PARAMETER PASSING IN CLIP

I promised at the Vienna meeting to try to produce a paper which reduced the concept of parameter passing to a limited number of distinct concepts. This is a first attempt.

The aspects that seem to need addressing are the nature of the formal parameters, the nature of the corresponding actual parameters, and the method of association between them. Each of these will be addressed below, but it seems to me that as far as parameter passing is concerned it can all be expressed in terms of passing values: everything else is in CLIP, and the familiar varieties of parameter calling, as listed in Ken Edward' "boxes" paper (N2--), can all be mapped into this. The key question for CLIP is when (before, during or at the end of call), by what (caller or server), and how often a parameter is used (evaluated for IN, reset for OUT). The key, I believe, to providing a simple model is the second of these three - whether it is the caller program processor or the server procedure processor which takes action. The "virtual contract" concept is capable of expressing the "who does what" aspects of parameter passing; the rest fall in place around that.

1. Simplification

First let us dispose of some side issues by means of some simplifications.

1.1 Function v. subroutine calls

Most languages, with differing terminology, allow both "function" procedures and "subroutine" procedures. Function procedures return a value of some datatype and in general can appear as part of an expression (i.e. operations can be performed on them as in

a + f (x,y) + b

Subroutine procedures, however, do not return a value and are called "stand-alone", either by invoking it by name, or by using an explicit invokation command (e.g. CALL in Fortran, PERFORM in Cobol).

The simplification in CLIP is to map the value returned into an additional (OUT) parameter, for "passing" purposes, and the language binding can decouple this implicit parameter from its explicit fellows when the call is completed. Once the NULL problem is resolved, the "subroutine" kind of procedure could be brought into the same model, if that is wished, though there seems little point in it. Perhaps it is more elegant in the abstract sense, but the creation of an extra parameter which is known in advance not to be capable of returning anything seem pointless. Specifically, I do not see the server processor needs to "know" the nature of the call that invoked it.

2.2 Global variables

Similarly, where a language with a nested block structure allows access to "global variables" and other entities defined in a surrounding block - or to a shared data area as with COMMON in Fortran or bricks in RTL/2 - then as far as CLIP is concerned these must be regarded as additional, implicit parameters to be passed. The CLIP standard may advise or even strongly recommend that CLIP-based services should not use such mechanisms, but the CLIP model must be general enough to cover it.

2.3 IN/OUT parameters

IN/OUT parameters can similarly be disposed of as parameters with the properties of (including constraints of) both IN and OUT. At one point I considered that perhaps IN and OUT could also be dispensed with, but concluded that, even if they were formally redundant, they were useful concepts, simple and easy to visualise, and closely related to the roles of caller and server.

3. The nature of parameters

The nature of formal parameters is straightforward - the CLIP model must allow any CLIP datatype as a formal parameter. CLIP-based services (like RPC) may well need to limit the range allowed. Language bindings on the server side will need to take account of the kinds of formal parameter supported by the language. The CLIP model itself, however, must cover everything - and indeed some languages come quite close to supporting everything.

Since the proposal here is to base the parameter passing solely on the passing of values, and the model needs to be complete, then an actual parameter may be any expression yielding a value of the datatype required by the call. Two points need to be noted here. One is that it is a "strongly typed" model. It has to be, just as CLIP has to be: some languages need strong typing, and weak typing can be accommodated by relaxing association rules and adding implicit type conversions in the bindings. This is much easier than, and preferable to, making the CLIP model weakly typed and adding the strong typing constraints in the bindings.

The other point to note is the use of the phrase "a value of the datatype required by the call" rather than "a value of the datatype of the formal parameter". This will be explained later, but it is necessary to ensure that false assumptions are avoided. Roughly, the difference is similar to the difference between the xs in x:=y and z:=x. The assignment operation is not symmetrical, and nor are the caller and server roles of the IN and OUT parameters in the CLIP model.

In particular in this respect, the proposal here is based on the concept that the destination for the value of and OUT parameter can be determined by evaluation of an expression - it does not have to be the destination itself, directly expressed, any more than the value to be assigned to an integer variable has to be an integer literal rather than an expression yielding an integer. (Some languages do support this concept.)

4. Kinds of parameter passing

The proposal here is that only four kinds of parameter passing need to be specified. Exactly what the consequences are depends on the kind of datatype of the corresponding formal parameter, and the particular form of parameter passing specified in the language. In the latter case, this is dealt with in the language binding and is discussed in section 5. In the former case, distinction needs to be made, at the implementation level and to some extent the conceptual level, between the four main classes of datatypes - primitive, aggregate, pointer, and procedure.

4.1 Call by value sent on initiation

In this, simplest, case the formal parameter of the server procedure requires a value of the datatype concerned. The virtual contract is that the caller evaluates the actual parameter and supplies the resulting value at the time of transfer of "control"; the server accepts it; and no further interaction takes place - the transaction is complete. (The situation where the provided value is unacceptable, e.g. out of range, is covered by exception handling.) To anticipate slightly, in the case at least of primitive datatypes, this is the formula "call by value" supported by many languages, though as we shall see it can have other uses.

4.2 Call by value sent on request

The virtual contract here is that the caller undertakes to evaluate the actual parameter and supply the resulting value, but only on receipt of a request to do so from the server procedure. The evaluation and passing of the actual parameter does not take place until and unless the server procedure requests it.

The essential difference from value-sent-on-initiation is that in some cases (not all) the value sent will be different. Date-and-Time is an obvious example but this can arise from side-effects of the server procedure using other parameters (e.g. printers). In a sense this could be regarded as call of an implicit procedure parameter - the procedure does the evaluation - except that is occurs only once. Any further reference in the server procedure to the formal parameter simply uses the value supplied - it does not issue a further request. Since CLIP deals essentially with static aspects of datatypes, it seems sensible to express this property in the essentially dynamic context of parameter passing mechanisms in CLIP.

4.3 Call by value returned as specified

This is the OUT equivalent to value-sent-on-initiation. The virtual contract is that the server will supply a value of the datatype of the formal parameter and the caller will accept it and act appropriately - a term deliberately left vague.

For a conventional OUT parameter it means that the destination is evaluated by the caller at the initiation of the call, and then sends the returned value to that destination at the conclusion of the call. In the case where the expression is simply the name of a variable, the timing of the trivial "evaluation" is of no significance, but is necessary to specify to accommodate the more general case, and make it clear that this is the usual "evaluation of actual parameters on initiation". Similarly it is necessary to specify that is the caller and not the server which sends the returned value to the destination, in order to accommodate situations, as in RPC, where caller and server are decoupled. In a closely coupled environment where providing the actual destination (such as a hardware address) to the server is a trivial task, there is no reason why the actual service contract at the implementation level should not include providing the actual destination to the server, which then sends its returned value directly there. That is an additional, service level function that the server contracts to perform for the caller, which does not affect the logical division of responsibility at the virtual contract (CLIP) level.

This kind of parameter passing also accommodates the return of a value for the procedure as a whole, in the case of function procedures. In section 1.1 it was noted that this could be regarded as an additional, anonymous parameter. Parameter passing by return-as-specified accommodates this, and directly reflects the conventional situation in "a + f(x,y) + b" and so on, where it is the caller that receives the returned value and "takes the appropriate action" - the reason for the vague phrasing in the first paragraph of this section.

4.4 Call by value returned when available

This is primarily here for completeness, to cover situations where the caller does not evaluate the destination for an OUT parameter on initiating the call, but delays it until the returned value is actually received. Hence the server can determine the timing of the evaluation to be any time after the returned value is available. It could be returned while the call is still in progress at the termination of the call (probably the most likely case), or even, in principle, at some later time. What time is chosen is determined by the binding of the CUP-based service and is not a matter for CUP itself; all the CUP model needs to do is to be able to accommodate the possibility. The virtual contract, then, is that the caller will receive the returned value when the server sends it, and then evaluae the destination and send the value there.

5. Conventional kinds of parameter passing

This section and the next briefly identify the main existing methods of parameter passing. Those most frequently encounered are covered in this section, others referred to in N2-- are covered in section 6. It is recognised that the distinction is to some extent arbitrary, and it is made only as an aid to readers: section 5 includes those parameter passing mechanisms which the author expects will be familiar to most people, while section 6 covers others that WG11 are aware of but which are believed to be less widespread. It is not normative! The important thing is for all significantly different methods to be covered, not any one person's perception of how "well known" each may be.

At this stage, it is sufficient to visualise each passed parameter as belonging to a primitive datatype. Parameters of aggregate, pointer and procedure datatypes are discussed in section 7.

5.1 Call by value (IN parameters)

This, the simplest of all passing mechanisms, appears directly in CLID as call by value sent on initiation (section 4.1). The virtual contract is fulfilled by the caller evaluating the actual parameter and sending the value to the server, and the server accepting it. No further action is required of the caller. The server does what it likes with the received value but can make no further demands on the caller with respect to the actual parameter that generated the value.

5.2 Call by value return (OUT parameter)

This also is supported directly in CLID by call or by value-returned-as- specified, which exactly describes the virtual contract. That involves "passing" only as undertaking to receive the value. If, in a specific language binding or CLIP-based service contract, a parameter is passed at language processor level, what is passed is an implicit pointer to a value of the datatype concerned, which the server contracts to set (but whose value, if any, prior to the call, the server contracts not to access).

In the common case where the actual parameter is the name of a variable the difference between call by value and call by value return is that between sending the value currently (on initiation of the call) held by the variable, as opposed to sending an access path to the variable itself.

Some languages, in their datatyping model, explicitly distinguish between the datatypes of values held by variables and those of the variables themselves: indeed, some have an explicit "dereference" (ie "obtain the value of" operator). For languages without such a model, CLIP allows that distinction to be made at the language binding service contract level without disturbing the virtual contract model.

5.3 Call by value send-and-return (copy-in-copy-out) (IN-OUT)

This is an IN/OUT passing mechanism where the actual parameter can be evaluated to a destination for call by value-return-on-initiation. (section 4.3). However, in the CLIP model it is regarded as a parameter with both that property and that of call by value-sent-on-initiation (section 4.1). Alternatively and equivalently, it can be expanded into two implicit parameters, one of each kind.

Hence the actual parameter corresponding to a formal parameter of a given datatype (T) must be capable, on evaluation, of yielding a destination for such a value, ie a pointer (explicit or implicit) to a value of datatype T. However, for the IN part of the IN/OUT the current value held in that destination on initiation of the call is retrieved (in the virtual contract) by the caller and relayed to the sender. The destination itself is also recorded (in general, if the result of evaluating an expression very often this will be trivial, eg when the actual parameter is the name of a variable of datatype T in the caller program). In the virtual contract the caller receives the returned value (the OUT part of the IN/OUT) from the server and sends it to that destination.

Where the language binding or service contract passes the destination itself to the server as part of copy-in-copy-out, the server must contract to retrieve the IN value immediately on transfer (making a copy for internal use while the procedure call is in progress) and then to send the returned (OUT) value to the detination on completion of the call. While the call is in progress the caller explicitly or implicitly marks the destination on completion of the call. While the call is in progress the caller explicitly or implicitly marks the destination as "read once only, write once only" and any attempt by the server to violate that condition is a breach of contract, ie. an exception.

5.4 Call by reference

In this case a formal parameter of datatype T is interpreted as an implicit "pointer to T" and the actual parameter must evaluate to such a pointer (explicit or implicit) accordingly. This pointer to T is then passed by value (its pointer, value, that is) as an IN parameter. Note, not as an IN/OUT, because that would imply an extra level of indirect addressing meaning that the "pointer to T" value, i.e. the destination, could be changed by the server, which is not the case with call-by-reference.

The virtual contract, then, is that the caller provides an access path to the destination (e.g. a variable of datatype T in the caller program). The destination is fixed, but the access path can be used by the server, both for reading and writing (IN/OUT) of values of datatype T, as many as it likes. In the close-coupled case the service contract may well involve passing the actual destination (eg. a machine address) with the caller needing to take no further action until the call is complete - hence the CLIP model of an IN parameter, call by value-sent-on-initiation of value of datatype pointer-to-T. In a loosely-coupled service environment the service contract will involve caller action during the call, responding to requests by the server (in effect, miniature reciprocal "calls", with the IN and OUT directions reversed) for a value of datatype T to be read or written.

Clearly, the reciprocal calls implied by call-by-reference in a loose-coupled environment represent a potential significant overhead, which may result in call-by-reference (and, by extension, pointer datatype parameters generally) not to be supported in such services.

6. Other kinds of parameter passing

(to be added)

7. Aggregate parameters

As far as formal parameters of aggregate datatypes are concerned, the virtual contract can be regarded as a passing, by the relevant mechanism, each of the individual elements of aggregate as a separate parameter of the appropriate datatype. In effect, the single parameter of the aggregate datatype is expanded, to a multiplicity of parameters. In contexts were parts of an aggregate may be marked as "mutable" or "immutable", the calling mechanisms will be different for the mutable and immutable elements.

Since the number of parameters for an expanded aggregate can be very large, which can represent a large overhead even in close-coupled environments, at the language binding or service level the expansion in practice will not occur and the service contract between caller and server will contain provisions to preserve the integrity of the call and provisions of the virtual contract. For example, a call by value-sent-on-initiation at the virtual contract level may well become call by value-sent-on-request at the service contract level, but with the caller and server contracting to "write-protect" (or denote "immutable") the actual aggregate parameter on initiation of the call.

8. Pointer Parameters

The case of pointer parameters can dealt with by extension from the discussion of intrinsic pointers and call-by-reference earlier. Pointers called by value-sent allow access (including write access) to the entity pointed to, but the pointer value itself cannot be changed so the pointer can refer to something else after the call. If the value sent is a pointer to a record, after the call the pointer still points to the same record, though the values in the fields of the record may have changed. If changing what the pointer refers to is needed, then another level of indirect referencing has to be invoked, either directly (as with modelling call-by-reference in CLIP) or indirectly by using call by value-returned.

In the (unlikely) case that there is any doubt, it is as well to spell out that an access path via pointer parameters, to whatever depth of indirect referencing, implies access to all lower levels, including the primitive datatype values referenced by the lowest level pointers.

9. Procedure parameters

All procedure parameters are effectively passed by call by value-sent, ie provision by the caller of an access path. Call by value-returned does not seem appropriate in CLIP, though perhaps it could be added of an extension of the model to coroutines were ever needed. The virtual contract is that the caller will evaluate the procedure whenever the server invokes it - i.e. will execute a reciprocal call. At the language binding or service level the service contract may allow the server direct access to the procedure concerned (ie that which is specified as the actual parameter), and this may even be automatic if the procedure is a library procedure. (Note that language bindings may in some cases have to deal with the practice of some implementations to provide in-line expansions - in effect macro implementations- of simple built-in library procedures) However, in some languages with a disjoint rather than a nested structure, existing services such as link-editors already provide much of the necessary contract framework.

10. Conclusions

This is very much a first draft, section 6 is left out at this stage, and there are doubtless errors and omissions. Nevertheless I hope it does fulfil the promise I made at the Vienna meeting to try to reduce the parameter passing model we discussed then to a relatively few basic concepts.

Brian Meek 8th October 1991