

On the Big-R Notation for Describing Iterative and Recursive Behaviors

Yingxu Wang, University of Calgary, Canada

ABSTRACT

Iterative and recursive control structures are the most fundamental mechanisms of computing that make programming more effective and expressive. However, these constructs are perhaps the most diverse and confusable instructions in programming languages at both syntactic and semantic levels. This article introduces the big-R notation that provides a unifying mathematical treatment of iterations and recursions in computing. Mathematical models of iterations and recursions are developed using logical inductions. Based on the mathematical model of the big-R notation, fundamental properties of iterative and recursive behaviors of software are comparatively analyzed. The big-R notation has been adopted and implemented in Real-Time Process Algebra (RTPA) and its supporting tools. Case studies demonstrate that a convenient notation may dramatically reduce the difficulty and complexity in expressing a frequently used and highly recurring concept and notion in computing and software engineering.

Keywords: basic control structures; cognitive informatics; computing; formal methods; iteration; loop; mathematical notations; RTPA; recursion; semantics; software engineering; syntax; the big-R notation

INTRODUCTION

A repetitive and efficient treatment of recurrent behaviors and architectures is one of the most premier needs in computing. Iterative and recursive constructs and behaviors are most fundamental to computing because they enable programming to be more effective and expressive. However, unlike the high commonality in branch structures among programming languages, the syntaxes of loops are far more than unified. There is even a lack of common

semantics of all forms of loops in modern programming languages (Louden, 1993; Wang, 2006a; Wilson and Clark, 1988).

When analyzing the syntactic and semantic problems inherited in iterations in programming languages, B. L. Meek concluded that: "There are some who argue that this demonstrates that the procedural approach to programming languages must be inadequate and fatally flawed, and that coping with something so fundamental as looping must therefore entail looking at computation in a different way rather than try-

ing to devise better procedural syntax. There are others who would argue that the possible applications of looping so it cannot simply be removed or obviated. As ever it is probably this last argument that will hold sway until (or unless) someone proves them wrong, whether with a brilliant stroke of procedural syntactic genius, or an effective and comprehensive new approach to the whole area" (Meek, 1991).

This article introduces the big-R notation that provides a unifying mathematical treatment of iterations and recursions in computing. It summarizes the basic control structures of computing, and introduces the big-R notation on the basis of mathematical inductions. The unified mathematical models of iterations and recursions are derived using the big-R notation. Basic properties of iterative and recursive behaviors and architectures in computing are comparatively analyzed. The big-R notation has been adopted and implemented in Real-Time Process Algebra (RTPA) and its supporting tools (Wang, 2002, 2003; Tan, Wang, & Ngolah, 2004). Application examples of the big-R notation in the context of RTPA will be provided throughout this article.

THE BIG-R NOTATION

Although modern high-level programming languages provide a variety of iterative constructs, the mechanisms of iteration may be expressed by the use of conditional or unconditional jumps with a body of linear code. The proliferation of various loop constructs in programming indicates a fundamental need for expressing the notion of repetitive, cyclic, recursive behaviors, and architectures in computing.

In the development of RTPA (Wang, 2002, 2003, 2006a, 2007a, 2007b), it is recognized that all iterative and recursive operations in programming can be unified on the basis of a big-R notation (Wang, 2006b). This section introduces the big-R notation and its mathematical foundation. It can be seen that a convenient notation may dramatically reduce the difficulty and complexity in expressing a frequently used and highly recurring concept and notion in programming.

The Basic Control Structures of Computing

Before the big-R notation is introduced, a survey of essential basic control structures in computing is summarized and reviewed below.

Definition 1. *Basic control structures (BCS's) are a set of essential flow control mechanisms that are used for modeling logical architectures of software.*











The most commonly identified BCS's in computing are known as the *sequential, branch, case (switch), iterations (three types), procedure call, recursion, parallel, and interrupt* structures (Backhouse, 1968; Dijkstra, 1976; Wirth, 1976; Backus, 1978; de Bakker, 1980; Jones, 1980; Cries, 1981; Hehner, 1984; Hoare, 1985, Hoare et al., 1987; Wilson & Clark, 1988; Loudon, 1993; Wang, 2002; Horstmann & Budd, 2004). The 10 BCS's as formally modeled in RTPA (Wang, 2002) are shown in Table 1. These BCS's provide essential compositional rules for programming. Based on them, complex computing functions and processes can be composed.

As shown in Table 1, the iterative and recursive BCS's play a very important role in programming. The following theorem explains why iteration and recursion are inherently vital in determining the basic expressive power of computing.

Theorem 1. *The need for software is necessarily and sufficiently determined by the following three conditions:*

- a. The *repeatability*: Software is required when one needs to do something for more than once.
- b. The *flexibility* or *programmability*: Software is required when one needs to repeatedly do something not exactly the same.
- c. The *run-time determinability*: Software is required when one needs to flexibly do something by a series of choices on the basis of varying sequences of events determinable only at run-time.

Table 1. BCS's and their mathematical models

Category	BCS	Structural model	RTPA model
Sequence	Sequence (SEQ)		$P \rightarrow Q$
Branch	If-then-[else] (ITE)		$(\diamond \text{exp} \mathbf{BL} = \mathbf{T}) \rightarrow P$ $ (\diamond \sim) \rightarrow Q$
	Case (CASE)		$\diamond \text{exp} \mathbf{RT} =$ $0 \rightarrow P_0$ $ 1 \rightarrow P_1$ $ \dots$ $ n-1 \rightarrow P_{n-1}$ $ \text{else} \rightarrow \emptyset$
Iteration	While-do (R^*)		$\overset{f}{R} (P)$ $\text{exp} \mathbf{BL} = \mathbf{T}$
	Repeat-until (R^+)		$P \rightarrow \overset{f}{R} (P)$ $\text{exp} \mathbf{BL} = \mathbf{T}$
	For-do (R^i)		$\overset{n}{R} P(i)$ $i=1$
Embedded component	Procedure call (PC)		$P \hookrightarrow Q$
	Recursion (R^v)		$P \cup P$
Concurrency	Parallel (PAR)		$P \parallel Q$
	Interrupt (INT)		$P \dashv Q$

Theorem 1 indicates that the above three situations, namely repeatability, flexibility, and run-time determinability, form the necessary and sufficient conditions that warrant the requirement for a software solution in computing (Wang, 2006a).

The Big-R Notation for Denoting Iterations and Recursions

The big-R notation is introduced first in RTPA (Wang, 2002) intending to provide a unified and expressive mathematical treatment of iterations and recursions in computing. In order to develop a general mathematical model for

unifying the syntaxes and semantics of iterations, the inductive nature of iterations needs to be recognized.

Definition 2. An iteration of a process P is a series of $n+1$ repetitions, R_i , $1 \leq i \leq n+1$, of P by mathematical induction, that is:

$$\begin{aligned} R_0 &= \emptyset, \\ R_1 &= P \rightarrow R_0, \\ \dots \\ R_{n+1} &= P \rightarrow R_n, n \geq 0 \end{aligned} \quad (1)$$

where \emptyset denotes a skip, or doing nothing but exit.

Based on Definitions 2, the big-R notation can be introduced below.

Definition 3. The big-R notation is a mathematical operator that is used to denote: (a) a finite set of repetitive behaviors, or (b) a finite set of recurring architectural constructs in computing, in the following forms:

$$(a) \mathbf{R}_{\text{exp} \mathbf{BL} = \mathbf{T}}^{\mathbf{F}} P \quad (2.1)$$

$$(b) \mathbf{R}_{\mathbf{i} \mathbf{N} = 1}^n P(i) \quad (2.2)$$

where \mathbf{BL} and \mathbf{N} are the type suffixes of Boolean and natural numbers, respectively, as defined in RTPA. Other useful type suffixes that will appear in this article are integer (\mathbf{Z}), string (\mathbf{S}), pointer (\mathbf{P}), hexadecimal (\mathbf{H}), time (\mathbf{TM}), interrupt ($\mathbf{\odot}$), run-time type (\mathbf{RT}), system type (\mathbf{i}), and the Boolean constants (\mathbf{T}) and (\mathbf{F}) (Wang, 2002, 2007a).

The mechanism of the big-R notation can be in analogy with the mathematical notations Σ and Π , or programming notations of while-loop and for-loop as shown in the following examples.

Example 1. The big- Σ notation $\sum_{i=1}^n$ is a widely used calculus for denoting repetitive additions. Assuming that the operation of addition is represented by $sum(x)$, the mechanism of the

big- Σ can be expressed more generally by the big-R notation, that is:

$$\sum_{i=1}^n x_i = \mathbf{R}_{i=1}^n sum(x_i) \quad (3)$$

According to Definition 3, the big-R notation can be used to denote not only repetitive operational behaviors in computing, but also recurring constructs of architectures and data objects, as shown below.

Example 2. The architecture of a two-dimensional array with $n \times m$ integer elements, A_{nm} , can be denoted by the big-R notation as follows:

$$A_{nm} = \mathbf{R}_{i=0}^{n-1} \mathbf{R}_{j=0}^{m-1} A[i, j] \mathbf{N} \quad (4)$$

Because the big-R notation provides a powerful and expressive means for denoting iterative and recursive behaviors and architectures of systems or human beings, it is a general mathematical operator for system modeling in terms of repetitive “to do” and recurrent “to be,” respectively (Wang, 2002, 2003, 2006a, 2006b, 2006c, 2007a, 2007b). From this point of view, Σ and Π are only special cases of the big-R for repetitively doing additions and multiplications, respectively.

Definition 4. An infinitive iteration can be denoted by the big-R notation as:

$$\mathbf{R}P \triangleq \gamma \bullet P \curvearrowright \gamma \quad (5)$$

where \curvearrowright denotes a jump, and γ is a label that denotes the rewinding point of a loop known as the fix-point mathematically (Tarski, 1955).

The infinitive iteration may be used to formally describe any everlasting behavior of systems.

Example 3. A simple everlasting clock (Hoare, 1985), $CLOCK$, which does nothing but tick, that is:

$$CLOCK \triangleq tick \rightarrow tick \rightarrow tick \rightarrow \dots \quad (6)$$

can be efficiently denoted by the big-R notation as simply as follows:

$$CLOCK \triangleq \mathbf{R}tick \quad (7)$$

A more generic and useful iterative construct is the conditional iteration.

Definition 5. A conditional iteration can be denoted by the big-R notation as:

$$\begin{aligned} \mathbf{R}_{\text{expBL}=\mathbf{T}}^{\mathbf{F}} P \triangleq & \gamma \bullet (\blacklozenge \text{expBL}=\mathbf{T} \rightarrow P \\ & \rightarrow \gamma \\ & | \blacklozenge \sim \rightarrow \emptyset \\ &) \end{aligned} \quad (8)$$

where \emptyset denotes a skip.

The conditional iteration is frequently used to formally describe repetitive behaviors on given conditions. Equation 8 expresses that the iterative execution of P will go on while the evaluation of the conditional expression is true ($\text{expBL}=\mathbf{T}$), until $\text{expBL}=\mathbf{F}$ abbreviated by ' \sim '.

MODELING ITERATIONS USING THE BIG-R NOTATION

The importance of iterations in computing is rooted in the basic need for effectively describing recurrent and repetitive software behaviors and system architectures. This section reviews the diversity of iterations provided in programming languages and develops a unifying mathematical model for iterations based on the big-R notation.

Existing Semantic Models of Iterations

Since the wide diversity of iterations in programming, semantics of iterations have been described in various approaches, such as those of the *operational*, *denotational*, *axiomatic*, and *algebraic* semantics. For example, the while-

loop may be interpreted in different semantics as follows.

- a. An *operational semantic description* of the while-loop (Louden, 1993) can be expressed in two parts. When an expression E in the environment Θ is true, the execution of the loop body P for an iteration under Θ can be reduced to the same loop under a new environment Θ' , which is resulted by the last execution of P , that is:

$$\frac{\langle E \parallel \Theta \rangle \Rightarrow E = \mathbf{T}, \langle P \parallel \Theta \rangle \Rightarrow \Theta'}{\langle \text{'while' } E \text{'do' } P \text{' ;' } \parallel \Theta \rangle \Rightarrow \langle \text{'while' } E \text{'do' } P \text{' ;' } \parallel \Theta \rangle} \quad (9)$$

where the while-loop is defined recursively, and \parallel denotes a parallel relation between an identifier/statement and the semantic environment Θ .

When $E = \mathbf{F}$ in Θ , the loop is reduced to a termination or exit, that is:

$$\frac{\langle E \parallel \Theta \rangle \Rightarrow E = \mathbf{F}, \langle P \parallel \Theta \rangle}{\langle \text{'while' } E \text{' do' } P \text{' ;' } \parallel \Theta \rangle \Rightarrow \Theta} \quad (10)$$

- b. An *axiomatic semantic description* of the while-loop is given below (McDermid, 1991):

$$\frac{\vdash \{ \Theta \wedge E \} P \{ Q \}}{\vdash \{ \Theta \} \text{while } E \text{ do } P \{ \Theta \wedge \neg E \}} \quad (11)$$

where the symbol \vdash is called the syntactic turnstile.

- c. A *denotational semantic description* of the while-loop by recursive if-then-else structures in the literature is described below (Louden, 1993; Wirth, 1976):

$$\begin{aligned} S \llbracket \text{'while' } E \text{'do' } P \text{' ;' } \rrbracket \parallel \Theta = & \\ S \llbracket \text{if } E[E] \parallel \Theta = \mathbf{T} & \\ \text{then } P[P] \parallel \Theta & \\ \text{else } \Theta & \\ \rrbracket \parallel \Theta & \end{aligned} \quad (12)$$

where P is a statement or a list of statements, S and P are semantic functions of statements, and E is a semantic function of expressions.

Observing the above classical examples, it is noteworthy that the semantics of a simple while-loop could be very difficultly and diversely interpreted. Although the examples interpreted the decision point very well by using different branch constructs, they failed to denote the key semantics of “while” and the rewinding action of loops. Further, the semantics for more complicated types of iterations, such as the repeat-loop and for-loop, are rarely found in the literature.

A Unified Mathematical Model of Iterations

Based on the inductive property of iterations, the big-R notation as defined in Equation 8 is found to be a convenient means to describe all types of iterations including the while-, repeat-, and for-loops.

Definition 6. The while-loop R^* is an iterative construct in which a process P is executed repeatedly as long as the conditional expression $expBL$ is true, that is:

$$\begin{aligned}
 R^*P &\triangleq \dot{R}P \\
 &= \gamma \bullet (\diamond \exp BL = T \\
 &\quad \rightarrow P \\
 &\quad \curvearrowright \gamma \\
 &\quad | \diamond \sim \\
 &\quad \rightarrow \emptyset \\
 &)
 \end{aligned}
 \tag{13}$$

where $*$ denotes an iteration for 0 to n times, $n \geq 0$. That is, P may not be iterated in the while-loop at run-time if $expBL \neq T$ at the very beginning.

According to Equation 13, the semantics of the while-loop is deduced to a series of repetitive conditional operations, where the branch “ $\curvearrowright \gamma \rightarrow \emptyset$ ” denotes an exit of the loop when $expBL \neq T$. Note that the update of the control expression $expBL$ is not necessarily to be explicitly specified

inside the body of the loop. In other words, the termination of the while-loop, or the change of $expBL$, can either be a result of internal effect of P or that of other external events.

Definition 7. The repeat-loop R^+ is an iterative construct in which a process P is executed repetitively for at least once until the conditional expression $expBL$ is no longer true, that is:

$$\begin{aligned}
 R^+P &\triangleq P \rightarrow \dot{R}P \\
 &= P \rightarrow \dot{R}P \\
 &= P \rightarrow \gamma \bullet (\diamond \exp BL = T \\
 &\quad \rightarrow P \\
 &\quad \curvearrowright \gamma \\
 &\quad | \diamond \sim \\
 &\quad \rightarrow \emptyset \\
 &)
 \end{aligned}
 \tag{14}$$

where $+$ denotes an iteration for 1 to n times, $n \geq 1$. That is, P will be executed at least once in the repeat loop until $expBL \neq T$.

According to Equation 14, the semantics of the repeat-loop is deduced to a single sequential operation of P succeeded by a series of repetitive conditional operations whenever $expBL = T$. Or simply, the semantics of the repeat-loop is equivalent to a single sequential operation of P plus a while-loop of P .

In Equations 13 and 14, the loop control variable $expBL$ is in the type Boolean. When the loop control variable i is numeric, say in type \mathbf{N} with known lower bound $n_1\mathbf{N}$ and upper bounds $n_2\mathbf{N}$, then a special variation of iteration, the for loop, can be derived below.

Definition 8. The for-loop R^i is an iterative construct in which a process P indexed by an identification variable $i\mathbf{N}$, $P(i\mathbf{N})$, is executed repeatedly in the scope $n_1\mathbf{N} \leq i\mathbf{N} \leq n_2\mathbf{N}$, that is:

$$\begin{aligned}
 R^iP(i\mathbf{N}) &\triangleq \dot{R}_{i\mathbf{N}=n_1\mathbf{N}}^{n_2\mathbf{N}} P(i\mathbf{N}) \\
 &= i\mathbf{N} := n_1\mathbf{N}
 \end{aligned}$$

$$\begin{aligned}
 &\rightarrow \gamma \bullet (\diamond i\mathbf{N} \leq n_2\mathbf{N} \\
 &\quad \rightarrow P(i\mathbf{N}) \\
 &\quad \rightarrow \uparrow(i\mathbf{N}) \\
 &\quad \curvearrowright \gamma \\
 &\quad | \diamond \sim \\
 &\quad \rightarrow \emptyset \\
 &\quad) \\
 &= i\mathbf{N} := n_1\mathbf{N} \\
 &\rightarrow \exp_{\mathbf{BL}}^{\mathbf{BL}} = (i\mathbf{N} \leq n_2\mathbf{N}) \\
 &\quad \rightarrow \mathbf{R}_{\exp_{\mathbf{BL}}^{\mathbf{BL}}} (P(i\mathbf{Z})) \\
 &\quad \rightarrow \uparrow(i\mathbf{N}) \\
 &\quad)
 \end{aligned} \tag{15}$$

where i denotes the loop control variable, and $\uparrow(i\mathbf{N})$ increases $i\mathbf{N}$ by one.

According to Equation 15, the semantics of the for-loop is a special case of while-loop where the loop control expression is $\exp_{\mathbf{BL}} = i\mathbf{N} \leq n_2\mathbf{N}$, and the update of the control variable $i\mathbf{N}$ must be explicitly specified inside the body of the loop. In other words, the termination of the for-loop is internally controlled.

Based on Definition 8, the most simple for-loop that iteratively executes $P(i\mathbf{N})$ for k times, $1 \leq i \leq k$, can be derived as follows:

$$\mathbf{R}^i P(i\mathbf{N}) \triangleq \mathbf{R}_{i\mathbf{N}=1}^k P(i\mathbf{N}) \tag{16}$$

It is noteworthy that a general assumption in Equations 15 and 16 is that i is a natural number and the iteration step $\Delta i\mathbf{N} = +1$. In a more generic situation, i may be an arbitrary integer \mathbf{Z} or in other numerical types, and $\Delta i\mathbf{Z} \neq +1$. In this case, the lower bound of a for-loop can be described as an expression, or the incremental step $\Delta i\mathbf{Z}$ can be explicitly expressed inside the body of the loop, for example:

$$\begin{aligned}
 \mathbf{R}^i P(i\mathbf{Z}) &\triangleq \mathbf{R}_{i\mathbf{Z}=0}^{-10} (P(i\mathbf{Z}) \\
 &\quad \rightarrow i\mathbf{Z} := i\mathbf{Z} - \Delta i\mathbf{Z} \\
 &\quad)
 \end{aligned} \tag{17}$$

where $\Delta i\mathbf{Z} \geq 1$.

MODELING RECURSIONS USING THE BIG-R NOTATION

Recursion is a powerful tool in mathematics for a neat treatment of complex problems following a fundamental *deduction-then-induction* approach. Godel, Herbrand, and Kleene developed the theory of recursive functions using an equational calculus in the 1930s (Kleene, 1952; McDerimid, 1991). More recent work on recursions in programming may be found in “Mendelson (1964);” “Peter (1967);” “Hermes (1969);” “Hoare (1985);” “Rayward-Smith (1986);” “Louden (1993);” and “Wang (2006a, 2007a).” The idea is to construct a class of effectively computable functions from a collection of base functions using fundamental techniques such as function composition and inductive referencing.

Properties of Recursions

Recursion is an operation that a process or function invokes or refers to itself.

Definition 9. A recursion of process P can be defined by mathematical induction, that is:

$$\begin{aligned}
 F^0(P) &= P, \\
 F^1(P) &= F(F^0(P)) = F(P), \\
 &\dots \\
 F^{n+1}(P) &= F(F^n(P)), n \geq 0
 \end{aligned} \tag{18}$$

A recursive process should be terminable or noncircular, that is, the depth of recursive d_r must be finite. The following theorem guarantees that $d_r < \infty$ for a given recursive process or function.

Theorem 2. A recursive function is noncircular, that is, $d_r < \infty$, iff:

- a. A base value exists for certain arguments for which the function does not refer to itself;
- b. In each recursion, the argument of the function must be closer to the base value.

Example 4. *The factorial function can be recursively defined, as shown in Equation 19.*

$$\begin{aligned}
 (n\mathbf{N})! &\triangleq \{ \blacklozenge n\mathbf{N} = 0 \\
 &\quad \rightarrow (n\mathbf{N})! := 1 \\
 &\quad | \blacklozenge \sim \\
 &\quad \rightarrow (n\mathbf{N})! := n\mathbf{N} \bullet (n\mathbf{N}-1)! \\
 &\quad \} \\
 &\hspace{15em} (19)
 \end{aligned}$$

Example 5. *A C++ implementation of the factorial algorithm, as given in Example 4, is provided below.*

```

int factorial (int n)
{
    int factor;
    if (n==0)
        factor = 1;
    else factor = n * factorial(n-1);
    return factor;
}
    
```

(20)

In addition to the usage of recursion for efficiently modeling repetitive behaviors of systems as above, it has also been found useful in modeling many fundamental language properties.

Example 6. *Assume the following letters are used to represent their corresponding syntactic entities in the angle brackets:*

- P* <program>,
- L* <statement list>,
- S* <statement>,
- E* <expression>,
- I* <identifier>,
- A* <letter>,
- N* <number>, and
- D* <digit>

The abstract syntax of grammar rules for a simple programming language may be recursively specified in BNF as follows.

$$\begin{aligned}
 E &::= E \text{ '+' } E \\
 &\quad | E \text{ '-' } E \\
 &\quad | E \text{ '*' } E \\
 &\quad | \text{ '(' } E \text{ ')' } \\
 &\quad | I \\
 &\quad | N \\
 I &::= I A | A \\
 A &::= \text{'a'} | \text{'b'} | \dots | \text{'z'} \\
 N &::= N D | D \\
 D &::= \text{'0'} | \text{'1'} | \dots | \text{'9'} \\
 &\hspace{15em} (21)
 \end{aligned}$$

It can be seen in Equation 21 that expression *E* is recursively defined by operations on *E* itself, an identifier *I*, or a number *N*. Further, *I* is recursively defined by itself or letter *A*; and *N* is recursively defined as itself or digit *D*. Because any form of *E* as specified above can be eventually deduced on terminal letters ('a', 'b', ..., 'z'), digits ('0', '1', ..., '9'), or predefined operations ('+', '-', '*', '(' , ')'), the BNF specification of *E* as shown in Equation 21 is well-defined (Louden, 1993).

The Mathematical Model of Recursions

Definition 10. *Recursion is an embedded process relation in which a process P calls itself. The recursive process relation can be denoted as follows:*

$$P \circlearrowleft P \hspace{15em} (22)$$

The mechanism of recursion is a series of *embedding* (deductive, denoted by \circlearrowleft) and *de-embedding* (inductive, denoted by \circlearrowright) processes. In the first phase of embedding, a given layer of nested process is deduced to a lower layer until it is embodied to a known value. In the second phase of de-embedding, the value of a higher layer process is induced by the lower layer starting from the base layer, where its value has already been known at the end of the embedding phase.

Recursion processes are frequently used in programming to simplify system structures

and to specify neat and provable system functions. It is particularly useful when an infinite or run-time determinable specification has to be clearly expressed.

Instead of using self-calling in recursions, a more generic form of embedded construct that enables interprocess calls is known as the procedural call.

Definition 11. A procedural call is a process relation in which a process P calls another process Q as a predefined subprocess. A procedure-call process relation can be defined as follows:

$$P \hookrightarrow Q \quad (23)$$

In Equation 23, the called process Q can be regarded as an embedded part of process P (Wang, 2002).

Using the big-R notation, a recursion can be defined formally as follows.

Definition 12. Recursion $R^\circ P^i$ is a multi-layered, embedded process relation in which a process P at layer i of embedment, P^i , calls itself at an inner layer $i-1$, P^{i-1} , $0 \leq i \leq n$. The termination of P^i depends on the termination of P^{i-1} during its execution, that is:

$$R^\circ P^i \triangleq R_0^{\circ} (\begin{array}{l} \diamond i\mathbf{N} > 0 \\ \rightarrow P^{i\mathbf{N}} := P^{i\mathbf{N}-1} \\ | \diamond \sim \\ \rightarrow P^0 \end{array}) \quad (24)$$

where n is the *depth of recursion* or embedment that is determined by an explicitly specified conditional expression $exp_{\mathbf{BL}} = \mathbf{T}$ inside the body of P .

Example 7. Using the big-R notation, the recursive description of the algorithm provided in Example 4 can be given as follows:

$$(n\mathbf{N})! \triangleq R_0^{\circ} (n\mathbf{N})! \\ = R_{i\mathbf{N}=n\mathbf{N}}^{\circ} (\begin{array}{l} \diamond i\mathbf{N} > 0 \\ \rightarrow (i\mathbf{N})! := i\mathbf{N} \bullet (i\mathbf{N}-1)! \\ | \diamond \sim \\ \rightarrow (i\mathbf{N})! := 1 \end{array}) \quad (25)$$

COMPARATIVE ANALYSIS OF ITERATIONS AND RECURSIONS

In the literature, iterations were often treated as the same as recursions, or iterations were perceived as a special type of recursions. Although, both iteration $R^i P(i)$ and recursion $R^\circ P^i$ are repetitive and cyclic constructs, the fundamental differences between their traces of execution at run-time are that the former is a linear structure, that is:

$$R^i P(i) = P_1 \rightarrow P_2 \rightarrow \dots \rightarrow P_n \quad (26)$$

However, the latter is an embedded structure, that is:

$$R^\circ P^i = P^n \circ P^{n-1} \circ \dots \circ P^1 \circ P^0 \circ P^1 \\ \circ \dots \circ P^{n-1} \circ P^n \quad (27)$$

The generic forms of iterative and recursive constructs in computing can be contrasted as illustrated in Figures 1 and 2 as follows.

It is noteworthy that there is always a pair of counterpart solutions for a given repetitive and cyclic problem with either the recursive or iteration approach. For instance, the corresponding iterative version of Example 7 can be described below.

Example 8. Applying the big-R notation, the iterative description of the algorithm as provided in Example 7 is shown below.

Figure 1. The linear architecture of iterations

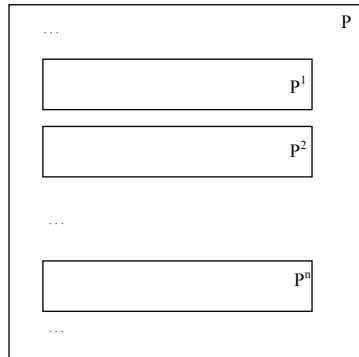
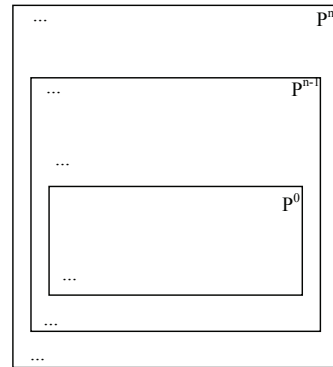


Figure 2. The nested architecture of recursions



$$\begin{aligned}
 (n\mathbf{N})! \triangleq \{ & \\
 & \text{factorial}\mathbf{N} := 1 \\
 & \rightarrow \mathbf{R}_{i=\mathbf{N}}^{\mathbf{n}}(\text{factorial}\mathbf{N} := i\mathbf{N} \bullet \text{factorial}\mathbf{N}) \\
 & \rightarrow (n\mathbf{N})! := \text{factorial}\mathbf{N} \\
 & \} \\
 & \qquad \qquad \qquad (28)
 \end{aligned}$$

It is interesting to compare both formal algorithms of factorial with recursion and iteration as shown in Equations 25 and 28.

Example 9. *On the basis of Example 8, an iterative implementation of Example 5 in C++ can be developed as follows.*

```

int factorial (int n)
{
    int factor = 1;
    for (int i = 1; i <= n; i++)
        factor = i * factor;
    return factor;
}
    (29)
    
```

The above examples show the difference between the recursive and iterative techniques for implementing the same algorithm for repetitive and cyclic computation. Contrasting Examples 4 and 8, or Examples 5 and 9, it can be seen that the recursive solution for a given problem is usually more expressive,

but less efficient in implementation in terms of time and space complexity than its iterative counterpart. As Peter Deutsch, the creator of the GhostScript interpreter, put it: “To iterate is human, to recurse divine.”

CONCLUSION

The efficient treatment of repetitive and recurrent behaviors and architectures has been recognized as one of the most premier needs in computing. However, surprisingly, there have been such a variety of iterative and recursive constructs in modern programming languages and there was still no settled consensus on some of the fundamental issues in their syntaxes and semantics.

This article has introduced the big-R notation that provides a unifying mathematical treatment of iterative and recursive behaviors and architectures in computing. Unified mathematical models of iterations and recursions have been derived using the big-R notation. Based on the big-R notation, the fundamental properties of iterative and recursive behaviors in computing have been comparatively analyzed. This article has demonstrated that a convenient notation may dramatically reduce the difficulty and complexity in expressing a frequently used and highly recurring concept and notion in computing. A wide range of applications of the big-R notation have been identified for effectively and rigorously modeling iterations and

recursions in computing, software engineering, and intelligent systems.

ACKNOWLEDGEMENT

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

REFERENCES

- Backhouse, R. C. (1968). *Program construction and verification*. London: Prentice Hall International.
- Backus, J. (1978). Can programming be liberated from the van Neumann Style? *Communications of the ACM*, 21(8), 613-641.
- Cries, D. (1981). *The science of programming*. New York: Springer-Verlag.
- de Bakker, J. W. (1980). *Mathematical theory of program correctness*. London: Prentice Hall International.
- Dijkstra, E. W. (1976). *A discipline of programming*. Englewood Cliffs, NJ: Prentice Hall.
- Hehner, E. C. R. (1984). Predicative programming, parts I and II. *Communications of the ACM*, 27(2), 134-151.
- Hermes, H. (1969). *Enumerability, decidability, computability*. New York: Springer-Verlag.
- Hoare, C. A. R. (1985). *Communicating sequential processes*. London: Prentice Hall International.
- Hoare, C. A. R., Hayes, I. J., He, J., Morgan, C. C., Roscoe, A. W., Sanders, J. W., et al. (1987, August). Laws of programming. *Communications of the ACM*, 30(8), 672-686.
- Horstmann, C., & Budd, T. (2004). *Big C++*. Danvers, MA: John Wiley & Sons.
- Jones, C. B. (1980). *Software development: A rigorous approach*. London: Prentice Hall International.
- Kleene, S. C. (1952). *Introduction to meta-mathematics*. North Holland, Amsterdam.
- Louden, K. C. (1993). *Programming languages: Principles and practice*. Boston: PWS-Kent Publishing.
- McDermid, J. (Ed.). (1991). *Software engineer's reference book*. Oxford: Butterworth Heinemann.
- Meek, B. L. (1991). Early high-level languages (Chapter 43). In J. McDermid (Ed.), *Software engineer's reference book*. Oxford: Butterworth Heinemann.
- Mendelson, E. (1964). *Introduction to mathematical logic*. New York: Van Nostrand Reinhold.
- Peter, R. (1967). *Recursive functions*. New York: Academic Press.
- Rayward-Smith, V. J. (1986). *A first course in computability*. Oxford: Blackwell Scientific.
- Tan, X., Wang, Y., & Ngolah, C. (2004, August). Specification of the RTPA grammar and its recognition. In *Proceedings of the 3rd IEEE International Conference on Cognitive Informatics (ICCI'04)*, Victoria, Canada, (pp. 54-63). IEEE CS Press.
- Tarski, A. (1955). A lattice-theoretic fixed point theorem and its applications. *Pacific Journal of Mathematics*, 5, 285-309.
- Wang, Y. (2002, October). The real-time process algebra (RTPA). *Annals of Software Engineering: An International Journal*, 14, 235-274. Kluwer Academic Publishers.
- Wang, Y. (2003). Using process algebra to describe human and software system behaviors. *Brain and Mind: A Transdisciplinary Journal of Neuroscience and Neurophilosophy*, 4(2), 199-213.
- Wang, Y. (2006a, March). On the informatics laws and deductive semantics of software. *IEEE Transactions on Systems, Man, and Cybernetics (Part C)*, 36(2), 161-171.
- Wang, Y. (2006b, May 8-10). A unified mathematical model of programs. In *Proceedings of the 19th Canadian Conference on Electrical and Computer Engineering (CCECE'06)*, Ottawa, ON, Canada, (pp. 2346-2349).
- Wang, Y. (2006c, July). Cognitive informatics and contemporary mathematics for knowledge representation and manipulation, invited plenary talk. In *Proceedings of the 1st International Conference on Rough Set and Knowledge Technology (RSKT'06)*,

- Chongqing, China, (pp. 69-78). Lecture Notes in Artificial Intelligence, LNAI 4062: Springer-Verlag.
- Wang, Y. (2007a). Software engineering foundations: A software science perspective. *CRC book series on software engineering* (Vol. II). Boca Raton, FL: CRC Press.
- Wang, Y. (2007b). Keynote speech: On theoretical foundations of software engineering and denotational mathematics. In *Proceedings of the 5th Asian Workshop on Foundations of Software*, Xiamen, China, (pp. 99-102).
- Wilson, L. B., & Clark, R. G. (1988). *Comparative programming language*. Wokingham, UK: Addison-Wesley Publishing.
- Wirth, N. (1976). *Algorithms + data structures = programs*. Englewood Cliffs, NJ: Prentice Hall.

Yingxu Wang is professor of cognitive informatics and software engineering, director of the International Center for Cognitive Informatics (ICfCI), and director of the Theoretical and Empirical Software Engineering Research Center (TESERC) at the University of Calgary. 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 was a visiting professor in the computing laboratory at Oxford University during 1995, and has been a full professor since 1994. He is editor-in-chief of International Journal of Cognitive Informatics and Natural Intelligence (IJCINI), editor-in-chief of the IGI book series of Advances in Cognitive Informatics and Natural Intelligence, and editor of CRC book series in Software Engineering. He has published over 300 papers and 10 books in software engineering and cognitive informatics, and won dozens of research achievement, best paper, and teaching awards in the last 28 years, particularly the IBC 21st Century Award for Achievement "in recognition of outstanding contribution in the field of Cognitive Informatics and Software Science."