mathcal{X}, \mathcal{P}, \mathcal{R})
\]

where

- \(\mathcal{X}\) is a set of 17 primitive types for modeling system architectures and data objects;
- \(\mathcal{P}\) a set of 17 meta-processes for modeling fundamental system behaviors;
- \(\mathcal{R}\) a set of 17 relational process operations for constructing complex system behaviors.

RTPA provides an algebraic notation system and mathematical rules for the specification and refinement of real-time/nonreal-time systems and safety-critical systems. Detailed descriptions of \(\mathcal{X}, \mathcal{P}, \) and \(\mathcal{R}\) in RTPA will be extended in the following subsections.

The Type System of RTPA

A type is a set in which all member data objects share a common logical property or attribute. The maximum range of values of variables in the set is the domain of a type. A type is always associated with a set of predefined or allowable operations in computing. A type can be classified as primitive and derived (complex) types. The former are the most elementary types that cannot be further divided into simpler ones; the latter are a compound form of multiple primitive types based on given type rules. In computing, 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 in computing. The set of 17 primitive types of RTPA, \(\mathcal{X}\), is elicited in computing and human cognitive process modeling, as summarized in Figure 2, from works in (Martin-Lof, 1975; Cardelli & Wegner, 1985; Stubbs & Webre, 1985; Mitchell, 1990; Wang, 2007). In Figure 2, the first 11 primitive types are for mathematical and logical manipulations of data objects, and the remaining 6 are for system architectural modeling.

Based on Figure 2, the set of primitive types of RTPA can be summarized as follows.

Definition 2. The RTPA type system \(\mathcal{X}\) encompasses 17 primitive types of computational objects, i.e.:

\[
\mathcal{X} \triangleq \{N, Z, R, S, B, H, P, TI, O, DT, BT, ST, \&e, \&T, \&i, \&O, \&BL\}
\]

where the primitive types stand for natural number, integer, real, string, Boolean, byte, hexadecimal, pointer, time, date, date/time, run-time determinable type, system architectural type, random event, time event, interrupt event, and system status.
It is noteworthy that although a generic computing behavior is constrained by the mathematical domain $D_m$ of types, an executable program is constrained by the language-defined domain $D_l$, and in most cases, it is further restricted by the user-defined domain $D_u$, i.e.:

$$D_u \subseteq D_l \subseteq D_m$$

### The Meta-Processes of RTPA

RTPA adopts foundationalism in order to elicit the most primitive computational processes known as the meta-processes. In this approach, complex processes are treated as derived processes from the meta-processes based on a set of algebraic rules for process compositions known as process relations. It is noteworthy that, although, CSP (Hoare, 1978, 1985), the Timed-CSP (Boucher & Gerth, 1987; Nicollin & Sifakis, 1991; Fecher, 2001), and other process algebra treat any computational operation as a process, RTPA distinguishes the concepts of meta-processes from those of complex and derived processes, which are composed by algebraic and relational process operations on the meta-processes.

**Definition 3.** A meta-process in RTPA is a primary computational operation that cannot be broken down to further individual actions or behaviors.

In RTPA, a set of 17 meta-processes has been elicited as shown in Figure 3, from essential and primary computational operations commonly identified in existing formal meth-
ods and modern programming languages (Higman, 1977; Aho et al., 1985; Hoare et al., 1987; Wilson & Clark, 1988; Louden, 1993; Woodcock & Davies, 1996; Lewis & Papadimitriou, 1998). Mathematical notations and syntaxes of the meta-processes are formally described in Table 2, while formal semantics of the meta-processes of RTPA may be referred to its deductive semantics (Wang, 2006, 2008e), denotational semantics (Tan & Wang, 2008), and operational semantics (Wang & Ngolah, 2008).

Definition 4. The RTPA meta-process system \( \mathcal{P} \) encompasses 17 fundamental computational operations, i.e.:

\[ \mathcal{P} = \{\text{\texttt{=, \texttt{\&}}, \texttt{\&\&}, \texttt{\|}, \texttt{\|\|}, \texttt{\neg}, \texttt{\|\|\|}, \texttt{\forall}, \texttt{\exists}, \texttt{\$}\} \]

(4)

where the meta-processes of RTPA stand for assignment, evaluation, addressing, memory allocation, memory release, read, write, input, output, timing, duration, increase, decrease, exception detection, skip, stop, and system, respectively.

As shown in Definition 4 and Figure 3, each meta-process is a basic operation on one or more operands such as variables, memory elements, or I/O ports. Structures of the operands and their allowable operations are constrained by their types as described in the preceding subsection. It is noteworthy that not all generally important and fundamental computational operations, as shown in Figure 3, had been explicitly identified in conventional formal methods such as the evaluation, addressing, memory allocation/release, timing/duration, and the system processes. However, all these are found necessary and essential in modeling system architectures and behaviors.

The Relational Process Operations of RTPA

Definition 5. A process relation in RTPA is an algebraic operation and a compositional rule between two or more meta-processes in order to construct a complex process.

A set of 17 process relations has been elicited from fundamental algebraic and relational operations in computing in order to build and compose complex processes. Syntaxes and usages of the 17 RTPA process relations are formally described in Figure 3. Formal semantics of RTPA process relations may be referred to (Wang, 2006, 2008e; Tan & Wang, 2008; Wang & Ngolah, 2008).

Definition 6. The RTPA process relations \( \mathcal{R} \) encompasses 17 fundamental algebraic and relational operations for composing complex processes in computing, i.e.:

\[ \mathcal{R} = \{\text{\texttt{-}}, \texttt{\&}, \texttt{\&\&}, \texttt{\|}, \texttt{\|\|}, \texttt{\neg}, \texttt{\|\|\|}, \texttt{\forall}, \texttt{\exists}, \texttt{\$}, \texttt{\texttt{\&}}, \texttt{\&\&}, \texttt{\|}, \texttt{\|\|}, \texttt{\texttt{\&}}, \texttt{\&\&}, \texttt{\|}, \texttt{\|\|}\} \]

(5)

where the relational operators of RTPA stand for sequence, jump, branch, while-loop, repeat-loop, for-loop, recursion, function call, parallel, concurrence, interleave, pipeline, interrupt, time-driven dispatch, event-driven dispatch, and interrupt-driven dispatch, respectively.

The RTPA Methodology for System Modeling and Refinement

RTPA provides both a coherent notation system and a formal engineering methodology for modeling both software and intelligent systems. In the RTPA methodology, the architecture and behaviors of any software system are modeled by the unified data model and unified process model, respectively.

Definition 7. 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 \( \tau \), i.e.
where $e_i$ is an arbitrary element of a given set $S_i$ that is constrained by the same property of $P_i$, $1 \leq i \leq n$, and $n$ is a natural number.

Any software system architecture, at the top level, can be specified as a list of UDMs and their relations. The counterpart computing model of tuples for UDMs is a record structure as follows:

$$UDM \triangleq \left\{ \forall e_i \in S_i, p_i(e_i) \right\}$$

(6)

The unified process model of RTPA can be rigorously described as follows.

**Definition 8.** A process $P$ is an embedded relation of a set of statements or processes of $n$ meta-statements $p_i$ and $p_j$, $1 \leq i < n$, $j = i+1$, according to certain composing relations $r_{ij}$, i.e.:

$$P = \bigcup_{i=1}^{n-1} (p_i \circ P_j)$$

(8)

where $r_{ij}$ is a set of relations or composition rules.

**Definition 9.** The unified process model (UPM) of a program $\wp$ is a composition of a finite set of $k$ processes according to the time-, event-, and interrupt-based process dispatching rules, i.e.:
Figure 4. RTPA Process Relations and Algebraic Operations

<table>
<thead>
<tr>
<th>No.</th>
<th>Process Relation</th>
<th>Notation</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Sequence</td>
<td>→</td>
<td>$P \rightarrow Q$</td>
</tr>
<tr>
<td>2</td>
<td>Jump</td>
<td>∩</td>
<td>$P \cap Q$</td>
</tr>
<tr>
<td>3</td>
<td>Branch</td>
<td></td>
<td>$\exp \text{BL} = T \rightarrow P$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>4</td>
<td>Switch</td>
<td>...</td>
<td>$\exp \text{T} =$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$i \rightarrow P_i$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>where $T \in {N, Z, B, S}$</td>
</tr>
<tr>
<td>5</td>
<td>While-loop</td>
<td>$R^*$</td>
<td>$f_{P}^{R^*}$</td>
</tr>
<tr>
<td>6</td>
<td>Repeat-loop</td>
<td>$R^+$</td>
<td>$P \rightarrow f_{P}^{R^+}$</td>
</tr>
<tr>
<td>7</td>
<td>For-loop</td>
<td>$R^i$</td>
<td>$\text{r}^{i}(f_{R}^{i})$</td>
</tr>
<tr>
<td>8</td>
<td>Recursion</td>
<td>∅</td>
<td>$0_{P^* \cup P^*}$</td>
</tr>
<tr>
<td>9</td>
<td>Function call</td>
<td>→</td>
<td>$P \rightarrow F$</td>
</tr>
<tr>
<td>10</td>
<td>Parallel</td>
<td>$</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Concurrence</td>
<td>$|$</td>
<td>$P | Q$</td>
</tr>
<tr>
<td>12</td>
<td>Interleave</td>
<td>$||$</td>
<td>$P | Q$</td>
</tr>
<tr>
<td>13</td>
<td>Pipeline</td>
<td>$\rightarrow$</td>
<td>$P \rightarrow Q$</td>
</tr>
<tr>
<td>14</td>
<td>Interrupt</td>
<td>$\uparrow$</td>
<td>$P \uparrow Q$</td>
</tr>
<tr>
<td>15</td>
<td>Time-driven dispatch</td>
<td>$\downarrow@$</td>
<td>$TM \downarrow P_i$</td>
</tr>
<tr>
<td>16</td>
<td>Event-driven dispatch</td>
<td>$\downarrow@$</td>
<td>$S \downarrow P_i$</td>
</tr>
<tr>
<td>17</td>
<td>Interrupt-driven dispatch</td>
<td>$\downarrow@$</td>
<td>$int \uparrow P_i$</td>
</tr>
</tbody>
</table>

\[\varphi = \sum_{i=1}^{n} \alpha \in S \rightarrow P_i = \sum_{i=1}^{n} \alpha \in S \rightarrow R(\alpha(k), s(k), j) = i + 1\]  

(9)

RTPA can be used to describe both logical and physical models of systems, where logic views of the architecture of a software system and its operational platform can be described using the same set of notations. When the system architecture is formally modelled, the static and dynamic behaviors that perform on the system architectural model can be specified by a three-level refinement scheme at the system, class, and object levels in a top-down approach (Wang, 2007, 2008c).
DESIGN OF THE RTPA CODE GENERATOR

The RTPA code generator (RTPA-CG) is a special compiler that transfers a mathematical model of a software system in RTPA into C++, Java, or other programming languages (with specific plug-in code builder modules). A set of semantic mapping strategies and patterns is designed to guide the processes of code generation under the support of the RTPA runtime library and the standard language libraries of C++/Java.

Design Considerations of the RTPA-CG

The challenges to automatic code generation based on RTPA specifications stem from the following issues:

a) As a denotational mathematical notation system, RTPA adopts prefix, postfix, and infix notations. Although these features of RTPA enable an excellent readability in system modeling, they are extremely difficult for machine processing.

b) Because the information about identifiers in an RTPA model is distributed throughout the specification, identifier identifications and their consistence checking become a very complicated problem in RTPA-based code generation.

c) As a general purpose process algebra, RTPA encompasses a rich set of process operations, such as real-time dispatching, dynamic memory manipulations, and device input/output. These features of RTPA require the support of a real-time operating system kernel as the runtime environment.

In order to cope with the above challenges, a two-pass code generation framework is adopted. The code generator first recognizes and processes a formal system model specified in RTPA, which produces a set of uniquely identified Abstract Syntax Trees (ASTs) (Aho et al., 1985). Then, the code generator transfers the ASTs into C++/Java code based on predefined mapping strategies and patterns. The workflow of RTPA-CG is designed as follows:

a) An RTPA model of a software system is specified by system analysts and architects.

b) The RTPA specification is processed by the RTPA lexical analyzer, parser, and type checker.

c) A set of ASTs is generated after the phase of type checking.

d) Grammar and constraint information, such as process priority levels and event usages, are analyzed based on the ASTs.

e) C++/Java code is generated for the given system model specified in RTPA supported by the RTPA runtime library.

Structure of the RTPA-CG

The structure of RTPA-CG is designed as shown in Fig. 5, which encompasses the RTPA lexer, parser, type checker, code builder, and the runtime library. RTPA-CG parses a set of ASTs derived from an RTPA specification in order to generate corresponding code in C++, Java, or other programming languages. The RTPA identifier tables are used to record all related information and their cross-references. Code is generated using the RTPA-C++/Java mapping strategies and patterns supported by the RTPA runtime library.

The RTPA Code Generation Techniques

The RTPA methodology specifies a software system in three-subsystems known as its architecture, static behaviors, and dynamic behaviors (Wang, 2007) as shown in Fig. 1. Corresponding to the RTPA model of a software system, the architecture of the system in RTPA is translated into C++/Java architectural class with object declarations. The UDM schemas in RTPA are translated into member classes in the architecture class, where each field of a schema is presented by corresponding member variables and each constraint is implemented by an
enumeration type in C++ or a special enumeration class in Java. The corresponding member variables are then declared using the defined classes. The UDM objects of RTPA schemas are transferred into global class constructors that act as initializations for the member classes obtained from the UDM schemas.

System static behaviors specified in RTPA are individual functions of a system and their interactions with the UDMs defined in the system architectures. Each static behavior specified in RTPA is modeled as a UPM, which is transferred into a corresponding method in C++/Java classes using a combination of meta-types, meta/complex processes, and process relations as defined in RTPA.

The dynamic behaviors of a software system describe the interactions between the defined static processes and their environment at run-time. In RTPA, the dynamic behaviors of a system are described in three refinement steps that can generally be grouped into process classification, process deployment, and process dispatch (Wang, 2007). RTPA-CG treats an RTPA process deployment and process dispatch as the main class in the target system, which implements precisely timed event/time/interrupt-driven behaviors at different priority levels of the system.

The RTPA runtime library consists of a kernel and an RTPA interface. The kernel of the RTPA runtime library provides the essential functionality of a real-time operating system kernel for multitasking and timing mechanisms. The kernel implements special RTPA processes such as parallel, concurrency, interrupt, real-time clock, and time/event driven mechanisms.

The second component of the RTPA runtime library, the RTPA interface, serves port I/O, timing, and time/event-driven dispatches based on the RTPA kernel. The RTPA real-time support environment is designed based on the formal models of the Real-Time Operating Systems (RTOS+) (Wang et al., 2010) and the support of the Micro Controller Operating System II (μC/OS-II) (Labrosse, 1999).
IMPLEMENTATION OF THE RTPA CODE GENERATOR

The technical approach to implementing the RTPA-CG is shown in Fig. 6. RTPA-CG transfers a formal system model specified in RTPA into C++/Java code. The RTPA tree parser guides the code builder to generate source code under the support of the RTPA runtime library and the auxiliary identifier tables.

Based on the design of the RTPA-CG as elaborated in the preceding section, both a C++ and a Java version of RTPA-CG are developed on the platform of DOS enhanced by μC/OS-II. This section describes the implementation of key components of RTPA-CGs, as well as the configuration and integration of the system.

Implementation of Key Components of RTPA-CG

The key components of RTPA-CG, such as the RTPA lexer, parser, type checker, AST parser, code builder, identifier table manipulator, and runtime library, are implemented as described in the following:

a) **The RTPA Lexer:** The front-end of the code generator employs a lexical analyzer (lexer) and a parser for system specifications in RTPA. The RTPA lexer analyzes the input of a formal system model in RTPA, and produces an output of a machine recognizable token stream. The RTPA lexer is generated by ANTLR (Parr & Quong, 1995; Parr, 2000) based on LL(k) grammar (Aho, 1985; Grune et al., 2000). In the RTPA syntactic analysis, most of the grammar rules for RTPA terminal symbols can be recognized by LL(k) algorithm. However, for those exceptional non-LL(k) grammar rules of RTPA, an arbitrary-length forward pre-trial algorithm, which will not consume the tokens in the input token stream, is adopted until a determinate choice can be made.

b) **The RTPA Parser:** The parser transfers RTPA specifications into abstract syntax in the form of a set of ASTs. The ASTs are then inputted to the code generator after type checking, which creates the executable code in C++/Java. The parser is built using ANTLR based on the predefined RTPA grammar rules in LL(k) (Wang, 2007). The key syntax of RTPA is defined by a set of 280 LL(k) grammar rules in EBNF (Tan, Wang, & Ngolah, 2004). To deal with the special non-LL(k) grammar rules of RTPA, the syntactic predicates provided by ANTLR are adopted to create guarded and extended rules. The same techniques used in the lexical analysis are applied in the parsing of the RTPA specifications.

Figure 6. Implementation of the RTPA code generator
c) **The RTPA Type Checker:** The type checker is an intermediate processor between the RTPA parser and code builder. Because RTPA is strongly typed, every identifier in an RTPA specification has a type suffix and every expression is evaluated to a definite type. Therefore, only those operations among compatible types are allowed, while the operations among incompatible types are forbidden unless explicit type casting is adopted. The type checking for RTPA specifications can be classified into three categories namely: (i) identifier type compliance, (ii) expression type compliance, and (iii) process constraint consistence. The first two tasks are considered as similar to those of programming language processing, while the third task is special due to the features of RTPA as a process-based algebraic notation system.

d) **The RTPA AST Parser:** A tree parser provides the means to walk through the ASTs generated by the preceding components (a) through (c) and transform them into corresponding code in C++/Java. The tree parser scans ASTs derived from a given system model, and transfers them into corresponding code in C++ or Java. With the ANTLR tool support (Parr, 2000); the first step to build the tree parser for RTPA is to specify the structure of ASTs generated by the RTPA Lexer and Type Checker in form of a set of ANTLR tree parser rules. Directed by the passing rules, the RTPA AST parser can be produced using the ANTLR tool automatically.

e) **The RTPA Code Builder:** The RTPA code builder is implemented based on the AST parser, which carries out corresponding semantic actions on each node of the given AST. A set of semantic actions is specified for expected functions in code generation such as RTPA identifier information manipulation and code transformation, which are implemented in the Java classes rtpaIDTable and rtpaCodeGeneration.

f) **The RTPA Runtime Library:** The RTPA runtime library encompasses the following RTPA functions: rtpaPortIn, rtpaPortOut, rtpaTiming, rtpaDuration, rtpaMakeDateTime, rtpaCreateEvent, rtpaEventSignal, rtpaEventWait, rtpaEventGroupSignal, rtpaEventGroupWait, rtpaTimeDispatch, and rtpaStop as shown in Table 1. The RTPA kernel is implemented based on DOS with the support of μC/OS-II, where the latter embodies the concurrency, interrupt, real-time clock, and time/event driven mechanisms. The RTPA interface functions implement the RTPA device-drive processes such as port I/O, timing, and time/event-driven dispatch processes.

The techniques applied to build a kernel of a real-time operating system are exploited in the implementation of the key RTPA real-time processes. Real-time task scheduling and event handling mechanisms are built into the core of the RTPA runtime library. The real-time kernel of RTPA guarantees that code under generation will meet the real-time requirements of a system specification.

g) **The Identifier Table and its Manipulations:** The identifier table class provides a set of internal records of all identifiers in a given RTPA specification. It supports the following functions: a) Building and maintaining the identifier tables; b) Creating corresponding variables from RTPA identifiers and constructing their internal names and types; c) Accessing and modifying identifier attributes; and d) Handling hierarchical structures when walking through a specification. Four types of identifier tables are implemented for type-checking and code generation. They are: (i) The general RTPA identifier table that holds all identifiers specified in an RTPA model; (ii) The event table that includes all events in the RTPA specification; (iii) The event relation group table that maintains event associations in the RTPA specification; and (iv) The process table that
contains all process priority information in the RTPA specification.

**Configuration and Integration of the Code Generator**

The configuration of RTPA-CG encompasses five Java classes such as RTPALexer, RTPATYPEChecker, RTPACodeBuilder, rtpaIDTable, and rtpaCodeGeneration. The relationships of the five classes can be described by a class diagram in Fig. 7.

Derived from the RTPA tree parser, RTPA-CodeBuilder generates code by walking through a given AST. With the guide of AST, corresponding code is automatically composed by invoking target functions provided in the classes rtpaIDTables and rtpaCodeGeneration.

The integration of RTPA-CG can be divided into the following steps as illustrated in Fig. 8:

a) Specify the rules of the RTPA tree parser;

b) Develop the five functional classes for RTPA code generation;

c) Insert ANTLR semantic actions to process a set of rules for the RTPA code builder;

d) Apply the ANTLR tool to generate the RTPA code generator class.

The automatically generated code is executable and effective under the support of the standard C++ library and the specially developed RTPA run-time library. Experimental results show that RTPA specifications can be rigorously checked and corresponding C++/Java code can be automatically generated from RTPA specifications using the toolkit.

**APPLICATIONS OF THE RTPA CODE GENERATOR**

This section describes applications of the RTPA-CG toolkit. A set of case studies on RTPA-based code generation in C++ and Java is presented.

---

**Table 1. RTPA runtime library functions**

<table>
<thead>
<tr>
<th>RTPA Process</th>
<th>Runtime Library Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>int rtpaPortIn(unsigned port)</td>
<td>Input data from a port</td>
</tr>
<tr>
<td>Output</td>
<td>int rtpaPortOut(unsigned port, int data)</td>
<td>Output data to a port</td>
</tr>
<tr>
<td>Timing</td>
<td>rtpaTM* rtpaTiming(rtpaTM *tm)</td>
<td>Take system time</td>
</tr>
<tr>
<td>Duration</td>
<td>rtpaTM* rtpaDuration(rtpaTM *tm, unsigned n)</td>
<td>Delay for n milliseconds</td>
</tr>
<tr>
<td></td>
<td>rtpaDATETIME rtpaMakeDateTime(char *dt, int type)</td>
<td>Make an RTPA Date/Time form string</td>
</tr>
<tr>
<td>Event detection</td>
<td>rtpaEVENT rtpaCreateEvent(unsigned int count)</td>
<td>Create a system event</td>
</tr>
<tr>
<td></td>
<td>rtpaEVENTSignal(rtpaEVENT x)</td>
<td>Report event x signaled</td>
</tr>
<tr>
<td></td>
<td>void rtpaEventWait(rtpaEVENT x, unsigned timeout, unsigned char *err)</td>
<td>Wait on an event x</td>
</tr>
<tr>
<td>Event-driven dispatch (individual)</td>
<td>OS_FLAGS rtpaEventGroupWait((OS_FLAG_GRP *pgrp, OS_FLAGS flags, unsigned char wait_type, unsigned timeout, unsigned char *err)</td>
<td>Wait on an event in event group pgrp</td>
</tr>
<tr>
<td>Event-driven dispatch (group)</td>
<td>OS_FLAGS rtpaEventGroupSignal(OS_FLAG_GRP *pgrp, OS_FLAGS flags, unsigned char opt, unsigned char *err)</td>
<td>Report an event in event group pgrp signaled</td>
</tr>
<tr>
<td>Time-driven dispatch</td>
<td>rtpaTM* rtpaTimeDispatch(rtpaTM *tm)</td>
<td>Time-driven dispatch</td>
</tr>
<tr>
<td>Exception detection</td>
<td>OS_FLAGS rtpaEventGroupSignal(OS_FLAG_GRP *pgrp, OS_FLAGS flags, unsigned char opt, unsigned char *err)</td>
<td>Report an event in event group pgrp signaled</td>
</tr>
<tr>
<td>Stop</td>
<td>void rtpaStop(void)</td>
<td>System stop</td>
</tr>
</tbody>
</table>
Automatic Code Generation using RTPA-CG

RTPA-CG compiles an RTPA specification into C++ or Java following the scheme as designed in Fig. 9. The program head of RTPA-CG is structured as an integration of the classes of RTPALexer, RTPATypeChecker, and RTPACodeBuilder in order to generate target code for a formal system specification modeled in RTPA.

A number of case studies on automatic code generation have been carried out using RTPA-CG. The real-world case studies cover large-scale real-time software systems 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., 2010), a Real-Time Operating System (RTOS+) (Wang et al., 2010), and a number of typical computing algorithms (Wang, 2007). The testing results demonstrate that RTPA-CG can atomically and seamlessly generate executable and effective C++ and Java code on the basis of formal system models in RTPA.
A Case Study on RTPA-Based Code Generation in Java

A case study on RTPA-based code generation in Java for the In-Between Sum (IBS) algorithm is presented in this subsection as a sample of applications of the RTPA-CG system. The IBS algorithm calculates the sum of digits between two given numbers \( A \) and \( B \) where \( A \) is the lower bound and \( B \) the upper bound. The formal model of the IBS algorithm specified in RTPA is given in Figs. 10 and 11 with both its architecture \textit{IBSAlgorithm.ArchitectureST} and behaviours \textit{IBSAlgorithm.StaicBehaviorsPC}, respectively.

Based on the formal models of IBS as given in Figs. 10 and 11, RTPA-CG can automatically generate source code for the IBS algorithm in Java as shown in Fig. 12 (Ngoalah and Wang, 2009). In Fig. 12, the first part of the code is the IBS structure and variable declarations; the second part of the code is the method of the IBS algorithm in Java; and the third part of the code is the corresponding dynamic behaviors of the algorithm when it is a part of another system. The Java version of RTPA-CG is implemented under the support of a real-time Java virtue machine (Bollella et al., 2002) and the Java Native Interface (JNI) (Sheng, 1999) with the plug-in modules of Java code generation classes and support libraries.

A Case Study on RTPA-Based Code Generation in C++

A case study on RTPA-based code generation in C++ for a simple real-time device driver (RTDD) is presented in this subsection as a sample of applications of the RTPA-CG system. RTDD is designed to drive a device represented by a set of six LEDs and a switch connected to the parallel port of a computer (Tan, Wang, & Ngoalah, 2006). The driver turns on the LEDs using different patterns controlled by the input of the switch. The driver checks the switch’s status periodically and lights the LEDs in specified patterns when a switch-on status is detected while the system handles other real-time tasks at the same time.

a) The Formal Model of RTDD in RTPA

The RTPA model of the RTDD system specifies the architecture, static behaviors, and dynamic behaviours of RTDD as given in Eq. 10, i.e.

```java
public static void main (String[] args) {
    try {
        FileReader fin = new FileReader(args[0]);
        RTPALexer lexer = new RTPALexer(fin);
        RTPATYPEChecker checker =
            new RTPATYPEChecker(lexer);
        AST t = checker.getAST();
        if (t == null) return;
        RTPACodeGenerator cgen =
            new RTPACodeGenerator(checker.idt);
        cgen.rtpa_specification(t);
        cgen.cg.rtpaSaveCode(args[1], cgen.idt);
    } catch (Exception e) {
        System.err.println("exception: "+e);
    }
}
```

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
The architecture of RTDD, RTDD§, is modeled as shown in Fig. 13, which consists of two unified data models (UDMs), also known as component logical models (CLMs), and a set of events and status. The UDM of SwitchST specifies the input devices of RTDD such as the structure and interface of the switch with specific initial values. The UDM of LEDsST models the output devices of RTDD with six LEDs as well as their shared port address and output patterns.

The static behaviors of RTDD are specified in Fig. 14 by three processes known as SystemInitialPC, RequestScanPC, and RequestHandlingPC. As modeled in RTDD§.StaticBehaviorsPC, RequestScanPC detects the switch status via the port interface and passes the device request to RequestHandlingPC via the global system status &RequestIdentifiedBL. When &RequestIdentifiedBL = T is identified, RequestHandlingPC turns on the six LEDs in predesigned patterns.

The dynamic behaviors of RTDD, TDD§. DynamicBehaviorsPC, are specified in Fig. 15. All real-time tasks of RTDD are deployed at two levels known as the base and interrupt levels. The former deals with routine and low priority tasks RequestHandlingPC, and other processes after the system is initialized and before it is shutdown. The latter handles high priority and timing-sensitive tasks such as RequestScanPC, which scans the status of the switch device per 100 milliseconds.

b) Code Automatically Generated for the RTDD System

A set of C++ programs of RTDD generated by RTPA-GC are listed in Figs. 16 and 17.
The first file presented in Fig. 16, class RTD, is the head file for the class RTDD. The RTDD class implemented in C++, RTDD::RTDD(), is shown in Fig. 17. With support of μC/OS-II and Borland C++ Compiler 5.1, the generated files were compiled under DOS larger mode. Then, the executable code for RTDD generated by RTPA-CG can be run as expected to respond to the device requests for displaying desired LED patterns and other application processes simultaneously.

The results of the sample case studies show that machine-based autonomic code generation based on formal system models in RTPA is technically feasible. As a formal notation and methodology, RTPA is not only easy to understand but also seamlessly transformable into high-level programming languages such as C++ and Java. Further case studies on RTPA meth-
**Figure 13. The architectural model of RTDD in RTPA**

```
RTDD$ . ArchitectureST ≜ \{ <Switch : ST | [1]>
                             || <LEDs : ST | [6]>
                             || <@Events : ST >
                             || <@Status : ST >
                         \}

                           \{ SwitchST ≜ (<InputPortAddr : H | InputPortAddrH = 037F>;
                                           <KeyInput : B | KeyInputB = 0000 0111B>,
                                           <KeyScanStatus : B | KeyScanStatusB = 0000 0000B>);
                          \}
                          \|| LEDsST ≜ (<NoLEDs : B | NoLEDsNB = 6>,
                           <LightPattern : B | LightPatternB = 0000 0000B>,
                           <OutputPortAddr : H | OutputPortAddrH = 0378H>);
                          \}
                          \|| @EventST ≜ ( @SwitchOnS
                                             || @SwitchOffS
                                             || @SysClock100msInt
                          \}
                          \|| @StatusST ≜ ( @RequestIdentifiedBL
                                             || @RequestIgnoredBL
                          \}
```

**Figure 14. The static behavioural model of RTDD**

```
RTDD$ . StaticBehaviorsPC ≜ ( SystemInitial(<I : ( )>; <O : LEDST>PC
                            || RequestScan(<I : SwitchST>; <O : @RequestIdentifiedBLPC
                            || RequestHandling(<I : @RequestIdentifiedBL>; <O : LEDST>PC

SystemInitial(<I : ( )>; <O : LEDST>PC ≜
\{ LEDST.LightPatternB := 0000 0011B
  → LEDST.LightPatternB := PORT(LEDSST.OutputPortAddrH)
\}

RequestScan(<I : SwitchST>; <O : @RequestIdentifiedBLPC ≜
\{ PORT(SwitchST.InputPortAddrH) := SwitchST.KeyInputB
  → ( @RequestIdentifiedBL := T
       || @RequestIdentifiedBL := F
  \}

RequestHandling(<I : @RequestIdentifiedBL>; <O : LEDST>PC ≜
\{ @RequestIdentifiedBL := T
   → 0000 1100B := Port(LEDSST.OutputPortAddrH) // Turn on LEDs 1 and 2
   → @TM := @TM + 1000ms // Delay 1 second
   → 0000 0000B := Port(LEDSST.OutputPortAddrH) // Turn off all LEDs
   → @TM := @TM + 1000ms
   → 0011 0000B := Port(LEDSST.OutputPortAddrH) // Turn on LEDs 3 and 4
   → @TM := @TM + 1000ms
   → 0000 0000B := Port(LEDSST.OutputPortAddrH) // Turn on LEDs 5 and 6
   → @TM := @TM + 1000ms
   → 1100 0000B := Port(LEDSST.OutputPortAddrH) // Turn on LEDs 5 and 6
   → @TM := @TM + 1000ms
   → 0000 0000B := Port(LEDSST.OutputPortAddrH)
\}
```
Figure 15. The dynamic behavioural model of RTDD

```
RTDD$\cdot$DynamicBehaviorsPC ⊳
{ // Base level processes
  @SysInitial$\triangleright$
    ← SystemInitialPC(⟨I:: (); ⟨O:: LEDsST⟩⟩)
    → $\Uparrow$
      RequestHandlingPC(⟨I:: RequestIdentifiedBL; ⟨O:: LEDsST⟩⟩)
      // Other processes
  $\Uparrow$
    …
  $\triangleright$
  || // Interrupt level processes
  @SysClock100msInt(⊙)
    ← RequestScanPC(⟨I:: SwitchST; ⟨O:: RequestIdentifiedBL⟩⟩)
    → @SysClock100msInt(⊙) := off
    $\triangleright$
  $\triangleright$
}
```

Figure 16. The head file of RTDD generated by RTPA-CG

```c
#include "rtdd_cpp.h"
class RTDD {

private:
  // Abstract code generation:
  RTDD();
  struct Type_ST {
    Type_ST *eptr;
    char *id_S;
    LEDStatusSet_BL_Type_ST LEDStatusSet_BL;
  };
  struct Switch_ST {
    Switch_ST *eptr;
    char *id_S;
    int InputPortAddr_H;
    char KeyInput_B;
    char KeyScanStatus_B;
  };
  struct LEDs_ST {
    LEDs_ST *eptr;
    char *id_S;
    int NoLEDs N;
    char LightPattern_B;
    int OutputPortAddr_H;
  };
  Type_ST Type_S;
  Switch_ST Switch_S;
  LED_ST LED_S;
  rtpaEVENT SwitchOn_S;
  rtpaEVENT SwitchOff_S;
  rtpaEVENT SysClock100msInt_S;
  rtpaEVENT rtpaConEvent1;
  BOOLEAN RequestIdentified_BL;
  BOOLEAN RequestServed_BL;
  void SysInitial();
  void RequestScan();
  void RequestHandling();
  void Deployment();
};
```
Figure 17. The code generated for RTDD based on the RTPA model

```c
#include <includes.h>
#include "RTDD.h"

RTDD::RTDD() {
    Switch_S.InputPortAddr_H = 0x379;
    Switch_S.KeyInput_B = 1;
    Switch_S.KeyScanStatus_B = 23;
    LEDs_S.NofLEDs_N = 6;
    LEDs_S.LightPattern_B = 0;
    LEDs_S.OutputPortAddr_H = 0x378;
    // Other processes
    RequestIdentified_BL = FALSE;
    RequestServed_BL = FALSE;
}

void RTDD::SystemInitial() {
    LEDs_S.LightPattern_B = 3;
    rtpaPortOut(LEDs_S.OutputPortAddr_H, LEDs_S.LightPattern_B);
}

void RTDD::RequestScan() {
    Switch_S.KeyInput_B = rtpaPortIn(Switch_S.InputPortAddr_H);
    if (Switch_S.KeyScanStatus_B == Switch_S.KeyInput_B) {
        RequestIdentified_BL = TRUE;
    } else {
        RequestIdentified_BL = FALSE;
    }
}

void RTDD::RequestHandling() {
    rtpaDATETIME(t_TIME);
    if (RequestIdentified_BL == TRUE) {
        LEDs_S.LightPattern_B = 12;
        rtpaPortOut(LEDs_S.OutputPortAddr_H, LEDs_S.LightPattern_B);
        rtpaPortOut(LEDs_S.OutputPortAddr_H, LEDs_S.LightPattern_B);
        rtpaPortOut(LEDs_S.OutputPortAddr_H, LEDs_S.LightPattern_B);
        rtpaPortOut(LEDs_S.OutputPortAddr_H, LEDs_S.LightPattern_B);
    }
}

void RTDD::DeploymentT1()
    SysInitial();
    // Other processes
}
```

CONCLUSION

This paper has reported the design and implementation methodologies and techniques for the Real-Time Process Algebra (RTPA) code generators in C++ and Java. RTPA has been introduced as a denotational mathematics for the algebraic modeling and manipulations of software system architectures and behaviors by the Unified Data Models (UDMs) and Unified Process Models (UPMs). The RTPA code generator (RTPA-CG) methodologies and RTPA-based code generation technologies have been reported in (Wang, 2007, 2009b; Wang et al., 2009, 2010a, 2010b). The case studies demonstrate that RTPA-CGs provide a seamless transformation technology from formal system models in RTPA to executable code in C++ and Java. Applications of RTPA-CGs may greatly improve program development productivity in software engineering and in the software industry.
has been configured encompassing the RTPA lexer, parser, type-checker, code-builder, and the real-time runtime library. The framework for RTPA code generation has provided a great flexibility by separating RTPA code generation into two passes of specification parsing and code building. It has been demonstrated that the system is practical and efficient to generate software code in C++ and Java, as well as other target programming languages with specific plug-in supporting modules. A number of experiments on code generations using RTPA-CG have been carried out. The results have shown that the approach adopted to automatically generate program code from formally modeled system specifications is achievable and practical, which can seamlessly transfer a software system model specified in RTPA into executable and effective code.

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


Yingxu Wang is professor of cognitive informatics and software engineering, Director of International Center for Cognitive Informatics (ICfCI), and Director of Theoretical and Empirical Software Engineering Research Center (TESERC) at the University of Calgary. He is a Fellow of WIF, a P.Eng of Canada, a Senior Member of IEEE and ACM, and a member of ISO/IEC JTC1 and the Canadian Advisory Committee (CAC) for ISO. He received a PhD in Software Engineering from The Nottingham Trent University, UK, in 1997, and a BSc in Electrical Engineering from Shanghai Tiedao University in 1983. He has industrial experience since 1972 and has been a full professor since 1994. He was a visiting professor in the Computing Laboratory at Oxford University in 1995, Dept. of Computer Science at Stanford University in 2008, and the Berkeley Initiative in Soft Computing (BISC) Lab at University of California, Berkeley in 2008, respectively. He is the founder and steering committee chair of the annual IEEE International Conference on Cognitive Informatics (ICCI). He is founding Editor-in-Chief of International Journal of Cognitive Informatics and Natural Intelligence (IJCINI), founding Editor-in-Chief of International Journal of Software Science and Computational Intelligence (IJSSCI), Associate Editor of IEEE Trans on System, Man, and Cybernetics (A), and Editor-in-Chief of CRC Book Series in Software Engineering. He is the initiator of a number of cutting-edge research fields and/or subject areas such as cognitive informatics, cognitive computing, abstract intelligence, denotational mathematics, theoretical software engineering, coordinative work organization theory, cognitive complexity of software, and built-in tests. He has published over 100 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.

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.

Copyright © 2010, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.