Acta Informatica 28:511–538 (1991)

Acta Informatica
© Springer-Verlag 1991

PROCOL

A concurrent object-oriented language with protocols delegation and constraints

Jan van den Bos[1] **and Chris Laffra**[2]*

[1] Department of Computer Science, Erasmus University, P.O. Box 1738, 3000 DR Rotterdam, The Netherlands
[2] SERC – Software Engineering Research Center, P.O. Box 424, 3500 AK Utrecht, The Netherlands

Received April 9, 1990 / March 15, 1991

Abstract. PROCOL is an object-oriented language with distributed delegation. It strongly supports concurrency: many objects may be active simultaneously, they execute in parallel unless engaged in communication. An object has exported operations, called Actions. Only one Action can be active at a time, however special interrupt Actions may interrupt regular Actions. Communication is performed via remote procedure call, or via a one-way synchronous message with short-time binding. In communications both client and server can be specified, either by object instance identifiers, or by type. Therefore client-server mappings may be $1-1$, $n-1$, or $1-n$, though only 1 message is transferred. PROCOL controls object access by an explicit per-object protocol. This protocol is a specification of the legality and serialization of the interaction between the object and its clients. It also provides for client type checking. The use of protocols in object communication fosters structured, safer and potentially verifiable information exchange between objects. The protocol also plays an important role as a partial interface specification. In addition it acts as a composition rule over client objects, representing relations with the client objects. PROCOL's communication binding is dynamic (run-time); it functions therefore naturally in a distributed, incremental and dynamic object environment. PROCOL also supports constraints, without compromising information hiding. An implementation is available in the form of a *C* extension.

1. Introduction

Objects provide self-contained state spaces and a collection of public operations on data private to that space. They were first introduced in Simula [7]. Smalltalk

* This author is supported by the Dutch Science Organization NWO

[28] further refined objects as language constructs and made object-oriented languages (OOL) popular. Actors [10] as well as CSP [11] offer computation models that try to provide 'laws' for communication. Because of their data and procedure encapsulation properties, objects are a natural construct for abstract data types, as in CLU [14] and Alphard [18]. The information hiding properties of abstract data types are considered very important in software engineering. They allow natural modular boundaries between weakly connected pieces of program. As a consequence, OOLs provide an attractive tool for programming projects.

Communication in general may be characterized by the type and duration of the binding between sender (client) and receiver (server). A coarse distinction is synchronous versus asynchronous communication. To be more precise, we distinguish 6 types of binding. In the first two, the operation (method, Action) executed is the basis of synchronization. The latter four use the message as the basis of synchronization (if any). Listed in order of decreasing binding time and thus increasing parallelism, they are:

- *Rpc*: remote procedure call – bound from the time of call until processing in the server completes and a result is returned: Smalltalk [28];
- Rpc^-: early return to client, with post-processing in server: Ada [12], Pool [2], PROCOL;
- $S_s R_s$: Synchronous send and receive: CSP [11], PROCOL;
- $S_a R_s$: Asynchronous send with synchronous receive: Plits [8];
- $S_s R_a$: Synchronous send with asynchronous receive;
- $S_a R_a$: Asynchronous send and receive.

It is hard to find examples of the last two categories, because in programming systems the asynchronous receive does not appear to be a very practical way to read messages, except perhaps in a polling situation. Ironically, the national Mail system operates under regime $S_a R_a$. Perhaps, there is a lesson to be learned here.

1.1 Access control

Controlling communication is the weak point of parallel programming. For OOLs this translates into a lack of explicit access control mechanisms to public operations used by client objects. In general, not all operations may be called at any time, or by any client. It often happens that one operation in an object must precede some other operation in the same object. For example, a file handler requires a file to be opened first before any read or write operation may begin. It may also happen that once a particular operation has been executed, access to some other operation is (temporarily) disallowed. It should be prevented that faulty or premature communication be the cause of erroneous situations. To better protect an object against unscheduled actions, some form of explicit access control is needed.

The lack of access control mechanisms in existing OOLs is directly connected to the lack of symmetry between sending and receiving. Whereas in the send operation the target object and its public operation must be specified, the role of the receiver is a passive one[1]. Its operations can apparently be accessed

[1] In the non-OOL area, CSP [11] and Plits [8] allow identification of both sender and receiver

indiscriminately, and at any time. The receiver is not even aware of the identity of the client, hence the appropriate name Server for the receiving object. This asymmetric situation, usually based on the *rpc*, introduces an undesirable master-slave relation between client and server. Such an artificial relationship is inconsistent with the OO paradigm in which (at least in first order) all objects are equal. Access control is a natural element of the autonomy of an object: it should therefore be exercized by the object itself.

2 The PROCOL model

2.1 Concurrency

Most present OOLs [28, 15, 20, 6, 17] are sequential languages. Their structure does not seem to cater for elegant concurrency extensions. This is unfortunate: as self-contained pieces of program, objects are natural constructs for parallel and even distributed processing. In principle, each object can be assigned its own (micro) processor with memory, to be run as a largely independent sequential process. A software system would in that case consist of a number of cooperating objects, with communication between objects performed on the basis of message exchange. Concurrency ought to be a basic requirement for an OOL, rather than an ad hoc added feature. This demand precludes the use of the regular *rpc* as the only communication primitive: in systems with one root object [28, 15, 20, 6, 17] its use leads by definition to a sequential language. An exception is Pool [2]. Its object instances may contain a permanently active *body*. However, in practice this body plays the role of object initialization as well as embedded protocol parser (see Sect. 2.5 and 4), blocking whenever a Pool method becomes available for access.

PROCOL (for PROtocol-controlled Concurrent Object Language), presented in this paper, is a parallel and distributed language based on objects. Brief, preliminary versions of PROCOL have been published elsewhere [25, 26, 27]. This paper presents a complete overview of the language. It adds language syntax, protocol syntax and semantics, a new communication primitive (request), parallel delegation as a dynamic reuse facility, parallel protocols and a language construct to define constraints, while persistent objects are being considered.

PROCOL's communication primitives are the one-way *send*, and the round-trip *request*, short for request-with-reply. Both primitives transfer messages between objects. Both primitives allow for concurrency, the *send* more than the *request*.

Internally an object executes sequentially. Externally, in relation to other objects, objects run in parallel, as long as they are not engaged in communication. The channeling of information from one object to another is accomplished by message exchange. As a mental model, it is perhaps best to consider each PROCOL object as being assigned its own processor-memory pair. A communication facility then provides the means to send synchronous messages between the processor-memory pairs.

For the sake of the following discussion an object is defined as an instance of an object type. This object type is similar to an abstract data type. The object contains local data structures and procedures, as well as public operations here called *Actions*. These Actions are similar to the Smalltalk *methods* [28] or Eiffel *routines* [15]. Furthermore an object type may carry attributes which are part of the local data. An object possesses a local state, defined by the local data. The values of the local data are preserved from one invocation of an object Action to another one. Communication excepted, objects are completely self-contained. Thus there exist no facilities for importing or exporting data types, data structures, or procedures, except in an indirect way via messages. Only through message transfer and the internal computation it triggers in an Action, can the state of the object be changed. Furthermore, only one copy of an instance exists, avoiding complex synchronization problems such as in Orca [3].

2.2 Delegation

Many OOLs offer inheritance, a facility to inherit and thus reuse methods and/or data structures. Inheritance, by its nature, exposes internals of an object. So in principle, it runs counter to the idea of information hiding. In addition, general reuse implies multiple inheritance, a feature that seems hard to accept to many OOL builders.

An alternative for reuse is delegation [1, 21, 19]. One approach is to start with a number of so-called prototype objects. Incarnations of prototype objects may contain additional methods. New incarnations may be added in a hierarchical way. But different from inheritance all these objects co-exist. An incarnation can handle both messages for its prototype methods, as well as for methods it possesses itself. In fact the messages for the prototype methods are delegated to the prototype, rather than processed by the incarnation itself. In the literature [19] inheritance and delegation are often equated; we will not enter into this discussion, since we are only interested in the reuse of methods.

We decided to use delegation in PROCOL for two reasons. The first one is better hiding of information. The second is that in a dynamic object environment contacts between objects, and thus reuse, are short-lived. Inheritance has a static nature. It is usually a compile-time feature: once inherited stays inherited. Delegation is more like dynamic linking and delinking, and therefore much more suitable to a dynamic object environment.

PROCOL may delegate entire Actions, acting as a kind of replacement behavior, such as in Actors [1]. But it also allows the delegation of parts of Actions. In other words PROCOL offers a form of distributed delegation. Delegation may be nested, and different parts of Actions may be delegated to different delegates. Whatever way it is used, the client is not aware of delegations in servers. Moreover, delegated Actions run in parallel with possible Actions in the delegating object.

2.3 Protocol

Few existing OOLs offer ways to *explicitly* control access to the object's Actions. Some control may be exercized in languages such as Sina [21] and Pool [2]. Implicit control may be effected by inspecting local switches or booleans, or by more or less explicit preconditions [15, 3], but by then it is already too late: the client has in effect accessed the object, possibly causing it to block. Explicit controls would regulate access to the object without first letting a client 'in'. It would keep a client for a certain Action pending, as long as the Action was not available for whatever reason (e.g. wrong object state, illegal client type). This would not only provide for cleaner code, but more importantly, it would give extra protection in the quest for information hiding.

PROCOL introduces a *Protocol* per object. It provides facilities for access control. The protocol offers the tools to (partially) order and restrict possible communications, and thus access to the object's Actions. It is specified in the form of augmented regular expressions. The protocol is influenced by the state of the object and the history of communication. The protocol is not an executable part of an object, but it functions as a declarative specification rule for legal communication between a server and its clients. At execution time the protocol is parsed. The state of the protocol determines the Actions that may be legally accessed at a particular time. Matching a client request to an object Action changes the state of the protocol. Thus the protocol provides the means to support the safe proceeding of accessing objects. However, the system designer exercizes his own discretion to what extent he uses these means. The protocol section of an object is an orthogonal addition: without it the object would be usable just as well, but the onus of access control would be on the client, rather than on the server where it should be under the object paradigm.

2.4 *Object types*

An object type is defined by means of a piece of (program) text. Its definition consists of the following parts:

Obj	*Name Attributes*
Description	natural language description
Declare	local data, procedures, type definitions
Protocol	(sender-message-action)-expressions
Init	section executed once at creation
Cleanup	section executed once at deletion
Actions	definition of public Actions
IntActions	definition of public interrupt Actions
EndObj	*Name*

Objects are created (allocated) by means of the `new` primitive, with the object variable as an argument, as follows[2]:

```
Declare z: OBJA;
new z (attr1, attr2, ...)
```

in which `z` is an object variable of the type `OBJA`, and the `attr1`, `attr2` are attributes of `OBJA`. After executing the statement, the variable `z` contains the identity of the object. Copying of object identities can be done via assignment.

[2] Henceforth capitals will be used to indicate object and other types, instantiations will have names written in mixed or lower case

PROCOL knows one special object type to indicate any from the universe of object types. The type has the name `ANY`. Variables of this type may not be the subject of a `new` operation.

An object may possess optional attributes, to tailor a particular object. Attribute types can be basic (real, int, etc.) or object types, including `ANY`. The attribute list is passed to the object as part of the `new` primitive. Creation also implies the (one-time only) execution of the **Init** section. The attributes may of course be used to tailor the initialization desired; this provides similar flexibility as multiple 'create' routines in some other OOLs.

The object issuing the `new` primitive is known as the `Creator` to the object created. The literal `Creator` may be used wherever an object type variable is allowed. Object removal is accomplished by the `del` primitive, to be issued by the `Creator` only. 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. The primitives `new` and `del` are actually builtin *requests.*

The **Declare** section contains the declarations of local variables, object instantiations, constants, and procedures. The message component variables are also declared here, and not in the Actions. The Actions themselves do not contain (local) declarations.

The **Protocol** section regulates access to the object. It is discussed in Sect. 4.

Actions and **IntActions** define the public operations of the object. They can be called by the *send, delegate* and *request* communication primitives. For further details see Sects. 3 and 5.

The entire language PROCOL could be built exclusively on object types. However, we preferred to have PROCOL coexist with some set of basic types as present in most languages. This avoids the quirky notation when a pure object orientation is adopted. It also makes it possible to graft PROCOL on a standard imperative language such as C (the host language of our current implementation, see Sect. 8), or Pascal, in order to use facilities like procedures, expressions, and assignments.

In summing up, the salient points of PROCOL are the following:

- high degree of concurrency and distribution;
- object is the grain of parallelism;
- communication primitives are *send* and *request*;
- communication is $1-1$, $1-n$, or $n-1$;
- for *send*, objects are only bound during message transfer;
- per object only one Action (method) can be active at a time;
- access is controlled by a per-object protocol;
- protocols allow for client type, identity checking;
- protocols provide a serialization mechanism.

2.5 Semantics

When the `new` primitive is executed on a variable of object type `O`, an instance of that type is created. Let us assume that the identity of this instance is `Oi`.

This identity is returned to the issuer of the `new` primitive. Creation may fail for a number of reasons. The system may not be able to find the definition of `O`, or there may be insufficient storage for a new instance. In that case a reserved identity is returned. If creation is successful, the identity is returned. Then the list of attributes specified as part of the `new` primitive is passed by value to `Oi` and there copied to a local attribute list. Next the **Init** section of `Oi` is executed. Subsequently, the protocol parser is given control. It determines which Actions are available for access by other objects. The parser remains in control until a legal communication (see Sect. 4) arrives. The message is processed by the Action specified in the communication. When processing is finished, control returns to the protocol parser, which determines the next legal communication(s) by inspecting the protocol. Parser and Actions execute in turn until the `Creator` issues the `del` primitive on `Oi`. First any Action execution in progress in `Oi` has to complete. Subsequently `Oi` executes its **Cleanup** section. Finally `Oi` is deleted.

All executable sections of a PROCOL object (**Init**, **Cleanup**, **Actions** and **IntActions**) consist of code derived from some imperative host language, the **Declare** section contains the declarations of variables and constants of the host language. In other words, PROCOL is not a completely new language, but rather an extension of an existing imperative language.

3 PROCOL actions

Both the **Actions** and the **IntActions** section in a PROCOL object contain definitions of public Actions: Actions to which other objects may send messages. The names of the Actions are known externally. Execution of an Action is triggered when the correct type of message, aimed at this Action, is received from the right source object, as specified in the protocol. Messages to other objects may be sent from within Actions, but also from the **Init** and **Cleanup** sections.

An Action has the following syntax:

$$ActionName = body$$

The body may contain any executable code. In particular it may include communication or constraints (see Sect. 6).

Actions are not mandatory. Sometimes the only thing an object does is to create other objects in its Init section.

3.1 Communication

The *send* primitive obeys the communication regime $S_s R_s$ (see Sect. 1). The sender of the message waits until the message has been accepted by an intended receiver. The potential receiver is likewise suspended until it acquires the required message. Receipt of the message consists of copying the values of the message's components to variables local to the object, as specified in the message part of the protocol. Immediately after receipt of the message the receiver starts the execution of the Action indicated, while the sender resumes execution. In other words Action execution starts (immediately) after the sender has been released. This (restricted synchronous) binding is identical to the communication binding in CSP [11]. Relaying messages for processing by other objects is quite possible. In contrast to most OOLs, communication mapping in PROCOL can be $1-1$, as well as $1-n$ and $n-1$, although in all cases only one message is exchanged. So strictly speaking, n refers to the number of potential senders or receivers.

The *request* primitive belongs to communication type rpc^{-} (see Sect. 1). It is comparable to the type of communication in ADA [12] and Smalltalk [28]. Sender and receiver are bound until a result is returned. In Smalltalk this implies that the execution of the method be completed. In Ada early return from the method is possible, while the server continues with post-processing. In PROCOL the result is returned by means of a *send.* This *send* may occur at any place in the Action where the result is known. This is analogous to early return. But in addition, only the client knows whether the server Action was triggered by a *request* or by a *send*, another contribution to information hiding.

Sending a message to an object, via a *send*, uses the syntax:

TargetObject. ActionName msg

ActionName is the name of the Action in *TargetObject* to which message *msg* is sent for processing.

The *request* uses a generalization of the *send* syntax:

$$TargetObject.\ ActionName\ msg \rightarrow mes$$

The result values returned by the *request* are deposited in the variables indicated in the list (message) *mes.* They must originate from a single *send* in the server.

Both *msg* (in its evaluated form) and *mes* are messages consisting of an ordered list of variable names. The variables may be of arbitrary type, including object types.

The *send* has to be matched by a *receive* (see Sect. 4.1). If this is not the case, the *send* eventually times out, as indicated by a return code.

In most cases *TargetObject* is an object variable containing the identity of the receiving object. However, in contrast to conventional OOLs, a receiver may also be selected from an indicated *set* of receivers. In PROCOL, the set is specified by the *name* of the object *type.* Although a single communication primitive always transfers only one message from sender to receiver, using a type name amounts to $1-n$ communication mapping. So in general, *TargetObject* can be the following:

- a variable containing the identity of a particular object;
- one of the constants `Creator`, `Receiver` or `Sender`;
- the name of an object type, indicating *any* instance of that type;
- ANY, indicating any instance from the universe of object types.

The first two cases correspond with $1-1$ communication mapping, the latter two with $1-n$ mapping. `Receiver` [22] is the primitive which yields the name of the object that actually received the latest message from the object issuing the primitive, while `Sender` is the primitive yielding the name of the object that sent the latest accepted message accepted by the object issuing this primitive.

`Creator`, `Receiver`, and `Sender` may be assigned to an object variable of the proper type.
ActionName can be the following:

– the name of an Action in *TargetObject*;
– empty: in this case the receiving object will dynamically bind the message to an appropriate Action, in conformance with the state of the protocol (see Sect. 4.1);
– a variable of type `string`, containing the name of an Action.

Only one Action per object can be in execution at a time. Normally the object processing an Action first completes it before it can receive any new message. However, interrupt Actions, if present, may temporarily interrupt an ongoing (non-interrupt) Action (see Sect. 5).

When an object is deleted by its `Creator`, the Cleanup section is executed. Among other things, the section may take care of the release of resources acquired in Actions and/or in the Init section.

For details on the syntax of these sections, see the Appendix.

3.2 Delegation

An Action may be *delegated* in its entirety or in part to one or more Actions in another object. This allows simple forwarding of messages. But messages may also be altered before forwarding. Moreover, the Action may be distributed over several delegates. The *delegation* primitive belongs to the communication type $S_s R_s$ (see Sect. 1). As a consequence, all distributed delegations may run in parallel. The *delegation* primitive uses the syntax:

@ DelegateObject. ActionName msg

in which *msg* is not necessarily identical to the message received by the delegating Action. For the client delegation is transparent. Its server (`Receiver`) remains the object that it communicated with initially. Analogously, for the delegate object the client (`Sender`) remains the original requestor. Technically, delegation is a *send* in which the sending object is disguised as the original client. The implication is that as soon as the message has been transferred, control returns to the delegating Action. Multiple levels of delegation (nesting) may occur. Request type delegation is not needed, because the physical server (the delegator) does not know whether the client triggered it by a *request* or by a *send* (see Sect. 3.1). There are no protocol consequences (see Sect. 4.1).

4 PROCOL protocol

4.1 Interactions

Every object with Actions requires an internal **Protocol** section. This section specifies a (possibly compound) protocol. A compound protocol consists of one or more (simple) protocols. A protocol in a object regulates the message traffic *to* the object by ordering access in time, and by allowing/disallowing messages.

Syntactically, a protocol is an expression over *interaction* terms. An interaction term couples the reception of a message (*receive*) to an Action[3]. To that end it specifies the client (sender), the message involved, and the Action to be executed. The expressions are regular expressions augmented by state controlled predicates called *guards*. The expressions provide for sequencing, alternatives, and (conditional) repetition of interactions. The state of the object and the type of communication and communicators influence the protocol. Variables in the protocol are set from the Init section or Actions, or from attributes or message fields; but, as stated earlier, a protocol is a declarative section and there is no such thing as executing a protocol.

The form of an interaction term is:

$$SourceObject\ mes \rightarrow ActionName.$$

The semantics is that upon receipt of *mes* from *SourceObject*, the Action with name *ActionName* will be executed. *ActionName* has the same syntax as in the *send* primitive (see Sect. 3.1), and *mes* serves to elaborate the message components.

SourceObject can be a variable of the appropriate object type, or one of the constants `Creator`, `Sender` or `Receiver`. These four cases correspond with $1-1$ communication mapping. But *SourceObject* may also be the name of an object type, or `ANY`. Use of `ANY` indicates that any sender will satisfy this interaction. In the latter cases *SourceObject* indicates a set of potential senders, hence this corresponds to $n-1$ communication mapping, even though only a single message is received (cf. the discussion of *TargetObject* in Sect. 3.1).

External communication with an object is allowed when the communication matches the current interaction term in one of its protocols. Matching occurs when the sending object and the Action requested correspond with the entities *SourceObject* and *ActionName* as specified in the interaction term in the protocol. The current interaction term in a protocol is determined by the protocol expression and the history of communications pertaining to this protocol. A compound protocol acts for the object as a number of parallel protocols.

The protocol is (repeatedly) traversed. The state of the protocol, and therefore of the object, changes whenever an interaction has been matched, and as a consequence a new term (interaction) in the protocol becomes the current one. This is analogous to the parsing process of a compiler with the protocol playing the role of grammar. Each (simple) protocol may be considered equivalent to a single grammatical production.

If the identity of the sending object is unknown (when `ANY` or the name of an object type is specified), the receiver may obtain it by issuing the `Sender` primitive from the Action, e.g. `Customer = Sender`, assuming that `Customer` is an object variable of the proper type.

Delegation is transparent to the protocol, in other words the protocol always specifies the physical source object.

Finally, when an object is deleted by its `Creator`, its protocol is interrupted after the present interaction has been completed.

The protocol plays a number of roles in an object:

[3] Only *receives* occur in the protocol; for a not entirely satisfactory experiment with *sends* and *receives*, see [24]

- it is an interface specification to other objects;
- it sequences interactions between objects;
- it controls access to the Actions of the object;
- it may perform type, identity checking on clients;
- it functions as a composition rule, because it specifies relations with client objects.

The serializing properties of protocols strongly depend on synchronous message exchange without queues, and the fact that access to the object is locked when an Action in the object is executing.

The protocol allows client sends and requests that do not specify the name of the desired Action: the state of the protocol determines the Action accessible. This could promote further information hiding in the server object.

Interactions in the protocol and communication primitives provide potential symmetry between sending and receiving: both can name their communication partner. In a world of autonomous objects, this symmetry appears to be a logical requirement.

4.2 Expressions

The protocol consists of expressions constructed with 4 operators: *selection* +, *sequence* ;, *repetition* *, and *guard*: (in increasing precedence). Given interaction 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

Evaluation of the expressions is from left to right, unless operators of higher precedence are encountered. Parentheses may be used to delimit subexpressions for reasons of clarity or precedence. The repetition operator is the equivalent of the Kleene star. The expressions are syntactically similar to the path expressions discussed in [5], and to the input expressions for human-computer interaction control proposed in [22, 23]. Semantically the expressions differ from [5] (but less from [22, 23]) in that they specify (ordered) communication patterns between senders and receivers.

An interaction may be indicated by `skip`. This means that the resulting expression is in effect empty and can therefore be skipped.

We will illustrate the expression operators with a few simple examples. Let us assume that we specify the protocol in a server object `S` with Actions `actionA` and `actionB` triggered by messages `msg1` and `msg2`. There also exist clients `A` and `B`. Then the following protocol in `S`

```
ANY(msg1)→actionA+ANY(msg2)→actionB
```

is equivalent to no protocol as far as access control is concerned, because it specifies that any client may send `msg1` to trigger `actionA`, or `msg2` to trigger `actionB`. The protocol does however play a role as interface specification for the outside world.

The following protocol in `S`

```
A(msg1)→actionA+B(msg2)→actionB
```

specifies that only clients `A` and `B` have access to object `S`; `A` can send a message that triggers `actionA` and `B` can send a message that triggers `actionB`. This with the exclusion of any other access patterns.

The following protocol in `S`

```
A(msg1)→actionA;A(msg2)→actionB
```

specifies that only client `A` is allowed access to object `S`; furthermore `A` first has to send a message `S.actionA(msg1)` to trigger `actionA`, before it can send a message `S.actionB(msg2)` to trigger `actionB`. Since the protocol repeats this doublet of messages repeats as well.

A slight change in the `S` protocol

```
A(msg1)→actionA;B(msg2)→actionB
```

specifies that first client `A` is allowed access to object `S`; once `A` has sent a message `S.actionA(msg1)` that triggered `actionA`, `B` obtains access rights to `actionB` of `S`, to be effectuated by the message `S.actionB(msg2)`.

Although the protocol as a whole repeats, it is sometimes necessary to repeat a subexpression in it as well. The following protocol accomplishes that:

```
A(msg1)→actionA*;B(msg2)→actionB
```

It specifies that client `A` may trigger `actionA` zero or more times, and that such a repetition always must be concluded by `B` triggering `actionB` once. Hence the interaction with object `B` serves as a terminator in this example.

Given interactions `V`, `X`, `Y`, `Z`, (the components of which do not interest us here). Then a protocol

```
V;Z+X;Y
```

specifies that interactions always occur in pairs `V` followed by `Z`, or `X` followed by `Y`. For example, over a period of time the sequence `V Z V Z X Y` is legal, while `V Z X Z X Y` is illegal, and can therefore not take place.

A compound protocol consists of two or more (simple) protocols separated by the || operator. All constituent protocols are simultaneously active. As an example take an object that performs interactions `OpenR`, `Read`, `CloseR`, `OpenW`, `Write`, and `CloseW`, for reading and writing files. By using the following compound protocol:

```
OpenR; Read*;  CloseR
||
OpenW; Write*; CloseW
```

read and write operations may occur interleaved, as long as each of them maintains the usual *open*, *read/write*, *close* temporal order.

4.3 Guards

To extend the power of the (so far regular) expressions predicates or guards may be used. The guard is a test preceding an interaction term or an interaction expression. This test may include any variable local to the object (i.e. occurring in the Declare section or attribute list). It is evaluated before any actual communication as specified in the interaction expression subjected to the guard takes place. A guard's evaluation yields true or false. It can be used to receive a message conditionally, and thus to execute the corresponding Action conditionally. If more than one guard occurs in a selection they are all evaluated. In general a guard is a function without side-effects, testing some aspect of the state of the object.

The form of a guard is a test expression followed by a colon (:). As an interaction expression operator, the guard operator has the highest precedence.

As an example, assume interactions X and Y, and two guards φ and ψ; then the protocol

$$\varphi\texttt{:X}+\psi\texttt{:Y}$$

specifies that those interactions are legal of which the guards are true. The guards are set from inside Actions. A ping-pong game may be played by alternatively setting one guard true and the other one false, and vice versa. Note that this is similar to the protocol X;Y, though not identical, since the guarded ping-pong protocol may start with an interaction Y. Clients which issue requests that cannot be honored remain pending, until they can be satisfied due to a change of state of the object that makes the corresponding guard true. Any Action that completes causes a re-evaluation of the pertinent guards.

Further details on protocol syntax and semantics may be found in the Appendix.

4.4 Discussion

The regular expressions of the protocol, augmented by guards, can specify quite complex access restrictions. However, the protocol is intended as support for access control and temporal ordering. It was not meant as an exhaustive restriction language. For instance, it does not allow the acceptance of a message to depend on the value of a message. It is also not possible to inquire after the number of senders waiting to be served by their intended receiver: the PROCOL programmer is not aware of any queuing. Priority interactions do also not occur in the language. These provisions do not belong at the language level. If need be, it could be easily programmed in an intermediate object serving as priority handler/assigner.

The protocol is meant as an orthogonal addition to a PROCOL object. In principle, objects could all run without protocols. In the implementation of PROCOL it is advisable, as we did, to make protocols mandatory (except in objects without Actions), because the protocol at a minimum serves as a clear interface specification of the object concerned.

5 Interrupt actions

Interrupt Actions are public Actions that may be accessed via a message, just as normal Actions. However, when a normal Action is in progress, the interrupt

Action in the same object has priority. The execution of the normal Action is suspended and the interrupt Action is executed. Once the latter is completed, execution resumes at the point of suspension in the normal Action.

To be allowed access, interrupts have to appear in a protocol of a compound protocol. Interrupt Actions cannot interrupt themselves or other interrupt Actions. Interrupts may be masked by any Action using the device of guards. For an example, assume that `A` and `B` are normal (inter-)actions, `I` is an interrupt Action, and φ is a guard.

```
A; B
||
φ: I
```

This protocol allows `I` to interrupt any executing `A` or `B` as long as φ remains true. `A`, `B` or `I` may mask the interrupt Action by setting φ to false.

6 Constraints

A *constraint* is a numeric or geometric relationship between objects [4, 13]. Constraints are described in terms of *visible* or exported aspects or value of the objects in question. Relations can be described easily in a declarative fashion when constraints are being used. Constraints are comprised of two aspects. One aspect is the *declarative* aspect: the definition of the constraint. The second aspect is the *procedural* aspect, namely the actions taken when the constraint has to be applied.

To incorporate constraints in PROCOL the language was extended with a simple construct: one-way constraints. We wanted an approach that did not require any modification in the code of the objects that are put under constraint. Therefore, the constraints are described outside of these objects. Also, we wanted to adhere to the encapsulation principle. Constraints are not defined in terms of status variables (which are hidden), but they are described by naming Action names. The original message mentioned in the constraint is sent without modification to the target. But a copy is sent to the constraining object. It can be manipulated by the object declaring the constraint and propagated to other objects that share relationships with the target of the message.

6.1 Propagator Constraints

The approach can be applied to any object-oriented language. The syntax of a propagator constraint is as follows:

`constraint` *TargetObject.ActionName msg* → *body*

The semantics is that every time an object sends the message *msg* to *ActionName* in object *TargetObject*, the code section defined in *body* is executed. The statement has a declarative function. It is a declaration of a piece of code that is to be executed at a later stage. In fact, it functions as a kind of inline procedure declaration. The code section describes the body of the procedure, the constraint trigger describes when this piece of code is to be executed. The propagator constraint can be placed at any place where a regular send statement can be placed. That is, in the Init and Cleanup sections, or at any place in an Action. Constraints can be canceled by specifying an empty body.

An example of a propagator constraint is:

```
constraint Line1.Draw( ) → {Line2.Draw( );}
```

This propagator constraint defines that whenever any object sends the message `Draw` to object `Line1`, then as a result the piece of code between the curly brackets is executed. Thus, the message `Draw` will be sent to object `Line2`. The constraint invocation is dynamic, and a form of second-order constraints could be defined, by placing the constraint statement inside an if-statement.

The propagator constraint

```
constraint Point1.Move (x, y) → {Point2.Move (x+5, y);}
```

assures that (graphical) object `Point2` is 5 units to the right of `Point1`. When `Point1` is moved to position (x, y), `Point2` is moved to the position `(x+ 5, y)`. The following example shows how propagator constraints can be used to animate algorithms. For instance, a sorting algorithm that has no graphical output or feedback can be made visible by applying constraints on its datastructure. Assume that the algorithm uses an `INTARRAY` object that provides an Action called `SetValue(index, value)`. The declaration of constraints allows an object to intercept all messages that are sent to this integer array, and propagate these messages to a visualizer (see Fig. 1).

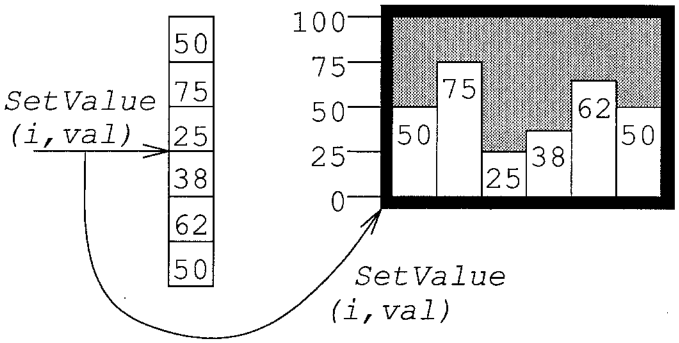

Fig. 1. Constraints used to animate an algorithm

For instance, the integer array could be coupled to a bar graph object, simple by defining one propagator constraint in the following object:

```
Obj      DEMO (INTARRAY array)
Declare  BARGRAPH bargraph;
         int i, val; /* index, value */
Init     {new bargraph;
          constraint array.SetValue (i, val) → {
             bargraph.SetValue (i, val);}
         }
EndObj   DEMO.
```

Strictly speaking our approach is more a propagation approach then a constraint definition approach. In fact, the constraints defined above extend the method of the object that has been put under constraint. Our approach uses simple, one-way constraints. Of course, the propagation techniques described here can be used to implement a more general constraint solving system. But then the constraint solving system becomes much more complex (see [13]).

A very interesting feature is that the contents of the original message can be changed and manipulated before being propagated to other objects. Propagators can be used to maintain relationships between objects, or to visualize nonvisual algorithms, or to simply monitor all accesses to a particular object. This can be useful for debugging environments or for profiling tools.

7 PROCOL examples

We now present two complete examples. They intend to illustrate the style and the flavor of PROCOL objects and the protocol governing the communication between the objects.

7.1 Mastermind

This example was derived from [22]. There is no intrinsic parallelism in this example. It serves to show the cooperation of 3 objects in solving a problem, as well as to demonstrate the use of protocols in ordering and allowing interactions. Communication is based on the *send* primitive only.

The familiar game is modeled here as a parent object, MASTERMIND, which creates two children, instances of PLAYER and OPPONENT. Object OPPONENT is created here with the attribute 20 indicating the maximum number of guesses PLAYER is allowed to make. MASTERMIND does not interfere with the communication between its two siblings, but waits until PLAYER and OPPONENT send a completion signal. When that occurs both children are deleted by the parent object. The function of the protocol of MASTERMIND is to restrict access to particular clients.

Obj MASTERMIND

Description Mastermind is played by a Player and an Opponent 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. The game continues until guess equals code (bulls == 4) or until the maximum number of guesses (maxguess) is exceeded.

```
Protocol    player → EndPlayer + opponent (result) → EndOpp
Declare     int result; PLAYER player;
            OPPONENT opponent;
            Note(){...}
Int         {new player;              /*create Player*/
             new opponent (player, 20); /*20 turns*/
            player. Start (opponent);} /*start player*/
Actions     EndPlayer={del player;}
            EndOpp={Note (result); del opponent;}
EndObj      MASTERMIND.
```

The next object, PLAYER, demonstrates the usefulness of protocols for client restrictions as well as for ordering and repetition of Actions in the object. When the Start Action is triggered it sends a random guess 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 (the number of turns allowed) has been exceeded. If so OPPONENT returns the score to Action Stop in PLAYER, and the repetition terminates. If bulls == 4 PLAYER celebrates by calling a local procedure Beep. Finally it sends a (completion) message to its creator MASTERMIND.

```
Obj       PLAYER
Protocol  Creator(opponent) → Start;
          opponent(bulls, cows) → Makeguess *;
          opponent(bulls) → Stop
Declare   OPPONENT opponent;
          int i, bulls, cows, guess[4];
          EducatedGuess(){...}
          Beep(){...}
Actions   Start = {for (i = 0; i < 4; i++) guess[i] = Random(1, 7);
                   opponent.Eval(guess);}
          Makeguess = {EducatedGuess(); opponent.Eval(guess);}
          Stop  = {if (bulls == 4) Beep(); Creator.EndPlayer();}
EndObj    PLAYER;
```

When OPPONENT is created the identity of its partner is recorded in object attribute player. Its protocol is in fact a repetition controlled by guard notend. It waits for a guess from player, evaluates it in Action Eval, where it also checks the number of turns taken and the new value of notend. If notend is true it returns the evaluation to PLAYER. As soon as notend is false OPPONENT will send the score to PLAYER, and a success signal (bulls is equal to 4) or a fail signal (bulls less than 4) to its creator MASTERMIND. If notend is true, the protocol expression may be repeated; otherwise the guarded protocol nicely blocks further (illegal) interactions with PLAYER. This blocking is permanent until object OPPONENT is deleted.

```
Obj       OPPONENT (PLAYER player; int maxguess)
Protocol  notend: player(guess) → Eval
Declare   int count, i, bulls, cows, code[4], guess[4];
          int notend; PLAYER player;
          Determinescore(){...}
Init      {for(i=0; i<4; i++) code[i]=Random(1, 7);
           count=1; notend=true;}
Actions   Eval={Determinescore(); count++;
                notend=(count≤maxguess&&bulls<4);
                if(notend)
                   player.Makeguess(bulls, cows)
                else{
                   player.Stop(bulls);
                   Creator.EndOpp(bulls==4);
                }
               }
EndObj    OPPONENT.
```

The specification of object variables in the protocol of the three objects demonstrates the use of the protocol as a composition rule over cooperating objects.

7.2 Ringbuffer

A more practical, yet still simple example is the object RINGBUFFER. It has a protocol with guards opening or closing access to Actions that insert (Put) a symbol sym in a cyclic buffer, or fetch (Get) a symbol from the same buffer:

```
Obj       RINGBUFFER (size:INT)
Protocol  count<size:ANY(sym) →Put + count> =0:ANY→Get
Declare   char sym, buf[size];
          int count, in_index, out_index;
Init      {count=in_index=out_index=0;}
Actions   Put={buf[in_index]=sym;
               in_index=(in_index+1)% size;
               count++;
              }
          Get={Sender.(buf[out_index]);
               out_index=(out_index+1)% size;
               count--;
              }
EndObj    RINGBUFFER;
```

in which `size` is the size of the buffer, and `count` is the number of buffer slots occupied. These two variables are set by Actions `Put` and `Get`. The upshot of the protocol is that no client is allowed to insert a symbol when the buffer is full, and no fetching is possible until at least one symbol is present.

7.3 Newton-Raphson pipeline

This is an example of pipelined concurrency and delegation. The pipeline consists of one object of type `SQROOT` and any number of pipeline elements `NRSTEP`. Together they compute an increasingly refined value for the square root of the original argument passed to `SQROOT`. A client communicates with object `SQROOT` by means of a *request* of the following form:

```
SQROOT.Compute (x) → (result)
```

in which `x` is the argument and `result` is the approximation to $\sqrt{x}$. `SQROOT` computes the square root by a series of approximations according to the Newton-Raphson method. It creates an object `NRSTEP` to which it later *delegates* the computation of the next estimate. This object creates a next `NRSTEP` for a computation of a new estimate by delegation. This creation and approximation process goes on until the present estimate differs less than `eps` from the previous one. This estimate is then returned *directly* (delegation!) to the client. As soon as Action `Compute` of `SQROOT` has delegated its message to the first instance of `NRSTEP`, it is ready to handle another square root request. Once the pipeline is filled, *n* computations are in progress simultaneously.

```
Obj         SQROOT
Protocol    ANY(x) → Compute
Declare     float x; NRSTEP Child;
Init        {new Child;}
Actions     Compute = {@Child.Compute(x, 0.5*x);}
EndObj      SQROOT;

Obj         NRSTEP
Protocol    Creator (x, Est) → Compute
Declare     float x, Est, NewEst, eps; int endpipe;
            NRSTEP Child;
Init        {eps = 0.0001; endpipe = true;}
Actions     Compute = {NewEst = 0.5 * (Est + x/Est);
                       if (abs(1-NewEst/Est) < eps)
                          Sender. (NewEst);
                       else{
                          if (endpipe) {
                            new Child; endpipe = false;}
                          @Child.Compute (x, NewEst);
```

```
                    }
EndObj      NRSTEP;
```

Note that objects SQROOT and NRSTEP are similar in structure. SQROOT functions as a kind of front end to the pipeline. It primes the pipeline by setting an initial estimate for the value of the square root. It then passes the argument and the estimate to the first NRSTEP object it has created.

NRSTEP receives these values from its creator, computes a new estimate, and determines if this value is within a value eps from the old estimate. If so the root is returned to the client via the *send* statement Sender. (NewEst). Because of the delegation statement in SQROOT and in the NRSTEP instantiations, the Sender primitive always holds the identity of the original client (the caller of SQROOT). Otherwise, if this NRSTEP instance is the end of the pipeline, indicated by boolean endpipe, NRSTEP creates a new instance of its own object type, and assigns it to the variable Child; subsequently it passes this child the original square root argument and a new estimate.

In a typical square root approximation this results in a string of NRSTEP objects, passing each other increasingly better approximations of the root, and terminating by passing the final approximation to the client.

Note that the protocols are in this example not very interesting, except perhaps in NRSTEP, where it indicates it only accepts its creator as accessor. Finally, note that NRSTEP objects remains in existence permanently once created. However, the length of the pipeline may increase dynamically, depending on the number of iterations required for any subsequent square root calculations.

7.4 *Polymorphic drawing*

Assume that the following three graphic objects have been defined: TRIANGLE, RECTANGLE, and POLYGON. All objects contain an Action Draw. Then the following object skeleton makes it possible to draw an arbitrary graphics picture by collecting object identities in a vector:

```
Obj      GENDRAW
Declare  ANY Geom[3]; TRIANGLE triangle;
         RECTANGLE box; POLYGON polygon;
Init     {...
          new triangle (.../*coordinates*/);
          new box (.../*coordinates*/);
          new poly (.../*coordinates*/);
          /*do a polymorphic Draw for these graphic objects*/
          Geom[1]=triangle;
          Geom[2]=box;
          Geom[3]=poly;
          for (i=0; i < 3; i++) Geom[i].Draw ( );
EndObj   GENDRAW;
```

The `for` statement effecting the polymorphic `Draw` will cause three `Draw` Actions that, although started in sequence, will run in parallel. But this example allows for more parallelism. It is quite imaginable that all graphic draws are executed line by line by some other object, say `LINE`. If every graphics object above creates as many `LINE` objects as it has vertices, and then delegates the drawing to these `LINE` objects, all lines will be computed and drawn in parallel to boot! For example the object `TRIANGLE's` Action `Draw` could contain the following (assuming `vtt` [1...3] is the vertex array):

```
new line1 (vtt[1], vtt[2]);
new line2 (vtt[2], vtt[3]);
new line3 (vtt[3], vtt[1]);
line1.Draw( );  line2.Draw();  line3.Draw();
```

with `line1`, `line2`, and `line3` objects of type `LINE`.

8 Implementation

Why did we not choose to superimpose protocols on an available OOL? In the first place almost all existing OOLs [28, 15, 6, 20, 17] are sequential *rpc*-based languages. These languages treat communication and corresponding Actions as a kind of extended procedure call. In particular this means that messages cannot be passed to other objects for further processing such that the results are directly returned by objects other than the original receiver of the message. It also means that sender and receiver are bound during message reception, processing, and return. ABCL/1 [29] (*rpc* and $S_a R_s$, see Sect. 1) and Actors (communication type $S_a R_s$), with their asynchronous message passing, as well as SINA [21] (rpc^- and $S_a R_s$) and PROCOL (rpc^- and $S_s R_s$), are exceptions to this long-term binding, and thus foster increased parallelism. In a world in which parallel processes prevail, concurrent OOLs are a prerequisite.

When a PROCOL program is being compiled, the PROCOL parser translates the parts different from *C* into *C*. A PROCOL object definition is thereby translated into a structure containing the state variables, and a set of *C* routines corresponding to the Actions in the object. When a message is being sent to an object, the structure belonging to the instance is available to the appropriate routine. As each message is subjected to the protocol of the receiving object, at some stage a test is needed whether a given message is valid. Therefore each object contains a protocol parser that will subject every message that is sent to this object to a run-time validation. The current protocol state of an object instance is saved in the same structure where the state variables of an object instance are kept. The code that performs the protocol parsing is the same for each object instance of a given type.

If a message is not wanted at a certain time, it gets delayed, and may finally time out. In this case the sender of the message can inspect the return value of the send primitive.

The extended regular expression in a protocol is translated into a finite state machine (see Fig. 2). When a message is being sent to an object instance, the protocol parser uses the current protocol state of the object instance to determine whether the message is legal or not. After accepting the message the protocol parser will update the protocol state of the object instance.

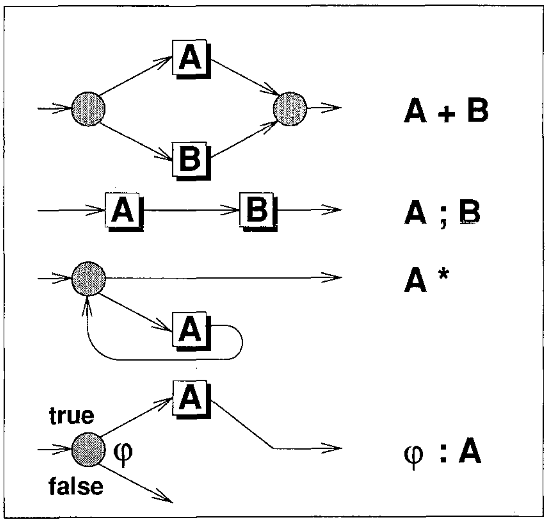

Fig. 2. Correspondence of finite state machine and protocol

In the process of compilation of a PROCOL program, a send statement will eventually be translated into a *C* function call. This function returns a value, indicating success of failure of the send statement. For example the send statement `box.Draw(...)` is translated into something like a call to the function `BOX_Draw(...)`, where `BOX` is the type of object `box`. At this stage we can discriminate between dynamic versus static binding. If the type of the target object and the Action to be executed are known at compile time, we can optimize the binding code. The send statement can then directly be replaced by the *C* function implementing the required Action. This method of binding is generally referred to as static binding. It is a method which has also been applied to optimize Smalltalk implementations.

A send statement which allows for less optimization than in the case of a $1-1$ mapping is the case when the target of the send statement is a *type* rather than an *instance* of a certain type ($1-n$ mapping). In this case the sender of the object is not interested which object instance is handling its request. The sender only indicates the type and the message. In this case the send statement is translated into a *C* function that tries all instances of the given type until one is willing to accept the message. If that happens the *C* function will return success. If no object instance of the given type wants to execute the specified action for the sender in question, the send request is suspended to be possibly satisfied later, or might finally time-out. Then failure is returned and the send statement indicates that the send has failed.

The third category of send statements involves those send statements that do indicate an object instance, but specify an empty Action name. The receiving object is in this case free to (dynamically) bind the message to any action it finds suitable. So, in this case the receiving object will have to inspect its protocol definition (and state) to determine the Action that will eventually handle the message. This kind of coupling of a send statement to an eventual Action is generally referred to as dynamic binding.

When a sender sends to a *type* specifying an empty Action name, all instances of that type will have to be tried, until success. In this case every try will involve dynamic binding as shown before.

When a message is sent to an object of which the type is unknown at compile time, its type needs to be determined at run time. After determining the type of the target object, the message can be dynamically bound to the appropriate Action.

The types of the individual items that make up the contents of a message are not checked against the parameter types of the receiving object, neither at compile-time, nor at run-time. In the case of dynamic binding, type-checking at compile-time is useless, as the eventual receiving Action cannot be determined. In the case of static binding, type-checking at compile-time could be performed. In both cases, type-checking could occur at run-time; that is, when the message is received and the contents of the message is copied to instance variables local to the receiving object. Type-checking is not implemented in the current version of the compiler. The programmer has to assure type compatibility.

A version of a PROCOL compiler has been implemented that translates every send statement into a *C* function call. As a consequence the semantics of a send is the same as for a request. Also, the resulting program no longer has any parallelism. All objects end up in one single process, and the process has only one thread.

A parallel extension of the implementation is under construction. In this implementation a PROCOL program will consist of a number of communicating processes, one for each processor in the network. Object instances are spread across the network based on some strategy. This strategy could involve load balancing or cluster preferences specified in startup files. When an object is sending a message to another object that resides within the same process, the message is handled locally. If the target object is outside the process of the sender, the message is sent over the network to the target process. A number of objects will reside in the same UNIX process. Light weight processes will be used to schedule objects. Each object will have its own thread.

At regular intervals, a process checks or is signalled whether there are messages from outside waiting to be handled. This (coarse grain) implementation strategy is much like the strategies adopted in Concurrent *C* [9], and Pool [2]. Inside each process, parallelism is simulated, but all processes do run in parallel. In general, when the message has been delivered, the sending process resumes execution directly and does not have to wait for the result.

Preliminary performance figures show that the sequential implementation has a throughput of 50000 messages per second, while the distributed version has a throughput of approximately 1000 messages per second when sender and receiver are not on the same processor.

9 Conclusion

PROCOL is a concurrent OOL with message based communication, synchronous only during message transfer or *request*. Apart from this concurrency, PROCOL's main contributions are its protocol, distributed delegation, and constraint facilities. The protocol is an explicit way of regulated access control to Actions inside the object, with specification facilities for sequencing, selecting, and repeating Actions that affect internal data structures. It also provides the means for client type checking. In addition it serves to specify relations between objects. Protocols further safe communication sequences, and are a step in the direction of verifiability. If all object protocols were available at compile-time, a first consistency check could be performed. Exhaustive verification is impossible because of the variable elements (types, guards) in a protocol, but above all because in a dynamic, incremental system not all object protocols may be accessible. Consequently, PROCOL's protocols are checked at run-time.

At first sight protocols take away some freedom in programming. On the other hand OOLs often encourage a programming style where ordering of and control over communications are hidden deeply in the code. But whereas the object code is a matter internal to the object itself, communication plays a role of global importance. It determines the smooth progression of execution of other objects. Hence there exist good reasons to better separate and control communication as an external, inter-object activity, from the internal activities of the object. This is especially important in the context of the global, and thus often disastrous effects that communication errors may cause.

Protocols as explicit, specification-type constructs in OOLs (sequential or parallel) are a novelty. An approximation of a protocol exists in the non OOL Ada [12]. It takes the form of an (explicit) selection protocol when accepting calls from other subtasks; in addition the selection can be controlled by guards. The ADA approach has been copied in Pool [2]. Pool does, but ADA does not have separately specified Actions. In ADA Actions occur in the middle of executable code; thus sequencing and repetition are (implicitly) derived from regular ADA language facilities.

A rudimentary protocol can also be found in ABCL/1 [29]. This OOL is based on the Smalltalk way of defining Actions, but it allows for hierarchical sets of Actions, such that the object could wait for a new message when in the middle of some Action triggered by an earlier message. In the terminology of PROCOL, this means a global selection (+ operator) protocol, in which the terms consist of sequences.

9.1 Present and future research

9.1.1 Global protocols. Protocols already offer substantial support for constraining communication. However, one could envisage further facilities for restriction and protection. An example is a group of cooperating objects. To prevent other objects from access to this group one might like to specify a kind of *inter-object protocol.* This (global) protocol would in fact specify various relationships and restrictions between the objects (rather than the Actions) in a group of logically connected objects. Such a protocol would be one level higher than the object protocols presented in this paper. The idea of inter-object protocols could be further generalized to inter-group protocols, and so on.

9.1.2 Persistent objects. Another desirable facility is persistent objects, persistence being defined as the ability of an object to outlive the execution time of a program. Persistent objects are of great use when objects accumulate large amounts of data, such as in database or CAD systems. They can be used in one or more other programs with their data still intact, obviating a possibly precious reloading of data. Persistent objects should be handled just as volatile objects. We have defined and implemented three new primitives [16] on an experimental basis. The primitive `persistent` makes a volatile object persistent, the primitive `volatile` make a persistent object volatile, and the primitive `retrieve` retrieves a persistent object created at an earlier stage. Refinements in the form of dataset keys make it possible to discriminate between datasets used for storage of persistent objects. We are also studying the use of *visible* (global) attributes to make it possible to search for a particular instance that satisfies a certain condition. The experimental implementation uses shared mapped memory (virtual files), so at the programming level there is no difference between persistent and volatile objects, which is an essential characteristic. The primitives take into account that persistent objects may create/use other persistent objects.

A. Appendix

In the extended syntax conventions used below square brackets enclose options, brackets ⟨ and ⟩ denote zero or more occurrences of the affected term, and the vertical bar (|) indicates an alternative. Literals (terminals) are printed bold. The alphabet of terminals consists of the following symbols:

=   .   →   @   (   )   {   } || ;   +   *   :
skip   new   del   constraint   Sender   Creator   Receiver

A.1 Actions syntax

Actions	::= ⟨Action⟩
Action	::= A-Name = body
body	::= {⟨[code] [communicate] [constraint]⟩}
communicate	::= send |request| delegation
new	::= **new** variable [msg]
del	::= **del** variable;
constraint	::= **constraint** send → body
send	::= Obj-Name. [A-Name] [msg];
request	::= Obj-Name. [A-Name] [msg] → mes
delegation	::= @ Obj-Name. [A-Name] [msg];
A-Name	::= identifier
Obj-Name	::= variable | literal | object-type
variable	::= identifier
literal	::= **Sender |Receiver| Creator**
object-type	::= identifier
msg	::= (expression ⟨, expression⟩)
mes	::= (identifier ⟨, identifier⟩)

The non-terminal *code* is understood to have its intuitive meaning. Apart from traditional programming constructs, it may contain `new` and `del`. The syntax of the Init and Cleanup section is:

```
Init    ::= body
Cleanup ::= body
```

A.2 Protocol syntax

The complete syntax for protocol specification is:

```
compound-protocol   ::= protocol ⟨||protocol⟩
protocol            ::= interaction-expr
interaction-expr    ::= interaction-term ⟨ + interaction-term⟩
interaction-term    ::= interaction-factor ⟨ ; interaction-factor⟩
interaction-factor  ::= interaction-primary | interaction-primary *
interaction-primary ::= [guard :] interaction | (interaction-expr) | interaction
interaction         ::= Obj-Name [mes] [ → A-Name] | skip
```

For the definition of the non-terminals *Obj-Name*, *mes* and *A-Name*, see above.

A.3 Protocol semantics

Part of the semantics of the expressions occurring in protocols has been explained, sometimes informally, above. We now present a more formal treatment of the semantics for the special cases such as guards, *skip*, and blocked expressions.

As syntactic sugar in guarded expressions a special guard *else* is provided. In a selection expression it leaves an alternative communication channel when all other guards evaluate to false. The guard *else* is syntactically equivalent to the complement of the logical or ($\vee$) of all guards on the simultaneously active terms in the particular selection with the *else*. E.g. given

$$\varphi: \quad A + \psi : B + else : C$$

the following identity holds:

$$else \equiv \overline{\varphi \vee \psi}$$

From the definition of the operators; + and *, various laws for associativity, distributivity and commutativity follow intuitively. Let A, B, and C stand for arbitrary interaction expressions, and let φ and ψ stand for two arbitrary guards.

Then arbitrary interactions, *skip* and blocked interactions Δ ($\forall A$: $\Delta \equiv false:A$) are subject to the following axioms:

associativity

1. $A;B;C \quad = A;(B;C) \quad = (A;B);C$
2. $A+B+C = A+(B+C) = (A+B)+C$
3. $\varphi + \psi + A = \varphi + (\psi + A)$

distributivity

4. $A;(B+C) = A;B + A;C$
5. $\varphi:(A+B) = \varphi:(A) + \varphi:(B)$

commutativity

6. $A+B \quad = B+A$
7. $\varphi + \psi + A = \psi + \varphi + A$

other

8. $A+A \quad = A$
9. $\Delta;A \quad = \Delta$
10. $\Delta + A \quad = A$
11. $skip;A \quad = A$
12. $skip* \quad = skip$
13. $\Delta* \quad = skip$
14. $\varphi:\Delta \quad = \Delta$

The expression $skip + A$ is to be interpreted as an optional occurrence of A (zero or one A).

More than one true guard could introduce (run-time) ambiguity (non-determinism). Sometimes it is possible to check ambiguity syntactically, but in general it requires semantic checking. For example the following expression would be ambiguous for φ and ψ both true (the parentheses only serve as textual delimiters):

$$(\varphi:A;B) + (\psi:A;C)$$

This situation is handled by allowing A and subsequently waiting for B or C. Temporary ambiguity can be accepted, but eventually an interaction term has to occur such that a unique alternative can be chosen from the selection expression. Without guards ambiguous expressions may also be defined. If the identity of the sending objects is known at compile time they could be handled by factoring out the ambiguous terms. But some other situations preclude a general solution.

References

1. Agha, G., Hewitt, C.: Concurrent programming using Actors. In: Yonezawa, A., Tokoro, M. (eds.) Object-oriented concurrent programming, pp. 37–53. Cambridge: MIT Press 1987
2. America, P.: POOL-T: A parallel object-oriented language. In: Yonezawa, A., Tokoro, M. (eds.) Object-oriented concurrent programming, pp. 199–220. Cambridge: MIT Press 1987

3. Bal, H.E., Tanenbaum, A.S.: Distributed programming with shared data. Proceedings IEEE Conference on Computer Languages, pp. 82–91. Washington: IEEE 1988
4. Borning, A., Duisberg, R.: Constraint-based tools for user interface. ACM Trans. Graphics **5**(4), 345–374 (1986)
5. Campbell, R.H., Habermann, A.N.: The specification of process synchronization by path expressions (Lect. Notes Comput. Sci., Vol. 16, pp. 89–102). Berlin Heidelberg New York: Springer 1974
6. Cox, B.J.: Object-oriented programming. An evolutionary approach. Reading: Addison-Wesley 1987
7. Dahl, O.-J., Myhrhaug, B., Nygaard, K.: Simula 67 common base language (NCC Publications S-52). Oslo: Norwegian Computing Center 1967
8. Feldman, J.: High-level programming for distributed computing. Commun. ACM **22**(6), 353–359 (1979)
9. Gehani, N.: The concurrent C programming language. Reading, Mass.: Addison-Wesley 1989
10. Hewitt, C.: Laws for communicating parallel processes. In: Gilchrist, B. (eds.) IFIP Information Processing, Vol. 77, pp. 987–992. Amsterdam: North-Holland 1977
11. Hoare, C.A.R.: Communicating sequential processing. Commun. ACM **21**(8), 666–677 (1978)
12. Ichbiah, J. et.al.: Rationale for the design of the ADA programming language. Sigplan Notices (ACM) **14**(6) (1979), part B (1980)
13. Leler, W.: Constraint programming languages. Their specification and generation. Reading Mass: Addison-Wesley 1988
14. Liskov, B., Snyder, A., Atkinson, R., Schaffert, C.: Abstraction mechanisms in CLU. Commun. ACM **20**(8), 564–575 (1977)
15. Meyer, B.: Object-oriented software construction. Englewood Cliffs: Prentice Hall 1988
16. Oosterom, P. van, Laffra, C.: Persistent graphical objects in PROCOL. In: Bézivin, J., Meyer, B., Nerson, J.M. (eds). TOOLS 2, the TOOLS'90 Proceedings, pp. 271–283, 1990
17. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., Wilpolt, C.: An introduction to Trellis/Owl. ACM Conference Proceedings OOPSLA'86, Portland, Special Issue. SigPlan Notices **23**(11), 9–16 (1986)
18. Shaw, M., Wulf, W.A., London, R.L.: Abstraction and verification in Alphard: defining and specifying iteration and generators. Commun. ACM **20**(8), 553–564 (1977)
19. Stein, L.A.: Delegation is Inheritance. ACM Conference Proceedings OOPSLA'87, Orlando, Special Issue. SigPlan Notices **22**(12), 138–146 (1987)
20. Stroustrup, B.: The C^{++} programming language. Reading, Mass.: Addison-Wesley 1986
21. Tripathi, A., Berge, E., Aksit, M.: An implementation of the object-oriented concurrent programming language SINA. Software – Pract. Exp. **19**(3), 235–256 (1989)
22. Bos, J. van den, Plasmeijer, M.J, Stroet, J.W.M.: Process communication based on input specifications. ACM-TOPLAS (Trans. Programm. Languages Syst.) **3**(3), 224–250 (1981)
23. Bos, J. van den: ABSTRACT INTERACTION TOOLS: A language for user interface management systems. ACM-TOPLAS (Trans. Programm. Languages Syst.) **10**(2), 215–247 (1988)
24. Bos, J. van den: PCOL – A protocol-constrained object language. SIGPLAN Notices **22**(9), 14–19 (1987)
25. Bos, J. van den: PROCOL – A protocol-constrained concurrent object-oriented language. SIGPLAN Notices, (Special Issue) **24**(4), 149–151 (1989)
26. Bos, J. van den, Laffra, C.: PROCOL – A parallel object language with protocols. ACM Conference Proceedings OOPSLA'89, New Orleans (Special Issue) SigPlan Notices **23**(11), 95–102 (1989)
27. Bos, J. van den: PROCOL – A protocol-constrained concurrent object-oriented language. Inform. Process. Lett. **32**, 221–227 (1989)
28. Xerox Learning Research Group: The Smalltalk-80 system. BYTE **6**(8), 36–48 (1981)
29. Yonezawa, A., Briot, J.-P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. ACM Conference Proceedings OOPSLA'86, Portland (Special Issue) SigPlan Notices **21**(10), 258–268 (1989)