SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 19(6), 553–578 (JUNE 1989)

The Joyce Language Report

PER BRINCH HANSEN
School of Computer and Information Science, Syracuse University, Syracuse, New York 13244, U.S.A.

SUMMARY

This paper defines a programming language for distributed processing called Joyce. Joyce is based on CSP and Pascal. A Joyce program defines concurrent agents which communicate through synchronous channels. The agents may be recursive.

KEY WORDS Programming languages Concurrent programming Communicating agents Input/output channels Polling Recursion

INTRODUCTION

This paper defines a small programming language called Joyce, designed for teaching the principles and practice of distributed processing. Joyce is a semantic variant of CSP with Pascal notation.[1–3]

A Joyce program consists of nested procedures which define abstract machines known as agents. The agent procedures may be recursive.

The execution of a program activates an initial agent. Agents may activate subagents dynamically. A subagent and its creator run concurrently.

The variables of an agent are inaccessible to other agents.

Agents communicate by means of symbols transmitted through channels. Every channel has an alphabet — a fixed set of symbols that can be transmitted through the channel. A symbol has a name and may carry a message of a fixed type.

Two agents match when one of them is ready to output a symbol to a channel and the other is ready to input the same symbol from the same channel. When this happens, a communication takes place in which a message from the sending agent is assigned to a variable of the receiving agent.

The communications on a channel take place one at a time. A channel can transfer symbols in both directions between two agents.

A channel may be used by two or more agents. If more than two agents are ready to communicate on the same channel, it may be possible to match them in several different ways. The channel arbitrarily selects two matching agents at a time and lets them communicate.

A polling statement enables an agent to examine one or more channels until it finds a matching agent. Both sending and receiving agents may be polled.

Agents create channels dynamically and access them through local port variables. When an agent creates a channel, a channel pointer is assigned to a port variable. The agent may pass the pointer as a parameter to subagents.

 Received 12 September 1988

When an agent reaches the end of its defining procedure, it waits until all its subagents have terminated before terminating itself. At this point, the local variables and any channels created by the agent cease to exist.

BNF NOTATION

The context-free grammar is defined in extended BNF notation.

A production

```
N = E .
```

defines a class of sentences named N by means of a syntax expression E. N consists of one or more letters. Example:

```
AssignmentStatement = VariableAccess ":=" Expression .
```

A syntax expression A|B|C defines sentences which are either A, B or C sentences, for example

```
"+" | "-" | "or"
```

A syntax expression ABC defines sentences consisting of an A sentence followed by a B sentence followed by a C sentence, for example

```
VariableAccess ":=" Expression
```

A syntax expression [A] defines sentences which are either A sentences or empty, for example

```
[ "(" ActualParameterList ")" ]
```

A syntax expression {A} defines sentences which are finite (possibly empty) sequences of A sentences, for example

```
{ Digit }
```

A syntax expression N denotes sentences defined by the production named N, for example

```
Digit
```

A syntax expression "xyz" defines a text string xyz as a basic symbol in the language (called a token). Examples:

```
"begin"    ":="    "true"    "1984"
```

TOKENS AND SEPARATORS

Every sentence in the language is a sequence of tokens and separators.

Graphic characters

```
GraphicCharacter = Letter | Digit | SpecialCharacter | Space .
Letter = SmallLetter | CapitalLetter .
SmallLetter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l"
  | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" |
  "z" .
CapitalLetter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K"
  | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X"
  | "Y" | "Z" .
Digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" .
SpecialCharacter = "!" | """ | "#" | "$" | "%" | "&" | "'" | "(" | ")" | "*" | "+"
  | "," | "-" | "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" | "[" | "\" |
  "]" | "^" | "_" | "`" | "{" | "|" | "}" | "~" .
Space = " ".
```

Graphic characters are used to form tokens and separators.

Tokens

```
Token = WordToken | SpecialToken | Name | Numeral | GraphicToken |
  ControlToken | StringToken .
WordToken = "agent" | "and" | "array" | "begin" | "const" | "div" | "do" |
  "else" | "end" | "if" | "mod" | "nil" | "not" | "of" | "or" | "ord" | "poll" |
  "record" | "then" | "type" | "var" | "while" .
SpecialToken = "(" | ")" | "*" | "+" | "," | "-" | "." | "/" | ":" | ";" | "<" |
  "=" | ">" | "[" | "]" | "!" | "?" | "&" | "|" | ".." | ":=" | "<=" | "<>" |
  ">=" | "->" .
Numeral = SimpleNumeral | RealNumeral .
```

A capital letter in a word token is equivalent to the corresponding small letter. In this paper, word tokens are printed in **boldface**.

Names, numerals, graphic tokens, control tokens and string tokens will be defined later.

Separators

```
Separator = Space | Newline | Comment .
Comment = "{" CommentText "}" .
CommentText = { GraphicCharacter | Newline | Comment } .
```

Every program line ends with a newline character.

Any token may be preceded by one or more separators. There must be at least one separator between two adjacent word tokens, names, numerals or control tokens.

The characters { and } cannot occur in a comment text (except as part of other comments within the comment text).

Example

```
{This is a { nested } comment
 that extends over two lines}
```

BLOCKS

The standard names

```
integer    boolean    char    real    false    true
```

denote predefined entities which may be used in any program.

All other named entities must be described by definitions. A definition is a sentence which introduces the names of similar entities and describes their common properties.

Related definitions and statements are combined into sentences called blocks. The standard entities are considered defined at the beginning of an imaginary block (the standard block) which contains the program. Any other block is either a program or a procedure. Blocks are nested as follows: the standard block contains every program; a program contains a single procedure; and a procedure may contain other procedures.

Names

```
Name = Letter { Letter | Digit } .
```

The word tokens cannot be used as names. Apart from that, names may be chosen freely. A capital letter used in a name is equivalent to the corresponding small letter.

Examples

```
x
true
RC4000
ConcurrentPascal
```

Scope rules

Constants, types, variables and procedures defined in the same block must have distinct names.

The use of a name to denote an entity is called a reference to the entity. That part of the program text in which a named entity can be referenced is called the scope of the entity. The entity is said to be known in its scope.

The following defines the scopes of named entities in terms of their apparent scopes and possibly homonyms. Homonyms are distinct entities with the same name defined in different blocks.

Constants, types and variables are non-recursive entities. Consider a non-recursive entity x introduced by a definition D in a block B: the apparent scope of x extends from the end of D to the end of B.

Procedures may be recursive. Consider a procedure x defined in a block B: the apparent scope of x extends from the beginning of x to the end of B.

If the apparent scope of a named entity x does not contain any blocks which define homonyms of x, the scope of x is the apparent scope. Otherwise, the scope is the apparent scope minus the scopes of the homonyms.

An entity x defined in a block B is said to be local to B and global to any blocks contained in its scope.

The scope rules do not apply to fields and symbols. A field or symbol is a subentity of another entity and cannot be identified by a single name only. (This will be explained later.)

Block activation

The language has one kind of procedures only, known as agent procedures.

An agent procedure P defines a class of abstract machines called agents. The creation and start of an agent defined by P is called an activation of P (or the activation of a P agent).

The activation of a P agent creates a new instance of every variable defined in the procedure P. These variable instances are called the own variables of the new agent. When the agent refers to a variable defined in the procedure, it refers to its own instance of the variable. The own variables of an agent are inaccessible to other agents. When an agent terminates, its own variables cease to exist.

An agent is always activated by another agent (called the creator). The new agent is called a subagent of its creator. After the creation, the subagent and its creator run concurrently at unpredictable speeds. When an agent reaches the end of the agent procedure, it waits until all its subagents have terminated before terminating itself.

The effect of executing a program is to activate and execute an initial agent. The initial agent is activated by an agent defined in another program (an operating system).

CONSTANTS

A constant denotes a fixed value of a fixed type.

Constant definitions

```
ConstantDefinitionPart = "const" ConstantDefinition { ConstantDefinition } .
ConstantDefinition = ConstantName "=" Constant ";" .
ConstantName = Name .
Constant = SimpleConstant | RealConstant | StructuredConstant .
```

A constant definition

```
c = d;
```

introduces a name c to denote a constant d. The type of the constant name c is the type of the constant d.

Example

```
const on = true; n = 10; nl = 10C; lf = nl; e = 2.718; none = nil stream;
```

Simple constants

```
SimpleConstant = SimpleNumeral | CharacterConstant | ConstantName .
```

A simple constant denotes a simple value. A constant name must denote a known, simple constant.

Simple numerals

```
SimpleNumeral = Digit { Digit } .
```

A simple numeral is a conventional decimal notation for a non-negative integer value.

Examples

```
0
1351
```

Character constants

```
CharacterConstant = GraphicToken | ControlToken .
GraphicToken = "'" GraphicCharacter "'" .
ControlToken = SimpleNumeral "C" .
```

A character constant denotes a value of type character. A graphic token denotes a graphic character. A control token denotes a character by its ordinal number.

Examples

```
'x'
10C
```

Real constants

```
RealConstant = RealNumeral | ConstantName .
```

A real constant denotes a real value. A constant name must denote a known, real constant.

Real numerals

```
RealNumeral = Mantissa [ Radix Exponent ] .
Mantissa = Digit { Digit } "." { Digit } .
Radix = "E" .
Exponent = [ Sign ] SimpleNumeral .
Sign = "+" | "−" .
```

A real numeral is a decimal notation for a non-negative real value. The letter E denotes the radix 10.

Examples

```
2.
5.72
23.1E−7
```

Structured constants

```
StructuredConstant = NilConstant | ConstantName .
```

A structured constant denotes a value of a structured type. A constant name must denote a known, structured constant.

Nil constants

```
NilConstant = "nil" TypeName .
```

The port value nil denotes a non-existing channel.
The constant

```
nil T
```

denotes a nil value of type T. T must be the name of a known port type.

Example

```
nil stream
```

TYPES

Every constant, variable, expression and symbol has a fixed type. The type of an operand is the set of all possible values which the operand may possess.

Type definitions

```
TypeDefinitionPart = "type" TypeDefinition { TypeDefinition } .
TypeDefinition = TypeName "=" NewType ";" .
TypeName = Name .
NewType = EnumeratedType | StructuredType .
```

A type definition

```
T = NT;
```

introduces a name T to denote a new type NT which is distinct from all other types.

Example

```
type
  task = (log, flow, scan);
  table = array [1..n] of integer;
  str = array [1..80] of char;
  buffer = record
             contents: table;
             head, tail, length: integer
           end;
  stream = [int(integer), eos];
```

Simple types

A simple type is a finite, ordered set of values (called simple values). The simple types are integer, boolean, character and enumerated types.

The values of a simple type can be mapped onto a set of successive integer values called ordinal numbers. The ordinal number of a simple value x is denoted

```
integer(x)
```

Type integer

The type name integer denotes the whole numbers in a system-dependent range. The ordinal number of an integer x is x itself, that is

```
integer(x) = x
```

Type boolean

The type name boolean denotes the truth values false and true. The ordinal numbers of these values are

```
integer(false) = 0     integer(true) = 1
```

Type character

The type name char denotes the ASCII character set.

Enumerated types

```
EnumeratedType = "(" ConstantName { "," ConstantName } ")" .
ConstantName = Name .
```

A type definition

```
T = (c0, c1, . . . , cn);
```

defines an enumerated type named T. The type introduces the constant names c0, c1, . . . , cn to denote the values of T.

The ordinal numbers of these values are

```
integer(c0) = 0    integer(c1) = 1 . . . integer(cn) = n
```

Example

```
(log, flow, scan)
```

Type real

The type name real denotes a system-dependent subset of the real numbers.

The results of real expressions are system-dependent approximations to the corresponding mathematical results.

Structured types

```
StructuredType = ArrayType | RecordType | PortType .
```

A structured type T is defined in terms of other types (the subtypes of T).

Array types

```
ArrayType = "array" "[" IndexRange "]" "of" ElementType .
IndexRange = LowerBound ".." UpperBound .
LowerBound = SimpleConstant .
UpperBound = SimpleConstant .
ElementType = TypeName .
```

A type definition

```
T = array [min. .max] of E;
```

defines an array type named T. Every array value is a sequence of max − min + 1 other values known as array elements. The number of elements is called the array length |T|.

Every array element is of type E and has an index which defines its position in the array value. The indices are the values

```
min, min+1, . . . , max
```

The array type comprises every possible sequence of |T| elements of type E.

The index range must be defined by two simple constants min and max of the same type (the index type), such that min ≤ max. E must be the name of a known type.

An array type in which the elements are of type char is called a string type. A string type defines an ordered set of values called strings.

Any other array type defines a set of unordered array values.

Examples

```
array [1..n] of integer
array [1..80] of char
```

Record types

```
RecordType = "record" FieldList "end" .
FieldList = RecordSection { ";" RecordSection } .
RecordSection = FieldName { "," FieldName } ":" FieldType .
FieldName = Name .
FieldType = TypeName .
```

A type definition

```
T = record f1:T1; f2:T2; . . . fn:Tn end;
```

defines a record type named T. Every record value is a sequence of n other values known as fields. Every field has a name fi that defines its position in the record value and a type Ti.

The record type comprises every possible sequence consisting of a value of type T1 followed by a value of type T2, and so on, ending with a value of type Tn. The record values are unordered.

The field names f1, f2, . . ., fn must be distinct, and T1, T2, . . . Tn must be names of known types.

A record section

```
f1, f2, . . ., fj:Tk
```

defines fields f1, f2, . . ., fj of the same type Tk.

Example

```
record
  contents: table;
  head, tail, length: integer
end
```

Port types

```
PortType = "[" Alphabet "]" .
Alphabet = SymbolClass { "," SymbolClass } .
SymbolClass = SymbolName [ "(" MessageType ")" ] .
SymbolName = Name .
MessageType = TypeName .
```

Agents communicate by means of values called symbols transmitted through entities called channels. The set of possible symbols that can be transmitted through a channel is called its alphabet.

Agents create channels dynamically and access them through variables known as port variables. The types of these variables are called port types.

A type definition

```
T = [s1(T1), s2(T2), . . ., sn(Tn)];
```

defines a port type named T. The port value nil T is of type T and denotes a non-existing channel. All other port values of type T denote distinct channels with the given alphabet. The port values (also known as channel pointers) are unordered.

The alphabet is the union of a fixed number of disjoint symbol classes named s1, s2, . . ., sn.

A symbol class

```
si(Ti)
```

consists of every possible value of type Ti prefixed with the name si. The Ti values are called messages.

A symbol class

```
sj
```

consists of a single symbol named sj without a message. The symbol is called a signal.

The symbol names s1, s2, . . ., sn must be distinct, and T1, T2, . . ., Tn must be names of known types. The message types cannot be (or include) port types.

Example

```
[int(integer), eos]
```

VARIABLES

A variable is an entity that has a changeable value of a fixed type.

Variable definitions

```
VariableDefinitionPart = "var" VariableDefinition { VariableDefinition } .
VariableDefinition = VariableGroup ";" .
VariableGroup = VariableName { "," VariableName } ":" TypeName .
VariableName = Name .
```

A variable definition

```
v1, v2, . . . vn:T;
```

in a procedure introduces names v1, v2, . . ., vn to denote distinct classes of variables of type T. T must be the name of a known type.

Every activation of the procedure creates a new instance of each of these variables, with unpredictable initial values.

Examples

```
var buf: buffer;
var x, y: integer; more: boolean; f: real; s: str; inp, succ: stream;
```

Variable access

```
VariableAccess = NamedVariableAccess | IndexedAccess | FieldAccess .
```

A variable access denotes a variable used as an operand.

When an agent executes a variable access, any index expressions occurring in the variable access are evaluated one at a time in the order written. Following this, the agent locates the variable among its own variables.

Named variables

```
NamedVariableAccess = VariableName .
```

A variable accessed by a name only is called a named variable. The type of a named variable is given by a variable or parameter definition.

An agent procedure P cannot access a variable which is global to P.

Example

```
x
```

Indexed variables

```
IndexedAccess = ArrayVariableAccess "[" IndexExpression "]" .
ArrayVariableAccess = VariableAccess .
IndexExpression = Expression .
```

A variable v:T of type

```
T = array [min..max] of E;
```

holds an array value. The variable is composed of |T| other variables called indexed variables. Every indexed variable holds a distinct element of the array value.

An indexed access v[e] denotes the indexed variable of v which holds the array element with index e. The variable access v must be of type T, and the index expression

e must be of the index type of T. The type of the indexed access is the element type E.

An indexed access is undefined if the index expression e denotes a value outside the index range min..max.

Example

```
buf.contents[buf.head]
```

Field variables

```
FieldAccess = RecordVariableAccess "." FieldName .
RecordVariableAccess = VariableAccess .
```

A variable v:T of type

```
T = record f1:T1; f2:T2; . . . fn:Tn end;
```

holds a record value. The variable is composed of n other variables called field variables. Every field variable holds a distinct field of the record value.

A field access v.fi denotes the field variable of v which holds the field named fi. The variable access v must be of type T, and fi must be one of the field names f1, f2, . . ., fn. The type of the field access is the type Ti of the field fi.

Example

```
buf.head
```

Port variables

```
PortAccess = VariableAccess .
```

A variable v:T of type

```
T = [s1(T1), s2(T2), . . ., sn(Tn)];
```

holds a port value.

If the value of v is nil T, a port access v denotes a non-existing channel; otherwise, it denotes a channel with the alphabet given by T. The type of the port access is T.

The channel itself is not a variable, but a communication device shared by two or more agents.

EXPRESSIONS

```
Expression = SimpleExpression | SimpleExpression RelationalOperator
  SimpleExpression.
RelationalOperator = "<" | "=" | ">" | "<=" | "<>" | ">=" .
```

An expression is a rule for computing a value of a fixed type.

The value of an expression is either the value of a simple expression or the value obtained by applying a relational operator to the values of two simple expressions:

1. If x and y are *simple* (or *real*) *operands* of the same type, the following expressions denote boolean values:

   x < y less
   x = y equal
   x > y greater
   x <= y not greater
   x <> y not equal
   x >= y not less

   If x and y are integer (or real) operands, the value of x < y is true if x is less than y, and is false otherwise. If x and y are operands of the same simple type, the value of x < y is the same as the value of

   integer(x) < integer(y)

   The other expressions are defined similarly.
2. If x and y are *array* (or *record*) *operands* of the same type, the following expressions denote boolean values:

   x = y x <> y

   The value of x = y is true if x and y consist of the same sequence of elements (or fields), and is false otherwise. The value of x <> y is the same as the value of not (x = y).
3. If x and y are *string operands* of the same type, the following expressions denote boolean values:

   x < y x = y x > y x <= y x <> y x >= y

   The elements of x and y are examined in the order of their indices until the index i = max or x[i] <> y[i]. The value of x < y is the same as the value of x[i] < y[i]. The other expressions are defined similarly. String operands are defined in the following.
4. If x and y are *port operands* of the same type, the following expressions denote boolean values:

   x = y x <> y

   The value of x = y is true if x and y denote the same channel (or nil value), and is false otherwise. The value of x <> y is the same as the value of not (x = y).

Examples

```
x - 1
x > 0
s <> str("edit")
inp = nil stream
```

Simple expressions

```
SimpleExpression = Term | SignOperator Term | SimpleExpression
  AddingOperator Term.
SignOperator = '+' | '-' .
AddingOperator = '+' | '-' | "or" .
```

The value of a simple expression is either the value of a term, or the value obtained by applying a sign operator to the value of a term, or the value obtained by applying an adding operator to the values of another simple expression and a term:

1. If x and y are *integer operands*, the following simple expressions also denote integer values:
   + x identity
   − x complementation
   x + y addition
   x − y subtraction
   If x and y are *real operands*, the above expressions denote real values.
2. If x and y are *boolean operands*, the following simple expression also denotes a boolean value:
   x or y disjunction
   The value is true if either x or y denotes true, and is false otherwise.

Examples

```
10
− (x + 1)
buf.tail mod n + 1
```

Terms

```
Term = Factor | Term MultiplyingOperator Factor.
MultiplyingOperator = "*" | "/" | "div" | "mod" | "and" .
```

The value of a term is either the value of a factor, or the value obtained by applying a multiplying operator to the values of another term and a factor:

1. If x and y are *integer operands*, the following terms also denote integer values:
   x * y multiplication
   x div y division (truncated quotient)
   x mod y division (remainder)
   The following relation holds for division
   x = (x div y) * y + (x mod y)
2. If x and y are *real operands*, the following terms also denote real values:
   x * y multiplication
   x / y division
3. If x and y are *boolean operands*, the following term also denotes a boolean value:
   x and y
   The value is true if x and y both denote true, and is false otherwise.

Examples

```
10
buf.tail mod n
```

Factors

```
Factor = Constant | VariableRetrieval | Constructor | "(" Expression ")" | "not"
  Factor
```

The value of a factor is either the value of a constant, variable, constructor or expression or the value obtained by applying a not operator to the value of another factor:

If x is a *boolean operand*, the following factor also denotes a boolean value:

```
not x
```

The value is true if x denotes false, and is false otherwise.

Examples

```
10
x
(x + 1)
not more
```

Variable retrieval

```
VariableRetrieval = VariableAccess .
```

A variable retrieval denotes the value of a variable.

The retrieval is undefined if no value has been assigned to the variable after its creation.

Constructors

```
Constructor = TypeName "(" ConstructorOperand ")" .
ConstructorOperand = Expression | StringToken .
StringToken = """ { GraphicCharacter } """ .
```

1. If x is a *simple operand*, the constructor
   integer(x)
   denotes the ordinal number of x.
2. If x is a *simple operand* and T is the name of a known simple type, the constructor
   T(x)
   denotes the simple value of type T with the ordinal number
   integer(x)
3. If x is an *integer operand*, the constructor
   real(x)
   denotes the real value corresponding to x.
4. If x is a *real operand*, the constructor
   integer(x)
   denotes the integer value corresponding to x.

An operand which is either an expression of a string type or a string token is called a string operand. A string token denotes a (possibly empty) sequence of graphic characters. The number of characters in a string operand x is called the string length |x|.

5. If x is a *string operand* and T is the name of a known string type, the constructor
   T(x)
   denotes a string of type T. If |T| > |x|, the string T(x) consists of x followed by |T| − |x| null characters; otherwise, it consists of the first |T| characters of x. The null character is denoted
   char(0)

Examples

```
char(x + integer('0'))
real(2)
integer(e)
str("This is a string of type str")
```

STATEMENTS

```
Statement =
  EmptyStatement | AssignmentStatement |
  AgentStatement | PortStatement |
  InputOutputStatement | IfStatement |
  WhileStatement | PollingStatement |
  CompoundStatement .
EmptyStatement = .
```

A statement denotes an action performed by an agent. The empty statement denotes the empty action.

Assignment statements

```
AssignmentStatement = VariableAccess ":=" Expression .
```

An assignment statement

```
v := e
```

denotes the assignment of a value to a variable. The variable access v and the expression e must be of the same type.

The assignment statement is executed in three steps:

1. The variable v is located.
2. A value is obtained by evaluating the expression e.
3. The value is assigned to v.

Example

```
buf.contents[buf.head] := x
```

Agent statements

```
AgentStatement = AgentName [ "(" ActualParameterList ")" ] .
ActualParameterList = ActualParameter { "," ActualParameter } .
ActualParameter = Expression .
```

An agent statement

```
P(e1, e2, . . ., em)
```

denotes activation of a new agent. P must be the name of a known agent procedure

```
agent P(a1:T1; a2:T2; . . . ; am:Tm);
var x1:U1; x2:U2; . . . ; xn:Un;
begin SL end;
```

The actual parameter list

```
e1, e2, . . ., em
```

must contain an actual parameter ei for every formal parameter ai in the formal parameter list

```
a1:T1; a2:T2; . . .; am:Tm
```

The order in which the actual and formal parameters are listed defines a one-to-one correspondence between them.

An actual parameter ei is an expression which must be of the same type Ti as the corresponding formal parameter ai.

When an agent executes the agent statement, a subagent is activated in two steps:

1. The own variables of the subagent are created as follows:
   (a) The formal parameters a1, a2, . . ., am are created one at a time in the order listed in the formal parameter list. Every formal parameter ai is assigned the value obtained by evaluating the corresponding actual parameter ei.
   (b) The variables x1, x2, . . ., xn defined in the procedure body are created with unpredictable initial values.
2. The subagent is started.

A port operand used as an actual parameter denotes a channel which is accessible both to the subagent and its creator. It is known as an external channel of the subagent.

After the execution of the agent statement, the subagent and its creator run concurrently.

An agent defined by a procedure may activate the procedure recursively. Several agents may also activate the same procedure simultaneously. Every activation creates a new agent with its own variables.

Example

```
sort(succ, out)
```

Port statements

```
PortStatement = "+" PortAccess .
```

The creation of a new channel is called the activation of the channel.
A port statement

```
+c
```

denotes activation of a new channel. The variable access c must be of a known port type T.

When an agent executes the port statement, a new channel with the alphabet given by T is created and a pointer to the channel is assigned to the port variable c. The agent is called the creator of the channel. The channel itself is known as an internal channel of the agent. The channel ceases to exist when its creator terminates. This is called the termination of the channel.

Example

```
+inp
```

Input/output statements

```
InputOutputCommand = OutputCommand | InputCommand .
OutputCommand = PortAccess "!" OutputSymbol .
OutputSymbol = SymbolName [ "(" OutputExpression ")" ] .
OutputExpression = Expression .
InputCommand = PortAccess "?" InputSymbol .
InputSymbol = SymbolName [ "(" InputVariable ")" ] .
InputVariable = VariableAccess .
InputOutputStatement = InputOutputCommand .
```

A communication is the transfer of a symbol from one agent to another through a channel. The sending agent is said to output the symbol, and the receiving agent is said to input the symbol. The agents access the channel through local port variables.

Consider an agent p which accesses a channel through a port variable b, and another agent q which accesses the same channel through a different port variable c. The port variables must be of the same type

```
T = [s1(T1), s2(T2), . . ., sn(Tn)];
```

An output command

```
b!si(ei)
```

denotes output of a symbol si(ei) through the channel denoted by the port variable b. si must be the name of one of the symbol classes of T, and the expression ei must be of the corresponding message type Ti.

An input command

```
c?si(vi)
```

denotes input of a symbol si(vi) through the channel denoted by the port variable c. si must be the name of one of the symbol classes of T, and the variable access vi must be of the corresponding message type Ti.

When an agent p is ready to output the symbol si on a channel, and another agent q is ready to input the same symbol from the same channel, the two agents are said to match and a communication between them is said to be feasible. If and when this happens, the two agents execute the output and input commands simultaneously. The combined effect is defined by the following sequence of actions:

1. p obtains a value by evaluating the output expression ei.
2. q assigns the value to its input variable vi.

(If the symbol si is a signal, steps 1 and 2 denote empty actions.)

After a communication, the communicating agents proceed concurrently.

When an agent reaches an input/output command which denotes a communication that is not feasible, the behaviour of the agent depends on whether the command is used as an input/output statement or as a polling command (defined in the following).

The effect of an input/output statement is to delay an agent until the communication denoted by the statement has taken place.

The communications on a channel take place one at a time. A channel can transfer symbols in both directions between two agents.

A channel may be used by two or more agents. If more than two agents are ready to communicate on the same channel, it may be possible to match them in several different ways. The channel arbitrarily selects two matching agents at a time and lets them communicate.

An input/output command is undefined if the port value does not denote an existing channel.

Examples

```
inp?eos
succ!int(x)
```

If statements

```
IfStatement = "if" BooleanExpression "then" Statement [ "else" Statement ] .
BooleanExpression = Expression .
```

An if statement

```
if B then S1 else S2
```

denotes execution of exactly one of the statements S1 and S2. The expression B must be of type boolean.

The if statement is executed in two steps:

1. A boolean value is obtained by evaluating the expression B.
2. If the value is true, S1 is executed; otherwise, S2 is executed. (If S2 is omitted, it denotes the empty action.)

Examples

```
if succ <> none then out!eos
```

```
if x > y then
  begin succ!int(x); x := y end
else succ!int(y)
```

While statements

```
WhileStatement = "while" BooleanExpression "do" Statement .
```

A while statement

```
while B do S
```

denotes zero or more executions of the statement S. The expression B must be of type boolean.

The while statement is executed by following these steps:

1. A boolean value is obtained by evaluating the expression B.
2. If the value is true, S is executed and afterwards steps 1 and 2 are repeated; otherwise, the execution of the while statement ends.

Example

```
while i <= n do
  begin
    io?readint(x); inp!int(x);
    i := i + 1
  end
```

Polling statements

```
PollingStatement = "poll" GuardedStatementList "end" .
GuardedStatementList = GuardedStatement { "|" GuardedStatement } .
GuardedStatement = Guard "->" StatementList .
Guard = PollingCommand [ "&" PollingExpression ] .
PollingCommand = InputOutputCommand .
PollingExpression = BooleanExpression .
```

A polling statement

```
poll
  C1 & B1 -> SL1 |
  C2 & B2 -> SL2 |
  . . .
  Cn & Bn -> SLn
end
```

denotes execution of exactly one of the guarded statements

```
Ci & Bi -> SLi
```

An agent executes a polling statement in two phases, known as the polling and completion phases:

1. Polling: the agent examines the guards C1 & B1, C2 & B2, . . ., Cn & Bn cyclically until it finds one with a polling command Ci that denotes a feasible communication and a polling expression Bi that denotes true (or is omitted).
2. Completion: the agent executes the selected polling command Ci followed by the corresponding statement list SLi.

While an agent is polling, it can be matched only by another agent that is ready to execute an input/output statement. Two agents polling at the same time do not match.

Example

```
poll
  user?P & x > 0 -> x := x - 1 |
  user?V -> x := x + 1
end
```

Compound statements

```
CompoundStatement = "begin" StatementList "end" .
StatementList = Statement { ";" Statement } .
```

A compound statement

```
begin SL end
```

denotes execution of the statement list SL.

A statement list

```
S1; S2; . . . ; Sn
```

denotes execution of the statements S1, S2, . . ., Sn one at a time in the order written.

Example

```
begin succ!int(x); x := y end
```

AGENT PROCEDURES

```
AgentProcedure = "agent" AgentName ProcedureBlock ";" .
ProcedureBlock = FormalParameterPart ";" ProcedureBody .
FormalParameterPart = [ "(" FormalParameterList ")" ] .
FormalParameterList = ParameterDefinition { ";" ParameterDefinition } .
ParameterDefinition = VariableName { "," VariableName } ":" TypeName .
ProcedureBody = [ ConstantDefinitionPart ] [ TypeDefinitionPart ]
  { AgentProcedure } [ VariableDefinitionPart ] CompoundStatement .
```

An agent procedure

```
agent P(a1:T1; a2:T2; . . . am:Tm);
  . . .
begin SL end;
```

introduces a name P to denote a block which defines a class of agents.

A new agent of this class is activated when another agent executes an agent statement that refers to the procedure.

The formal parameter list

```
a1:T1; a2:T2; . . . am:Tm
```

defines the formal parameters of the agent. T1, T2, . . ., Tm must be the names of known types.

A parameter definition

```
ai:Ti
```

introduces a name ai to denote a formal parameter of type Ti.

A parameter definition

```
a1, a2, . . ., aj:Tk
```

defines formal parameters a1, a2, . . ., aj of the same type Tk.

Every formal parameter is a local variable that is assigned the value of an actual parameter (an expression) when the agent is activated.

After its activation, the new agent executes the procedure body in two steps:

1. The agent executes the compound statement SL.
2. The agent waits until all its subagents (if any) have terminated. At this point, the own variables and internal channels of the agent cease to exist, and the agent terminates.

Example

```
agent semaphore(x: integer; user: PV);
begin
  while true do
    poll
      user?P & x > 0 -> x := x - 1 |
      user?V -> x := x + 1
    end
end;
```

PROGRAMS

Program = [ConstantDefinitionPart] [TypeDefinitionPart] AgentProcedure .

A program is a block which contains a single agent procedure.

A program denotes activation and execution of a single agent defined by the procedure (the initial agent). The activation of the initial agent is the result of executing an agent statement in another program (an operating system). A program communicates with its operating system through the external channels of the initial agent (the system channels). The program execution terminates when the initial agent terminates.

Example

```
{ The program uses a recursive pipeline to sort and output n integers. The port
  variable io denotes a system channel to the operating system. The sorting
  agent is explained in Reference 3. }

type iosym = [readint(integer), writeint(integer)];

agent sortprogram(io: iosym);
type stream = [int(integer), eos];

  agent sort(inp: stream; out: iosym);
  var more: boolean; x, y: integer;
    succ: stream;
  begin
    poll
      inp?int(x) -> +succ;
        sort(succ, out); more := true !
      inp?eos -> more := false
    end;
    while more do
      poll
        inp?int(y) ->
          if x > y then
            begin succ!int(y); x := y end
          else succ!int(y) |
        inp?eos -> out!writeint(x);
          succ!eos; more := false
      end
  end;
```

```
var i, n, x: integer; inp: stream;
begin
  +inp; sort(inp, io);
  io?readint(n); i := 1;
  while i <= n do
    begin
      io?readint(x); inp!int(x);
      i := i + 1
    end;
  inp!eos
end;
```

GRAMMAR

```
Program = [ ConstantDefinitionPart ] [ TypeDefinitionPart ] AgentProcedure .
AgentProcedure = "agent" AgentName ProcedureBlock ";" .
ProcedureBlock = FormalParameterPart ";" ProcedureBody .
FormalParameterPart = [ "(" FormalParameterList ")" ] .
FormalParameterList = ParameterDefinition { ";" ParameterDefinition } .
ParameterDefinition = VariableGroup .
ProcedureBody = [ ConstantDefinitionPart ] [ TypeDefinitionPart ]
{ AgentProcedure } [ VariableDefinitionPart ] CompoundStatement .
ConstantDefinitionPart = "const" ConstantDefinition { ConstantDefinition } .
ConstantDefinition = ConstantName "=" Constant ";" .
TypeDefinitionPart = "type" TypeDefinition { TypeDefinition } .
TypeDefinition = TypeName "=" NewType ";" .
NewType = EnumeratedType | ArrayType | RecordType | PortType .
EnumeratedType = "(" ConstantName { "," ConstantName } ")" .
ArrayType = "array" "[" IndexRange "]" "of" TypeName .
IndexRange = SimpleConstant ". ." SimpleConstant .
RecordType = "record" FieldList "end" .
FieldList = RecordSection { ";" RecordSection } .
RecordSection = FieldName { "," FieldName } ":" TypeName .
PortType = "[" Alphabet "]" .
Alphabet = SymbolClass { "," SymbolClass } .
SymbolClass = SymbolName [ "(" TypeName ")" ] .
VariableDefinitionPart = "var" VariableDefinition { VariableDefinition } .
VariableDefinition = VariableGroup ";" .
VariableGroup = VariableName { "," VariableName } ":" TypeName .
Statement = Empty | AssignmentStatement | AgentStatement | PortStatement
  | InputOutputStatement | IfStatement | WhileStatement | PollingStatement |
  CompoundStatement .
AssignmentStatement = VariableAccess ":=" Expression .
AgentStatement = AgentName [ "(" ActualParameterList ")" ] .
ActualParameterList = Expression { "," Expression } .
PortStatement = "+" PortAccess .
PortAccess = VariableAccess .
InputOutputStatement = InputOutputCommand .
```

```
InputOutputCommand = OutputCommand | InputCommand .
OutputCommand = PortAccess "!" OutputSymbol .
OutputSymbol = SymbolName [ "(" Expression ")" ] .
InputCommand = PortAccess "?" InputSymbol .
InputSymbol = SymbolName [ "(" VariableAccess ")" ] .
IfStatement = "if" Expression "then" Statement [ "else" Statement ] .
WhileStatement = "while" Expression "do" Statement.
PollingStatement = "poll" GuardedStatementList "end" .
GuardedStatementList = GuardedStatement { "|" GuardedStatement } .
GuardedStatement = Guard "->" StatementList .
Guard = InputOutputCommand [ "&" Expression ].
CompoundStatement = "begin" StatementList "end" .
StatementList = Statement { ";" Statement } .
Expression = SimpleExpression [ RelationalOperator SimpleExpression ] .
RelationalOperator = "<" | "=" | ">" | "<=" | "<>" | ">=" .
SimpleExpression = [ SignOperator ] Term { AddingOperator Term } .
SignOperator = "+" | "-" .
AddingOperator = "+" | "-" | "or" .
Term = Factor { MultiplyingOperator Factor }.
MultiplyingOperator = "*" | "/" | "div" | "mod" | "and" .
Factor = Constant | VariableAccess | Constructor | "(" Expression ")" | "not"
  Factor .
Constructor = TypeName "(" ConstructorOperand ")" .
ConstructorOperand = Expression | StringToken .
StringToken = """ { GraphicCharacter } """ .
VariableAccess = VariableName { Selector } .
Selector = IndexedSelector | FieldSelector .
IndexedSelector = "[" Expression "]" .
FieldSelector = "." FieldName .
Constant = SimpleConstant | RealConstant | StructuredConstant .
SimpleConstant = SimpleNumeral | GraphicToken | ControlToken |
  ConstantName.
SimpleNumeral = Digit { Digit } .
GraphicToken = "'" GraphicCharacter "'" .
ControlToken = SimpleNumeral "C" .
RealConstant = RealNumeral | ConstantName .
RealNumeral = Mantissa [ "E" Exponent ] .
Mantissa = Digit { Digit } "." { Digit } .
Exponent = [ "+" | "-" ] SimpleNumeral .
StructuredConstant = "nil" TypeName | ConstantName .
Name = Letter { Letter | Digit } .
```

REFERENCES

1. C. A. R. Hoare, 'Communicating sequential processes', *Comm. ACM*, **21**, 666–677 (1978).
2. N. Wirth, 'The programming language Pascal', *Acta Inform.*, **1**, 35–63 (1971).
3. P. Brinch Hansen, 'Joyce — a programming language for distributed systems', *Software—Practice and Experience*, **17**, 29–50 (1987).