Communicating Process Architectures 2002
James Pascoe, Peter Welch, Roger Loader and Vaidy Sunderam (Eds.)
IOS Press, 2002

The "Honeysuckle" Programming Language: Event and Process

Ian EAST

Department of Computing, School of Technology
Oxford Brookes University, Wheatley Campus, Oxford, England OX33 1HX

ireast@brookes.ac.uk

Abstract. A new language for programming systems with *Communicating Process Architecture* [1] is introduced which builds upon the success of occam [2]. Some of the principal objectives are presented and justified. The means employed to express behaviour are then described, including a *transfer* primitive, which conveys object ownership as well as value [3], and an *alternation*[1] construct. The latter replaces PRI PAR and PRI ALT, and affords explicit expression of conflict-free prioritized reactive (event-driven) behaviour, including exception response [4]. HPL also offers source-code modularity, object encapsulation, and the recursive definition of both object and process. Despite such ambition, a primary aim has been to retain simplicity in abstraction, expression, and implementation.

1 Introduction

Over the last decade, object-orientation has come to dominate both the literature and the lore of programming [5]. The origin of object-orientated programming (OOP) can be traced back to the late 1960s, when Simula established the principles of *hierarchical data abstraction* and *encapsulation* [6-8]. However, these powerful ideas have become inextricably mixed up with their interpretation, and have been carried to an extreme. Abstraction is limited to passive objects.

Communicating Process Architecture (CPA) [1], on the other hand, implies hierarchical *process* abstraction. CPA is made manifest via dedicated theory (CSP) [9] and the occam programming language [2]. These combine to offer the possibility of unprecedented software integrity, making CPA far better suited to the demands of safety-critical, and consumer, applications. Unfortunately, exploitation of its formal basis currently requires additional tools and skills. As a result, it rarely achieved its full potential.

occam suffers from a few other problems which, together, help to explain its limited commercial success. Perhaps the most serious is the inefficiency that can arise out of the ability to communicate only a value and never a reference. Objects are immobile and must always be copied, rather than passed, between processes. occam also offers only very limited data abstraction, placing it in apparent conflict with the OOP juggernaut.

Finally, occam lacks many features now considered *de rigueur* for commercial use, such as source-code modularity and exception-handling. It also fails to deliver any decisive advantage in the rapidly expanding market for high integrity reactive (event-driven) systems. For example, the expression of prioritized *alternation*[1] [9, #5.4.3] is imprecise and obscure.

---

[1] Not to be confused with the *alternative* construct, which merely selects a process once.

Honeysuckle PL (HPL) has been designed with three aims. First, it exploits *design rules* which encapsulate formal analysis and obviate additional tools or skills. Any system programmed in HPL is automatically guaranteed never to deadlock. All necessary analysis is performed *a priori*. All that is required of the programmer is the definition of *service protocol* between all communicating processes. This is described in detail in a companion paper [10].

Second, HPL restores the balance between process and object abstraction. In doing so, it eliminates inefficiency in inter-process communication and enhances regularity. HPL emphasizes old wisdom [11]:

$$programs = processes + objects$$

Just as one passes either value or object to a Pascal procedure, in HPL one may pass either value or object to a concurrent process. HPL introduces regularity in expressing the transfer of ownership between processes, running in sequence or parallel. This is achieved without introducing explicit reference, thus rendering object identity and value distinct.

Explicit transfer lies in contrast with the implicit transfer which follows distinguishing mobile from immobile data upon declaration [12]. It is argued that implicit transfer is both irregular and less transparent. It also interferes inappropriately with object abstraction.

Third, HPL facilitates professional software development by including source-code modularity and exception-handling. It also introduces the means by which prioritized alternation may be directly programmed, without the possibility of conflict [4]. This novel construct provides the basis for programming reactive behaviour in general, and exception response in particular. (Exceptions are regarded as signals made by a *watchman* process to alert us to impending disaster, as would a neighbour seeing our house on fire.)

Inclusion of explicit alternation reflects three guiding principles:

- *transparency*;
- alternation is the essence of a reactive system, not concurrency;
- priority must be defined for any reactive system.

Transparency implies clarity. The meaning of any command should be immediately apparent to the reader. It should also imply ease of expression, without the need to introduce unwarranted operational complexity, which will obscure intent. Finally, it implies that the evolution of process state is clearly visible through the program text (procedure).

Unlike occam, HPL does not permit prioritization local to any single process. Instead, prioritization forms part of the protocol *between* communicating processes. In other words, it is an attribute of a relation, not of a process. This denies the possibility of conflict.

The principles above lie in addition to those which clearly guided the design of occam, as they did its precursors, Algol and Pascal – above all, *simplicity* in structure and abstraction, and *security*, as defined by Hoare [13].

Dijkstra's famous criticism of the GOTO command [14] perhaps best argues the case for both abstraction and transparency. However, Knuth later argued effectively that both abstraction and efficiency demand control structure beyond that offered by Algol, and thus justify the controlled use therein of GOTO. To meet (most of) the needs he described, HPL defines a single repetition construct allowing multiple points of exit, positioned at the start of, end of, or *within*, an embedded sequence. Like occam, HPL affords no direct reference to a point within either control or data memory.

Lastly, HPL aspires to gaining a worthy programming environment (PE). One of the many lessons from the occam experience was the potential for, and of, tight integration between compiler and PE. Folding was an innovation as important as any in occam itself. Note, however, that a PE should *not* extend to the maintenance of both 'debug' and 'release'

versions of a program, which Hoare once likened to wearing a life-jacket right up to the time one puts to sea [13]. Security should instead be sought in the vessel.

Space permits only a summary here of the elements of HPL which owe the most to occam and CPA. Implementation will be reported when enough experience has been gained. It is hoped that the reader will find none of the simplicity, security, or sheer elegance, of occam sacrificed "on the altar of ambition".

2 Block Structure

2.1 Sequential and Parallel Composition

A block in HPL is a process composition combined with its *context*, which describes how components communicate.

```
{                                   {
  ...  declare objects                ...  declare channels

  sequence                            parallel
    ...  do this                        ...  do this
    ...  do that                        ...  do that
}                                   }
```

Components of a sequential composition communicate asynchronously via objects. Each object acts as a buffer between a pair of processes, running in sequence. Components of a parallel composition communicate synchronously via channels.

Objects and channels may be declared *only* above sequential and parallel composition respectively. They are declared before use to aid transparency and readability, by rendering the context of any block explicit. Object *creation* is postponed until first assignment.

Many errors are denied by constraining the way in which each object and channel may be used. First, the type of message conveyed is defined, though it need not be unique. Variant protocol is sometimes useful but requires caution. Pascal supported variant objects, and occam variant channels. HPL allows both but subject to certain constraints.

Channels are not declared directly. Instead, a *service* is declared, which defines, not just a set of channels, but also *the order in which they are used*. Multiple services may be declared above each parallel composition. Furthermore, they may each be attributed a relative *priority*. The theory and consequences of service protocol are addressed in a companion paper [10].

Process replication is permitted, with automatic declaration and creation of cardinal indices.

```
{                                   {
  ...  declare services               ...  declare services

  parallel                            parallel
    for each i from m for n             for each i from m for n
      ... do this                         replicate
    for each j from p for q                 for each j from p for q
      ... do that                             ...  do this
}                                   }
```

Neither *m* nor *n* (expressions) need be computable upon translation.

2.2 Object Declaration, Lifespan, and Visibility

Explicit variable declaration, and the association of scope and construct ("block structure"), was introduced in 1960 with the revolutionary Algol-60 [15, 16]. Some time later, Dijkstra published a thorough discussion of how process context may, and should, be programmed [17]. He demanded that:

- the *entire* context of every block be rendered apparent;
- references are *not* automatically acquired[2] from the context of an outer block;
- proper (not arbitrary) initialization of every variable be verified *upon compilation*.

The advantages of meeting these demands are profoundly significant:

- the opportunity for variable misuse is reduced as far as possible;
- transparency is greatly enhanced;
- a procedure becomes merely a named block (thus naming a process).

These seem well worth exchanging for a little convenience. Note that it still remains possible to misuse a variable. One may omit reassignment prior to reuse. However, failure to properly initialize is by far the most common error. Block structure tends to reduce reuse.

Declaration, in HPL, does *not* indicate the start of an object's life. This would imply one of two consequences. First, the value of a variable (state of an object) would have to be undefined between declaration and first assignment. Exposing a variable without proper value is obviously dangerous. Status may be rendered determinable if the domain of each variable type is extended to include an appropriate 'undefined' pseudo-value. Dijkstra rejected any such notion, suggesting, for example, that "one might discover a case of bigamy when meeting two bachelors married to the same nobody". Second, one might postpone declaration until the moment the object is required, allowing its initial value to be immediately assigned. He rejected this also, on the grounds that it would promote block nesting to unacceptable depths. It would also fragment context nomenclature.

The design of HPL follows Dijkstra's advice, enforcing simultaneous creation and first assignment via a dedicated primitive. Once created, an object may be explicitly destroyed or transferred to another process (see below). Otherwise it is destroyed automatically at block end.

```
{
  cardinal x

  sequence
    ...  x doth not exist
    x != 4
    ...  x now exists
    destroy x
    ...  x doth not exist
}
```

```
{
  cardinal x

  sequence
    ...  x doth not exist
    x != 4
    ...  x now exists
    transfer x via pipe
    ...  x doth not exist
}
```

Each object must be created before it can be:

- used in a repetition or selection;
- given as an argument to any function;
- transferred to another process.

[2] Dijkstra referred to variables being "inherited" by an embedded block. Here, we favour "borrow" partly to avoid confusion with type inheritance, and partly for clarity. An inheritance is always a *gift*. See #4.2.

Objects can be declared over neither repetition nor selection, because it is never necessary so to do. In the case of a repetition, an embedded block may introduce its own private state. Because a repetition is inherently sequential, it may itself require a variable via which to pass information between iterations. However, any such variable must inevitably be read before assignment upon each and every iteration, including the first. Its prior intialization is thus a logical necessity. As a result the repetition must form part of a sequential composition.

Equally, there is no sensible justification for creating an object within a selection, except where its scope may be confined within a single clause. In which case, it may be the property of a block subtended there. Conditional creation would require subsequent conditional use, inviting disaster, and preventing the required compile-time guarantee that every reference is to an initialized variable. There remains the possibility that initial value is subject to selection, even though creation is not. Some might then prefer to express creation/ initialization in every clause. However, a single creation, followed by a selection between substitute 'initial' values is just as logical, retains simple rules, and removes the need to further complicate the compiler.

Objects acquired from an outer (calling) block may be renamed upon declaration, exactly as a formal parameter may rename an actual parameter upon Pascal procedure invocation. However, since, unlike a procedure (named block), an embedded (unnamed) block is invoked just once, it is rarely as attractive to do so.

```
{
  cardinal x

  sequence
    x != 4
    {
      borrow z alias x
      integer y

      sequence
        y != z
        ... use z and y

    }
    ... use x
}
```

```
{
  cardinal x

  sequence
    {
      return z alias x
      integer y

      sequence
        z != 4
        y != z
        ... use z and y

    }
    ... use x
}
```

Each procedure above expresses precisely the same computation, conducted in precisely the same way. Only the expression differs. On the left, the object *x* (alias *z*) is created and then transferred to the embedded block, on loan. On the right, the same object is created within the embedded block and then passed back to the parent.

Note the automatic numeric type promotion, from (unsigned) cardinal to (signed) integer.

2.3 Procedures and Functions

In Pascal or occam, a procedure[3] binds to every variable 'free' upon definition. As a result it may carry a lot of excess baggage, all of which is accessible. HPL simply limits the baggage to that required, and compels a baggage-list. This means that procedures may be

[3] occam used the term 'PROC' to blur the distinction between procedure and process. With HPL, the term 'procedure' is preferred, partly because it is familiar to a wider (Pascal) audience, but also to preserve the distinction between run-time behaviour (process) and its description (procedure).

defined without reference to *any* external context. They stand alone, and may thus be located, or relocated, within in any *collection*.

So, if we have:

```
procedure dispense
{
  receive cardinal value denomination      // constant parameter
  borrow cardinal balance                  // variable parameter
  return cardinal notes                    // return

  sequence
    notes := balance div denomination
    balance := balance mod denomination
}
```

Then:

```
{
  cardinal myBalance
  cardinal twentyCount

  sequence
    myBalance != 104
    dispense (20, myBalance, twentyCount)
    ...
}
```

is precisely equivalent to:

```
{
  cardinal myBalance
  cardinal twentyCount

  sequence
    myBalance != 104
    {
      cardinal value denomination is 20
      borrow balance alias myBalance
      return notes alias twentyCount

      sequence
        notes != balance div denomination
        balance := balance mod denomination
    }
    ...
}
```

HPL, like occam, affords only constant and variable parameters. Each parameter names, or renames, either an expression or an object. There is no equivalent to a "value parameter" in C or Pascal, which is free to vary within the subroutine. However, a HPL procedure can also give birth to new objects, named by its parent.

Functions can be defined, under the same constraints as in occam. For example, …

```
cardinal function dispensed
  of
    cardinal value denomination
    integer value balance
  is
    notes
  {
```

```
    sequence
      notes != 0
      if
        balance > 0
          notes := balance div denomination
        otherwise
          skip
  }
```

Recursion is permitted in both procedure and function. However, the compiler is required to detect and record this, so that the PE may inform the programmer that the memory requirement may only be determined dynamically. This remains an attached attribute.

3 Programming Constructs

3.1 Repetition

occam opted for regularity with replication, allowing replicated sequence and parallel in syntactic harmony. However, regularity should not be confused with simplicity. HPL puts clear blue water between sequence and repetition. A specific construct must be employed for each, rendering repetition explicit and as transparent as possible.

The history of the development of structured programming (procedure abstraction) is now a matter for textbooks, e.g. [18]. One might begin with Dijkstra's famous missive concerning the use of GOTO [14]. He argued therein that the relation between a position in the program text ("textual index") and process state should remain at all times apparent. The unrestrained use of GOTO quickly destroyed this. Dijkstra went on to embellish his ideas in a seminal work [19]. Here, we incorporate his constraint in our requirement for *transparency*.

The debate concerning the use of jumps within the program text can be traced further back in time to the earliest attempts to improve upon machine language. The desire is to simplify both intuitive and formal reasoning about programs through control flow abstraction. Knuth published a lengthy assessment, often misinterpreted as favouring the retention of GOTO [20]. In fact, he advocated retention only until a set of programming constructs has been identified which obviates its use.

Knuth's corollary is perhaps best understood as adding that only *unconstrained* use of GOTO leads to a hyper-dimensional mapping from procedure to process, and thus unmanageable complexity. Constructs do not eliminate jumps, they merely constrain them to certain patterns. Without a complete set, explicit use of GOTO remains justifiable. The RISC revolution in processor design is highly relevant. A similar revolution in process design is called for – a reduced, carefully chosen, but complete, command set is needed.

Maddux [21] analysed control flow (as visualized using the familiar, but now unfashionable, "flow chart") constrained to *proper programs*, which have:

- a single entry arc
- a single exit arc
- a path from entry to exit through each node.

He then enumerated *prime programs*, which cannot be subdivided. After eliminating those which do nothing, or which may never terminate, one is left a set of control structures from which to compose any program. Curiously, one construct has been repeatedly ignored by language designs over the years since. It is often referred to as DO-WHILE-DO (Figure 1).

The "Structure Theorem" [22] guarantees that any proper program can be converted to an equivalent composed solely using SEQUENCE, WHILE, and IF. However, the price is

measured all too often in both efficiency and clarity. Knuth, among others, has pointed out its limited significance [20]. Effective abstraction needs more.

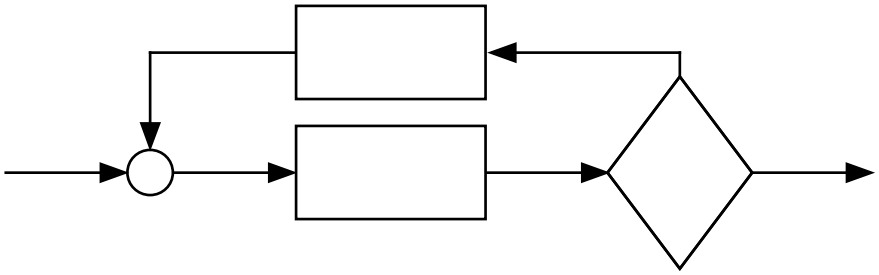

Figure 1. Control flow in DO-WHILE-DO construct.

HPL offers a single repetition construct, strongly inspired by DO-WHILE-DO syntax which Knuth recommended, and attributed to Ole-Johan Dahl [20].

```
repeat
  while thisCondition
    ...  do this
```

Experience testifies to the utility of *multiple* exit points within an embedded sequence. HPL provides them, with the embedded sequence implicit (as in Pascal's REPEAT-UNTIL).

```
repeat
  while thisCondition
    ...  do this
  while thatCondition
    ...  do that
  ...
```

Exit occurs at the first failed condition. The embedded sequence may thus be *broken*. It may well be concern over this that led so many language designers to exclude DO-WHILE-DO.

A null guard, or null command, may be omitted. DO-WHILE (Pascal REPEAT-UNTIL) and DO-WHILE-DO can thus be expressed easily and transparently.

```
repeat                                  repeat
    ...  do this                            ...  do this
  while thatCondition                     while thatCondition
                                            ...  do that
```

Indexing may be added, with the index created automatically at point of first use, with scope confined to clauses which lie beneath. Any type derived from CARDINAL (the set of natural numbers) may be used.

```
repeat
  for each Week day
    ...  do this
  for each day from Monday for 3
    ...  do that
  while (pay > Minimum)
```

A little care is needed to avoid confusion with C or Pascal syntax. Only REPEAT causes repetition. FOR merely introduces further description, governing a single clause.

Indexing over an entire subrange or enumeration, as in line 2 of the above, allows a tie between data and control structure. One can easily express array traversal without defining the same bounds twice.

3.2 Selection

A choice between alternative behaviours may be expressed via a single construct that allows any mixture of *three* different kinds of criteria: condition, value of an expression, or readiness to communicate a value. To ease expression, certain criteria may be combined on consecutive lines. An offer to communicate may be prefaced by a precondition, which must conclude on the preceding line.

```
if
  enCongé and fineWeather
    ...  play
  inTheMood and
  receive money via creditLine
    ...  work
  otherwise
    ...  relax
```

```
if
  asciiCode & #20
    'a'..'z'
      ...  add to buffer
    #2D
      ...  swap buffers
    otherwise
      ...  beep
```

As with occam, the first successful guard, from the top of the list down, will be selected. Any offer to communicate, not excluded by a failed precondition, will remain open until a selection is made. If no guard succeeds then the result is STOP.

HPL draws no distinction between a condition and an expression of any other type. It just allows omission when the expression type is Boolean. If one wished, one could list 'true' and 'false' alternatives below the condition.

Replication is permitted, of either expression or communication, indexing an element of an object or channel array respectively. A replication may suffer a precondition.

```
if
  cashRequired and
  for one i from 1 for n
    acceptable[i]
      ...  thank donor[i]
  otherwise
    skip
```

```
if
  for one i from 1 for n
    secure[i] and
    receive orders via line[i]
      ...  carry 'em out
  otherwise
    ...  improvise
```

Control will pass via the successful guard with the lowest index.

Unlike occam, which allows only conditional receipt via a channel, HPL allows either sending or receiving to be subject to selection. To avoid the need for negotiation [23], HPL instead bars selection at both ends of any single communication. Service protocol, which governs the *use* of channels, enables compile-time verification .

3.3 Alternation

Any reactive system must respond to events signalled by its environment, *alternating* its behaviour accordingly. These shared events $\{g_i\}$ fully describe process-environment interaction, and exhibit a varying degree of *priority* (which, say, increases with i). Note that any such ordering is a property of neither process nor environment. It describes a *protocol* between the two.

A common misconception is to confuse alternation with an *'alternative'* construct – ALT or PRI ALT in occam. It is possible only to approach the required behaviour using this:

```
WHILE running
  PRI ALT i = 0 FOR n
    input[i] ? request
      ...  respond to request
```

```
PRI PAR i = 0 FOR n
  WHILE running
    SEQ
      input[i] ? request
      ...  respond to request
```

To obtain pre-emption, a programmer typically resorts to prioritized scheduling of multiple concurrent processes – PRI PAR in occam. However, it is only necessary, and meaningful, to introduce priority when concurrency is *denied*. The one thing two alternating processes cannot be is concurrent. Furthermore, applying prioritization *locally* invites conflict.

A fundamental axiom of CSP is that no two events are ever considered simultaneous. However, the model does allow two events to be *offered* simultaneously. Even without prioritization, any parallel composition of ALT and/or PAR processes, sharing more than one event, requires resolution via either protocol or negotiation. HPL prefers protocol.

Service protocol established above each parallel construction defines channels, the order in which they are used, the data types of values and objects conveyed, and a prioritization between distinct services. Prioritized response is implemented using a dedicated alternation construct.

```
when
  ...  this happens
    ...  respond
  ...  that happens
    ...  respond
```

```
when
  send copy via postBox
    ...  celebrate
  acquire draft via letterBox
    ...  edit it
```

At least two clauses must be given. The lowest priority clause may employ a null (SKIP) guard. No clause is re-entrant. Part of each response may be to deny further interruption by either the same, or any *other*, event. Note that a disabled clause does not terminate until *all* interruption is denied. An alternation terminates only when all guards are disabled.

Clauses are listed in order of priority, even though prioritization is dictated by protocol. This might usefully be assisted by the PE, which could sort clauses according to protocol.

Priority is defined by interruptibility. Hoare [9] denotes a process P_1, interruptible by P_2, by:

$$P_1 \hat{} P_2$$

The process thus formed starts and continues behaving as specified by P_1 until some event with which P_2 can start occurs. It then behaves as P_2. A natural meaning to priority can be inferred directly. Since each interrupting event is necessarily unique, an ordering can be imposed which precisely interprets prioritization. This is reflected simply in the suffix of the component process:

$$((P_1 \hat{} P_2) \hat{} P_3) \ldots \hat{} P_n$$

Following interruption, no process resumes, and only the last of the given list may terminate. In practice, we wish no response to disappear after completion. The solution is to consider each process *cyclic* about interruption, as indeed most interrupt service routines effectively are. Thus, P_2 starts with interruption of P_1. Upon completion of response, it awaits another instance of its guard event, while P_1 resumes. Such behaviour may be precisely defined, and denoted by a new operator:

$$((P_1 \lfloor P_2) \lfloor P_3) \ldots P_n$$

When process P_2 interrupts P_1, one must consider the possibility that P_1 is blocked awaiting synchronization with the environment. P_1 may also comprise multiple concurrent processes, each of which may be blocked. Upon interruption, *all* offers of shared events

must be withdrawn, pending completion of the response, whereupon they must be re-established.

A more detailed discussion of the new construct[4], its relation to interruption and alternation in CSP, and its use in programming exception response, is available [4].

4 Primitives

4.1 Assignment

Any program describes the behaviour of a system that is discrete in both time and space. States are delimited in time by a single class of physical event, where:

- *first:* one or more registers are read;
- *second:* some function of the values represented by their state is computed;
- *third:* the result is written into one or more registers.

Such an event is considered indivisible (atomic), and is commonly referred to as an assignment. It may be thought of as a bead upon a thread of control.

Process abstraction identifies a single control thread with a specific context. Each thread retains full control over its own private context. When a variable belonging to one process must be assigned a value that is a function of the context of another, synchronization takes place. One may imagine then a bead lying upon both threads.

occam and HPL require three primitive commands to describe assignment – direct assignment, send and receive:

```
day := Monday                          receive appointment via pipe
send (day + 2) via pipe
```

Expressions are formed in a manner very similar to occam. The notation used for direct assignment follows the Algol/Pascal/occam tradition. However, transparency is best served by employing a familiar word, rather than cryptic symbols for communication. This is considered well worth a little extra typing.

Algol notation for assignment is now familiar to generations of professional programmers, and is usually readily accepted by the dwindling minority who are happy with mathematics. However, there is a desperate need to empower a wider population with the ability to program. So-called "script" languages, beginning with Apple's Hypercard™ [26], have demonstrated what can be achieved.

HPL thus also accepts a second, more verbose, syntax for creation and assignment:

```
create thisDate with value [4, July, 2002]
assign thisDate.day value 24
```

4.2 Transfer

In abstraction, *all* objects are dynamic. They appear and disappear over time. This is as true with C or Pascal as it is with HPL. Block structure merely emphasizes the fact. Hence, one should think of the surface beneath a control thread as varying in width along its length. HPL adds one valuable refinement – the boundary between contexts can move. Any such change is inherently synchronous.

---

[4] … with slightly different syntax.

It should be immediately apparent that a *transfer of ownership* has no effect upon the context itself. It merely allows secure expression of certain behaviour. As a result, only appropriate synchronization requires implementation.

As stated earlier, the motivation here is efficiency. A common observation of occam was the inefficiency which can result from channel communications between processes. An occam communication is always conducted by *copying* a value. Experience with other languages strongly suggests passing a reference instead. The choice mirrors that between the two types of Pascal procedure parameter. A reference parameter represents controlled aliasing, and reduces to simple renaming. Introducing the communication of references between processes, however, has the potential for uncontrolled aliasing and to wreak havoc with abstraction.

Traditional "structured" programming relies simply upon procedure abstraction. Any program may be expressed as a composition of standard constructs, and thereby reduced. Data is modelled very simply, in a manner familiar to anyone who has ever used algebra. A name refers only to a value, hiding the complications associated with machinery.

Object-oriented programming (OOP) changes this picture, somewhat subtly. It adds new meaning to a name. Sometimes it refers solely to a value – that which may be assigned. On other occasions it refers to something which can be created, destroyed, and even moved between contexts. Such responsibilities imply an *owner*.

HPL abstraction regards *every* object as dynamic, but admits no detail regarding memory allocation. After declaration, an object begins life only when explicitly created, when it is simultaneously assigned value. If not transferred, an object disappears when the end of its block is encountered, or when it is explicitly destroyed.

Just as there is no need for pointers in HPL, there is no need either for the notion of a reference. When invoking a procedure, one passes either a value or an object. When transmitting to another process, one has the same choice. In designing the syntax for HPL, the possibility of distinguishing between references to value and object was seriously considered. However, this was rejected for two reasons. First, the elegance and clarity of…

```
name := "fred"                    x := 4
```

would be lost. Second, distinguishing operator, rather than operand simplifies the language, yet appeals equally to intuition.

```
{                                 {
  this Thing                        that Thing
  ...                               ...
  sequence                          sequence
    ... create this                   ...
    transfer this via pipe            acquire that via pipe
    ...                               ... use that
}                                 }
```

Note that a name must be declared for an object in both contexts, sending and receiving.

Just as communication and assignment are equivalent in their effect, transfer *between* processes is precisely equivalent to renaming *within* a process. However, there is one important difference. When an object is renamed, the effect lasts only until end-of-block, exactly as happens with a reference parameter in a Pascal procedure. The object is only on *loan*. An object transferred across a channel is a *gift*. The effect lasts until it dies.

Each is appropriate to its use. For example, it is often useful, and efficient, to transfer a single object between processes, each of which uses or modifies it differently, as on a production line. Within a process, lending affords recursive processing of an object recursively defined.

4.3 Use of Transfer

Transfer and Repetition:

Suppose a process repeatedly creates and then transfers an object. It thus forms a *source* of new objects, all of which must be found a new home. A similar loop in a recipient might act as a *sink* for objects generated in this way. Each must employ either repetition or recursion.

```
// Process A
repeat
  {
    this thing
    sequence
      create this ...
      transfer this via pipe
      ...
  }
```

```
// Process B
{
  that thing
  sequence
    ...
    repeat
      acquire that via pipe
      destroy that
    ...
}
```

It is *not* legitimate to acquire the same object twice in succession. One may, however, acquire a second object, with the same name, *after* destroying or transferring the first.

Constraints placed upon the interface between processes, together with the illegitimacy of repeatedly acquiring the same object, ensure both sense and transparency.

Transfer and Selection:

Should a transfer be rendered conditional:

```
if
  someThingOrOther
    transfer that ...
  ...
...  now dangerous to refer to that
```

one invites the equivalent of the dangling pointer problem. However, it makes no sense to refer to an object which may not then exist. Sense demands the later reference to lie in an alternate clause within the selection. All reference to an object is therefore denied following a transfer, even a conditional one. Adherence to this constraint can be verified by the compiler. No dynamic verification is required.

5 Conclusions

An introduction has been given to that part of a new programming language which derives most of its inspiration from occam. It is at a highly experimental stage. Help is required to determine all the consequences of its novel features, and particularly any that arise from their combination. A language is necessarily holistic.

Among novel features introduced is a primitive that allows the transfer of ownership of objects between concurrent processes. This allows a degree of symmetry with their passing between processes in sequence (procedure invocation), and removes (arguably) the primary objection to occam – the inability to efficiently pipeline object processing. Transfer should also improve abstraction by clarifying the relation between process and object.

While experience with occam provides evidence of the utility of transfer, that of the two new control constructs (alternation, and repetition with multiple exits) can only be demonstrated in practice. They should both improve algorithmic efficiency and deny any need for GOTO. A companion paper [10] details with the abstraction of object (data) and protocol.

Acknowledgements

The author is grateful for the helpful insight offered in discussions with Jeremy Martin and David Lightfoot, and acknowledges the overwhelming influence of C. A. R. Hoare, David May, and Peter Welch. He hopes he understood them.

References

[1] East, I. R., *Parallel Processing with Communicating Process Architecture*. 1995: UCL Press. ISBN 1-85728-239-6.

[2] Inmos Ltd., *occam 2 Reference Manual*. 1988: Prentice Hall International. ISBN 0-13-629312-3.

[3] East, I. R., Towards a Successor to occam, in *Communicating Process Architectures 2001*, A. Chalmers, M. Mirmehdi, and H. Müller (Editors). 2001, IOS Press. pp. 231-241. ISBN 1-58603-202-X.

[4] East, I. R., Programming Prioritized Alternation, in *Proceedings of the 2002 International Conference on Parallel and Distributed Processing Techniques and Applications*, H.R. Arabnia (Editor). 2002, CSREA Press.

[5] Meyer, B., *Object-Oriented Software Construction*. 1997: Prentice-Hall. ISBN 0-13-629155-4.

[6] Dahl, O.-J., Simula - An ALGOL-Based Simulation Language. Comm. ACM, 1966. **9**(9): pp. 671-678.

[7] Dahl, O.-J. and C.A.R. Hoare, Hierarchical Program Structures, in *Structured Programming*. 1972, Academic Press. pp. 175-220. ISBN 0-12-200550-3.

[8] Nygaard, K. and O.-J. Dahl, The Development of the Simula Languages, in *History of Programming Languages*. 1981, Academic Press.

[9] Hoare, C.A.R., *Communicating Sequential Processes*. 1985: Prentice-Hall. ISBN 0-13-153289-8.

[10] East, I. R., The "Honeysuckle" Programming Language: Object and Protocol, in *Communicating Process Architectures 2002*. 2002, IOS Press.

[11] Wirth, N., *Algorithms + Data Structures = Programs*. 1976: Prentice-Hall. ISBN 0-13-022418-9.

[12] Welch, P.H. and F.R.M. Barnes, Mobile Data Types for Communicating Processes, in *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, H.R. Arabnia (Editor). 2001, CSREA Press. pp. 20-26. ISBN 1-892512-66-1.

[13] Hoare, C.A.R., Hints on Programming Language Design, in *State of the Art Report: Computer Systems Reliability*, C.J. Bunyan (Editor). 1974, Pergamon/Infotech. pp. 505-534. Reprinted in [24].

[14] Dijkstra, E. W., The GOTO Statement Considered Harmful. Comm. ACM, 1968. **11**(3): pp. 147-148.

[15] Naur, P. (Ed.), Report on the Algorithmic Language Algol 60. CACM, 1960. **3**(5): pp. 299-314.

[16] Naur, P. (Ed.), Revised Report on the Algorithmic Language Algol 60. CACM, 1963. **6**(1): pp. 1-17.

[17] Dijkstra, E., An Essay on the Notion "The Scope of Variables", in *A Discipline of Programming*. 1976, Prentice-Hall. pp. 79-93. ISBN 0-13-215871-X.

[18] Pratt, T.W. and M.V. Zelkowitz, *Programming Languages: Design and Implementation*. Fourth ed. 2001: Prentice-Hall. ISBN 0-13-027678-2.

[19] Dijkstra, E.W., Notes on Structured Programming, in *Structured Programming*. 1972 (first published 1969), Academic Press. pp. 1-72.

[20] Knuth, D., Structured Programming with go to Statements. ACM Computing Surveys, 1974. **6**(4): pp. 261-301. Reprinted in [25].

[21] Maddux, R., A Study of Program Structure. PhD Thesis. 1975, University of Waterloo.

[22] Böhm, C. and G. Jacopini, Flow Diagrams, Turing Machines, and Languages with only Two Formation Rules. Comm. ACM, 1966. **9**(5): pp. 366-371.

[23] Jones, G., On guards, in *Proceedings of the 7th Technical Meeting of the occam User Group: International Workshop on the Parallel Programming of Transputer-Based Machines*. 1987.

[24] Hoare, C.A.R. and C.B. Jones (Editor). *Essays in Computing Science*. 1989, Prentice-Hall. ISBN 0-13-284027-8.

[25] Knuth, D., *Literate Programming*. CSLI Lecture Notes, Vol. 27. 1992: Center for the Study of Language and Information, Stanford University. ISBN 0-937073-81-4

[26] Apple Computer, HyperCard Script Language Guide: The HyperTalk Language. 1988, Addison-Wesley. ISBN 0-201-17632-7.