PROCOL
A Protocol-Constrained Concurrent Object-Oriented Language

Jan van den Bos
Department of Computer Science
University of Leiden
P.O. Box 9512, Leiden
The Netherlands
E-mail: vdbos@hleru15.bitnet

Abstract

PROCOL is a simple concurrent object-oriented language supporting a distributed, incremental and dynamic object environment. Its communication is based on unidirectional messages. Objects are only bound during actual message transfer and not during the processing of the message. This short duration object binding promotes parallelism. The communication leading to access has to obey an explicit protocol in each object. It takes the form of a specification of the occurrence and sequencing of the interaction between the object and its communication partners. The use of such protocols fosters structured, safer and potentially verifiable communication between objects.

1 The Protocol-Constrained Concurrent Object Model

PROCOL is a concurrent and distributed language based on objects. Internally these objects are purely sequential programs. Externally objects run in parallel, as long as they are not engaged in communication. The communication is accomplished by message exchange. In PROCOL message transfer is synchronous. The sender of the message waits until the message has been accepted by an intended receiver. A potential receiver is likewise suspended until it acquires the required message. Immediately thereafter both the sender and the receiver resume execution.
This (restricted synchronous) binding is identical to the type of communication binding in CSP. Any processing of the received message is done after the sender has been released. In other words, this synchronous transfer is not equivalent to an extended procedure call such as in ADA or Smalltalk, because that type of binding includes the processing of the message until a result can be returned. In PROCOL a particular message transfer is in one direction only. Hence, relaying messages for processing by other objects can be done routinely.

As a mental model, it is perhaps best to consider each PROCOL object as being assigned to its own processor-memory pair. A communication facility then provides the means to send synchronous messages between the processor-memory pairs.
In PROCOL the exportable object services are internally specified as **Actions.** These actions are similar to the Smalltalk methods or Eiffel routines. The actions are the only interface with the outside world. In general, these actions may depend on one another or on communication with some other object. In PROCOL eligible actions may also depend on the state of the object, because a **Protocol** constrains possible communications, and thus access to the object's actions. Per object, this protocol takes the form of a specification rule over interactions, an interaction being an entity consisting of sender, message, and internal action. The protocol determines the allowable actions in a given state as well as the (partial) ordering of the actions. A run-time communication is legal when the object is in the right state, and when the actual sender and requested action correspond with sender and action specified in the protocol interaction(s) for that object state. Receipt of the message denoted in the interaction triggers the action. The form of the protocol has been derived from our work on input expressions (*ACM-Toplas 10,2,1988*) controlling man-machine interaction, which were inspired by the path expressions of Habermann.
An object type (abstract data type) is defined by means of a (program) text. Objects become active when created (allocated) by means of the **new** primitive. Attributes may be used to tailor a particular object. The attribute list is passed to the object as part of the **new** primitive, and bound to the object through the initialization routine defined in the **Init** section. Example:

> **Declare** *z : ObjA;*
> *z* := **new** *ObjA (attr1, attr2,...)*

in which *ObjA* is a defined object type, *z* a variable of the type *ObjA*, and *attri* attributes of *ObjA*. After succesfully executing the statement, the variable *z* contains the identity of the object.
The object issuing the **new** primitive is known as the *Creator* to the object created. Object removal is accomplished by means of the del primitive, to be issued by the *Creator*. Before the object is physically removed the **Cleanup** section is executed (only once). Object creation imposes a certain hierarchy between objects. Originally, the identity of the created object is only known to its *Creator*. The identity may however be passed to other objects as part of an attribute list or a message.

Objects in PROCOL coexist with some set of basic types as present in most languages. PROCOL object types do not emulate these basic types. PROCOL uses facilities such as procedures, expressions, and assignments of some imperative host programming language.
PROCOL knows one special object type to indicate a collection of regular object types. The type has the name *any* with optional parameters enclosed in square brackets. The parameters indicate the object types involved. If the parameter list is absent, the universe of object types is meant. For example:

Declare *z : any; x,y : any [ObjA, ObjB]*

As a result x and y can be of type *ObjA* or of type *ObjB*, but not simultaneously. The variable z can be any object type, but again one at a time. The actual type is determined by explicit assignment or implicitly via a message. It is also in this connection that the object *ANY* is employed. This name is optionally used with a parameter list equal to the one with the type any. For example, with the declarations just given in force:

x := ANY [ObjA];
y := ANY [ObjA, ObjB];
z := ANY

means that x is any instance of type *ObjA*, y is any instance of type *ObjA* or *ObjB* (the assignment to y may be replaced by the shorthand *y:=ANY*), and z stands for any instance of all object types. The literals *Creator* and *ANY* may be used wherever an object type variable is allowed.
A PROCOL object definition consists of the following parts:

OBJ	*Name Attributes*
Description	natural language description
Declare	local type definitions, data, procs
Init	section executed once at creation
Protocol	(sender-message-action)-expression
Cleanup	section executed once at deletion
Actions	definitions of local actions
Endobj	*Name*

2 Actions

The so-called **Actions** section in a PROCOL object definition contains the definitions of the actions to which other object may send messages. The names of the actions are known externally. The action is executed when the correct type of message is received from the right source object, as indicated in the protocol. Messages to other object may be sent from within the actions specified in the **Actions** section, but also from within the **Init** and **Cleanup** sections. Reception, meaning acceptance of the message, can only take place when the action is allowed according to the protocol. An action body may contain any executable code. In particular it may include sending a message to an object using the syntax (square brackets enclose options):

[rtc:=] OtherObject.[Action-Name] [msg]

OtherObject contains the identity of the created object. *Action-Name* is the name of the action in *OtherObject* to which message *msg* is sent for processing. The success or failure of the send is transmitted through the boolean variable *rtc*. Failure occurs when the object does not exist.
Actions have an atomic nature: the object processing an action first completes it before it can receive any new message. In other words only one action per object can be in progress at a time.

3 PROCOL Protocol

During the life of a PROCOL object, the corresponding protocol repeats until the object is deleted. This protocol is an expression over *interaction* terms. An interaction term specifies the communication partner, the type of message involved, and the action to be executed. The form of an interaction term is:

SourceObject [msg] $\rightarrow$ *Action.*

The semantics is that upon receipt of message *msg*, from source *SourceObject*, action *Action* will be executed.
The protocol contains expressions constructed by the following four operators *selection* +, *sequence* ;, *repetition* *, and *guards* (in increasing precedence). The guard is a boolean expression opening or closing a gate to the interaction E. Given expressions E and F, and *guard* φ, their meaning is as follows:

E	+	F	*selection:*	E or F is selected
E	;	F	*sequence:*	E is followed by F
E	*		*repetition:*	Zero of more times E
φ	:	E	*guard:*	E only if φ is true

4 PROCOL Examples

4.1 Square Root Pipeline

This pipeline consists of one object of type *Sqroot* and any number of pipeline elements *NRStep*. The square root of x is calculated by a series of approximations according to the Newton-Raphson method. *Sqroot* creates an object *NRStep* which computes the next estimate. This object creates a next *NRStep* for a new estimate. This creation and approximation process goes on until the present estimate and the previous one differ less than some value *eps*. The present estimate is then returned to the user. As soon as *Sqroot* has passed its argument to the first instance of *NRStep*, it is ready to handle a second square root request, because the pipeline remains in existence. Once the pipeline is filled, n computations are in progress simultaneously.

```
OBJ       Sqroot

Declare   x : REAL; initpipe : BOOL;
          Client : any; Child : NRStep

Init      initpipe := TRUE

Protocol  ANY(x)→Compute

Actions   Compute = { Client := sender;
                      IF initpipe THEN new Child;
                         initpipe := FALSE FI;
                      Child.Compute(x,0.5*x+1,Client) }
EndOBJ Sqroot.
```

```
OBJ       NRStep

Declare   x, Est, NewEst, eps : REAL; initpipe, done: BOOL;
          Client : any; Child : NRStep

Init      eps := 0.0001; initpipe := TRUE

Protocol  Creator(x,Est,Client)→Compute

Actions   Compute = { NewEst := 0.5*(Est+Est/x);
                      done := abs(1-NewEst/Est) < eps;
                      IF done THEN Client.(NewEst);
                      ELSE IF initpipe THEN new Child;
                              initpipe := FALSE FI;
                           Child.Compute(x,Est,Client)
                      FI }
EndOBJ NRStep.
```

4.2 Mastermind

This familiar game is modeled here as a parent object, ***Mastermind***, which creates two children, instances of ***Player*** and ***Opponent***. ***Mastermind*** does little else but wait for a ready signal from its two siblings. Player and an Opponent play with pawns in 7 colors. Opponent determines a sequence of 4 pawns, called the code. Player tries to guess the code. Opponent evaluates the guess and informs Player of the number of bulls (position and color correct) and cows (color correct, not including the bulls). Player now determines a new guess. ***Player*** and ***Opponent*** are presented only.

Player's protocol starts by sending a random guess from action *Start*, to its opponent. ***Opponent*** monitors the number of turns allowed and evaluates the correctness of the guess, which is sent to ***Player***. ***Player*** determines a new guess in action ***Makeguess*** and sends it to ***Opponent***. This may be repeated (*) a number of times, until either the code is guessed or ***maxguess*** has been exceeded. If so ***Opponent*** returns the score to action ***Stop*** in ***Player***, and the repetition terminates. If ***bulls=4*** ***Player*** sends a celebration signal to action ***blow*** of any existing object of type ***Horn***. Finally it sends a (completion) message to its creator Mastermind.

```
OBJ       Player

Declare   opponent : Opponent;
          i, bulls, cows: INT;
          guess : ARRAY [1..4] OF INT;
          proc EducatedGuess = ...

Protocol  Creator(opponent)→Start;
          opponent(bulls, cows)→Makeguess *;
          opponent(bulls)→Stop

Actions   Start     = { FOR i:=0 TO 4
                           guess[i] = Random(1, 7);
                        opponent.Eval(guess); }
          Makeguess = { EducatedGuess;
                        opponent.Eval(guess); }
          Stop      = { IF (bulls = 4) THEN
                           ANY[Horn].blow;
                        FI;
                        Creator.EndPlayer; }

EndOBJ    Player.
```

```
OBJ       Opponent

Declare   player : Player; count, i, bulls, cows : INT;
          code, guess : ARRAY [1..4] OF INT;
          notend : bool;
          proc Determinescore = ...

Init      FOR i:=0 TO 4 code[i] := Random(1,7);
          player := ANY[Player]; count := 1;
          notend : true;

Protocol  notend: player(guess)→Eval

Actions   Eval = {IF count=1 THEN player:=sender FI;
                  Determinescore; count:=count+1;
                  notend:=count≤maxguess and bulls<4;
                  IF notend THEN
                        player.Makeguess(bulls,cows);
                  ELSE player.Stop(bulls);
                        Creator.EndOpp(bulls=4)
                  FI };
EndOBJ    Opponent.
```