JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2005**

Vol. 4, No. 2, March-April 2005

Implementing the π-Calculus in Java

Liwu Li, University of Windsor, Canada

Abstract

Mobile communicating systems are ubiquitous in the modern world. The π-calculus proposed by Milner et al [Milner et al. 1992] sets a theoretical foundation for modeling mobile communicating systems such as the Internet, in which links can be sent from processes to processes and a process can use a received link to interact with another process. Here, we present a language named π-language for programming π-calculus processes and a compiler named `pi2j` for translating the π-programs to Java code, which can be compiled with the Java compiler to JVM bytecode and, then, executed on the Java Virtual Machine. Thus, we implement the π-calculus in the Java language.

1 INTRODUCTION

The π-calculus was introduced by Milner et al. [Milner et al 1992] for modeling the changing connectivity inside mobile communicating systems, in which links between concurrently running processes can be passed from processes to processes and a process can use a received link to communicate with another process. It is similar to the λ-calculus as a theoretical model of sequential computation, the π-calculus can be used to model modern concurrent systems [Milner 1999; Sangiorgi and Walker 2001]. The π-calculus has aroused intensive interests in research to study its applicability, extensibility, and other properties. For example, it has been shown that the π-calculus is powerful enough to model various data structures, object-oriented programs, and communicating systems. The π-calculus is also the basis of several experimental programming languages such as Pict [Pierce and Turner 2000], Join [Fournet and Maranget 1997], and TyCO [Vasconcelos and Bastos 1998]. In this paper, we shall denote a π-calculus process expression in the ASCII directly by denoting Greek letters and typographical notations used in the π-calculus process expression with ASCII words. Then, we show how to translate the ASCII code of the π-calculus process expression to the Java language. Thus, we realize the dynamic communication mechanism of the π-calculus by invoking the Java multithreading mechanism.

We present an ASCII-based language called π-language for coding π-calculus process expressions. Using keywords in ASCII to denote the Greek leters and other non-

Cite this article as follows: Liwu Li: "Implementing the π-Calculus in Java", in *Journal of Object Technology*, vol. 4, no. 2, March-April 2005, pp. 157-177
http://www.jot.fm/issues/issue_2005_03/article5

ASCII symbols, the π-language represents π-calculus process expressions faithfully. Its grammar improves the syntactic rules of the π-calculus by eliminating ambiguities in the syntax rules of the π-calculus. We can compile the translated π-calculus process expressions into Java to explore the multithreading and synchronization mechanism of Java. We describe the Java concurrent classes that we developed to support the communication mechanism of the π-calculus. Based on the Java classes, we present a compiler named pi2j to translate the so-called π-programs into the Java language. Then, we can compile and run the Java code that implements π-calculus process expressions.

This paper is organized as follows. The next section introduces the π-calculus as specified in [Milner 1999; Sangiorgi and Walker 2001]. In Section 3, we briefly introduce the multithreading and synchronization constructs of the Java language, which are used to implement the π-calculus. We present the π-language for coding π-calculus process expressions in Section 4. We describe the various Java objects that support the dynamic communication mechanism of the π-calculus in Section 5. The compiler pi2j attached with the paper is described in Section 6. This paper is concluded in Section 7.

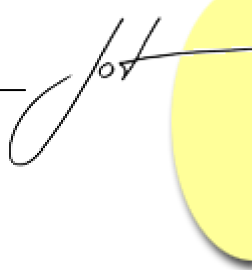
2 THE π-CALCULUS

The π-calculus is founded on three notions: name, (atomic) action, and process [Milner 1999, p. 87; Sangiorgi and Walker 2001, p. 11]. It assumes a countably infinite set N of names, which are denoted by lower case letters x, y, z, … with possible subscripts. A *name* can be thought of as the name or label of a communication link. It has no internal structure. In the π-calculus, a mobile communicating system is represented with a *process*, which may be composed of processes recursively and which performs *actions*. Processes use names to interact each other and pass names through the interactions.

An (atomic) action π in the π-calculus takes one of the following four forms [Sangiorgi and Walker 2001, p. 11]:

π	::=	$x(y)$	receives name y through name (link) x (parameter y is optional)
		$\overline{x}\langle z\rangle$	sends name z out via name x (argument z is optional)
		τ	performs an unobservable internal action
		$[x=y]\pi$	performs action π if names x and y are the same

(Only the first three atomic actions are listed in [Milner 1999, p. 11].) A process P takes one of the following four forms [Milner 1999, p. 87]:



P	::=	$\sum_{i \in I} \pi_i.P_i$	chooses one of alternative processes $\pi_i.P_i$ for a finite index set I
		$P_1 \mid P_2$	executes processes P_1 and P_2 concurrently
		new $z\ P$	declares a private (bound) name z for process P
		$!P$	supplies (an infinite number of) copies of process P

The dot symbol '.' is a process constructor in π-calculus expressions. A process expression $P.Q$ schedules a sequential execution of the processes P and Q; i.e., process Q can proceed only after process P is exercised. For instance, an addend $\pi_i.P_i$ in the summation $\sum_{i \in I} \pi_i.P_i$, indicates that action π_i must be completed before process P_i can start. We say that process P_i is guarded by action π_i [Milner 1999, p. 87].

In the summation expression $\sum_{i \in I} \pi_i.P_i$, binary operator '+' is used to connect the addends $\pi_i.P_i$ for $i \in I$. The choice and execution of action prefix π_i of a term $\pi_i.P_i$ in the summation renders other terms $\pi_j.P_j$ with $j \neq i$ void [Sangiorgi and Walker 2001]. If the index set I is empty, we denote the summation $\sum_{i \in I} \pi_i.P_i$ with zero symbol 0, which does nothing and is called an inaction [Sangiorgi and Walker 2001, p. 12]. The expression 0 is also used in the π-calculus to terminate a special sequence of dot-separated actions [Sangiorgi and Walker 2001].

A composition expression $P_1 \mid P_2$ indicates that the component processes P_1 and P_2 can proceed independently. The composition operator '$\mid$' is commutative and associative [Sangiorgi and Walker 2001, p. 20]. The component processes P_1 and P_2 may communicate via a (channel) name x if one of them has an input action $x(y)$ as prefix and the other has an output $\overline{x}\langle z \rangle$ as prefix for some names x, y, and z. The communication inside the process $P = P_1 \mid P_2$ is regarded as an unobservable internal action in process P. Component processes P_1 and P_2 can communicate if their action prefixes are x and $\overline{x}$, respectively, for some name x.

For example, process expression $x(y).\overline{y}\langle z \rangle \mid \overline{x}\langle w \rangle.w(x)$ can be transmitted to a process $\overline{y}\langle z \rangle\{w/y\} \mid w(x)$ by an internal communication via the name x. The substitution $\{w/y\}$ applied to expression $\overline{y}\langle z \rangle$ requires replace the free occurrences of name y in process $\overline{y}\langle z \rangle$ with name w. The resulting process is congruent to process $\overline{w}\langle z \rangle \mid w(x)$,

which permits another internal communication action via the name w. Particularly, the two components in process $\overline{w}\langle z\rangle | w(x)$ can communicate via name w and the process will proceed to $0|0=0$. Thus, the process $x(y).\overline{y}\langle z\rangle | \overline{x}\langle w\rangle.w(x)$ executes.

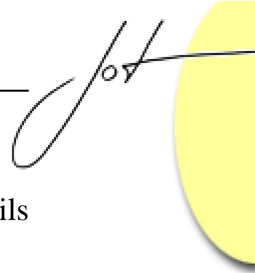
In the above syntax rule for processes, the new-prefix $new\ z$ in the process expression $new\ z\ P$ introduces a private (local) bound name z for the process P. By the syntax rule, all the free occurences of name z in the process expression P refer to the name z declared by the new-prefix $new\ z$. For example, both the third and fourth occurrences of the name z in process expression $z(y).new\ z\ \overline{x}\langle z\rangle.z(v)$ denote the bound name z declared in the new-prefix inside the expression. They are different from the first occurrence of the name z, which is a free occurrence of name z in the process expression. To avoid accidental capture of bound names, α-conversion can be applied to rename bound names and input parameters [Sangiorgi and Walker 2001, p. 15]. We can use a new-prefix in the form $new\ z_1 \ldots z_k$ to abbreviate a series of new prefixes $new\ z_1 \ldots new\ z_k$. The new-prefix introduces multiple bound names $z_1,\ldots,z_k$ with k ≥ 1. In [Sangiorgi and Walker 2001], Greek letter ν is used for the keyword new.

A replication expression $!P$ is composed of the exclamation symbol '$!$' and a process expression P. It provides the power of a parametric recursive process [Milner 1999, p. 88]. It is equivalent to the composition $P|!P$. In the following discussion, we shall not shrink a process expression $P|!P$ to the expression $!P$. The expression $!P$ is expanded to the expression $P|!P$ only when the component P may communicate with another process.

In the π-calculus [Sangiorgi and Walker 2001, p. 15], prefixing operator '$\cdot$', operator new, condition $[x=y]$, and replication operator '$!$' bind more tightly than composition '$|$', and prefixing binds more tightly than the summation operator '$+$'. A pair of parentheses can be used to enclose a process expression P for creating a scope. For example, we can represent process expression $x(y).\overline{y}\langle z\rangle | \overline{x}\langle w\rangle.w(x)$ equivalently as $(x(y).\overline{y}\langle z\rangle) | (\overline{x}\langle w\rangle.w(x))$.

3 MULTITHREADING IN JAVA

A thread of execution is denoted in a Java program with a *thread* object, which is an instance of the standard class `Thread`. The Java Virtual Machine allows a program to spawn multiple threads, which run concurrently. The Java language defines a synchronization mechanism for programmers to prevent concurrent threads from interfering with each other [Arnold et al 2000]. In addition to synchronization, Java provides methods `wait`, `notify`, and `notifyAll` in the standard class `Object` to support communication between threads [Arnold et al 2000, p. 244]. For self-containment,



we briefly introduce thread creation, synchronization, and communication. More details on thread can be found in [Arnold et al 2000, Chapter 10].

Thread Creation

There are two ways to create a thread in a Java program. First, a subclass of the class `Thread` can be instantiated to create threads. In the subclass, we override method `run` of class `Thread` with specific functionality for the threads, which run concurrently with other threads.

For example, we shall use an object p of class Process to implement a π-calculus process. The class Process inherits class Thread. In subclass Process, we override method run of class Thread with operations that realize the atomic actions specified in the π-calculus process. The following Java statement can be used to start the running method run of object p:

```
p.start();
```

The other way to create a thread is to declare a class that implements the standard interface `Runnable`. The class should implement method `run` to specify functionality. We can instantiate the class to create an instance `r`, instantiate the class `Thread` with the `Runnable` object `r` as the constructor argument, and start to run the created `Thread` object. Thus, method `run` of object `r` is executed and the specified functionality is realized. For example, since class `Thread` implements interface `Runnable`, the above Java statement can be replaced with Java statement

```
new Thread(p).start();
```

Thread Synchronization

Java keyword `synchronized` can be used to qualify a method or statement. Each object has a lock, which is acquired implicitly through the call of a `synchronized` method of the object and explicitly through the execution of a `synchronized` statement [Arnold et al 2000, p. 235]. In a Java program, we can synchronize a set of threads by following the protocol that before operating on an object, each of the threads acquires the object's lock. Thus, only one of the threads can hold the lock and proceed; others are blocked by the lock. The thread holding the object's lock can complete an atomic transaction before it releases the lock.

When a thread invokes a `synchronized` method on an object, the thread acquires the object's lock. If the thread already possesses the lock, it will not be blocked and the method execution proceeds. If the acquired lock is being held by a different thread, the thread is blocked until the lock is released by the other thread. When an object's lock is released, there may be several threads blocked by the lock. An object's lock is released by a thread automatically when method `run` of the thread returns. A `static synchronized` method of a class acquires the class object's lock when it is invoked. For example, in the implementation of the compiler `pi2j`, we define the following `static` method `newPrefix()` in class `Blackboard` for handling new-prefixes,

which are denoted by the parameter np. An invocation of the method newPrefix acquires the lock of the class Blackboard.

```
public static synchronized void newPrefix(NewPrefix np)
{
      RepeatedSequence rseq = np.seq0.parent;
      rseq.addBoundVars(np.names);
      np.removeFlag();
}
```

A synchronized statement explicitly specifies an object for acquiring the object's lock. The object to be locked may be different from the current object [Arnold et al 2000, p. 238]. It is locked before the synchronized statement is executed and is released after the statement execution is over. For example, we can remove the synchronized keyword from the above method signature and turn the method body into a synchronized statement that acquires locking the class Blackboard. That is, we can translate the above method to the equivalent method definition:

```
public static void newPrefix(NewPrefix np)
{
   synchronized(Blackboard.class)
   {
      RepeatedSequence rseq = np.seq0.parent;
      rseq.addBoundVars(np.names);
      np.removeFlag();
   }
}
```

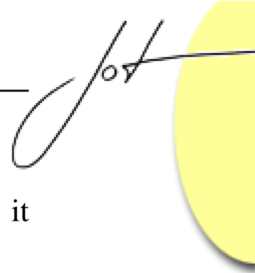
The standard class Thread defines methods interrupt and interrupted, which are inherited by subclasses of class Thread. In one part of a Java program, we can call method interrupt for a running thread to request cancelling the thread; in another part, we can invoke method interrupted to detect the cancellation request. The execution of method interrupt does not halt the running thread. If the running thread is executing method sleep or wait for some object when it is interrupted by the method interrupt, the sleep or wait method throws an InterruptedException [Arnold et al 2000, p. 256].

Thread Communication

Java threads may communicate by invoking the methods wait, notify, and notifyAll of objects. The methods, defined in the standard class Object, are available in each object. The parameterless wait method has signature

```
public final void wait() throws InterruptedException
```

It causes the current thread to wait until another thread invokes the notify or notifyAll method for the object. For the current thread to execute the method for an object, it must own the object's lock (monitor). By executing the wait method, the thread releases the monitor and waits until another thread notifies threads waiting on the object's monitor by calling the notify or notifyAll method of the object. The other



thread that executes the notify or notifyAll method may need to wait until it regains the monitor.

The notifyAll method has signature

```
public final void notifyAll()
```

for waking up all the threads that are waiting on this object's monitor. The awakened threads compete in the same way with other threads that compete to lock this object. The notify method has signature

```
public final void notify()
```

After the method is executed for an object, arbitrary one of the threads waiting on the object's monitor is awakened. The awakened thread competes with other threads that actively compete to synchronize on this object. Like the method wait, the current thread must own an object's monitor before it can invoke the notifyAll or notify method for the object.

For example, in the implementation of the compiler pi2j, we represent a sequence of dot-separated actions with an object of class Sequence. In the run method of the Sequence object, the following Java code is used to realize a parameterless atomic input action x. The synchronized statement is used for the current thread to acquire the lock of object inputAgent, which is an instance of class InputAgent. After locking the object, the current thread executes the wait method for the object inputAgent to wait until the flag in object inputAgent is removed.

```
inputAgent = new InputAgent(this, "x");
inputAgent.setupFlag();
Blackboard.communicateInput(inputAgent);
synchronized(inputAgent) {
   try {
      while (inputAgent.flag) inputAgent.wait();
   } catch(InterruptedException ie) { return; }
}
```

4 A LANGUAGE FOR CODING π-CALCULUS EXPRESSIONS

We now present the ASCII-based π-language for coding π-calculus expressions. We follow [Milner 1999, p. 89] to encode input and output atomic actions in the π-language. ASCII text editors can be used to program in the π-language.

Atomic Actions

The π-calculus expression $\overline{x}\langle y\rangle.P$ uses output prefix $\overline{x}\langle y\rangle$ to send name y via (channel) name x. We replace the typographical symbol $\overline{x}$ with ASCII expression out x and code the action expression $\overline{x}\langle y\rangle$ with expression out x<y> in the π-language. We regard the

name `y` as an argument of the output action `out x`. The π-language allows using capital letters in keywords. For example, all the following three expressions in the π-language denote the output atomic action $\overline{x}\langle y\rangle$. They are equivalent.

```
out x<y>
Out x<y>
OUT x<y>
```

The following three equivalent expressions represent argumentless output action $\overline{x}$.

```
out x<>
Out x
OUT x<>
```

The π-calculus process expression $x(z).Q$ uses an action prefix $x(z)$ to input a name from name x and assigns the inputted name to the bound name (parameter) z. We shalll denote the "input operator" with ASCII expression `in x` and code atomic action expression $x(z)$ with expression `in x(z)` in the π-language. We regard the name `z` as the parameter of the input action `in x`. For example, all the following three π-language expressions denote the same input action $x(z)$.

```
in x(z)
In x(z)
IN x(z)
```

The following three expressions represent the same parameterless input action x.

```
in x()
In x
IN x()
```

In the π-calculus, process expression $\tau.R$ using Greek letter τ denotes an action that is invisible to the environment outside the agent that executes the action. Following the Pict language and other applications [Pierce and Turner 2000; Canal et al 2003], we denote the atomic action τ with keyword `tao` in the π-language. For example, the π-language expression

```
tao.in x(z).out z<y>
```

denotes a process. It schedules an internal action τ, an input action $x(z)$, and an output action $\overline{z}\langle y\rangle$, which sends name y via the name received by the input action $x(z)$. The keyword `tao` is also caseless. For example, the following π-language expression denotes the same π-calculus process as the above π-language expression.

```
TAO.IN x(z).OUT z<y>
```

Other than the keywords such as `in`, `out`, and `tao`, names in the π-language are case sensitive. For example, the following π-language expression is different from the above π-language expression.

```
TAO.IN X(Z).OUT Z(Y)
```

In the π-calculus [Milner 1999], keyword *new* is used in the prefix $new\,\vec{w}$ in a process expression to declare a series of private bound names $\vec{w} = w_1 ... w_m$ with $m > 0$. We keep the keyword *new* and denote the π-calculus prefix $new\, w_1 ... w_m$ with π-language prefix

```
new w1 … wm
```

The keyword `new` is caseless. For example, the π-language expression `new x y new z out w<x>.out w<y>.out w<z>` denotes π-calculus process $new\; x\; y\; new\; z\; \overline{w}\langle x\rangle.\overline{w}\langle y\rangle.\overline{w}\langle z\rangle$. It uses two new-prefixes to introduce names `x`, `y`, and `z`. It sends the private names out through the name `w`. The expression is equivalent to `new x y z out w<x>.out w<y>.out w<z>`.

Syntactic Rules

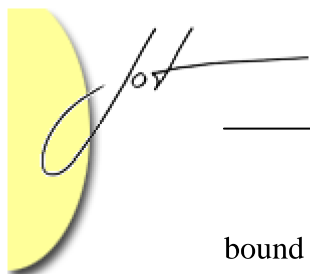
The grammar of the π-language consists of the following production rules, in which terminal symbols are printed in color `blue`. A pair of curl braces { and } followed by an asterisk '*' is used to enclose a grammar component that may be repeated zero or more times. A pair of curl braces followed by the plus symbol '+' encloses a component that is repeated at least once. A pair of square brackets [and] encloses an optional component. A Java identifier denoted with nonterminal *id* consists of letters, digits, and underscore '`_`' and can be started only with a letter.

```
<process> ::= <summation> { | <summation> }*
<summation> ::= <repeatedSequence> { + <repeatedSequence> }*
<repeatedSequence> ::= [ <actionSequence> ] { ! <actionSequence> }*
<actionSequence> ::= { <restriction> }* <atomicAction>
                                    { . { <restriction> }* <atomicAction> }*
<atomicAction> ::= <outAction> | <inAction> | <silentAction> | ( <process> )
<restriction> ::= < newPrefix > | <condition>
<newPrefix> ::= new { <id> }+
<condition> ::= [ <id> = <id> ]
<outAction> ::= out <id> [ < [ <id> ] > ]
<inAction> ::= in <id> [( [ <id> ] ) ]
<silentAction> ::= tao
```

In the above grammar, operators introduced at lower levels bind more tightly than the ones at a higher level. Hence, the grammar is compatible with operator precedence ordering:

```
|
+
!, ., new, in, out, tao, [, =, ], <, >, (, )
```

A difference between the above grammar and the π-calculus [Milner et al. 1989] is about the precedence ordering of operators '`|`' and '`+`'. In the above grammar, the *composition* operator '`|`' is assigned the lowest precedence because in system modeling for, say, the jobshop [Milner 1999, p. 62] or the storage system [Sangiorgi and Walker, p. 32], a system usually consists of indepednent, concurrent agents (components), which are



bound together using the operator '`|`'. Choice decisions supported by the operator '`+`' are lower-level decisions made by indivisual agents.

The above grammar makes a syntactic simplification for the π-calculus. In the π-calculus, the repetition operator '!' is prefixed to an atomic action expression π and the dot operator '.' can connect an action π_1 and a repeated action $!\pi$ in the π-calculus expression $\pi_1.!\pi$. Here, we omit the connector '.' inside the process expression $\pi_1.!\pi$. For example, the π-calculus expression $!x(z).!\overline{y}z.0$ shown in [Sangiorgi and Walker 2001, p. 13] will be coded with expression `! in x(z) ! out y<z>` in the above grammar. An advantage of replacing the two-symbol operator ".!" in the π-calculus with the single-symbol operator '`!`' is simplification of parser construction. Logically, occurrences of the repetition operator '`!`' in a sequence of actions separate the sequence into *maximal* sequences that do not contain operator '`!`'.

In the above grammar, nonterminal *<atomicAction>* denotes an input $x(y)$, an output $\overline{x}\langle z\rangle$, a silent action τ, or a process expression enclosed within a pair of parentheses `(` and `)`. The nonterminal *<restriction>* denotes either a prefix $new\ w_1...w_m$ or a condition $[x=y]$. Multiple restrictions denoted by the nonterminal *<restriction>* may be prefixed to the atomic actions in an action sequence, which is denoted with the nonterminal *<actionSequence>* and which does not include any of the operators '`!`', '`+`' and '`!`'. The action sequences, denoted by nonterminal *<actionSequence>*, can be connected with operator '`!`' to form a repeated sequence, which is denoted with nonterminal *<repeatedSequence>* and which may be prefixed with the operator '`!`'. Several repeated sequences, denoted with nonterminal *<repeatedSequence>*, can be connected with operator '`+`' to form a summation, which is denoted with *<summation>*. Several summations , denoted with the nonterminal *<summation>*, can be connected into a process expression denoted with nonterminal *<process>*. Since a process expression enclosed within a pair of parentheses `(` and `)` is an atomic action, the above grammar defines process expressions recursively.

The π-language can be used to encode an empty process, which amounts to the inaction process 0. The empty process denoted as 0 can be deduced as follows:

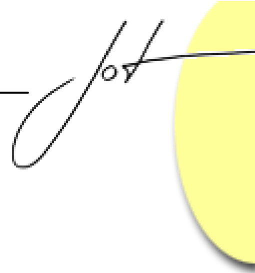
<process> ⇒ <summation>

⇒<repeatedSequence>

⇒ [<actionSequence>]

⇒ε

We can represent the empty process ε with a pair of parentheses `()` in the π-language. In fact, the process `()` is an instance of the nonterminal *<atomicAction>* of the above grammar. Similarly, expression `().()` is also a π-language process expression. By the above discussion, we omit the inaction symbol 0 in the π-language.



5 TRANSLATING π-PROGRAMS TO JAVA

Process Decomposition

We follow the syntactic structure of the π-language grammar to compile a π-calculus process expression into Java objects. Specifically speaking, we use the objects of the following Java classes to organize the grammar components identified in a π-language process expression. The following table shows that the Java classes correspond to nonterminal symbols in the π-language grammar. We ascribe responsibilities to the Java objects in the following discussion.

Nonterminal Symbol	Java class
process	`Process`
summation	`Summation`
repeatedSequence	`RepeatedSequence`
actionSequence	`Sequence`
newPrefix	`NewPrefix`
condition	`Condition`
outAction	`OutputAgent`
inAction	`InputAgent`
silentAction	`SilentAgent`

Table 1: Nonterminals in the π-language grammar and their Java incarnations

Each of the classes `Process`, `Summation`, `RepeatedSequence`, and `Sequence` inherits the standard class `Thread`. Therefore, the `run` method of an object of any of the classes can be started as a new thread. We enclose the object's responsibility in the objects' `run` method.

We use an object of class `Process` to represent a π-language process expression. Its `run` method encloses the translations of all the *summation* components of the π-language process and, thus, all the actions specified in the process.

An object of class `Summation` is used to represent a π-calculus summation expression $\sum_{i \in I} \pi_i.P_i$. An addend or term $\pi_i.P_i$ in the summation is parsed as

repeatedSequence component. The `run` method of the `Summation` object encloses translations of all the *repeatedSequence* components in the summation and, thus, encloses all the actions specified for the summation.

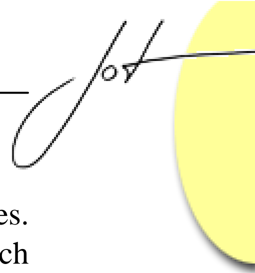
We use a `RepeatedSequence` object to realize a maximal sequence of atomic actions that are connected with either the dot operator '`.`' or the repetition operator '`!`'. Each of the atomic actions in the sequence can be qualified with conditions and new-prefixes, which are represented with objects of classes `Condition` and `NewPrefix`, respectively. The sequence corresponds to an addend $\pi_i.P_i$ of the summation $\sum_{i\in I}\pi_i.P_i$. The `run` method defined in the `RepeatedSequence` object is explained as follows.

A `RepeatedSequence` object is responsible to keep the current value `w` for each name `y` declared locally in the repeated sequence. The value `w` is assigned to the name `y` by an input action such as `in x(y)`. It is represented as a substitution $\{w/y\}$ applied to the repeated sequence in a π-calculus process expression. We use a dictionary named `substitution` in the `RepeatedSequence` object to keep the current values w of names y. In addition to the name `w`, we also keep an object `sequ` of the `RepeatedSequence` class in which the name `w` is declared. The name `w` and the object `sequ` are encapsulated in an object, which is the value of key `y` in dictionary `substitution`.

In the root process object of a Java program that represents a π-program, we use an instance variable `root` to hold an object of class `RepeatedSequence`. The object `root` keeps all the free variables of the π-calculus process expression and the current values of those free variables that have been changed by input actions.

The action sequence represented by the `RepeatedSequence` object is divided by the repetition operator '`!`' into maximal sequences of atomic actions that do not include the repetition operator. We represent each of the atomic action sequences with an object of class `Sequence`. The `run` method of the `Sequence` object encodes the π-language translations of all the actions in the action sequence. We represent the actions with objects of classes `OutputAgent`, `InputAgent`, and `SilentAgent`. The conditions and new-prefixes in front of the action expressions are represented with objects of classes `Condition` and `NewPrefix`.

While translating a π-language process expression, we create anonymous subclasses of the classes `Process`, `Summation`, `RepeatedSequence`, and `Sequence`. In the `run` methods of the unique objects of the anonymous classes, we enclose the Java translations of components of the π-language process expression. For example, the π-language process `in x.out x<y> + out x | out y` consists of two summations `in x.out x<y> + out x` and `out y`. The first summation consists of two repeated sequences `in x.out x<y>` and `out x`. The second summation consists of only one repeated sequence, which is an action sequence of length one. The repeated sequence `in x.out x<y>` contains no replication operator '`!`' and, therefore, is composed of an



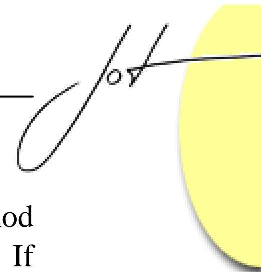
action sequence. Thus, the given π-language process consists of three action sequences. The π-language process expression is compiled to the following Java program, in which three `Sequence` objects are created.

```
import java.util.*;
public class PiProgram {
   public static void main(String args[]) {
      new Process(2) {
         public void run() {
            new Summation(this) {
               public void run() {
                  new RepeatedSequence(this)    {
                  {   seqs = new Sequence[1]; }
                     public void run() {
                        seqs[0] = new Sequence(this) {
                           public void run() {

              inputAgent = new InputAgent(this, "x");
              inputAgent.setupFlag();
              Blackboard.communicateInput(inputAgent);
              synchronized(inputAgent) {
              try {
                 while (inputAgent.flag) inputAgent.wait();
              } catch(InterruptedException ie) { return; }
              }
              if (dead()) return;

              outputAgent = new OutputAgent(this, "x", "y");
              outputAgent.setupFlag();
              Blackboard.communicateOutput(outputAgent);
              synchronized(outputAgent) {
              try {
                 while (outputAgent.flag) outputAgent.wait();
              } catch(InterruptedException ie) { return; }
              }
              if (dead()) return;
                           }
                        };
                        new Thread(seqs[0]).start();
                     }
                  }.start();
                  new RepeatedSequence(this) {
                  {   seqs = new Sequence[1]; }
                     public void run() {
                        seqs[0] = new Sequence(this) {
                           public void run() {
               outputAgent = new OutputAgent(this, "x");
               outputAgent.setupFlag();
               Blackboard.communicateOutput(outputAgent);
```

```
                    synchronized(outputAgent) {
                    try {
                       while (outputAgent.flag) outputAgent.wait();
                    } catch(InterruptedException ie) { return; }
                    }
                     if (dead()) return;
                                }
                            };
                            new Thread(seqs[0]).start();
                         }
                      }.start();
                    }
                 }.start();
                 new Summation(this) {
                    public void run() {
                       new RepeatedSequence(this)    {
                       {   seqs = new Sequence[1]; }
                       public void run() {
                          seqs[0] = new Sequence(this) {
                                public void run() {
                    outputAgent = new OutputAgent(this, "y");
                    outputAgent.setupFlag();
                    Blackboard.communicateOutput(outputAgent);
                    synchronized(outputAgent) {
                    try {
                       while (outputAgent.flag) outputAgent.wait();
                    } catch(InterruptedException ie) { return; }
                    }
                     if (dead()) return;
                                 }
                             };
                             new Thread(seqs[0]).start();
                          }
                       }.start();
                     }
                  }.start();
               }
            }.start(); // root process
         } // main
      } // PiProgram
```

Action Realization

Inside the `run` method of a `Sequence` object, we create objects to perform atomic actions. For example, the input action `in x` presented in the π-language process `in x.out x<y> + out x | out y` is realized with the following Java code, which instantiates class `InputAgent` to create an object `inputAgent` that encapsulates the



name x . It sets the flag value in the object inputAgent. The method communicateInput of class Blackboard is invoked to handle the input action. If the input action can be communicated with an existing output action, the communication takes place; otherwise, the static method communicateInput enters the object inputAgent into a queue. The flag value in object inputAgent is removed by the method communicateInput after the input action is communicated with an output action $\overline{x}$ presented in a different summation.

```
inputAgent = new InputAgent(this, "x");
inputAgent.setupFlag();
Blackboard.communicateInput(inputAgent);
synchronized(inputAgent) {
   try {
      while (inputAgent.flag) inputAgent.wait();
   } catch(InterruptedException ie) { return; }
}
if (dead()) return;
```

As shown in the above Java code, before the input action is communicated, the running Sequence thread that has the input action as prefix waits by invoking the wait method of the object inputAgent. While the Sequence thread is waiting, if the class Blackboard finds an output action to communicate with the object inputAgent, the flag value of the object inputAgent is removed so that the thread will continue its operation. As described in Section 2, the class Blackboard may need to render the Sequence thread void. In the latter case, the Sequence thread invokes method dead to detect whether it is void. If the test results true, the Sequence thread terminates with a return statement.

Parent Relation

In the above Java program, a thread object is created with the keyword this as constructor argument. The constructor relates the created thread object and the current object with a parent relation. In the classes Process, Summation, RepeatedSequence, and Sequence, an instance variable parent of type Sequence, Process, Summation, or Repeated-Sequence is used to hold the parent of an object. The parent relation at the class level is shown in the UML class diagram in Fig. 1. As indicated in the above Java program, the parent relation in a Java program that realizes a π-calculus process expression has a hierarchical structure, the root of which is an object of the class Process.

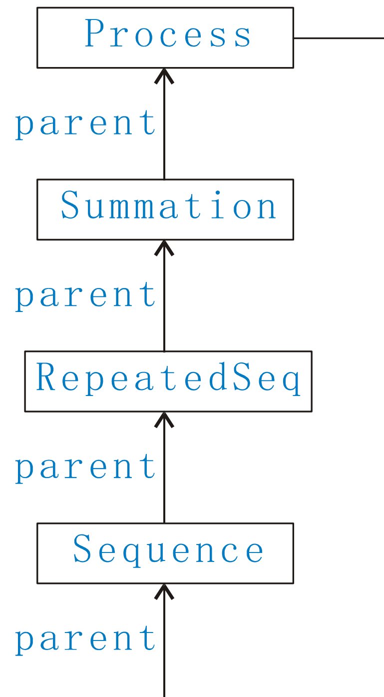


Figure 1 The parent relation for π-programs

For example, a process is composed of summations. A reference of the corresponding Process object is assigned to the parent attributes of the Summation objects. In the above Java program, the constructor invocation of each anonymous class uses keyword this to relate the created object and its parent.

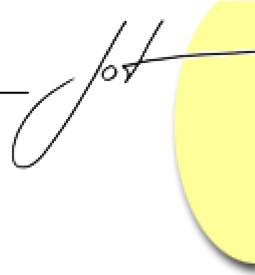
Action Synchronization

Running threads may interfere each other. We realize an action, new-prefix, or condition with a static synchronized method of class Blackboard. The static methods acquire the lock of the class object of the class Blackboard and are, thus, synchronized. We use the static method communicateInput of class Blackboard to illustrate the synchronization mechanism. The method signature is

```
public static synchronized void communicateInput(InputAgent ia)
```

Assume the argument object ia of the communicateInput method represents input action in x(z). The object ia initially records only the Sequence object seq0 of which the input action is the current prefix and the names x and z. Note that the names x and z may have been replaced with other bound names declared in the same or different repeated sequence. The current values of the names x and z can be found in the RepeatedSequence objects that are ancesters of object seq0. An instance method named setup of class InputAgent is invoked by the communicateInput method to find the current values arg0 and arg1 of the names x and z recorded in object ia.

The communicateInput method looks for a queued object oa of class OutputAgent such that



- The object oa represents an atomic action out y<w>,
- The current values of names y and w recorded in object oa are denoted with arg0 and arg1, respectively, and
- The value arg0 in ia and value arg0 in oa are the equal.

If the search for a queued object oa succeeds, the communicateInput method performs a communication between the threads that have ia and oa as prefix actions, respectively. Then, the threads can continue their respective run methods. Otherwise, the object ia is entered into a queue to wait for an output action to which it can communicate. The communication process consists of the following activities:

- If the name z is declared in RepeatedSequence object sequ, the value of the key z in the dictionary substitution in object sequ will be replaced with the value arg1 in the object oa. Thus, the output argument is assigned to the input parameter.
- If any queued input or output action that stores an arg0 or arg1 value for the name z declared in RepeatedSequence sequ, the arg0 or arg1 value is replaced by the value arg1 in object oa.
- Remove the flag values in the objects ia and oa. The flag removals permit the threads waiting for the flags to continue their executions.

6 COMPILER PI2J

Usage of the Compiler

Compiler pi2j is pronounced as "π to J(ava)", which means translating π-language process expressions to Java programs. It is a Java application generated with JavaCC [JavaCC 2003]. Suppose the π-language program of a π-calculus process is in the file pi-program.txt. We can use command

```
java pi2j pi-program.txt
```

to invoke the Java application pi2j for translating the file pi-program.txt to a Java program, which has default name PiProgram.java. If a class name MyClass is placed in the command

```
java pi2j pi-program.txt MyClass
```

The above command will define a class MyClass in Java file MyClass.java. The Java file PiProgram.java and MyClass.java can be compiled with the Java compiler by issuing one of the commands

```
javac PiProgram.java
javac MyClass.java
```

The resulting JVM bytecode file can be executed with one of the commands

```
java PiProgram
```

java MyClass

The java application is compressed into the jar file pi2j.jar with the command

jar -cmf pi2j.jar *.class

The compiler pi2j can be applied to translate the file pi-program.txt to the JVM bytecode file PiProgram.java with the command

java -jar pi2j.jar pi-program.txt

Action Queue

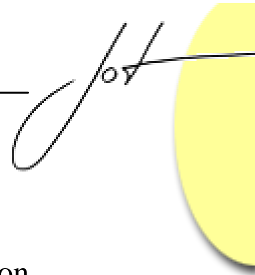
In the execution of a π-language process expression, if an input or output action π cannot be communicated with a waiting output or input action, the action π is entered into a queue to wait for a new output or input action for the purpose of communication. We now use the π-calculus process expression $new\ x\ y\ new\ z\ \overline{w}\langle x\rangle.\overline{w}\langle y\rangle.\overline{w}\langle z\rangle$ discussed in Section 4 to illustrate the queuing functionality of the compiler pi2j.

Assume the file pi-program.txt consists of the π-language code `new x y new z out w<x>.out w<y>.out w<z>`. We compile the π-program pi-program.txt into the Java program PiProgram.java with compiler pi2j and compile PiProgram.java to bytecode file PiProgram.class. An execution of the bytecode file PiProgram.class displays the following information on the standard output.

```
Keep into queue output action:
   An out action in sequence: Thread[Thread-5,5,main]
   the parameters are w and x
   the parameter values are "Thread[Thread-1,5,main] + w"
   and "Thread[Thread-4,5,] + x".
```

The above output indicates that output action prefix `out w<x>` is placed into a queue by the class Blackboard. It also indicates that the output action belongs to a thread object denoted by expression Thread[Thread-5,5,main], the arg0 value for the free name w is an encapsulation of thread object Thread[Thread-1,5,main] and name w, and the arg1 value for name x is an encapsulation of thread object Thread[Thread-4,5,] and name x. Note that the bound name x is declared in thread Thread[Thread-4,5,]. The execution of the bytecode file will not terminate naturally since the Sequence thread Thread[Thread-5,5,main] has action `out w<x>` waiting for communication.

Similarly, if the file pi-program.txt consists of the π-language code `in x.out x<y> + out x | out y`, an execution of the JVM byte code that is compiled from the π-program places objects that represent the action prefixes in x, out x, and out y into queues by the class Blackboard. The execution will not terminate due to three waiting sequence threads.



I/O Communication

We use a π-calculus process $\overline{w}\langle x\rangle + x(z).\overline{w}\langle y\rangle \mid w(r)$ to illustrate action communication supported by the compiler `pi2j`. Assume file pi-program.txt consists of π-language code `out w<x> + in x(z).out w<y> | in w(r)`, which encodes the π-calculus process. An execution of the JVM bytecode compiled from the file pi-program.txt displays the following information on the standard output. The first operation displayed on the standard output queues the output action $\overline{w}\langle x\rangle$, which is the action prefix of the first repeated sequence in the process expression. The second operation queues the input action $x(z)$, which is the prefix of the second repeated sequence. The prefix action $w(r)$ of the second summation can communicate with the waiting output action $\overline{w}\langle x\rangle$. The following information uses word `Communicate` and a pair of curl braces to show the action communication. Since the performance of the prefix of the first repeated sequence renders the second repeated sequence void, there will be no active `Sequence` thread left in the queue after the communication. Thus, the execution of the JVM bytecode compiled from the π-program `out w<x> + in x(z).out w<y> | in w(r)` terminates.

```
Keep into queue output action:
   An out action in sequence: Thread[Thread-8,5,main]
   the parameters are w and x
   the parameter values are "Thread[Thread-1,5,main] + w"
   and "Thread[Thread-1,5,main] + x"
Keep into queue input action:
   An in action in sequence: Thread[Thread-10,5,main]
   the parameters are x and z
   the parameter values are "Thread[Thread-1,5,main] + x"
   and "Thread[Thread-1,5,main] + z"
Communicate {
 input action:
   An in action in sequence: Thread[Thread-12,5,main]
   the parameters are w and r
   the parameter values are "Thread[Thread-1,5,main] + w"
   and "Thread[Thread-1,5,main] + r"
output action:
   An out action in sequence: Thread[Thread-8,5,main]
   the parameters are w and x
   the parameter values are "Thread[Thread-1,5,main] + w"
   and "Thread[Thread-1,5,main] + x"
```

7 CONCLUSION

The π-calculus proposed by Milner et al [Milner et al 1992] is a theoretical model of mobile communicating systems. Here, we present an implementation of the π-calculus in the Java language by exploiting the Java multithreading and thread communication

mechanism. We decompose a π-calculus process into different components and implement the components with classes `Process`, `Summation`, `RepeatedSequence`, and `Sequence`. The `Sequence` objects are execution threads, which enclose the actions specified in π-calculus processes and which can run concurrently.

The Java implementation of the π-calculus makes it possible for using the π-calculus to control and simulate real-world mobile communicating systems. The Java implementation of the π-calculus handles only the communication actions but no control actions or other operations. A research topic is to incorporate controlled operations into the π-processes.

REFERENCES

[Arnold00] K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language – Third Edition*, Addison-Wesley, Boston 2000.

[Canal03] C. Canal, L. Fuentes, E. Pimentel, J.M. Troya, and A. Vallecillo, *Adding roles to CORBA objects,* IEEE Transactions on Software Engineering, Vol. 29, No. 3, March 2003, 242-260.

[Fournet97] C. Fournet and L. Maranget, *The Join-Calculus Language (release 1.05)*, Institut National de Recherche en Informatique et Automatique, http://pauillac.inria.fr/join/manual/index.html.

[JavaCC03] JavaCC developers community, *Java Compiler Compiler (JavaCC) – The Java Parser Generator, Version 3.2*, Sun Microsystems Inc., 2003, https://javacc.dev.java.net/.

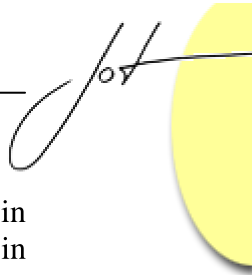
[Vasconcelos98] V. Vasconcelos and R. Bastos. *The TyCO Programming Language*, http://www.ncc.up.pt/~lblopes/tyco/.

[Milner92] R. Milner, J. Parrow, and D. Walker, *A calculus of mobile processes (Parts I and II)*, Information and Computation, Vol. 100, No. 1, pp. 1-77, 1992.

[Milner99] R. Milner, Communication and Mobile Systems: *The π-Calculus*, Cambridge University Press, Cambridge, UK 1999.

[Pierce00] B. Pierce and D. Turner, *Pict: A programming language based on the Pi-calculus*, In Proof, Language and Interaction, MIT Press 2000. The Pict compiler is available at http://www.cis.upenn.edu/~bcpierce/papers/pict/.

[Sangiorgi01] D. Sangiorgi and D. Walker, *The π-calculus: A Theory of Mobile Processes*, Cambridge University Press, Cambridge UK 2001.



[Smolka95] G. Smolka, *The Oz programming model*, in Current Trends in Computer Science, Jan van Leeuwen (editor), Lecture Notes in Computer Science, Volume 1000, Springer-Verlag, Berlin 1995.

About the author

Dr. Liwu Li is 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. He can be reached at liwu@uwindsor.ca and http://www.uwindsor.ca/liwu.