Communicating Process Architectures 2001
Alan Chalmers, Majid Mirmehdi and Henk Muller (Eds.)
IOS Press, 2001

From safe concurrent processes to process-classes?
PLUSSING new code by ROLLING out and compile?

Øyvind TEIG
Kongsberg Maritime Ship Systems, Ship Control (KMSS-SC)
7005 Trondheim, Norway
oyvind.teig@kmss.no

Abstract. This article expands a concurrent language to support implementation inheritance by making block structures of the super process-class pluggable, and to interface inheritance by making the language's protocol inheritable. The parallel "object-based" concurrent language occam 2 has been used as a catalyst for the concept, causing the language in fact to become (almost?) "object-oriented" (OO). The result is white-box reuse between a "process-class" and its sub process-class, and black-box reuse seen from the client side. Since occam is aliasing free and considered a "safe" concurrent language, the expansion we discuss here keeps those properties - somewhat unusual for an OO system. This new language should be well suited in safety critical systems, since it has inherited the static (compile-time) and analysable properties from occam proper. Basically, two new keywords are defined: PLUSSING and ROLLING. The language feature suggestion is on sketch level only and therefore not complete, no BNF description exists and no compiler has been written.

1 - Introduction

In [1]

> "*Allen Holub suggests that the Java programming language's threading model is possibly the weakest part of the language. It's entirely inadequate for programs of realistic complexity and isn't in the least bit object oriented.*"
> (Underlined here)

This paper tries to define something “threading” and quite a bit object-oriented. It investigates if it is possible *not* to use class-subclass & *method* interface often used in object-oriented languages. Instead *code-insertion* into "thread processes" is proposed, thereby taking a static concurrent language from being "object-based" to a more static type of "object-oriented" concurrent language. Holub's main complaint about Java is that tasks/threads are not first rate citizens, amongst other things a *task* keyword is missing. We will try to add inheritance to a language where tasks (called processes) and safe synchronisation, i.e. *concurrent* behaviour, is as much a first rate citizen as *sequential* behaviour.

The paper suggests a methodology for object orientation somewhere between C++'s *templates*, Java's (and any other OO language's) *implementation inheritance*, and *pattern matching* - for final code insertion and compilation. We need to expand the occam 2 language definition [2] somewhat to achieve the goal - not to have to take a source code file and rewrite it just a little to be able to reuse it, and end up with two distinct programs. Occam actually turned up to be a very interesting language for this exercise, it was more suited than I at first thought.

1.1 Aim and limitations of paper

The aim of the paper is to deliver ideas.

> *The paper tries to "think aloud". It is not part of any ongoing research, contains ideas only, collected by a software engineer working in industry. As such it may be considered "off piste" (dangerous skiing off the downhill marked and beaten track).*

The paper represents a first iteration of a set of ideas. If you like what you see maybe *you* do the next iteration? The paper tries to present a set of coherent ideas, but academically they need "tidying up". The *R*ationale for the ideas has not been discussed, except that "object orientation" and CSP/ occam are, per definition in this context, worth considering as rather sound foundations. And, there are no examples to justify the idea. However, none of this should free *me* from not trying to be as sound as possible when presenting the ideas!

2 - Background

2.1 - Taxonomy

Wegner's taxonomy of object languages [3] is often used to differentiate between several programming models, paving the way towards *object orientation*. This is discussed in [4], here is a copy of the intro:

> *The first object-oriented language was Simula, developed in Scandinavia in the late 1960's and early 1970's and it was truly and completely object-oriented. Several languages have been developed since then that implement the Simula object model more or less completely.*

(1) *No object facilities*, e.g. C and Pascal.

There is no facility for encapsulating state and process simultaneously. State may be encapsulated in records/structs, and process may be encapsulated in functions.

(2) *Object based -- system uses objects*, e.g. Self, Modula-2.

Objects encapsulate state information along with the processes that manipulate that state. These objects may be defined using various mechanisms, including direct construction from parts. The procedures/functions that manipulate the state are called methods. Asking an object to execute a method is called a message.

(3) *Class based -- system uses objects that are defined by classes*, e.g. Actor, CLU, and Ada-8x.
The objects in the system are defined by a type definition mechanism called a class. A class describes a family of similar objects, with identical structure. Each object defined by a single class has the same methods and state variables.

(4) *Inheritance enabled -- system uses classes and inheritance*, e.g. C++ without virtual methods.

Classes within the system may be defined as extensions and specializations of other classes making it possible to build hierarchical (or DAG based) taxonomic type structures in the language. Methods may be statically (by the compiler) dispatched. Methods may be overridden.

(5) *Object-oriented -- system uses classes, inheritance and polymorphic (virtual) methods*, e.g. Smalltalk, Simula, Beta, Java, Eiffel, Modula-3, Oberon II, Ada 95, Cobol 97.

Focus of execution control switches from client to server. A message sent to an object is interpreted by the object which then chooses a procedure to execute. Every message sent to an object is interpreted in the context of the most specific type of the object. Methods are dynamically dispatched.

Observe that concurrency is not discussed. The occam 2 language is a running subset of a *process algebra* called CSP (Communicating Sequential Processes [5]). The basic entity here is a *process*. Traditionally, if we must categorise occam according to the taxonomy above, occam has been labelled *object-based* with the clear feeling that processes do not lend themselves willingly to be lined up in Wegner's taxonomy. However, some would say that processes are the *real* objects. This paper will try to move occam towards the object-oriented (OO) mindset, and keep process-thinking along the line. Inheritance is added, and safe processes are kept.

For this a new term is used: *process-class*.

OO has been solving real software problems for years, the methodology is rather mature. But it is not an exact science since it is far too easy to use it incorrectly and introduce errors. This may perhaps be caused by inadequate implementation languages, the future will tell.

According to Kuhn's terminology [6], one science can have only one paradigm at a time. If *programming* is a science and the ruling paradigm was functional programming, it is probably correct to say that it presently is OO. Then, says Kuhn, when one cannot get further with the present paradigm, a crisis is reached since the problems have been defined so well that one knows what went wrong. So far, OO has not been merged with *concurrent programming*. When the problems are understood well enough - and this includes the need to understand that OO and concurrency need to be merged - will the next computer "paradigm shift" arrive when OO and concurrency merges?

2.2 - Process-classes

Let me try to compare:

Processes (as in occam)

Each process is a separate program which runs in its own "thread" of actions. They communicate by sending messages to each other, and this is their only means of communication after they have started running, so encapsulation is water-tight. Static.

Objects are on land.

1. You are yourself proper, a Homo Sapiens - no super process.
2. No talking over your head - you see all that is going on.
3. No talking through you - no super to pass difficult questions to.
4. If several sequential talkers, none may unknowingly modify others' view of

you - no aliasing error (more about aliasing later).
5. Several concurrent talkers OK, none may interfere with each other - thread-safe by birth.

Objects (as in OO)

When "classes" have started running (instantiated), they are called "objects". They run in their user's thread, not in their own thread. They communicate with access methods or by taking advantage of holes in encapsulation.
Implementation inheritance is enforced, to that the programmer may make a new class by building on similar code functionality of a super class. Dynamic.
Objects are floating around.

1. You are an animal first, then Homo Sapiens - super object exists.
2. Talking over your head: somebody may be talking with your super, change super's mind, and confuse you - super class may have own state.
3. Talking through you - super to pass difficult questions to.
4. If several sequential talkers, one may unknowingly modify others' view of you - aliasing errors may occur.
5. Several concurrent talkers not defined, may interfere with each other - not thread-safe by birth.
6. Threads and objects not "orthogonal".

Process-classes (as in this paper)

All properties of occam processes are kept, plus implementation- and interface inheritance. Static.
Objects are floating, but attached to land.

1. You are yourself proper, but internally you are animal first, then Homo Sapiens - you are compiled into super process-class.
2. Talking over your head: not possible since it is impossible to distinguish between you and your super.
3. Talking through you - super process-class takes difficult questions.
4. If several sequential talkers, none may unknowingly modify others' view of you - no aliasing error.
5. Several concurrent talkers OK, none may interfere with each other - thread-safe by birth.
6. Are processes and process-classes "orthogonal"?

The "objects are.." statements are imaginative catch phrases to visualise the differences. I have landed on the term process-*class* rather than process-*object*, to indicate that there is only one entity "alive". Where objects are "living" instances, their templated source-code classes are "dead". With process-classes it is the process which "live", and there is no explicit "class" term in this scheme.

I will be the first to admit that *maybe* "process-class" is an impossibility. By splitting *my* personality I will try to get to know it - at least it responds in some way. I am, however, bewildered whether it is an idea which scales or just some syntax that comes right. Also, knowing the cognitive properties of the suggested scheme would be interesting - how will a programmer understand and use it, compared with occam 2 and "standard" OO languages?

2.3 - occam 2 as a catalyst language

The occam 2 ("occam") language was designed as, and has proven to be, a "safe" language. It features:

- No pointers.
- Concurrent usage rules enforced.
- May be considered a "runnable" version of the process algebra CSP - analysable.
- No recursive calls - possible to calculate exact stack size.
- Array index overflow caught at compile or run-rime.
- No aliasing errors allowed
- Etc.

Aliasing is when several different identifiers refer to the same object, and this gives rise to subtle and often serious errors when there is a reader/writer role conflict. In a typical case, it may be shown that in most programming languages neither the compiler nor the run-time system will catch a simple example, which results in x becoming *zero* (instead of being unmodified) after these two assignments: `x:=x+a` then `x:=x-a`. In occam the compiler catches it. See [7] and the seminal paper "The Geneva Convention On The Treatment of Object Aliasing" [8]. Recently, a free of aliasing error *and* dynamic data type has been bound into occam, with two new keywords MOBILE and CLONE [9].

Since the suggestions of inheritance discussed here ends up with a new language where all points above should also be valid, we have ended up with a quite "safe" OO language.

2.3.1 - Process "method" interface in occam is not very useful

The occam language doesn't have a *method* interface to concurrent processes, so we couldn't reinvent standard OO. Standard OO defines methods in classes, and when you want to reuse the class, you define a derived child class which has access to the base- or super-class methods. A derived class also often has access to some of the internal objects of the super class, this may be everything from wanted to unsafe. With occam it is somewhat different - to communicate with a process you send it messages, it's *the only way* to do this dynamically, there are no internal entities inside another process to have access to. In OO jargon, *thread-safe* access is *all* there is.

The programming community has seen how complex the method interface has become when objects start to live on their own, as threads, tasks or processes. Back to Java again, it is such a wonderful metric to compare with. It does have thread support, it is possible to make objects thread-safe, but it is difficult to get it right. The language has *synchronized*, *wait()*, *notify()* and *notifyAll(),* but programming thread-safely boils down to using coding *conventions*. Language support is present, but sparse. Now, let us look at an occam example.

```
PROTOCOL SetGet
  CASE
    set.NoRe; INT
    get.Re
    get.End; INT
:
PROC SetMethod (CHAN OF SetGet to, VAL INT Data)
  to ! set.NoRe; Data
:
PROC GetMethod (CHAN OF SetGet to, from, INT data)
  SEQ
    to ! get.Re
    from ? CASE get.End; data
:
PROC Server ([]CHAN OF SetGet in, out)
  INT data:
  SEQ
    data := 0
    WHILE TRUE
      ALT i = 0 FOR SIZE in
        in[i] ? CASE
          set.NoRe; data
            SKIP
          get.Re
            out[i] ! get.End; data
:
```

```
PROC Client (VAL INT Me,
  CHAN OF SetGet in, out)
  INT id, noOfMe:
  SEQ
    noOfMe := -1
    WHILE TRUE
      SEQ
        GetMethod (out, in, id)
        IF
          id = Me
            noOfMe := noOfMe + 1
          TRUE
            SKIP
        SetMethod (out,Me) -- ..
        out ! set.NoRe;Me  -- same
:
PROC Test()
  VAL Num IS 10:
  [Num] CHAN OF SetGet north, south:
  PAR
    Server (north, south)
    PAR i = 0 FOR Num
      Client (i,south[i],north[i])
:
```

Example 1

The example illustrates a typical system: a *Server* is serving request from 10 *Client*s. Two "methods" have been wrapped around Server, the *SetMethod* and *GetMethod* do channel I/O.

Server access is thread-safe because of the occam process model and the ALT mechanism. There are no semaphores or "synchronized" involved. In one of the occam compilers we use (SPoC), SetMethod and GetMethod are started and stopped as temporary processes on each "call", to flatten out the process hierarchy. So, how SetMethod and GetMethod are run is just an implementation issue, as long as the implementation is thread-safe.

If we would want to make a new process called *Server2*, which inherited Server we see at least two problems. 1) SetMethod and GetMethod are not defined to *belong to* Server, and 2) if they were and we were able to override them with new methods with new functionality - is this all we need? Would we want to *inherit more*, in a way, so that we would be able to *redefine more*? Point 1 boils down to saying that we have no CLASS definition in occam - this has indeed been suggested by Barrett (see chapter 5.10).

So, SetMethod and GetMethod are not much worth except for readability. They only add run-time overhead.

However, observe that even if SetMethod and GetMethod are not defined to belong to Server, they are *reusable components along another line*: for the *SetGet protocol*. Reusability is a context sensitive trait.

2.3.2 - Sub-classing through occam process layering is not practical

The usual way to "sub-class" an occam process is to start one instance of it in a layered process which consists of two processes: the "super process" and the "new" sub-process which takes over some of its functionality. This is "sub-classing" through process layering. It is not very flexible, since *every* "function" (tag of the protocol) must be re-implemented and the messages either passed to the super-process and its reply handled, or handled differently locally. There is no way whereby a super process will take over the messages not handled by the layering process.

In other words, so called *implementation inheritance* is not directly supported, even if we can program it. We cannot in occam state that a "house *is a* building" easily.

2.3.3 - Polymorphism through occam anonymous communication scheme is nice

Polymorphism is plug-in functionality set into system, since it makes it possible to use the same plug in several types of receptacles. Some times we do not want to know what kind of dog is barking when we state *Dog.Bark()*, but we can certainly hear the difference. We can use the same verb to any listener, and they will respond individually.

In occam we can likewise send the message *Bark* over a CHANnel. A process has no idea who receives the message, it could be a Beagle or a Terrier process. Occam communicates over *named channels* with *anonymous processes*, so the process on the other end of a channel might be a Dog of any type.

The dog owner (client) could in fact query Dog which type of Dog it is, if this was defined in the protocol or interface, an important trait in interface based programming. This should give the same functionality as some languages' *run-time type inspection*.

So, occam does have a system for polymorphism, even if it is not method naming polymorphism. Occam does not seem to need dynamic binding to facilitate this (more later).

But, the way to actually plug the different *Dog* processes is somewhat more difficult in occam than in standard OO languages. Maybe this has to do with how occam processes are stopped, they do not stop when they go out of scope, they *have stopped* when they go out of scope (implicit "join"). We can not "kill" an occam process from the outside. We have to send it a message to have it exit. This message will only be accepted when the process is not engaged in some other activity. And after a process has been stopped, another user could legally try to communicate with it, the semantic is to wait forever. We have no null-pointer equivalent for an attempted pathological CHANnel communication. Also, we do not have a way to differentiate between *normal waiting forever* and *pathological waiting forever*.

None of the problems mentioned above have been solved with the suggested solution I discuss in this document. But do observe, the immediate occam solution does give a, say, 90% thread-safe system. Since occam is a runnable subset of the CSP - we can go to 100% by using design patterns or model the system in a CSP based tool. This is an entirely opposite picture as compared to using Java's Threads and basic concurrency primitives.

2.4 - Concurrent OO and "inheritance anomaly"

If we wanted to inherit ALT, one of occam's synchronisation constructs, then a problem "more severe than violation of class encapsulation in sequential language" could fast surface [10]. (ALT is similar to the *select* calls in Unix and the *select* statement in Ada.) *Inheritance anomaly* is the problem which arises from inheriting synchronisation code in "object-oriented concurrent languages". In [11] Meseguer points out that "synchronization code is often hard to inherit and tends to require extensive redefinitions". Meseguer solves the problem by rewriting synchronisation parts in such a way that the synchronisation code actually falls out of the formulae. With no synchronisation, there is no inheritance anomaly either.

The occam inheritance model we define here will make possible "inheritance" of synchronisation, but since the solution is within the present occam process model, I see no reason why inheritance anomaly should appear. I quote "inheritance" because I have a feeling some will say that what I suggest below is not really inheritance. But then, what is it?

2.5 - White-box and black-box reuse

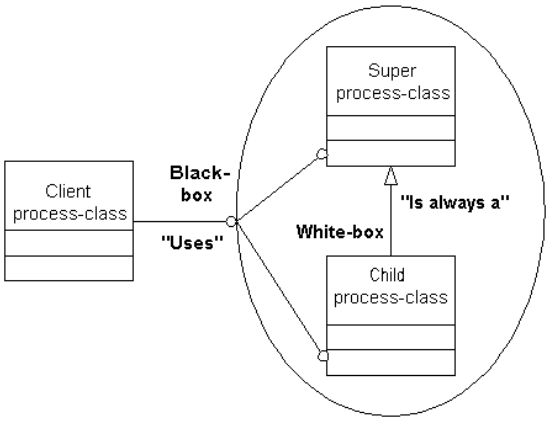

Fig. 1

The suggested scheme gives both *white-box* and *black-box* reuse.

White-box reuse (implementation inheritance) is when a child process-class knows about the internal details of the super process-class. The reuse "operand" is an # in association with IF, ALT etc. (much more about the #-modified keywords later). The problems associated with white-box reuse is minimised by not allowing unclear semantics to creep into the super process-class.

Black-box reuse (interface programming) where details are never revealed to a client, is the standard occam 2 CHANnel interface contract. This will come into play *whenever a process* is instantiated, there is no other way. Occam also allows and enforces Concurrent Read Exclusive Write (CREW) access rights to entities like arrays etc. - this is within the black-box regime. We now also have included interface *inheritance*.

This should give the user a tool to enforce reusability, maintainability and extensibility.

3 - "Plussing" processes

To Walt Disney "plussing" was to add detail [12]. "Many of the statues then undergo a 'plussing' process, consisting of detailing the sculptures with special materials."

What we try to do here is the "plussing process" of "PLUSSING" code to an existing process.

3.1 - The PLUSSING and ROLLING operators # and @

I suggest the following occam keywords and combinations to be "plussable" (and "rollable", later):

Basic	Flow etc.	Processes	PROTOCOL	Conditional
IF	TRUE	SKIP	CASE	CASE
ALT	FALSE	STOP		ELSE
PAR	FALSE & SKIP			
SEQ	TRUE & SKIP			

Table 1 - Suggested occam PLUSSING keywords

These keywords may be appended with a trailing # or @. If the occam compiler sees an `IF#` it will simply remove the # and compile as usual. But it will "export" some knowledge that a child process-class may override the `IF#` block with what is defined by the corresponding `IF@` block of the sub process-class.

It is tempting to coin hash the "plussing operator". I do realise that naming, both with respect to keywords and operators, is a risky business: bad names often turn us off.

We term the @ "operator" for the "rolling operator", because this is where the compiler in a sub process-class will "roll out" the code defined into the super process-class. So, all entries in the table above are also "rollable". (Should we say "if plus" for `IF#` if we need to say it, and "if roll" for the `IF@` if we need to say it?)

Even if SKIP is a valid component in several occam constructs, like PAR and SEQ, we cannot drop PAR and SEQ from the list, as we would like to override existing code blocks.

I have chosen *not* to suggest to inherit WHILE. The reason for this is that I feel it hard to understand how a WHILE TRUE or WHILE (condition) would behave inherited. Perhaps some structures are good to inherit, some not. In one of the examples below, I have shown that it is still possible to plug in *exit* functionality. I have also chosen *not* to define CHAN#, PROC#, INT# etc., as I think it perhaps not necessary and overly complex. Defining a complete "occam-pluss" shadow language is perhaps not such a good idea? The suggestion I have for inheriting a protocol is not with PROTOCOL# - more later.

4 - Suggestions

I do not show examples of all suggested PLUSSING keywords. Of course, taking the ideas further than this would require a substantial amount of work.

4.1 - Inherit process with plug-in of blocks

Now, to the suggestions. The main idea is to export unnamed blocks of code, and give the user the possibility to override default code in these blocks with new code, or to add code where such a block is intended to be.

```
PROC Buffer (CHAN OF INT in, out) PLUSSING (TRUE)
  INT data:
  WHILE TRUE
    SEQ
      in ? data
      IF
        data <> 0
          out ! data
        TRUE# -- Plussing operator on TRUE
          SEQ
            in ? data  -- Default is to read a second time..
            out ! data -- .. and send whatever we then get
:
PROC BufferChild() ROLLING Buffer
  TRUE@ -- Rolling operator on corresponding TRUE
    out ! data -- Don't read a second time
:
PROC Buffers (CHAN OF INT in, out)
  CHAN OF INT local:
  PAR
    Buffer      (in, local)
    BufferChild (local, out)
:
```

Example 2

The Buffer process contains an IF branch which the designer wants to be able to override in a child process-class later on. This is done with the new TRUE# keyword. When the compiler sees a TRUE# during compilation of Buffer, it will remove the hash and compile. The code must pass standard occam compilation. PLUSSING (TRUE) informs the user that there is a block which may be overridden.

Observe that some keywords will not by themselves completely define which context they stand in. TRUE is one of those. The context of *any* PLUSSING keyword must be known by the sub process-class designer; more about this later.

BufferChild is a child process-class which is "ROLLING Buffer". The compiler will compile Buffer splicing it with BufferChild and produce BufferChild. At this level this is mere text replacement, provided we have a compiler and the source code of Buffer available.

Buffers has one instance of Buffer and one of the BufferChild process-class.

- BufferChild adds no new parameters to Buffer, so an empty pair of brackets is shown.
- BufferChild could have values parameterised into any of the blocks it defines.

4.2 - Inherit process with plug-in of synchronisation

Now, let us look at a program which has been designed to accept alternative service channels.

```
PROC ALTer ([]CHAN OF INT in, out) PLUSSING (ALT)
  INT data:
  SEQ
    data := 0
    WHILE TRUE
      ALT
        ALT i = 0 FOR SIZE in
          in[i] ? data
            INT result:
            SEQ
              -- Process something
              out[i] ! result
        ALT#
          FALSE & SKIP
            SKIP
:
PROC ALTerChild (CHAN OF INT inspect.out, CHAN OF BOOL inspect)
ROLLING ALTer
  BOOL now:
  ALT@
    inspect ? now
      inspect.out ! data
:
CHAN OF INT  a,b,c:
CHAN OF BOOL d:
PAR
  AProducer  (..)
  ALTerChild (a,b,c,d)
  AConsumer  (..)
```

Example 3

ALTer accepts input from a number of clients on the array of channels called *in*. An open ALT branch space has been left for some other things to happen. The ALTerChild adds a "method" to inspect the internal data. The only traces ALTer alone will have of this code, is the necessary hook it needs to be able to insert the block later on. If ALTerChild is compiled completely anew, with no "incremental" insertion into ALTer, ALTer is quite happy with zero traces of the extension hook.

In the example a local variable is declared above the ALT@. I am not certain where it should be placed, but imagine that the suggested placement both should be OK for a compiler and give acceptable syntax.

We now have a mechanism for *pluggable thread-safe access functions*. We see that the added functionality only to a small degree has to be designed into the base process-class. We set up a flag saying "insert as much as you want here".

As we see here, ALTerChild has *access to all internals of ALTer*. The bad thing is that this is like making everything *public* to the sub process-class, so that any rewriting of the super process-class could break the sub process-class. Not to break the *client* is imperative in OO design. But this does not break the client. However, in *any* class design, designing it right is crucial. Since the new sub process-class is standard occam and as such 100% encapsulated, the sub process-class breakage is "acceptable".

Some words need to be said about how parameters should be stated in sub process-classes. I suggest that the compiler should enforce strict equal sequences for super and child. So, if super has *(a,b,c)* and child uses redefined versions of *b* and *c* and adds the new parameter *d*, then child's layout must be *(b,c,d)* (optionally *(a,b,c,d)* - more later), and client's use *(a,b,c,d)*. If f.ex. child specified *(d,b,c)* it would be a syntax error, one cannot add new parameter inside super's parameter list.

We shall see below that we really do not need to inherit the ALT in this case.

4.3 - Inherit process with plug-in of data

How could we make it possible to insert a timeout in an ALT? We insert extra places for data declaration, data initialisation and an ALT component for it:

```
PROC TIMEOUTer ([]CHAN OF INT in, out) PLUSSING (SKIP, SEQ, ALT)
  INT data:
  SEQ
    SKIP# -- Any sub process-class data
    SEQ
      SEQ# -- Any sub process-class data initialisation
      data := 0
      WHILE TRUE
        ALT
          ALT i = 0 FOR SIZE in
            in[i] ? data
              INT result:
              SEQ
                -- Process
                out[i] ! result
          ALT# - Any sub process-class timeout action
            FALSE & SKIP
              SKIP
:
PROC TIMEOUTerChild (CHAN OF INT inspect.out) ROLLING TIMEOUTer
  SKIP@
    TIMER clock:
    INT time:
  SEQ@
    clock ? time
    time := time PLUS 1000
  ALT@
    clock ? AFTER time
      SEQ
        inspect.out ! data
        time := time PLUS 1000
:
```

Example 4

This clearly is not elegant! The base process TIMEOUTer has to know "too much" about any future expandability of the process. The so-called "fragile super class scenario" [13] hits. In the example we see three PLUSSING keywords being exported - they should be in the same sequence in that list as they are in the code.

Below is an alternative approach, which also takes the suggestions further than to mere text replacement. Observe that if some kind of run-time code insertion is used, not even the above examples are "text replacement".

```
PROC TIMEOUTer2 ([]CHAN OF INT in, out) PLUSSING (FALSE&SKIP)
  INT data:
  SEQ
    data := 0
    WHILE TRUE
      ALT
        ALT i = 0 FOR SIZE in
          in[i] ? data
            INT result:
            SEQ
              -- Process
              out[i] ! result
        FALSE# & SKIP# -- For sub process-class
          SKIP
:
PROC TIMEOUTer2Child (CHAN OF INT inspect.out) ROLLING TIMEOUTer2
  INITIAL
    TIMER clock: -- Assume PROC..
    INT time:    -- ..global scope
    SEQ
      clock ? time
      time := time PLUS 1000
  FALSE@ & SKIP@
    clock ? AFTER time
      SEQ
        inspect.out ! data
        time := time PLUS 1000
:
```

Example 5

Something more elegant than "FALSE# & SKIP#" is perhaps needed, but I have used it to be loyal to the statement that code should be compilable if the # is removed. It is not legal occam just to parenthesise "(FALSE & SKIP)" like this. I have done two things here. First, I have used the occam 3 [14] INITIAL keyword and included runnable code with it. Second, I have removed the extra ALT level in ALTer.

In this example I have allowed the child to have own executable code which is not tagged to concise # statements in the base process. This would be safe, since it is paired with precise semantics through the INITIAL keyword. Likewise, the occam 3 FINAL keyword could also be used. INITIAL and FINAL would function as a type of process-class constructor and destructor.

4.4 - Inherit interface with plug-in of new protocol

We now have a mechanism where inheritance of PROTOCOL is viable:

```
PROTOCOL SetGet PLUSSING (CASE)
  CASE#
    set.NoRe; INT -- Tag=0 (input)
    get.Re        -- Tag=1 (input)
    get.End;  INT -- Tag=2 (output)
:
PROTOCOL SetGetChild ROLLING SetGet
  CASE@
    get.Re;   INT -- Tag=1 (input)  Overridden
    mask.Re;  INT -- Tag=3 (input)  New
    mask.End; INT -- Tag=4 (output) New
    kill.NoRe     -- Tag=5 (input)  New
:
```

Example 6

The *get.Re* tag is allowed by the compiler to be overridden only if *all original usage* of it in the base process has been removed by the sub process-class.

A client using the original *get.Re* is unaware that a child process-class may use another version of *get.Re*. Introducing a child process-class will not break this client.

The compiler will ensure that all PROTOCOL tags are unique, with no overlap.

4.5 - Inherit process with plug-in of new protocol handling

This is where we have reached at interface based *inheritance*. Interface based *programming* we have always had with occam, according to the best practices: full encapsulation. We now have interface based polymorphism, as opposed to the implementation based polymorphism we demonstrated with the other examples.

```
PROC PROTer ([]CHAN OF SetGet in, out) PLUSSING (CASE)
  INT data:
  BOOL running:
  SEQ
    data := 0
    running := TRUE
    WHILE running
      ALT i = 0 FOR SIZE in
        in[i] ? CASE#
          set.NoRe; data
            SKIP
          get.Re
            out[i] ! get.End; data
:
PROC PROTerChild ([]CHAN OF SetGetChild in) ROLLING PROTer
  -- Only input channel redefined
  CASE@
    INT extra:
    get.Re; extra -- Redefined
      out[i] ! get.End; data + extra
    kill.NoRe -- New
      running := FALSE
:
CHAN OF SetGetChild a:
CHAN OF SetGetChild b: -- Could also have been PROTOCOL SetGet
PAR
  AProducer   (..)
  PROTerChild (a,b)
  AConsumer   (..)
```

Example 7

Observe that the protocol CASE statements of the base process-class have not been augmented, and that we do not need to repeat the base process-class statements in the sub process-class. In order to facilitate this we need the compiler to do some kind of pattern matching so that the new code is inserted correctly.

PROTerChild's use of the *out* channel still handles the super protocol only, so we have not parameterised *out* into PROTerChild. Maybe PROTerChild would be more readable if we had included the *out* channel as well - this could probably be optional. I have done so in the following examples.

Channel *b* could also have been of PROTOCOL type *SetGet*, since any code only uses elements from this super protocol. Observe that occam 2 does not require all tags from a PROTOCOL to be used, the compiler would not know the directionality of a tag, it could be used for input, output or both.

4.6 - Inherit and inherit again: nesting

Nesting is legal and possible, even if CASE@# in the example looks like a cartoon figure crying out in pain - is there any such thing as a syntactic singularity?

```
PROC PROTerChild ([]CHAN OF SetGetChild in, out) PLUSSING (CASE) ROLLING PROTer
  -- Super from example 7
  CASE@#
    INT extra:
    get.Re; extra
      out[i] ! get.End; data + extra
    kill.NoRe
      running := FALSE
:
PROC PROTerChildChild() ROLLING PROTerChild
  CASE@
    kill.NoRe
      running := BOOL (data)
:
```

Example 8

Below is an example of PLUSSING operator # nested inside ROLLING operator @. PROTerChild designer has decided that only *mask.Re* processing may be redefined, not *get.Re* code:

```
PROC PROTerChild ([]CHAN OF SetGetChild in, out) PLUSSING (SEQ) ROLLING PROTer
  -- Super from example 7
  CASE@
    INT extra:
    get.Re; extra
      out[i] ! get.End; data + extra
    INT mask:
    mask.Re; mask -- New
      SEQ#
        data := data BITAND mask
        out ! mask.End; data
    kill.NoRe
      running := FALSE
:
PROC PROTerChildChild() ROLLING PROTerChild
  SEQ@
    data := data BITOR mask
    out ! mask.End; data
:
```

Example 9

5 - Discussion

5.1 - Dynamic vs. static

Occam 2 is a *static* language, too static, many say, we don't even have a choice. In occam 2 we cannot send instances of processes over channels. This seems to be worked upon by several researchers. I assume that this static version of an object-oriented language would be a nice flower in the language flora, even if we can not *new* an object - only *start* a process-class type of object. This line of thinking should apply for safety critical systems, *this* new language has inherited that trait from occam.

> *One of the major features of object-based and object-oriented programming is late binding of methods, which means that we can write code in terms of abstract operations without knowing exactly which concrete operations will be executed at run time.* [15]

Named channels of occam are "*connected* late", even if the other and anonymous end of a channel is known at compile-time. They are connected to servers at run-time. It is not like a subroutine which is statically linked. As mentioned earlier, we can have any unknown Dog bark over that scheme. Even if "bound late" requires dynamic linking and "connected late" does not, in many respects the functionality would be the same for the programmer. Here is one of the arguments for calling this occam object-oriented.

There is a slight difference with standard method based OO, though. In both cases the client controls what *service* is to be executed, and the server which *code* is executed, but with this scheme the client might need to know something of the outside connection. In occam the client may send on a single channel, in which case the servers must start and stop themselves in straight sequence. A client can also send on one of an array of channels, in which case the servers could run concurrently.

```
PROC Connector1()
  CHAN OF INT chan:
  PAR
    INT i: -- Client1 starts here
    SEQ
      i := 0
      WHILE TRUE
        SEQ
          i := i + 1
          chan ! i -- To any dog
    WHILE TRUE -- Server1 starts here
      INT bark := 0:
      SEQ -- PAR not legal
        WHILE ((bark REM 10) <> 0)
           -- Terrier1 sequential server
          chan ? bark
        WHILE ((bark REM 100) <> 0)
          -- Beagle1 sequential server
          chan ? bark
:
```

⇑ Server1 SEQ only, error reported by compiler

Server2 PAR only, deadlocks at runtime if SEQ ⇒

```
PROC Connector2()
  [2]CHAN OF INT chan:
  PAR
    INT i: -- Client2 starts here
    SEQ
      i := 0
      WHILE TRUE
        SEQ
          i := i + 1
          VAL Index IS i REM 2:
          chan[Index] ! i -- To any dog
          PAR -- Alternatively:
            chan[0] ! i -- To Terrier2
            chan[1] ! i -- To Beagle2
    WHILE TRUE -- Server2 starts here
      PAR -- Not SEQ here!
        WHILE TRUE
          -- Terrier2 concurrent server
          INT bark:
          chan[0] ? bark
        WHILE TRUE
          -- Beagle2 concurrent server
          INT bark:
          chan[1] ? bark
:
```

Example 10

In this case, to avoid deadlock, the PAR in Client2 is only correct if there is a corresponding PAR in Server2.

This is an example where occam does not completely have "What You See Is What You Get" semantics, where one can program a client or server and *only* know the protocol (message sequencing) involved, not how the other part is programmed. Here we have to know about the "topography" (the PAR) at the sibling process level.

That being said, WYSIWYG semantics is not at all present with Java's concurrency features, where *wait*, *notify* and *notifyAll* always have to match 100% correctly between client and server, two programs which should be independently coded.

The aliasing-error free dynamic occam described in [9] enhances occam with copy-by-ownership-moving semantics. An idea which sprang from this has been described in [16], where access rights to memory objects are given to a process over a channel-like mechanism.

5.2 - Garbage collection

The suggestions discussed, as implemented with occam 2 as catalyst language, will require no garbage collection. Process-class life span is completely known at compile-time.

5.3 - Unnamed blocks vs. named methods

With method based interface to classes we are able to precisely see what we may redefine. With the ROLLING out scheme suggested here, we still need a precise interface definition. The PLUSSING block has no name and no explicit interface "parameter list". So, is this worth anything?

There are at least three scenarios:

1. We design both super and child at the same time.
   - This should be OK, and should be no worse off than subclassing in Java. A purpose-built (folding) program editor would be a nice treat!
2. We design child later on, but have access to super's code.
   - As above.
3. We design child later on, but have only access to some kind of interface description.
   - The problem here is the interface description. Here again, any *Java* method which needs to be overridden, or any local variable which needs to be accessed, must *also* be known. When we say this, we *also* need some information of what the methods and variables are there for, their semantic meaning.
   - An "abstract" PLUSSING operator interface description would have to describe what the original designer wants to tell: (1) what is in scope, (2) what the base process-class PLUSSING block contains, and (3) which kind of *new* functionality the base process-class designer had in mind for the sub process-class designer.
   - Thus, protecting investment in the base class by not giving a user more information than really needed, should be possible.

I cannot see that the lack of an *explicit* name and interface hinders what the inheritance mechanism may be useful for.

Observe that even if we have no method interface to a process, this does not mean that the *process* doesn't have a precise interface defined. The occam process interface *is* precisely defined.

Probably the model sketched here is quite suitable for graphical tools, since it could build up the super process-class interface description dynamically. Maybe the model actually needs such a tool.

5.4 - Limitations of scope of this paper

This paper only discusses inheritance in concurrent PROCesses started with PAR. Perhaps the same principles could be used for sequential PROCesses started with SEQ, or for FUNCTION. Occam does not have a taxonomy to differentiate between "concurrent" and "sequential" PROC, it is the contents and the usage which implicitly decide. This will not be discussed here.

Also, the newer constructs of occam 3 (or occam 2.1) have not been discussed (except briefly INITIAL and FINAL).

5.5 - Generic buffers

I have found no way to make generic buffers or multiplexers which would handle *any* PROTOCOL. This is a basic feature not present in occam. I have missed it. Without a primitive mechanism for generic PROTOCOL handling there is little point in trying to invent an inheritance system to support it. Occam does not accept CASE inputs on a CHANnel with the ANY protocol, this makes it no easier.

Off the record, we have been able to make a generic buffer and mux/demux in standard occam. For a new project, where occam is running on MS-Windows, we have written general protocol pack and unpack processes which may be connected back-to-back, and which will buffer messages of any occam PROTOCOL. Even if the code is 100% occam (we use the SPoC occam compiler for this), we have had to make a protocol description by hand (as a two-dimensional integer array), and this is parameterised into the process at run-time. However, this is not *proper* generic protocol handling.

5.6 - Some implementation issues

Some other *ideas* (I am still off piste):

- Perhaps it is possible to send off a process-class as a zipped encrypted source file and have the compiler compile against this "source".
- This could perhaps be done by a Just In Time (JIT) compiler, perhaps making it possible to load sub third-party process-classes over the net.

5.7 - Explicit and implicit PLUSSING

Implicit PLUSSING would mean that we, according to certain rules, are able to write a sub process-class which would be "rolled out" on the super class with no explicit PLUSSING operator in the super class. With the simple syntax as occam 2, such a scheme could be possible. There is no problem in understanding that a glove and not a sock fits onto a hand, even if we can put a hand inside a sock. Put under the regime of a graphical tool this could be even more viable.

5.8 - Closures

The parameterised block of Ruby [17] and Smalltalk and functional programming languages, also called *closure*, should perhaps be looked upon in this context. They are highly dynamic concepts, where a local block of code may be sent over as a parameter and then converted back to a block of code, and then run within the callee's context. The rather unusual concept of closure as implemented in Perl OO is excellently described in [18].

This paper was much inspired by the concept of closures. I have an intuitive feeling that doing some more thinking along this line would be interesting.

5.9 - Names of child process-classes

With occam 2's weak module concept (it supports only the #USE separate compilation into a "flat namespace" [19] - better in occam 3), it would be *allowed* to write:

```
PROC PROTer() ROLLING PROTer
```

but it would only be of use if we want the super PROTer to go out of scope once child PROTer has been declared.

5.10 - occam CLASS

In [19] Barrett outlines an occam CLASS concept as a means of grouping together different kind of objects like CHANnels, variables and PROCesses. A CLASS could be used within any TYPE definition. No inheritance mechanism was described. Later, with the occam 3 draft [14], Barrett did not include CLASS, unknown for what reason.

This paper has avoided any CLASS definition by piggy-backing on occam 2's PROCess.

6 - Conclusion: "PLUSSING new code by ROLLING out and compile"

Making block structures of the super *process-class* code or super *protocol* description pluggable, seems like a scheme which may work to facilitate reuse by way of inheritance.

However, this paper does not state that inheritance is A Good Thing, or A Bad Thing. In [20] Microsoft points out the problems with implementation inheritance; and their widely popular Component Object Model (COM) does not support it. Instead COM focuses on clear interface definitions. The scheme described here contains both.

I have tried to discuss how a kind of *implementation inheritance* may be added to a concurrent language. Since this "*is always a*" relationship is static, as such it is "safe". A kind of *interface inheritance* is also discussed. The "safe" concurrent language occam seemed like a good catalyst for the ideas. Two kind of encapsulation emerged: white-box (weak) between process-class and its super class, and black-box (strong) between the new process-class and its client. The latter constitutes a "*uses*" relationship.

Since the *occam/CSP process model* was the basis and the result of all this, there is no process-class which may be used by several thread classes in an unsafe way. All instances of any process-class live within their own thread, with no possibility of conflicts caused by erroneous type of encapsulation.

The result is a (1) concurrent, (2) aliasing-free, (3) static (compile-time built) and an (4) object-oriented (-like?) language, so far only sketched to this point - with more answers left than questions posed.

6.1 - Afterword - in need of a taxonomy

In "Pitfalls of Object-Oriented Development" [21] Webster outlines OO. I have *tried* to set his list in context:

Trait	**Comment**	**occam 2**	**This paper**
Object		Process	Process-class
Abstraction		Yes	⇐
Encapsulation		Channel interface	⇐
Instantiation	*new* operator	Start process	Start process-class
Inheritance (**is a, is kind of**) (interface, implementation, single, multiple)	Some consider this not wanted!	No	Yes, both interface and implementation, single
Specialisation (adding, overriding, blocking inherited)		No	Yes (adding, overriding, blocking inherited?)
Polymorphism (interface, protocol, by hand)		No	Yes (interface, protocol)
Type Checking (static/ strong, dynamic/weak)		Static/strong. Dynamic through protocol	⇐
Message Binding (early/compile-time, late/run-time/message dispatching)		Early, some run-time. (Read as "channel connection")	⇐
Composition (**has a**)	Also called *aggregation*	Yes. Compile a process inside a process	⇐
Containment (**holds a**)	Typically hash table or container	No. Cannot send a process to a process	⇐
Association (**knows about**)		Through channels	⇐

Table 2

Again, concurrency is not in [21].

I have discussed some of these concepts here, but often it felt like comparing apples and bananas and discuss which taste best. We need a similar (to table 2) taxonomy of *concurrent processes* first and for *concurrent "OO" processes* next.

Acknowledgements

I would like to thank Claude Petitpierre of École Polytechnique Fédérale de Lausanne, Switzerland, on leave at the Indian Instutute of Technology, Delhi, for reading a draft of this paper, and helping with valuable comments. He has designed a very interesting CSP-based object-oriented language called sC++ [22]. I would also like to thank Åge Stien of KMSS-SC for his valuable comments.

References

[1] Allen Holub, "If I were king: A proposal for fixing the Java programming language's threading problems." October 2000. At: www-4.ibm.com/software/developer/library/j-king.html?dwzone=java

[2] "occam2 Reference Manual". INMOS Ltd, Prentice Hall, 1988 (C.A.R. Hoare is series editor) ISBN 0-13-629312-3. Also see occam archive at archive.comlab.ox.ac.uk/occam.html.

[3] P.Wegner: "Dimensions of Object-Based language design". In *Proc.of the OOPSLA ´87 Conf. on Object-Oriented Programming Systems, Languages and Applications*, 1987.

[4] Joseph Bergin Pace University, "What IS Object-Oriented Programming--Really?", 1997. At: csis.pace.edu/~bergin/papers/oop.html

[5] C.A.R. Hoare, "Communicating Sequential Processes". Prentice Hall, 1985. Also see the CSP archive at archive.comlab.ox.ac.uk/csp.html

[6] Kuhn, T. S. (1996), "The structure of scientific revolutions" (3rd ed.). Chicago: University of Chicago Press.

[7] Øyvind Teig, Kongsberg Maritime Ship Systems, Ship Control, "Mission impossible? Encapsulate that aliased alien! Between need and bleed: Aliasing in computer languages". At: www.autronica.no/pub/tech/rd/PublicationList.htm

[8] John Hogg et.al., University of Toronto, "The Geneva Convention On The Treatment of Object Aliasing", August 1991. At: g.cs.oswego.edu/dl/aliasing/aliasing.html

[9] F.R.M. Barnes and P.H.Welch, "Mobile Data Types for Communicating Processes", CSREA Press, June 2001. The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)

[10] Rajeshwari SusaiMichael, "Inheritance Anomaly in OOCP Laguages". At: www.engr.csufresno.edu/Personal/CSci/Students/Grad/Rajeshwari_SusaiMicheal/ia.html

[11] Jose Meseguer, "Solving the Inheritance Anomaly in Concurrent Object-Oriented Programming", p. 220-246, ISSN 0302-9743 In: Nierstrasz, O. M. (ed.); ECOOP '93 - Object-Oriented Programming. Proceedings.; p.1-3; ISBN 3-540-57120-5; Berlin, Heidelberg, New York, etc.: Springer Verlag (1993).

[12] Walt Disney and "plussing" on the Disney collectables web site www.wdcc.net/making/plussing.htm.

[13] Ted Pattison, "Understanding Interface-based Programming", Microsoft Corporation. At: msdn.microsoft.com/library/techart/ifacebased.htm

[14] Geoff Barrett, "occam 3 reference manual", Inmos. Available at wotug.ukc.ac.uk/parallel/occam/documentation/. Occam 3 was never implemented. However, occam 2.1 was implemented in the Inmos D7405 occam Toolset (mainly data types was added). The occam compiler "KRoC" is based on this, and SPoC occam->C compiler has limited 2.1 support.

[15] Paul R. Wilson, "An Introduction to Scheme and its Implementation", University of Texas at Austin. At: www.cs.utexas.edu/users/wilson/schintro/schintro_133.html

[16] Øyvind Teig, "CHANnels to deliver memory? MOBILE structures and ALTing over memory?", submutted to CPA 2001.

[17] Ruby on the web: www.pragmaticprogrammer.com/ruby/links.html

[18] Tom Christiansen, Perl OO Tutorial. see "Closures as Objects" in elib.cs.berkeley.edu/~loretta/perl/nmanual/pod/perltoot/Closures_as_Objects.html

[19] Geoff Barrett, "The Development of Occam: Types, Classes and Sharing". In: *Real-Time Systems with Transputers, Proceedings of the 13th occam User Group*, York, 1990. H.S.M. Zedan (ed.) ISNB 90-5199-041-3

[20] Williams, Kindel, COM: The Component Object Model: A Technical Overview", Microsoft, October 1994. At: msdn.microsoft.com/library/techart/msdn_comppr.htm

[21] Bruce F. Webster, "Pitfalls of Object-Oriented Development", M&T Books, 1995, ISBN 1-55851-397

[22] Claude Petitpierre, "Synchronous C++: A Language for Interactive Applications", *IEEE Computer, Sept98*. Also see ltiwww.epfl.ch/sCxx/sC++_toc.html.
Also see their *Synchronous Java* language.

Øyvind Teig is Senior Development Engineer at *Kongsberg Maritime Ship Systems, Ship Control*. He has worked with embedded systems for 25 years, and is especially interested in real-time language issues. See http://home.no.net/oyvteig/ for publications.