

An Implementation of the π -Calculus on the .NET

Liwu Li, University of Windsor, Canada

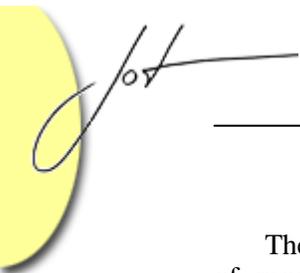
Abstract

Here, we show how to refine the grammar of the π -calculus so that we can implement the π -calculus by decomposing a given process expression into a hierarchy whose nodes are responsible to communicate links and execute the basic actions specified in the process expression. The nodes in the hierarchy add new child nodes when they need to handle lower-level process expressions. They are removed from the hierarchy when the processes assigned to them are completed. Hence, the hierarchy for executing a process expression of the π -calculus grows and shrinks dynamically at runtime. We implement the nodes with concurrently executing threads in the managed C++ on Microsoft .NET. The communication requests from the nodes are coordinated and resolved with a class named `CommunCenter`. Thus, we present an operational semantics for the π -calculus and implement the semantics in the .NET.

1 INTRODUCTION

The π -calculus, introduced by Milner et al. [Milner et al. 1989, Milner et al. 1992], models the changing connectivity of a communication system, in which links to sites are sent between sites and used by the sites for communication. It has been studied extensively as a theoretical model and shows a powerful capacity in modeling data structures, mobile communication systems, and other real-world systems [Milner 1999, Sangiorgi and Walker 2001].

Here, we present an approach to implement the π -calculus. In the approach, we realize a π -calculus process expression by decomposing the expression into a hierarchy, whose nodes manage, coordinate, and perform the basic actions that are specified in the process expression. When the process expression is executed, new nodes are added to the hierarchy to account for lower-level process expressions. Existing nodes in the hierarchy whose tasks are completed are removed from the hierarchy. Thus, the execution of the π -calculus expression is realized and visualized with a dynamically changing hierarchy. To implement the approach, we associate a node of the hierarchy with an execution thread in the managed C++ language on the Microsoft .NET Framework and realize the hierarchy with the communicating concurrent threads. A class named `CommunCenter` is used to match input and output requests from the threads. The approach represents a feasible framework for a full-fledged distributed system that is based on the π -calculus to specify the dynamically changing connectivity of the components in the system.



The π -calculus is commonly regarded as an algebraic language. It uses algebraic expressions of processes to specify the communication tasks to be realized by the components in a communication system and use operators to manipulate the algebraic expressions. Our approach to implementing the π -calculus is based on a reformulation of the syntactic rules of the π -calculus so that we can unambiguously decompose a process expression into a hierarchical structure, which allows a natural divide-conquer style in implementing the π -calculus with the .NET execution threads. This approach is also different from the author's another paper to be published in Journal of Object Technology [Li 2005]. Here, we reformulate the syntax of the π -calculus and represent a given process expression with a dynamically changing hierarchy of nodes. The nodes can be realized with concurrently communicating threads and the execution of a process expression by a node may dynamically dispatch execution threads. Thus, a communication system that executes a π -calculus process expression can be realized with a distributed system, whose components are dynamically changed, added, and removed. The work presented in [Li 2005] transforms a π -calculus process expression into a Java program, which is to be interpreted by the Java virtual machine.

This paper is organized as follows. We introduce the π -calculus and reformulate the syntactic and semantic rules of the π -calculus in Section 2. Based on the reformulated syntax of the π -calculus, we present an ASCII language for coding process expressions of the π -calculus and decomposing the coded expressions into hierarchies in Section 3. Based on the hierarchical representation of a given process expression, we describe an operational semantics for handling the hierarchies in order to execute the π -calculus process expression in Section 3. We describe the concurrency mechanism supported by the .NET in Section 4 and specify how to apply the mechanism to implement the π -calculus in Section 5. The paper is concluded in Section 6.

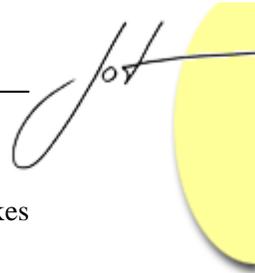
In the following presentation, **blue font** is used to present source code, **brown font** presents syntactic constructs appeared in grammar rules, and *green italic font* shows non-terminal symbols in grammar rules and in running text. Particularly, a pair of braces followed by an asterisk `{ }`* is used in grammar rules to enclose a syntactic component that may be repeated zero or multiple times. The source code files of the .NET implementation of the presented approach to realizing the π -calculus process expressions are compressed into a zip file, which is attached with this submission. The executable code of the implementation for Windows is also attached with the submission. Its usage will be described in Section 5.

2 THE π -CALCULUS

As a computational apparatus, the π -calculus has syntactic and semantic rules. Here, we briefly introduce the grammar and the operational semantics of the π -calculus based on the monograph [Sangiorgi and Walker 2001]. Then, we reformulate the syntax of the π -calculus for the purpose of an implementation of the π -calculus. Detailed elaboration of the π -calculus as an algebraic notation can be found in the monographs [Milner 1999, Sangiorgi and Walker 2001].

An Introduction to the π -Calculus

In the π -calculus [Milner 1999, p. 87], we can assume an infinite set \mathcal{N} of channel names, which are denoted by lowercase letters x, y, z, \dots with possible subscripts. Process expressions in the



π -calculus are composed of basic actions, summations, and processes. A (*basic*) *action* π takes one of the following forms [Sangiorgi and Walker 2001, p. 11]:

$\pi ::= x(\bar{y})$	receive names from channel x through parameters $\bar{y} = y_1, \dots, y_k$
$\bar{x}\langle\bar{z}\rangle$	send $m \geq 0$ channel names $\bar{z} = z_1, \dots, z_m$ into channel x
τ	execute an unobservable internal (silent) action
$[x = y]\pi$	enable action π if names x and y are same

A *summation* M takes one of the forms:

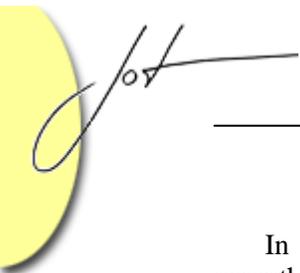
$M ::= 0$	do nothing (an inaction process)
$\pi.P$	execute action π and proceed to process P
$M_1 + M_2$	choose one of the summations M_1 and M_2 to execute

A *process* P takes one of the forms

$P ::= M$	execute summation M
$P_1 P_2$	execute processes P_1 and P_2 concurrently
$\nu \bar{z} P$	declare $n \geq 1$ channel names $\bar{z} = z_1, \dots, z_n$ for process P
$!P$	supply (an infinite number of) copies of process P

In the above syntactic rules, the π -calculus uses operators ‘.’, ‘+’, ‘|’, and ‘!’ to compose process expressions. With the dot operator ‘.’ as a process constructor, a process expression $\pi.P$ schedules a sequential execution of a basic action π and a process P so that the action π must be completed successfully before the process P can be started. We say the process P is *guarded* by the action π [Milner 1999, p. 87]. The addition operator ‘+’ in process expression $M_1 + M_2$ is used to separate the alternative processes M_1 and M_2 so that an execution of one of the two processes renders the other void. Due to the precedence of composition operator ‘|’ over addition operator ‘+’, neither of the process addends M_1 and M_2 in the choice expression $M_1 + M_2$ can show a top-level occurrence of the operator ‘|’. The composition operator ‘|’ in process expression $P_1 | P_2$ is used to dictate a parallel execution of the processes P_1 and P_2 . The repeat operator ‘!’ in process expression $!P$ indicates that copies of the process P can be supplied on demand. Expression $!P$ is structurally equivalent to the composite process expression $P | !P$ [Sangiorgi and Walker 2001, p. 20].

The π -calculus uses symbol 0 to denote a nullifying process or a null, which does nothing but terminating the execution of the process that encounters the null 0. In the π -calculus, the null 0 is used as a placeholder symbol to indicate positions in *process contexts*, which are used as templates to generate process expressions [Sangiorgi and Walker 2001, p. 19]. The above syntactic rules allow an occurrence of the null 0 to be followed by any basic actions as well as by an operator ‘+’ or ‘|’. The execution of a process expression can terminate naturally when it exhausts the actions specified in the expression, and action expressions that follow null 0 will be ignored in execution. We could safely ignore occurrences of the nullifying process 0 from process expressions [Sangiorgi and Walker 2001, p. 20] and simplify a process expression such as $\nu w \bar{x}\langle w \rangle.0$ to $\nu w \bar{x}\langle w \rangle$.



In the π -calculus [Milner 1999, p. 88; Sangiorgi and Walker 2001, p. 16], a pair of parentheses can be used to enclose a process expression to clarify the scope of an operator such as '+', '|', or '.'. For a valid process expression R , which must be terminated with null 0, we regard the expression (R) as a basic action. An occurrence of an operator such as '+', '|', or '.' inside (R) is an embedded but not top-level occurrence of the operator with respect to the process expression P that includes the basic action expression (R) . A channel name x declared in the subexpression (R) in process expression P hides the same channel name x that is declared in P prior to the basic action (R) .

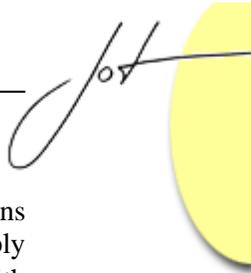
In the π -calculus, a *prefix* of a process expression is a ν -prefix $\nu \bar{z}$ in process expression $\nu \bar{z} P$, an action-prefix π in $\pi.P$, or a match-prefix $[x = y]$ in $[x = y] \pi.P$. By convention [Sangiorgi and Walker 2001, p. 16], a prefix has a higher precedence than operators '+' and '|'. Hence, the scope of the restriction ν -prefix $\nu \bar{z}$ in process $\nu \bar{z} P$ is not extended over any top-level occurrence of operator '+' or '|' in the process expression P . The choice operator '+' has a higher precedence than the composition operator '|'. For example, in the process expression $\nu z \bar{x}\langle z \rangle.(\nu z \bar{x}\langle z \rangle.0).z(\nu).0 + \bar{z}(\nu).0$, the first ν -prefix νz restricts the second and the fifth occurrence of the name z , the second ν -prefix νz restricts the fourth occurrence of the name z , and the last occurrence of the name z is a free name in the whole process expression.

If we regard the above set of syntactic rules as a grammar Θ with P as its goal, the grammar is neither LR(1) nor LL(1) as defined in [Aho et al. 1985] but ambiguous. For example, the π -calculus process expression $\nu z \bar{x}\langle z \rangle.0 | x(y).0$ can be derived from the goal P through two different top-down leftmost derivations:

$$\begin{aligned}
P &\rightarrow P_1 | P_2 \rightarrow \nu z P_3 | P_2 \rightarrow \nu z \pi.P_5 | P_2 \rightarrow \nu z \bar{x}\langle z \rangle.P_5 | P_2 \rightarrow \nu z \bar{x}\langle z \rangle.M_1 | P_2 \\
&\rightarrow \nu z \bar{x}\langle z \rangle.0 | P_2 \rightarrow \nu z \bar{x}\langle z \rangle.0 | M_2 \rightarrow \nu z \bar{x}\langle z \rangle.0 | \pi.P_6 \\
&\rightarrow \nu z \bar{x}\langle z \rangle.0 | x(y).P_6 \rightarrow \nu z \bar{x}\langle z \rangle.0 | x(y).M_3 \rightarrow \nu z \bar{x}\langle z \rangle.0 | x(y).0 \\
P &\rightarrow \nu z P_4 \rightarrow \nu z P_3 | P_2 \rightarrow \dots \text{ (Omitted derivation is same as the above derivation.)}
\end{aligned}$$

A Reformulation of the π -Calculus

Following the above presentation of the syntactic and semantic specification of the π -calculus, we can characterize π -calculus process expressions by introducing several notions. We say that an occurrence of an operator '+' or '|' in a process expression is a *top-level* occurrence if it is not enclosed within any pair of parentheses; otherwise, it is an *embedded* occurrence. A *choice* process expression is a process expression that does not contain any top-level occurrence of the composite operator '|'. A *sequential* process expression is a process expression that does not contain any top-level occurrence of the operator '|' or '+'. We extend the syntactic rules of the π -calculus recursively with a basic action expression (R) for any process expression R . We regard the expression (R) as an instance of the symbol π in the above syntactic rules. Note that the basic action expression (R) is not a complete process expression in the π -calculus since it is not ended with an inaction symbol 0.



We shall use the following recursive syntactic specification of π -calculus process expressions to construct and parse the process expressions. Here, the term *basic action expression*, or simply action expression, represented with symbol A denotes inaction 0 , an expression as denoted with the symbol π in the above syntactic rules, or an expression (R) for a process expression R .

S1. A *composite* process expression P consists of one or more choice process expressions M that are separated with the composite operator '|'; i.e., the syntax of a composite process expression is defined with rule

$$P ::= M \{ | M \}^*$$

S2. A *choice* process expression M consists of one or more sequential process expressions S that are separated with the choice operator '+'; i.e.,

$$M ::= S \{ + S \}^*$$

S3. A *sequential* process expression S consists of basic actions that are separated with the dot operator '.' and each of which is possibly prefixed with a sequence of matches, restrictions, and/or repeat symbols. It is ended with a null action 0 . In the following rules, we use symbol E to denote a match, and D a restriction.

$$\begin{aligned} S &::= \{ B . \}^* 0 \\ B &::= \{ E | D | ! \}^* A \\ A &::= 0 | \pi | (P) \end{aligned}$$

We shall use the term a *binding expression* γ to refer to either an expression $x(\bar{y})$, which consists of an input action $x(\bar{y})$ followed by the prefixing operator '.', or a ν -prefix (restriction) $\nu \bar{y}$. The binding expression γ , which is either $x(\bar{y})$ or $\nu \bar{y}$, in a sequential process expression γS binds (the scope of) the channel names \bar{y} to the sequential process S and we call γ a binding prefix of the sequential process γS . For simplicity of presentation [Sangiorgi and Walker 2001, p. 47, Convention 1.4.10], we require that if the sequential process expression γS includes a binding expression γ' inside S , the channel names bound by γ' be different from those bound by γ . In a sequential process expression, names that are not bound by any binding expression are *free*. We require the bound names in a sequential process expression be different from the free names in the expression. For example, the sequential process expression $\nu z \bar{x}\langle z \rangle.(\nu z \bar{x}\langle z \rangle.0).z(\nu).0$ does not satisfy the first requirement. In the π -calculus [Sangiorgi and Walker 2001, p. 15], two process expressions P and Q are α -convertible if we can transform one of them to the other by changing bound names, and α -convertible process expressions are regarded as equal. For example, we can transform the sequential process expression $\nu z \bar{x}\langle z \rangle.(\nu z \bar{x}\langle z \rangle.0).z(\nu).0$ to an equal sequential process expression $\nu z \bar{x}\langle z \rangle.(\nu y \bar{x}\langle y \rangle.0).z(\nu).0$ by applying α -conversion.

The π -calculus realizes a communication between a pair of input and output processes with a substitution of the output arguments received from the output process for the input parameters used in the input process. A *substitution* is a function $\sigma: \mathcal{N} \rightarrow \mathcal{N}$ that maps names to names and that is the identity except on a finite set of names. If a substitution σ maps names \bar{a} to names \bar{b} ,

the mapping can be denoted with $\sigma \vec{a} = \vec{b}$. Two sequential processes $\bar{x}(\vec{y}).S_1$ and $x(\vec{z}).S_2$ can *communicate* if and only if the number of output arguments \vec{y} equals the number of input parameters \vec{z} and the mapping that maps each parameter z_i in the list \vec{z} to the corresponding argument y_i in the list \vec{y} can be extended to a substitution σ , which can be the identity on names that are not in list \vec{z} . The communication will transform the processes $\bar{x}(\vec{y}).S_1$ and $x(\vec{z}).S_2$ to processes S_1 and $S_2 \sigma$, respectively. The expression $S_2 \sigma$ denotes a process expression resulting from S_2 by replacing occurrences of parameters z_i with corresponding arguments σz_i . Thus, the channel names \vec{y} are passed from a process $\bar{x}(\vec{y}).S_1$ to another process S_2 .

Assume a choice process expression $\tau.S + M'$. After the sequential process expression $\tau.S$ in the choice process expression is chosen to execute, the front internal action τ is executed and the choice process is reduced to the sequential process S . Thus, executing the silent action τ in the first subprocess $\tau.S$ in the given choice process expression $\tau.S + M'$ renders the alternative process M' void.

3 REALIZING π -CALCULUS PROCESSES

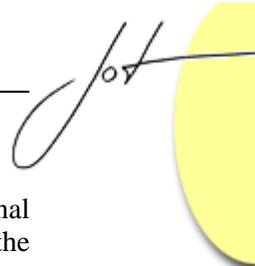
Process Expressions

To code π -calculus processes in ASCII, we use keywords `in`, `out`, and `tao` to signify the operational semantics of input, output, and internal actions. We use keyword `new` to introduce channel names declared by a ν -prefix restriction. The language for coding π -calculus processes in ASCII is specified with the following grammar, which follows the reformulated syntax of the π -calculus and which has the goal *composite_process* for producing process expression.

```

<composite_process> ::= <choice_process> | <composite_process> |
<choice_process> ::= <sequential_process> + <choice_process> |
<sequential_process> ::= <action> . <sequential_process> |
new <identifier> <names> <sequential_process> |
[ <identifier> = <identifier> ] <sequential_process> |
! <sequential_process> |
0
<action> ::= in <identifier> <names> |
out <identifier> <names> |
tao |
( <composite_process> ) |
0
<names> ::= <identifier> <names> |
ε

```



Note that due to the occurrences of the null 0 in both the production rules for non-terminal symbols *sequential_process* and *action*, the above grammar is still ambiguous. However, the grammar allows us to distinguish *composite_process*, *choice_process*, and *sequential_process* expressions uniquely in a given process expression. In the above grammar, non-terminal *composite_process* corresponds to a π -calculus process P , which may show top-level occurrences of operators ‘|’ and ‘+’. The top-level occurrences of the composite operator ‘|’ in the *composite_process* expression are used to decompose the *composite_process* expression into choice processes, which are represented with the non-terminal *choice_process* and which may show top-level occurrences of operator ‘+’ but not ‘|’. The top-level occurrences of the choice operator ‘+’ in a *choice_process* expression decompose the *choice_process* expression into sequential processes, which are represented with the non-terminal *sequential_process*. A *sequential_process* expression does not include any top-level occurrence of operator ‘|’ or ‘+’. It must be ended with an inaction 0.

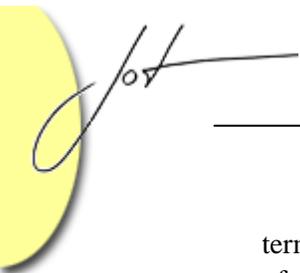
In the above grammar, the symbol ε denotes an empty sequence of identifiers. Keyword **new** in a *sequential_process* expression represents a ν -prefix by declaring channel names, which are denoted with $\langle \text{identifier} \rangle \langle \text{names} \rangle$. We call an expression that is composed of the keyword **new** and one or more channel names a *new-term*, an expression in form $[\langle \text{identifier} \rangle = \langle \text{identifier} \rangle]$ a *match-term*, and an expression **!**, which consists of the single character ‘!’, a *repeat-term*. We use the notion of a *term* to denote a new-, match-, or repeat-term.

The keyword **in** in an expression **in** x y_1 ... y_k with $k \geq 0$ introduces an input action $x(\bar{y})$ with bound variables $\bar{y} = y_1 \dots y_k$. The keyword **out** in an expression **out** x z_1 ... z_k introduces an output action $\bar{x}\langle \bar{z} \rangle$. Keyword **tao** denotes an internal action τ . A composite action (R) in the π -calculus is represented with $(\langle \text{composite_process} \rangle)$ in the above grammar. We regard the inaction 0 as a special type of action and use the term *action* to refer to an input, output, silent, composite action, or an inaction. A *sequential_process* expression consists of a sequence of actions that are separated with the dot operator ‘.’ and is ended with an inaction 0. Each of the actions in a *sequential_process* expression may be prefixed with a sequence of terms. For example, we can code the input and output actions $x(y \ x_2 \ x_1)$ and $\bar{x}\langle x_1 \ x_2 \ y \rangle$ with action expressions **in** x y x_2 x_1 and **out** x x_1 x_2 y , respectively. The π -calculus process expression $\nu y \bar{x}\langle y \rangle.0$ can be coded in *sequential_process* expression **new** y **out** x y .0.

Process Hierarchy

Given a process expression P coded in the above ASCII language, we construct a hierarchy, denoted as $\mathfrak{S}(P)$ or simply \mathfrak{S} , to represent the expression P as follows:

- H0. The given expression P is regarded as a *composite_process* expression and denoted with the root node of \mathfrak{S} .
- H1. For each node ρ in \mathfrak{S} that denotes a *composite_process* expression Q which is not 0, we denote each of the *choice_process* expressions M in the expression Q with a child node ρ' of ρ in \mathfrak{S} .
- H2. For each node ρ' in \mathfrak{S} that denotes a *choice_process* expression M , we denote each of the *sequential_process* expressions S in the expression M with a child node ρ'' of ρ' in \mathfrak{S} .
- H3. For each node ρ'' in \mathfrak{S} that represents a *sequential_process* expression S , we can handle the



terms in front of the first action in the sequential process S at runtime. (The runtime handling of the terms will be detailed after we specify the data structures of the nodes in hierarchy \mathfrak{T} .) Then, if the first action in S is a composite action (R), we add a unique child node for the node ρ'' to denote the *composite_process* expression R and apply the rule H1 recursively with Q equal to R ; otherwise, the node ρ'' is a leaf in \mathfrak{T} .

We can characterize the hierarchy \mathfrak{T} as follows:

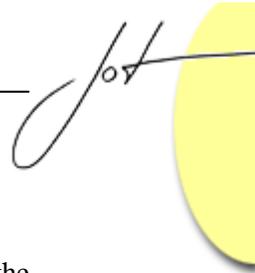
- T1. A *composite_process* node ρ has at least one *choice_process* child ρ' . A non-root *composite_process* node ρ that represents expression R has a unique *sequential_process* parent whose first action is (R).
- T2. A *choice_process* node ρ' has a unique *composite_process* parent and has at least one *sequential_process* child ρ'' .
- T3. A *sequential_process* node ρ'' has a unique *choice_process* parent and represents a *sequential_process* expression S . The node ρ'' is a leaf in \mathfrak{T} if the first action in S is not a composite action.

Each node in the hierarchy \mathfrak{T} keeps track of its parent and children. In addition to the parent-child relation, a *sequential_process* node ρ'' also holds the following data structures, which are modified at runtime.

- A list `procExpr` of characters for holding the *sequential_process* expression S to be executed by the node ρ'' ,
- A list `declaredNames` of channel names declared in new-terms that have been handled by the node ρ'' ,
- A collection `repeatedNodes` of *sequential_process* nodes, which are created by node ρ'' when the node handles repeat-terms, and
- A mapping `inputValue` that maps input parameters z to node-name pairs (ζ'', y) such that an input action performed by ρ'' has received the output argument y from another *sequential_process* node ζ'' through the input parameter z . The performance of input and output operations are specified in the following subsections.

The character list or string `procExpr` in ρ'' is initialized to a given *sequential_process* expression when the node ρ'' is created. The data structures `declaredNames`, `repeatedNodes`, and `inputValue` are initially empty when ρ'' is created.

When a *sequential_process* node ρ'' evaluates a match term $[n_1 = n_2]$, it needs to decide whether names n_1 and n_2 are free names, input parameters received or channel names declared by node ρ'' or by ancestors of ρ'' . We use symbol `inputValuePlus` at the *sequential_process* node ρ'' to denote a mapping that combines mapping `inputValue` at node ρ'' with the mappings `inputValue` at its *sequential_process* ancestors; i.e., value `inputValuesPlus(n)` at node ρ'' equals the value `inputValue(n)` at ρ'' if n is in the domain of the mapping `inputValue` at node ρ'' or equals `inputValue(n)` at lowest ancestor ζ'' of ρ'' if n is in the domain of the mapping `inputValue` at ζ'' . If the name n is free with respect to the whole hierarchy, we shall construct a value $(0, n)$, which consists of the pointer 0 and name n , for the function `inputValuesPlus(n)`.



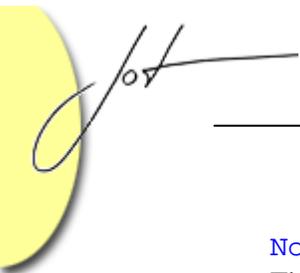
An Operational Semantics

When a process expression P is executed, we parse the expression P automatically through the steps H0-H4 to construct the hierarchy $\mathfrak{S}(P)$, which consists of *composite_process* nodes ρ , *choice_process* nodes ρ' , and *sequential_process* nodes ρ'' . The nodes collaborate to execute tasks required by the given process expression P . We specify the tasks for different types of node as follows.

- S1. A *composite_process* node ρ monitors its *choice_process* child nodes ρ' so that it is removed from the hierarchy \mathfrak{S} when all its children complete their tasks.
- S2. A *choice_process* node ρ' manages its *sequential_process* children. If any one of its children progresses by performing a basic action, the node ρ' removes its other *sequential_process* children and, thus, realizes a choice. The *choice_process* node is removed from the hierarchy \mathfrak{S} when it has no more children.
- S3. A *sequential_process* node ρ'' maintains a *sequential_process* expression `procExpr`. It handles the terms that are in front of the first action in `procExpr` and, then, processes the first action δ in the expression `procExpr` as follows:
 - If δ is a composite action (`R`), the hierarchy \mathfrak{S} is extended as described in Step H3 by adding a *composite_process* child node for the node ρ'' . The new child holds the *composite_process* expression `R`. The node ρ'' can progress only after its child completes the execution of the process `R` and is removed from the hierarchy \mathfrak{S} .
 - If δ is an input or output action, the node ρ'' is a leaf in hierarchy \mathfrak{S} and it requests the class `CommunCenter` to perform the input or output action. Node ρ'' continues its operation only after the class `CommunCenter` fulfils the input or output request. As described in Step S2, the continuation of the node ρ'' will request the *choice_process* ancestors of ρ'' to remove their *sequential_process* children except the *sequential_process* ancestors of node ρ'' .
 - If δ is a silent action `tao`, the node ρ'' progresses and requests its *choice_process* ancestors to remove their *sequential_process* children except the *sequential_process* ancestors of the node ρ'' .
 - If δ is inaction `0`, the node ρ'' is removed from the hierarchy \mathfrak{S} . As described in Steps S1 and S2, the removal of node ρ'' may trigger removal of its ancestors from \mathfrak{S} .

We now describe how a *sequential_process* node ρ'' handles the new-, match-, and repeat-terms $\varphi_1, \dots, \varphi_k$ with $k \geq 0$ that are in front of the first action δ in the *sequential_process* expression `procExpr`. The node ρ'' handles the terms $\varphi_1, \dots, \varphi_k$ by modifying the data structures `declaredNames`, `inputValue`, and `repeatedNodes` as follows. After handling a front term in the expression `procExpr`, node ρ'' removes the term from the expression `procExpr`. We shall use the notation $\rho''(n)$ for a name `n` to represent the value `inputValuesPlus(n)`.

- A new-term `new y1 ... yk` is handled by appending the declared channel names `y1, ..., yk` into the list `declaredNames` in node ρ'' .
- A repeat-term `!` is handled by creating a *sequential_process* node ζ'' such that the values in the data structures `procExpr`, `declaredNames`, and `inputValue` in the new node ζ'' are copied from the corresponding data structures in node ρ'' but the collection `repeatedNodes` in the new node ζ'' is empty. The new node ζ'' is inserted into the list `repeated-`



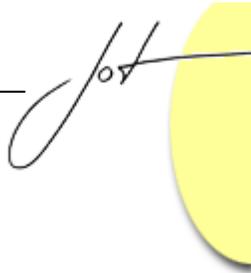
Nodes in node ρ ". It will be added into the hierarchy after an action in node ρ " is performed. Thus, we realize a π -calculus transformation from expression $!P$ to $P|!P$.

- A match-term $[n_1 = n_2]$ is handled by comparing values $\rho(n_1)$ and $\rho(n_2)$ of the names n_1 and n_2 . If the values $\rho(n_1)$ and $\rho(n_2)$ are equal, the node ρ " removes the match-term from the expression `procExpr` and continues executing the expression `procExpr`; otherwise, node ρ " terminates its execution and is removed from the hierarchy \mathfrak{S} .

The collection `repeatedNodes` holds *sequential_process* nodes that have no parent or child. Data structure `inputValue` is a dictionary that maps input parameters to output arguments received through input actions by the node ρ ". When the node ρ " uses input action `in x y1 ... yk` to communicate with an output action `out x z1 ... zk` of another node ζ ", the key-value pairs $(y_1, \zeta(z_1)), \dots, (y_k, \zeta(z_k))$ will be added into the dictionary `inputValue` in node ρ ". Thus, parameters y_1, \dots, y_k and arguments $\zeta(z_1), \dots, \zeta(z_k)$ values are related by the function `inputValue`. After the communication, nodes ρ " and ζ " discard their performed actions from their `procExpr` expressions and continue the processing of their remaining actions.

Process Communication

We use class `CommunCenter` to coordinate communication requests proposed by the concurrently executing *sequential_process* nodes in the hierarchy \mathfrak{S} . The class `CommunCenter` defines lists `inputRequests` and `outputRequests` to hold the input and output requests that cannot be fulfilled immediately. Particularly, when a *sequential_process* node ρ " submits an input request r_0 to the class `CommunCenter`, the class `CommunCenter` first searches the list `outputRequests` for an output request r_1 such that r_1 uses the same communication link as r_0 and the length of the parameter list of r_0 is equal to the length of the argument list of r_1 . If the search succeeds and the input and output requests are proposed by nodes n_0 and n_1 , respectively, the class `CommunCenter` allows the nodes n_0 and n_1 to realize their actions as follows; if the search fails, the class `CommunCenter` saves the input request r_0 into the `inputRequests` list. The node n_0 realizes its input action by extending the function `inputValue` so that the function `inputValue` maps the parameters in r_0 to the corresponding arguments in r_1 . As described in Step S3, the node n_0 also asks its *choice_process* ancestors to remove their *sequential_process* childrens except the ancestors of the node n_0 . Node n_1 also asks its *choice_process* ancestors to do the same thing. Similarly, when a *sequential_process* node ρ " submits an output request, the class `CommunCenter` searches the list `inputRequests` to decide if the output request can be communicated immediately. If the output request can be fulfilled for a saved input request, the output and input actions are performed by the *sequential_process* nodes that proposed the output and input requests, respectively; otherwise, the output request is placed into the list `outputRequests`.



4 MULTITHREADING IN THE .NET

We now introduce the multithreading mechanism of the .NET Framework for implementing the nodes in a hierarchy $\mathfrak{I}(\mathbf{P})$ for a π -calculus process expression \mathbf{P} . We also describe the standard .NET classes `Monitor` and `Mutex`, which are designed for thread synchronization. We shall use the facilities provided in the class `Monitor` to synchronize the execution threads that execute *composite_process* and *sequential_process* expressions. Accesses to the class `CommuncCenter` are also synchronized with the class `Monitor`. The class `Monitor` can be replaced with the class `Mutex` in the implementation of the π -calculus.

.NET Threads

In addition to its `main` thread, an application in the .NET can create one or more concurrent execution threads, which are represented with objects of the standard class `Thread`. To create a thread, we need a `ThreadStart` delegate to encapsulate a method that will be executed by the created thread. For example, the following statement creates a thread `cpThread`. The thread encapsulates a *composite_process* node referenced by variable `cp` and a named `processing` defined in class `CompositeProcess`.

```
Thread *cpThread = new Thread(new
    ThreadStart(cp, &CompositeProcess::processing));
```

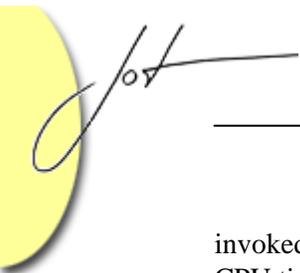
The method `processing` defined in class `CompositeProcess` performs the activities of a *composite_process* node as described in the operational semantics in Section 3. The above statement encapsulates an object of class `CompositeProcess` pointed at by the first argument `cp` of the constructor of the `ThreadStart` delegate. To encapsulate a static method in a `ThreadStart` object, the first argument of the constructor for the created `ThreadStart` delegate is the pointer `0`. We can use the standard method `GetHashCode` in expression `cpThread->GetHashCode()` to return a value for identifying the thread `cpThread`. In our implementation of the π -calculus, we also create threads to encapsulate objects of class `SequentialProcess` along with the instance method `processing` defined in the class `SequentialProcess`.

In the execution of an application, a thread is always in one or a combination of the states represented with values of the standard enumeration type `ThreadState`. The initial state of a newly created thread such as `cpThread` is `Unstarted`. After the instance method `Start()` of thread `cpThread` is executed in statement

```
cpThread->Start();
```

thread `cpThread` enters the `Running` state, in which the encapsulated method `processing` is executed concurrently with other threads.

When a running thread executes method `Sleep()`, `Wait()`, or `Join()`, the thread enters state `WaitSleepJoin` and its execution is blocked. We can use an integer argument `s` to invoke the static method `Sleep` of class `Thread` so that the current thread will be blocked for the number `s` of milliseconds before it regains the `Running` state. The method `Sleep()`



invoked in the following statement with integer 0 as argument just yields the rest of the thread's CPU time slice:

```
Thread::Sleep(0);
```

Methods `Wait()` and `Join()` specify a thread to wait for or join. For example, a running thread can execute the statement

```
cpThread->Join();
```

to enter state `WaitSleepJoin`. The thread will be in the `WaitSleepJoin` state until the target thread `cpThread` terminates.

A thread in state `WaitSleepJoin` leaves the state and reenters state `Running` when the sleep period expires, the waited or joined thread calls method `Pulse` or `PulseAll`, or another thread executes the function `Interrupt` for the thread.

A thread can invoke the method `Suspend()` or `Abort()` of another thread to request the other thread to enter a `Suspended` or `Stopped` state. A `Suspended` thread can reenter the `Running` state when another thread invokes its `Interrupt` method. A thread in the `Stopped` state can no longer execute and is subject to garbage collection.

Class Monitor

We can use the class `Monitor`, defined in namespace `System::Threading`, to synchronize multiple threads that access an object. A thread uses the static methods `Enter`, `TryEnter`, and `Exit` of class `Monitor` to get or release the lock of an object, which is pointed at by a method argument. For example, we shall use an object `cc` of the class `CommunCenter` to encapsulate the lists `inputRequests` and `outputRequests`, which are discussed in Section 3. A thread that implements a *sequential_process* node can get the exclusive lock of the object `cc` by executing the statement

```
Monitor::Enter(cc);
```

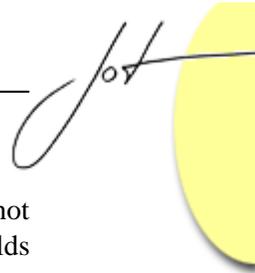
If the lock of the intended object `cc` is not possessed by another thread, the thread will hold the lock and continue its execution; otherwise, its execution is blocked until the lock is released by the other thread that is holding the mentioned lock. A thread releases the lock of object `cc` through statement

```
Monitor::Exit(cc);
```

The method `TryEnter` invoked in the `bool` expression

```
Monitor::TryEnter(cc)
```

does not block the current thread, which is evaluating the expression. Its evaluation returns either `true` or `false` immediately depending on whether or not the current thread can get the lock of object `cc`.



Note that an execution of the static method `Enter` by a thread may block the thread but not change the `Running` state of the thread to `WaitSleepJoin`. If a `Running` thread that holds the lock of object `cc` evaluates the static method `Wait` in statement

```
Monitor::Wait(cc);
```

it releases the lock and enters the state `WaitSleepJoin` for the lock. A thread in state `WaitSleepJoin` for the lock of object `cc` cannot get the lock unless another thread that is holding the lock invokes the static method `Pulse` or `PulseAll` to signal the thread and then releases the lock by evaluating the `Exit` method of class `Monitor`.

For example, the implementation of the π -calculus presented in this paper uses the lock of the object `cc` to synchronize the input and output requests issued by `SequentialProcess` objects. When the class `CommunCenter` receives an input request, it performs the statement

```
Monitor::Enter(cc);
```

to lock the object `cc`. Assume the class `CommunCenter` can find an output request that has been saved in the list `outputRequests` and that matches the input request. If the found output request was issued by a thread that encapsulates the `SequentialProcess` object `sp`, the class `CommunCenter` uses the following code to lock the object `sp`, change the `WaitSleepJoin` state of the thread to the `Running` state so that the thread can resume its computation.

```
Monitor::Enter(sp);  
Monitor::Pulse(sp);  
Monitor::Exit(sp);
```

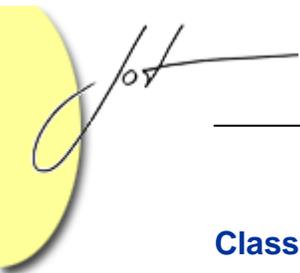
Then, the class `CommunCenter` executes the following statement to release the lock of object `cc` before it returns.

```
Monitor::Exit(cc);
```

If the class `CommunCenter` cannot find an output request saved in the list `outputRequests` to satisfy the input request, it saves the input request into the `inputRequests` list and executes the following code for the current thread that is executing the input request into the `WaitSleepJoin` state. Here, we assume the input request was generated from the `SequentialProcess` object `sp1`.

```
Monitor::Exit(cc);  
Monitor::Enter(sp1);  
Monitor::Wait(sp1);  
Monitor::Exit(sp1);
```

The class `CommunCenter` uses another static method to handle an output request from `SequentialProcess` objects. The execution thread of the static method also uses the lock of object `cc` to synchronize the `SequentialProcess` objects that issue input and output requests. We use the class `Monitor` to synchronize the activities of a `SequentialProcess` object and its child of class `CompositeProcess` so that only after the child completes its execution, the `SequentialProcess` parent can continue its execution.



Class Mutex

An object of class `Mutex`, defined in namespace `System::Threading`, can be used to synchronize threads that are created in the same process and in different processes. It can be used to enforce exclusive access to a resource. It does not provide all the wait and pulse methods as defined in the class `Monitor`. We can invoke methods `WaitOne`, `WaitAny`, and `WaitAll` defined in the class `Mutex` to wait for a `Mutex` object or for some `Mutex` objects to be available before the current thread can continue its execution. A thread releases a `Mutex` object by invoking the instance method `ReleaseMutex` of the `Mutex` object. When a thread finishes normally, it releases the `Mutex` objects that are held by it so that waiting threads can hold them. In the implementation of the π -calculus presented in this paper, we do not use the class `Mutex` to synchronize input and output actions. However, if the *composite_process* and *sequential_process* nodes in a hierarchy are implemented with operating system processes, we need to use the class `Mutex` to synchronize the input and output requests of the nodes.

For example, we can use the statement

```
Mutex* inputMut = new Mutex(true);
```

to create a new `Mutex` object `inputMut`. The constructor argument `true` indicates that the current thread that is executing the above statement holds or owns the newly created `Mutex` object. Another thread can compete for the ownership of the `Mutex` object with statement

```
inputMut->WaitOne();
```

A thread that holds the `inputMut` object can release it by executing statement

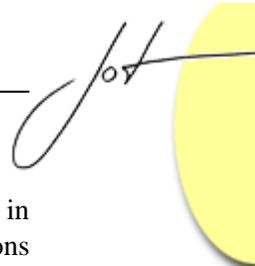
```
inputMut->ReleaseMutex();
```

5 THE .NET IMPLEMENTATION OF THE π -CALCULUS

Nodes and Threads

The hierarchy $\mathfrak{S}(P)$ for a given π -calculus process expression `P` is composed of objects of the three classes `CompositeProcess`, `ChoiceProcess`, and `SequentialProcess`, which implement the *composite_process*, *choice_process*, and *sequential_process* nodes, respectively. To maximize the concurrency of an execution of the π -calculus process, we encapsulate `CompositeProcess` and `SequentialProcess` objects in .NET threads, which are instances of the standard class `Thread`. The starting methods of the threads follow the responsibilities of the *composite_process* and *sequential_process* nodes as described in Section 3. They are detailed as follows.

Class `Node` is the common base of classes `CompositeProcess`, `ChoiceProcess`, and `SequentialProcess`. It introduces instance variables `parent` and `children` for a node to hold its parent and children, respectively, and member methods that manage the `parent` and `children` variables.



An object `cp` of the class `CompositeProcess` keeps the string `procExpr`, described in Section 3. The thread starting method of the object `cp` identifies the *choice_process* expressions from the string `procExpr` using top-level occurrences of the composite operator `|` in the string. For each of the *choice_process* expressions, an object `dp` of class `ChoiceProcess` is created to process the *choice_process* expression, which is used to initialize the instance variable `procExpr` of the created object `dp`. The thread starting method of the object `cp` adds the created object `dp` into the `children` list of `cp`.

The constructor of an object `dp` of the class `ChoiceProcess` divides the *choice_process* expression `procExpr` held in `dp` into *sequential_process* expressions, each of which is used to create an object `sp` of class `SequentialProcess`. The created object `sp` is assigned to a thread, which is responsible to process the actions specified in the *sequential_process* expression `procExpr`.

For example, assume the *composite_process* expression `(in x y.out y z.0).0 | out z.0 + out x a.in a w.0` is used to create a `CompositeProcess` object `cp0`. The thread starting method of object `cp0` will create two children `dp1` and `dp2` of class `ChoiceProcess` to handle the *choice_process* expressions `(in x y.out y z.0).0` and `out z.0 + out x a.in a w.0`, respectively. The object `dp1` will create an object `sp1` of class `SequentialProcess` to handle the *sequential_process* expression `(in x y.out y z.0).0`. The object `dp2` will create two objects `sp2` and `sp3` to handle the *sequential_process* expressions `out z.0` and `out x a.in a w.0`, respectively. When the thread starting method of the object `sp1` handles the composite action `(in x y.out y z.0)`, it create an object `cp1` of class `CompositeProcess` to handle the *composite_process* expression `in x y.out y z.0`. The object `cp1` creates an object `dp3` of class `ChoiceProcess`, which creates an object `sp4` of class `SequentialProcess` for processing the *sequential_process* expression `in x y.out y z.0`. In summary, a handling of the process expression `(in x y.out y z.0).0 | out z.0 + out x a.in a w.0` needs to create the above mentioned objects, which form the hierarchy shown in Fig. 5.1. Here, we introduce the object names such as `cp0` and `dp1` for illustration. The source code of the implementation of the π -calculus does not use the object names.

A `CompositeProcess` object `cp` such as the objects `cp0` and `cp1` shown in Fig. 5.1 holds its children, which are objects of class `ChoiceProcess`, with the instance variable `children`. If the list `children` becomes empty and the object `cp` is the root node of the hierarchy \mathfrak{S} , the execution is completed. If the list `children` in object `cp` becomes empty but the object `cp` is not the root, the node `cp` invokes the `removeChild` method of its parent object before it completes the execution specified by the thread starting method `processing` defined in class `CompositeProcess`.

Similar to a composite process object, a choice process object `dp` of class `ChoiceProcess` such as the node `dp1`, `dp2`, or `dp3` shown in Fig. 5.1 uses a list named `children` to hold its children, which are objects of class `SequentialProcess`. If the list `children` becomes empty, the object `dp` winds up its computation by invoking the `removeChild` method of its `CompositeProcess` parent to remove itself from the hierarchy.

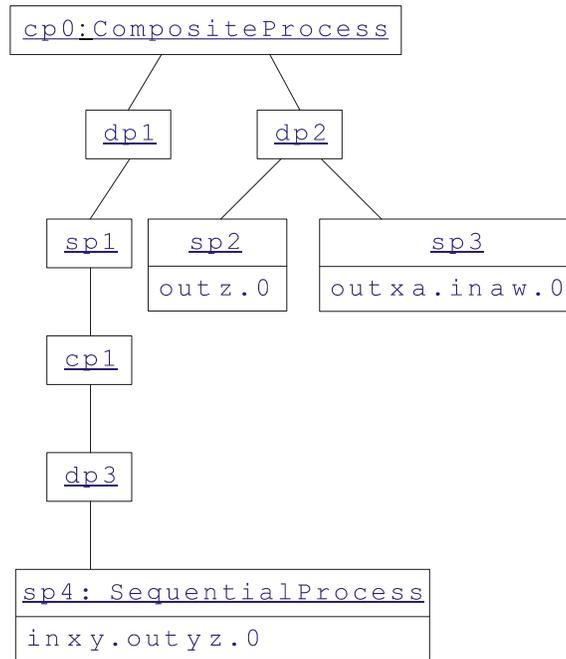
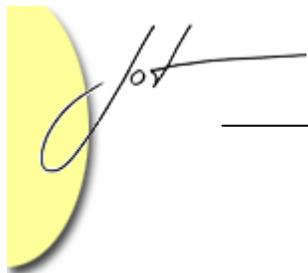
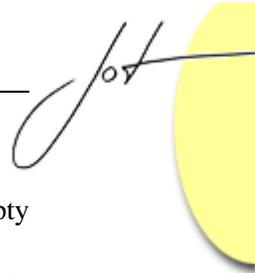


Figure 5.1 A hierarchy for executing a π -calculus process expression

An object `sp` of class `SequentialProcess` such as the node `sp1`, `sp2`, `sp3`, or `sp4` shown in Fig. 5.1 encapsulates methods for handling the different terms and basic actions specified in the *sequential_process* expression `procExpr` that is stored in the object `sp`. The instance methods are described as follows. We shall describe the communication between the `SequentialProcess` objects in the following subsection.

- The method `handle_new()` is invoked to handle a new-term that is in the front of the expression `procExpr`. It adds the names declared by the new-term into the `declaredNames` list.
- The method `handle_repeat()` handles a repeat term `!` by creating a new `SequentialProcess` object whose `declaredName` list and `inputValue` function has equal values as the same named data structures in the object `sp`.
- The method `handle_match()` handles a match-term `[n1 = n2]` by retrieving the values of the names `n1` and `n2` in the following way. The method tries to determine if the name `n1` is an input parameter in a lowest ancestor `sp'` of the node `sp` by searching the function `inputValue` in the ancestors of `sp`. If the search succeeds in a `SequentialProcess` object `sp'`, the desired value is equal to the value `inputValue(n1)`; otherwise, method `handle_match()` tries to determine if the name `n1` is a name declared in a lowest ancestor `sp'` of the node `sp` by searching the `declaredNames` list in the ancestors of `sp`. If the search succeeds in object `sp'`, the retrieved value is equal to `(sp', n1)`; otherwise, the name `n1` is free with respect to the whole hierarchy and the retrieved value is equal to `(0, n1)`. The method `handle_match()` retrieves a value for the other name `n2` similarly. If the two retrieved values have same components, the method `handle_match()` completes

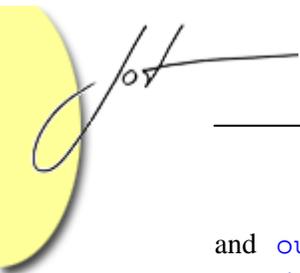


its evaluation successfully; otherwise, it changes the `procExpr` expression to the empty string `S` " " so that the execution of object `sp` can wind up.

- The method `performedAction()` is invoked after an input, output, or internal action is performed by the object `sp`. It simply requests the `ChoiceProcess` ancestors of the node `sp` to remove all the children but the `SequentialProcess` ancestors of the node `sp`. Thus, the method enforces the choice operations at the `ChoiceProcess` ancestor nodes of the node `sp`.
- The method `after_action()` is invoked after an input, output, or internal action is performed by the object `sp`. It installs the `SequentialProcess` nodes stored in the `repeatedNodes` list in the object `sp`. Particularly, for each object `sp'` in the `repeatedNodes` list, the method `after_action()` creates a `ChoiceProcess` object `dp'`, sets the `parent` field in `dp'` with the parent of parent of the node `sp`, and sets the `parent` field in object `sp'` with `dp'`. It also creates a thread for the object `sp'`. The method `after_action()` will empty the `repeatedNodes` list in the object `sp` after it adds all the nodes in the list to the hierarchy.
- The method `handle_inaction()` is invoked if the front action in the process expression `procExpr` in object `sp` is the inaction `0`. The method simply sets the `procExpr` variable in node `sp` with the empty string `S` " " and returns.
- The method `handle_tao()` is invoked to handle the front `tao` action in the `procExpr` expression in object `sp`. It simply removes the front action from `procExpr` and invokes the methods `performedAction()` and `after_action()` for object `sp`.
- The method `handle_input()` is invoked for object `sp` if the front action in expression `procExpr` in `sp` is an input action. It uses the communication link and parameters shown in the input action expression to invoke the `InputRequest()` method defined in class `CommunCenter`, which decides whether the input action can be honored immediately or should be saved in the `inputRequests` list in the object `cc`. After the `InputRequest()` method is executed, the methods `performedAction()` and `after_action()` of object `sp` are executed.
- The method `handle_output()` of object `sp` for handling the front output action in expression `procExpr` of object `sp` is similar to the above method `handle_input()`. It uses the communication link and arguments coded in the output action expression to invoke the `OutputRequest()` method defined in class `CommunCenter`, which decides whether the output action can be honored immediately or should be saved in the `outputRequests` list in the object `cc`.
- The method `handle_paren()` is invoked if the front action in the expression `procExpr` in object `sp` is a parenthesized process expression `(R)`. It creates a new object `cp` of class `CompositeProcess` to encapsulate the `composite_process` expression `R` and creates a thread to execute the `processing` method for the object `cp`. It then waits for the object `cp` to complete its execution by invoking the method `Join()` of `cp`.

Process Communication

We use the class `CommunCenter` to coordinate and synchronize the input and output requests issued by `SequentialProcess` objects. Particularly, the class uses a static variable `cc` to hold the unique object of the class `CommunCenter` for encapsulating the `inputRequests`



and `outputRequests` lists. The static methods `InputRequest()` and `OutputRequest()` defined in the class `CommuCenter` with signatures

```
void InputRequest(NodeName*, ArrayList*, SequentialProcess*);  
void OutputRequest(NodeName*, ArrayList*, SequentialProcess*);
```

are responsible to accept input and output requests from `SequentialProcess` objects. The first argument of the above methods denotes a communication link in the form $(sp, name)$ such that the pointer 0 for `sp` indicates `name` is free in the whole hierarchy and a valid pointer `sp` indicates `name` is declared in the node `sp` with a new-term. The second argument carries either the parameter list or the argument list. The third argument points to the `SequentialProcess` object that issued the input or output request.

The object `cc` not only stores the lists `inputRequests` and `outputRequests` but also is used to synchronize accesses to the lists by the `SequentialProcess` objects. Particularly, the above two static methods must request the lock of object `cc` before they can access or modify the lists. Thus, we keep the integrity of the lists.

The .NET Executable Code

The .NET implementation of the π -calculus is compiled to an application named `pi-calculator.exe` for Microsoft Windows. The application allows a user to enter a π -calculus process expression coded in the ASCII language presented in Section 3. For example, after starting the application with command

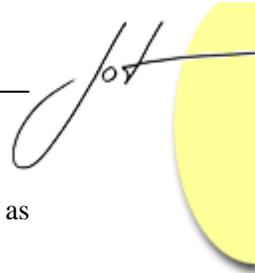
```
pi-calculator
```

we can enter the process expression $(in\ x\ y.out\ y\ z.0) \mid out\ z.0 + out\ x\ a.in\ a\ w.0$, which was displayed in Fig. 5.1. The application will display the following information to indicate the communications performed in the execution of the process:

```
A communication is realized through link: node free and name x:  
    Parameter name: y argument: node free name a;  
Communication is completed.  
A communication is realized through link: node free and name a:  
    Parameter name: w argument: node free name z;  
Communication is completed.
```

In the first sentence, the term `free` is used to indicate the channel name `x` is a free name with respect to the whole composite process expression. The term `free` is used similarly in the following sentences. The above display shows the second *sequential_process* expression `out x a.in a w.0` in the second *choice_process* expression `out z.0 + out x a.in a w.0` communicates an output request and an input request with the first *choice_process* expression. Thus, the first *sequential_process* expression `out z.0` is rendered void by the activity of the second.

The composite process expression $(in\ x\ y.out\ y\ z.0) \mid out\ z.0 + new\ a\ out\ x\ a.in\ a\ w.0$ introduces a new-term into the above *composite_process* expression. Its execution will display the following information, which indicates that the node 11 declared the name `a`.



The declared name `a` is communicated as an argument in the first communication. It is used as the communication link in the second communication.

A communication is realized through link: node `free` and name `x`:

Parameter name: `y` argument: node `l1` name `a`;

Communication is completed.

A communication is realized through link: node `l1` and name `a`:

Parameter name: `w` argument: node `free` name `z`;

Communication is completed.

6 CONCLUSIONS

The π -calculus proposed by Milner et al. [Milner et al. 1989, Milner et al. 1992] has been studied extensively as a theoretical model of mobile communication systems. Here, we reformulate the syntax of the π -calculus and describe how to implement the π -calculus based on the reformulated syntax. The operational semantics of the π -calculus based on the reformulated syntax is realized on the .NET Framework for the construction of a Windows application, which accepts π -calculus process expressions and executes them by dispatching threads.

The presented syntax and semantics of the π -calculus implies that we can realize π -calculus process expressions, which specify the behaviour of mobile communication systems, with distributed systems that are based on the CORBA or Web Services.

REFERENCES

- [Aho85] A.V. Aho, R. Sethi, and J. D. Ullman: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1985.
- [Arnold00] K. Arnold, J. Gosling, and D. Holmes: *The Java Programming Language – Third Edition*, Addison-Wesley, Reading, Massachusetts, 2000.
- [Li05] L. Li: “Implementing the π -Calculus in Java”, to appear in 2005 March/April issue in *Journal of Object Technology*, <http://www.jot.fm/>.
- [Milner89] R. Milner, J. Parrow, and D. Walker: “A Calculus of Mobile Processes”, Parts I and II. Technical Report ECS-LFCS-89-85 and -86, University of Edinburgh, 1989.
- [Milner92] R. Milner, J. Parrow, and D. Walker: “A Calculus of Mobile Processes”, Parts I and II. *Information and Computation*, Vol. 100(1), pp. 1-77, 1992.
- [Milner99] R. Milner: *Communication and Mobile Systems: The π -Calculus*, Cambridge University Press, Cambridge, UK, 1999.
- [Sangiorgi01] D. Sangiorgi and D. Walker: *The π -Calculus: A Theory of Mobile Processes*, Cambridge University Press, Cambridge, UK, 2001.

About the author

With great regret we report the passing of Liwu Li on April 21, 2005 at the age of 58.



Dr. Liwu Li was a professor in School of Computer Science at University of Windsor, Canada. His research interests include object-oriented language design and implementation, object-oriented software analysis and design, and software process design and execution.