# A Modular Language for Concurrent Programming

Peter Grogono
Concordia University
Montreal, Quebec, Canada
grogono@cse.concordia.ca

Brian Shearing
The Software Factory, UK
ShearingBH@aol.com

## ABSTRACT

How will programmers respond to the long-promised concurrency revolution, which now appears both inevitable and imminent? One common answer is "by adding threads to objects". This paper presents an alternative answer that we believe will reduce rather than add complexity to the software of the future. Building on the ideas of an earlier generation, we propose a modern programming language based on message passing. A module cannot invoke a method in another module, but can only send data to it. Modules may be constructed from other modules, thus permitting processes within processes. Our goal is to provide the flexibility and expressiveness of concurrent programming while limiting, as much as possible, the complexity caused by nondeterminism.

The principle innovations reported in the paper derive from bringing together ideas—some well known, but others almost forgotten—found in the historical software literature, and combining these ideas to solve problems facing modern software developers. In addition, at least one idea reported here appears to be novel, namely the introduction of an interface hierarchy based not on data elements or methods, but on *path expressions*, on the actual flow of control within a module. It is more natural to classify components of a process-oriented system by control flow rather than data content.

Another novel feature is the integration of unit tests into the source of each component, thus reducing the possibilities for testing to get out of step with development.

## 1. INTRODUCTION

For many years, we have been reading about the "software crisis". During the same period, we have been reading about new ways of solving the software crisis, often put forward by the very people who complain about it. Whether there is in fact a "crisis" in the sense of a particular hurdle that must be cleared is perhaps questionable [13]. However, there is clearly a *software gap* between what can easily be developed and what is actually needed. Software engineers are always building at the limit of, or sometimes just beyond, their competence.

Programs tend to grow incrementally. As they grow, they become harder to maintain. The simple elegance of the original architecture is engulfed by hastily-added accretions. Programming languages evolve in a similar way. From simple beginnings, they grow into complex monsters. Like old and brittle programs, they must eventually be scrapped.

The popular languages in current use are showing signs of their age. Each change leads to greater complexity, and further changes are announced at regular intervals. We propose an alternative path.

## 2. WHAT WILL WE NEED?

Each generation of systems makes greater demands on software development techniques than its predecessors. Below, we list a few of the areas in which software developers are working at the limit of software development technology and urgently need better techniques. The discussion is framed in terms of the programming languages that are currently used for application development such as Java and C++. The arguments do not necessarily apply to areas for which specialized tools have already been developed, such as operating systems, databases, telecommunications, and aerospace.

Programming has reached a level of abstraction at which programmers are not concerned with details of implementation such as bus width of memory, sector and track sizes of disks, or the instruction set of the processor. Memory management is on the cusp, automatic in many languages but still manual in a few. Future programming languages should free programmers from concerns with distributing processes over memory spaces, communication between processors, and low-level management of concurrency.

As in any engineering discipline, trade-offs are necessary: an acceptable system must be efficient, reliable, and affordable. In software, this means that resources such as memory, disks, communication channels, and displays must be managed by appropriate, well-designed algorithms. Future programmers will expect automatic management to be applied not only to memory and disk access, but to higher level aspects of systems such as load balancing, multiplexing, and network topology.

Software development tools have evolved from unadorned compilers and linkers to full-fledged IDEs. Today's compiler knows more about the code than any other tool and most of the developers. It is wasteful to use the compiler only to detect obvious errors and generate code. Future compilers will assist developers in refactoring, documenting, and testing.

Testing is recognized as an essential part of modern software development. Process models such as XP require tests to be written before code. Automated unit testing has become popular. But application code and test code are still separate entities. The programmer who, rushing to meet a deadline, has time to write either application code or test code will write the application code and skip the tests. In future, tests will be an integral part of the code, like assertions and comments, making them harder to omit.

Since software is a major investment, there is a strong incentive to keep it running for as long as possible (and sometimes a little longer). As time passes, things change: requirements, hardware, and deployment. The software must adapt to the changes, by enhancement, when new features are needed, or by refactoring, when the functionality does not change but the environment does. Enhancement and refactoring are both difficult in current systems. A project as simple in principle as replacing fat clients by thin clients, or multiplexing a component to handle increased loads, may founder as time, money, or expertise run out. Future languages will be designed specifically to support enhancement and refactoring.

There was a time when a program started, ran for a while, and stopped. No longer: most applications today run continuously, with downtime of the order of minutes per year, if that. For future systems, incorporating changes without interrupting service will be at least highly desirable and often mandatory.

In summary, we expect the compiler of the future to accept directives such as these:

- *Compile package X in its own address space*
- *Compile Y as a remote server using CORBA for communication*
- *Compile Z as middleware with multiplexing and load-balancing*
- *Compile and execute the system tests*
- *Show me the structure of system component K*
- *Show me the dataflows that occur when P executes*

We also expect that development environments will improve to match the capabilities of the compiler.

These are some of the challenges facing the next generation of software development. Can we meet them?

## 3. WHAT'S WRONG WITH OBJECTS?

The concept of modules, and the criteria for making them reusable, were introduced by McIlroy [26] and Parnas [30]. Objects were advertised as having the required characteristics. Alan Kay, the inventor of Smalltalk, said that after reading Simula code, no procedural program would ever look the same [20]. He stated the principle that an object with data and code was a recursive decomposition of a computer, thus taking software one step towards scale-free organization.

Today, the prevailing paradigm is object-oriented programming (OOP) or, more precisely, OOP with a sprinkling of concurrency. OOP has served us well; the transition to OOP that occurred during the 80s and 90s enabled software that would never have got off the ground with earlier paradigms. Consider our earlier criteria of enhancement and refactoring. Companies often proffer a portfolio of products that are variations on a theme; such as the accounts offered by a bank. If software to support products is organised in an OOP manner, with an abstraction named *Product* as supertype, then adding a new product can be done simply and with little or no disruption of software for existing products. Enhancement of this kind is well handled by OOP. It is also the case that refactoring the methods of an inheritance tree can often be a rewarding process, yielding simplifications, and providing the basis for further enhancements in the future. However, these examples of enhancement and refactoring relate to programming in-the-small. OOP is of little help in tackling major enhancement, and a positive barrier to refactoring in-the-large. Because code often relates to its class, and sometimes to its super-classes, in intimate ways, changing one class structure into another can be tantamount to a rewrite rather than a refactoring.

In a historical context, it is clear that OOP was a natural and important step forward, just as structured programming was for an earlier generation of programmers [11]. It is also clear that for the complex requirements posed by today's requirements of flexibility and scale, OOP is running out of steam. It has inherent weaknesses that will not be fixed by patching. Symptoms of these weaknesses are obvious: although inheritance is a cornerstone of OOP, it has no clear, uncontroversial definition; `Java` has no less than fourteen ways of controlling access to variables;[1] the inability to handle cross-cutting concerns has led to aspect-oriented programming, adding a further layer of complexity.

Objects are passive. They just sit there, waiting to be told what to do next. An object with several methods has no control over the order in which the methods are invoked and cannot even prevent interleaving. Suppose that a method in an object `myObject` executes the code

```
x = 5;
yourObject.doIt();
// What is the value of x?
```

in which `x` is a private instance variable of `myObject` and `yourObject` is another object. It is possible (and, in a large system, not unlikely) that `doIt` could invoke another method in `myObject`, perhaps changing the value of `x`. Objects have no control over their own state!

[1] C++ is probably no better: it's just harder to count the ways.

The interface of an object is usually identified as its set of public methods. But this is only part of the story. If any method invokes `new`, the object requires a memory manager. If a method raises an exception, the object requires a handler. If a method sends a message, there must be another object ready to receive the message. The object may need the support of other objects that already exist or that it creates itself. Thus, an object may have complex interactions with other parts of the system that are not revealed by its interface.

Methods do not scale well. Popular advice is to keep methods short, but the practical effect of many short methods is just to distribute complexity through the system.

The response of the OO world to the need for concurrency has been to add threads to objects. But as Lee observes [23]:

> We must and can build concurrent computation models that are far more deterministic, and we must judiciously introduce nondeterminism where needed. . . . Threads take the opposite approach. They make programs absurdly nondeterministic and rely on a programming style to constrain that nondeterminism to achieve deterministic aims.

It is not surprising that novice programmers are taught that functions and methods are easy and natural but processes are difficult and unnatural. The reckless programmer who tinkers with concurrency will be punished with deadlock, race conditions, and non-reproducible bugs.

Perhaps the best-known software engineering mantra is "low coupling, high cohesion". Objects cannot enable low coupling because the glue between them is too strong. Adding concurrency just strengthens the glue.

## 4. LESSONS FROM THE PAST

The history of software development has many paths not taken. Often, this is a good thing: the chosen paths were often better and the alternative paths were abandoned for good reasons. Occasionally, however, a promising approach is abandoned for reasons that either became invalid or were invalid in the first place.

Such is the story of concurrency. Pioneer programmers such as Dijkstra, Brinch Hansen, and Hoare, working in the 60s, were inevitably involved in the implementation of operating systems. They recognized that concurrency or, more precisely, processor multiplexing, had to be done properly. They developed sound methods for reasoning about concurrency and algorithms to match. But "a subsequent generation has lost that understanding" [17].

Coroutines, introduced by Conway [10] and used in Simula [29] and Smalltalk [20], demonstrated early recognition of the advantages of processes over procedures. Since most programs at this time were executed on single processors, true concurrency with its accompanying problems was obviously not required. Coroutines provided an effective way of simplifying program structure without sacrificing efficiency. But coroutines have never become popular.

In fact, processes are in many respects easier to work with than procedures. Hoare showed this by example in CSP [15, 16]. Jackson advocated modeling systems as processes and then transforming the models into procedural implementations [18, 19]. The reason that Jackson's method required transformation was efficiency. When he proposed his methodology, process switching was slower than procedure invocation by orders of magnitude. Although hardware designers have worked hard to speed procedure invocation but have done little to speed up process switching, we now know how to change contexts quickly; systems with thousands of processes are now feasible [2, 34].

There are a few fundamental problems, such as those involving producers, filters, and consumers, for which the process model is obviously simpler than the procedural model. Consider a slightly harder problem, such as coding a tree iterator. A procedural implementation must save state between calls. The state will probably be represented as a stack of pointers to tree nodes, simulating the processor's own, more efficient, stack. A process implementation, on the other hand, is a straightforward tree traversal with a 'send' as each node is visited. A compiler provides a more complex example. In the procedural implementation, each component must be designed in a specific way to communicate with other components: the scanner must provide one token each time it is called, and so on. In the process implementation, each component is a process that can be coded in any way that is compatible with its input and output requirements but is independent of the overall organization of the compiler. Finally, consider GUI programming. The rat's nest of method registration and callbacks can be replaced by simple processes that wait for external events and send messages to appropriate modules when they arrive.

Experienced UNIX$^{TM}$ programmers are, of course, familiar with the advantages of processes. They write small programs, with input and output statements positioned naturally in accordance with the underlying logic, and connect these programs with pipes. The programs can then be used in a variety of ways to build useful application [21]. They are, in fact, reusable components.

Specialized areas of software development have always provided features that have not become part of mainstream application programming. Operating systems must run multiple applications, giving each application the resources it needs, preventing it from accessing resources it does not own, avoiding deadlock, and doing all of this efficiently. Databases must provide rapid access to large amounts of data to many clients simultaneously, while maintaining transaction integrity and surviving system failures. Telecommunications software must provide high levels of concurrency and reliability with minimal downtime. A modern programming language should incorporate techniques from software developed for specialized applications.

The knowledge is there; we just need to tap into it.

# 5. WHAT COMES NEXT?

As Jackson and others recognized, modeling with processes is more natural, for many applications, than modeling with procedures. Kay saw objects as small computers [20], but processes make even better small computers. As well as data and code, a process has a life of its own. Implementation with processes does not have to be significantly less efficient than implementation with procedures. We expect that the next programming paradigm will be based on processes.

Accordingly, the goal of our research is a language and supporting tools, all based on concurrent processes. The provisional name for the system is Erasmus.[2]

An Erasmus module is called a *cell*, a term with intentionally biological nuance. A cell is an instantiation of a process. A cell is to a process as an object is to a class. A process may be strictly sequential, or may be a composition of other processes. When instantiated, a cell whose text is composite has the appearance of a container of nested cells, all of which are executing independently and in parallel.

Multiple threads are permitted within a cell, but not true concurrency. A cell executes in a single processor and does not need true concurrency. Thus cells have a single thread of control but provide a form of coroutine. A single thread of control significantly simplifies cell design and implementation. A typical cell runs a main process and several daemons, all accessing shared data. Processes and daemons have lower coupling than regular coroutines because they swap control by exchanging signals and messages rather than by the conventional `suspend/resume` mechanism. See Section 5.9 for more on this feature of the Erasmus model.

Cells communicate solely by exchanging data. This gives cells more autonomy than objects, because each cell is in full control of its actions and state. The Erasmus language is precise about the semantics of cells but somewhat vague about their deployment. A program may state that cell $C$ communicates with cell $C'$, specify the type of data, the order of transfer, and so on. But the means of communication—direct memory copy, transfer across partition boundaries, LAN, WAN, or general internet protocol—is not decided until later; perhaps at link time or even at run-time.

## 5.1 Data Transfer

Programs spend much of their time moving data about. Erasmus assigns meanings to three common types of transfer: copy, move, and alias.[3] These meanings are independent of the physical nature of the transfer, which might be an assignment in a single memory space, a communication between continents, or anything in between.

We describe the transfer of data from a source *src* to a destination *dst*. We suppose that there is an object of some kind at *src* initially. After the transfer, there are three possibilities, depending on the transfer mode:

**Copy:** there are two distinct objects, the original at *src* and a copy of it at *dst*.

**Move:** there is one object that is accessible at *dst* but not at *src*.

**Alias:** there is one object that is accessible from both *src* and *dst*.

Data transfer requires a sending process and a receiving process. (These processes are not necessarily distinct.) The sending process executes $p\,!\,mode\ src$ to send data *src* using port $p$, where *mode* is one of `copy`, `move`, or `alias`. The receiving process executes $p\,?\,dst$ to receive data using port $p$.

The sending and receiving processes must agree to communicate at a particular time. The agreement results in a *rendezvous*;[4] during the rendezvous, data are transferred from sender to receiver. If the sending and receiving processes share the same address space the rendezvous reduces to a conventional assignment. In the general case, the first process to reach the rendezvous blocks until the other process is ready. Basic transfers of this kind are *unbuffered*. We use unbuffered transfers as the primitive operation, as in CSP [16], because it is easy to define buffered transfer using unbuffered primitives.

Communication semantics are defined in terms of several basic operations that are outlined informally below.

In defining the semantics of data transfer operations, we assume the most general case: *src* and *dst* are on different processors with different architectures. In order to effect the transfer, the data at *src* is first *marshalled* into an architecture-independent format, sent over a network, and then *unmarshalled*.

Formally, there is a generic *marshalling function* parameterized by the type $T$ of the object to be transferred

$$\mathcal{M}(T) \quad : \quad T\backslash S \rightarrow Secret$$

where $T\backslash S$ is the type $T$ without its transient fields, and an inverse *unmarshalling function*

$$\mathcal{M}^{-1}(T) \quad : \quad Secret \rightarrow T.$$

The actual moving of data across the network is called *transporting*. Since data are always transported in marshalled form, the only values that can be transported are those of type *Secret*. We write

```
transport x to y
```

to indicate that data of type *Secret* is transported from $x$ at *src* to $y$ at *dst*.

The third kind of data transfer, aliasing, requires that only a single object, or the illusion of a single object, exists after the transfer. The operation that achieves this is named *synch*. The effect of

[2] Desiderius Gerhard Erasmus, 1466–1536. "The fox has many tricks. The hedgehog has but one. But that is the best of all."

[3] Java distinguishes different kinds of aliasing, such as hard, weak, and soft. Eventually, Erasmus may also recognize these distinctions.

[4] The term is from Ada: [1, §9.5.2].

`synch` $x$ `with` $y$

is to maintain objects $x$ and $y$ in a state of mutual equality. This effect can be achieved in various ways. For example, by making $x$ and $y$ references to the same memory location or by maintaining two copies of the object in different memory spaces and notifying one copy whenever the other copy is updated. Synchronization is transitive and, in the general case, an arbitrary number of objects may have to be maintained in the same state.

We use $\leftarrow$ to denote a primitive assignment operator. The statement $x \leftarrow e$ binds the name $x$ to the value of the expression $e$.

When an object is moved from *src* to *dst*, it is no longer accessible at *src*. Also, if a variable $v$ refers to an object $o$, and is changed to reference another object $o'$, then $o$ is no longer accessible via $v$. In both cases, an access path to an object is lost and the object becomes inaccessible if there are no other access paths to it. We indicate that the access path $v$ is lost by writing

`dispose` $v$

Processes communicate by matching a send operation $p\,!\,mode\ src$ from a port $p$ in one process to a receive operation $p\,?\,dst$ in another process. We write the communication as

$$p\,!\,mode\ src \quad \longrightarrow \quad p\,?\,dst$$

in which *mode* indicates the protocol, `copy`, `move`, or `alias`.

The fundamental operation, $p\,!\,$`copy` $src \longrightarrow p\,?\,dst$, is performed in four steps as follows:

| | Source | Network | Destination |
|---|---|---|---|
| 1. | $x \leftarrow \mathcal{M}(src)$ | | |
| 2. | | transport $x$ to $y$ | |
| 3. | | | dispose *dst* |
| 4. | | | $dst \leftarrow \mathcal{M}^{-1}(y)$ |

The `move` and `alias` operations are defined in terms of `copy`; Figure 1 illustrates the effect of each operation. The dashed line in `alias` mode represents synchronization.

$p\,!$`move` *src* $\longrightarrow p\,?$ *dst* $\equiv$
    $p\,!$`copy` *src* $\longrightarrow p\,?$ *dst*
    `dispose` *src*

$p\,!$`alias` *src* $\longrightarrow p\,?$ *dst* $\equiv$
    $p\,!$`copy` *src* $\longrightarrow p\,?$ *dst*
    `synch` *src* `with` *dst*

These definitions assume that *src* is the name of (i.e., reference to) an object. If *src* is an expression *exp*, we evaluate the expression, store the result in a temporary location, treat that as the source of the operation, and dispose the temporary after the transfer. For each of the modes above:

$p\,!$ *mode exp* $\longrightarrow p\,?$ *dst* $\equiv$
    *tmp* $\leftarrow$ `eval`(*exp*)
    $p\,!$ *mode*(*tmp*) $\longrightarrow p\,?$ *dst*
    `dispose` *tmp*

Assignment is written *dst* := *mode src* and defined in terms of the data transfer operations by:

*dst* := *mode src* $\equiv$
    $p$ : *port*($T$)
    $p\,!$ *mode src* $\longrightarrow p\,?$ *dst*
    `dispose` $p$

where *mode* is one of `copy`, `move`, or `alias`, and $T$ is the type of *src*, and the declaration $p$ : *port*($T$) introduces a new port capable of transferring instances of $T$.

If *exp* is an expression:

*dst* := *exp* $\equiv$
    *tmp* $\leftarrow$ `eval`(*exp*)
    *dst* := *move tmp*

As an example of how these definitions work together, the result of expanding the simple assignment *dst* := *exp* is:

    *tmp* $\leftarrow$ `eval`(*exp*)
    $p$ : *port*($T$)
    $p\,!$`copy` *tmp* $\longrightarrow p\,?$ *dst*
    `dispose` *tmp*
    `dispose` $p$

In practice, the compiler generates the best code for the particular transfer, which could be anything from a simple memory move to full internet protocol. In order to do this, it needs information in addition to the source code. Current languages provide such information by means of pragmas, but pragmas tend to be embedded in source code. An Erasmus compiler will read both source code and, from an independent stream, directives indicating how the processes are to be deployed. Looking further ahead, we expect that these decisions will often be made dynamically, by a just-in-time (JIT) compiler generating code for the current configuration.

## 5.2 Types and Typestates

The type system of Erasmus is based on standard principles. There is a conventional set of basic types and a collection of rules for constructing other types, such as arrays and records from them. User-defined types may be parameterized. There is a relation on types that respects Liskov's "behavioral subtyping" [24]. One application of subtypes is to data transfer, where an object of type $S$ can be sent through a port of type $T$ provided that $S$ is a subtype of $T$.

Following Hermes [32], Erasmus performs typestate analysis. The *typestate* of a variable describes both its type and its state: uninitialized, initialized, or (for complex types) perhaps partially initialized. A declaration introduces an uninitialized variable. Each operation has an implicit precondition specifying the typestate it requires and a postcondition specifying the typestate it ensures. The compiler uses flow-analysis to check the preconditions and postconditions statically and to insert run-time checks when necessary.

## 5.3 Aliasing

In earlier times programmers were encouraged to employ aliasing wherever possible 'in the interests of efficiency.' We were taught that a parameter of a subroutine that was a string or an array should be passed by reference rather than

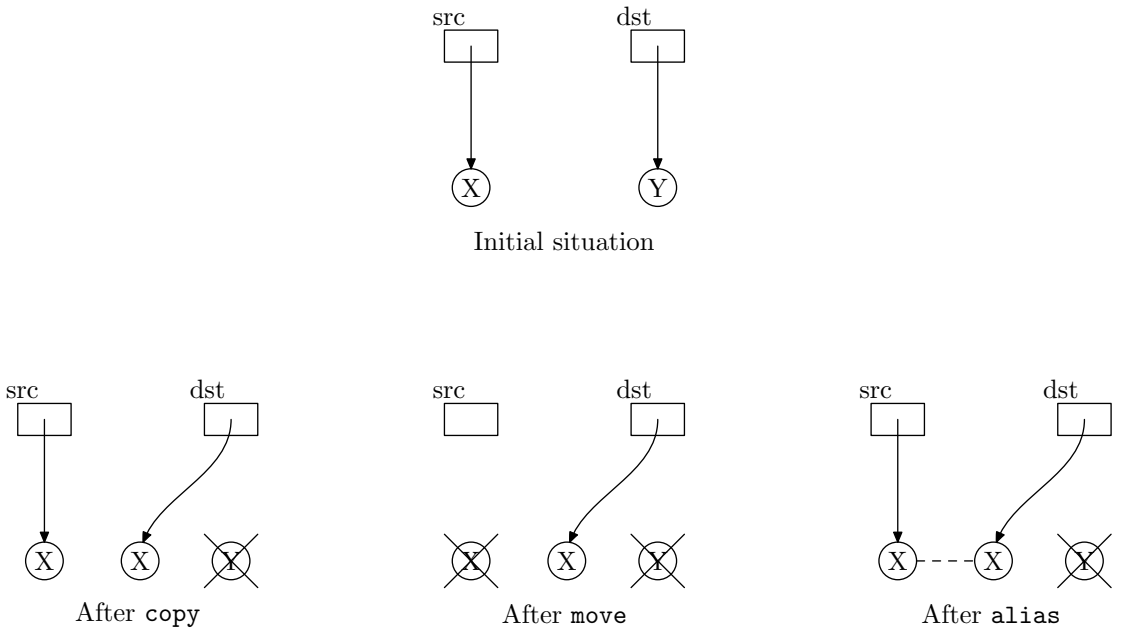


**Figure 1: Transfer operations**

by value 'to avoid copying', even though the parameter might logically be a value not a reference. Object-oriented programming has placed this style of programming at its heart; almost everything is passed as an object pointer, and shared by client and service. Yet aliasing is a mixed blessing. Apart from the danger of corruption by a rogue client the claims to efficiency can sometimes be misplaced, for example, when the time taken for collecting garbage is considered. Indeed, if it were possible to devise a style of programming that avoided aliasing altogether, the heap could be managed by calls injected by the compiler, and the garbage collector would be redundant. But of course, variables do need to be aliased on occasion. Shared data is at the heart of most business applications, and we must make provision for it. Nevertheless, Erasmus discourages the use of aliasing except where the business model mandates it. Copying and moving are the preferred modes of data transfer.

The policy of discouraging aliasing might appear to impose an undue burden of inefficiency on situations such as passing large arrays as parameters, but this is not necessarily so. The typestate analysis discussed in Section 5.2 above is usually capable of detecting when a large structure can safely be passed by reference rather than copied. It is time to take decisions such as these out of the hands of humans and leave them to an automaton.

Aliasing in the presence of processes does however pose a new problem, that of introducing the possibility of race conditions and deadlocks. In Joyce [8] shared variables are simply not allowed. We take a more pragmatic approach, and permit aliased variables but insist that the programmer declare aliasing clearly. One mechanism for avoiding real-time bugs is for the compiler to associate a lock with every aliased variable, and to arrange that if a process refers to aliased variables then it most acquire the locks of each. In effect, processes that refer to aliased variables become monitors. This strategy causes locks to be held for longer than manual locking might achieve, as in [14] for example, but the approach is safe, and follows our policy of applying gentle disincentives to aliasing.

## 5.4 Capabilities

When a cell is created, it is provided with the capabilities that it needs to perform its tasks. These capabilities allow the cell to access a particular region of memory, communicate with specified peripherals, send and receive data through typed ports, and so on. Once started, a cell can only communicate with other cells; it cannot acquire new resources directly, but may receive further capabilities via its ports.

A simple Erasmus program might start like this:[5]

```
type MyProtocol = ....
type MyCellType =
[
    main: process(init: port(MyProtocol)) =
        var caps: MyProtocol;
        init ? caps;
    ....
]
```

In this example, the type `MyProtocol` defines the type of an object that can represent the capabilities needed by a cell.

[5]This example uses the syntax of the prototype language. Later versions of the language will provide syntactic sugar to sweeten this kind of code.

`MyCellType` is the type of some kind of cell that will be used in the program. Each cell contains a process called `main` that is the principal process of the cell. This process has a parameter that is used by the cell to read the capabilities into the variable `caps`. The cell does not have access to any resources other than the ones in `caps`.

Another cell can create an instance of `MyCellType` by executing

```
var caps: MyProtocol = ....;
new MyCellType ! caps;
....
```

The `new` statement creates a new cell but does not introduce a variable denoting the new cell. If the creating cell wants to communicate with the new cell, it must include a suitable port amongst the capabilities. The creating cell will usually do this but, in principle, it might pass only capabilities for communicating with other cells, and have no way of communicating directly with the new cell after creating it. In this way, unnecessary coupling is avoided.

## 5.5 Scale-Free Structure

Most current programming language have a fixed hierarchy of structure. The first level consists of expressions and statements. These are grouped into named functions, procedures, or methods. These two levels are shared by most languages. At the third level, the way in which functions are grouped, into modules or classes, tends to characterize the language. At higher levels, there is even less consistency but we find, for example, packages in Ada and Java and clusters in Eiffel [27]. The hierarchy makes it easier to design small programs than large programs. Small program components, up to the class level, receive all the benefits of the structuring features of the language. Large program units have no such benefits, although patterns such as Façade [12] can help.

In principle, we would like to have a language in which programs of all sizes look similar: in modern jargon, *scale-free programming*. This is harder to do than it seems at first sight. Curiously, one of the earliest languages was scale-free: in Algol 60, the principal structuring feature was the *Algol block*, which contained statements and functions, and which could be nested. The freedom from scale of Algol programs, however, is technical but not practical. Separate compilation, for example, is infeasible.

Erasmus would be scale-free if cells could be nested. However, we have rejected this approach for several reasons. First, it seems to lead to the same problems as Algol's nested blocks and nested classes in more recent languages. Second, it destroys the concept of a cell as a simple unit with a single process and controlled access to its environment. Instead, we say that an Erasmus program of any size, or any part of a program, is a collection of communicating cells. The large-scale organization of a program is determined by the patterns of communication. Extending the biological analogy, groups of cells are like the organs of a body.

Section 5.2 above discusses the general principles of types in Erasmus. The following sections illustrate the three concrete kinds of type that we employ to build a system. We dub the types *message*, *interface*, and *process*. We reiterate that no significance should be attached to the concrete syntax of the examples below. The style is chosen to expose underlying mechanisms rather than for convenience of expression. Choice of a concrete syntax will be take place only once the underlying model has stabilised.

## 5.6 Messages

A message is a data structure that passes from one process to another. Messages are defined in an hierarchy of types much as classes are defined in an object-oriented system. The following example defines a message employed to initialise a process that is a *filter*—which is a process that reads from one port and writes to another. The initialisation method passes the ports to the process. In this example the type of the items of data that pass through the filter is integer.

```
message FilterInit = [
    producer, consumer: port(int)
];
```

As with other kinds of type in Erasmus, messages can be generic. The following example illustrates a version of the filter initialisation message that can be used for filters of any type.

```
message FilterInit(T) = [
    producer, consumer: port(T)
];
```

Like objects, messages can have methods as well as data. The following example adds methods *read* and *write* to the filter initialisation message.

```
message FilterInit(T) = [
    producer, consumer: port(T);
    function read(): T =
        var v: T;
        producer ? v;
        producer ! ack;
        return v
    end;
    procedure write(v: T) =
        consumer ! v
    end
];
```

The statement '`producer ! ack`' in method *read* above marks the end of the rendezvous between two communicating processes.

Various things can go wrong during a rendezvous, and the possible failures can be categorised as follows:

- genuine hardware failure;
- failure of communication protocol;
- violation of a process protocol (See Section 5.7 below);
- violation of assertion; or
- explicit return of a `nak` under program control.

In the Erasmus model the signalling of an acknowledgment, an '`ack`', or of a '`nak`', is explicit. This provides a unified framework for raising and handling communication failure, whatever its nature. See Section 5.8 below for examples of `ack`s and `nak`s, and of how the programmer can respond to a `nak`.

## 5.7 Interfaces and Protocols

In a large system, or library, many processes will have similar overall behaviour. For example there will be many processes that act as filters. We capture this commonality by stating that a process satisfies the definition of a named *interface*. Unlike an interface in a language such as Java, an Erasmus interface is not defined in terms of its methods but in terms of its ports. For example, a filter process has three ports: an initialisation port, a port to read values from, and a port to write filtered values to. The following example show how we might declare Interface *Filter*, which is generic in Type *T*.

```
interface Filter(T) = [
    init: port(FilterInit(T));
    inp, out: port(T)
]
```

As well as having a signature defined by its ports, an interface has a *protocol*, which is a *path expression* defining the allowable sequence in which messages may be read from or written to its ports. The following elaboration of Interface *Filter* declares that its sequence is an initialisation followed by an indefinite alternation of reads and writes.

```
interface Filter(T) = [
    init: port(FilterInit(T));
    inp, out: port(T);
    protocol init?, (inp?, out!)*
]
```

A path expression defines a *Finite State Machine* against which the compiler can check that only valid sequences occur. The compiler will often be able to make such checks statically, but even when it is necessary to generate code to carry out the checks during execution, the number of machine instructions is but one or two for each read or write.

The strength of an interface can be further enhanced by attributing pre- and post-conditions to its ports, as in the following:

```
interface Filter(T) = [
    init: port(FilterInit(T));
        pre "Producer defined":
            init.producer ≠ null;
        post "Consumer defined":
            init.consumer ≠ null;
    inp: port(x: T);
        pre "Given value defined":
            x ≠ null;
    out: port(y: T);
        post "Filtered value defined":
            y ≠ null;
    protocol init?, (inp?, out!)*
]
```

Interfaces form an inheritance hierarchy based on their protocols. The following example defines Interface *Bifilter*, which acts like Interface *Filter* except that it reads two items for every one that it writes.

```
interface Bifilter(T) extends Filter(T) = [
    protocol init?, (inp?, inp?, out!)*
]
```

Figure 2 illustrates inheritance of interfaces. The supertype of all interfaces is the do-nothing process defined by Interface *Stop*.

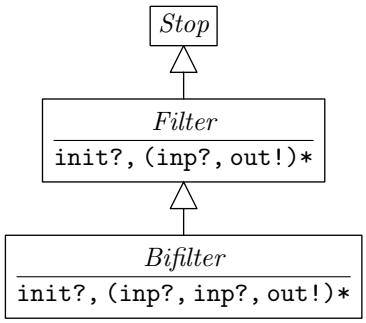


**Figure 2: Inheritance of Protocol-based Interfaces**

## 5.8 Processes

With an initialisation message defined, and the signature and protocol of an interface to satisfy, we can write a process. The following is a filter named *Inc* that reads a stream of integers and writes each integer, incremented by unity.

```
process Inc implements Filter(int) =
    var parms: FilterInit(int); init ? parms;
    out :=alias parms.consumer;
    connect inp to parms.producer;
    init ! ack;
    loop
        var v: int; inp ? v; inp ! ack;
        out ! v+1
    end
end;
```

Alternatively we could employ methods *read* and *write* defined in Message *FilterInit* to write the loop of the process as follows.

```
process Inc implements Filter(int) =
    ...
    loop
        var f: FilterInit(int);
        f = FilterInit(int)(inp, out);
        f.write(f.read() + 1)
    end
end;
```

Pre- and post-conditions can be included in processes as well as in interfaces. Figure 3 is an example of a Bifilter named *Sum* that reads consecutive pairs of integers and writes their sum. It includes the artificial constraint that the range of values that it can process is limited to three digit values.

```
process Sum implements Bifilter(int) =
    pre "Only three-digit values allowed":
        0 ≤ inp.x < 1000;
    pre "Only three-digit result allowed":
        0 ≤ out.y < 1000;
    var parms: FilterInit(int); init ? parms;
    out :=alias parms.consumer;
    connect inp to parms.producer;
    init ! ack;
    loop
        var u: int; inp ? u; inp ! ack;
        var v: int; inp ? v; inp ! ack;
        out ! u+v
    end
end;
```

**Figure 3: Process with pre- and post-conditions**

Although trying to add 600 to 700 would cause an exception—as it should—the failure would not necessarily manifest itself in the most convenient manner, as this check is a post-condition on the send to the output channel. A thoughtful programmer might choose to arrange that the input be faulted rather than the output. Leaving the existing post-condition in place—in the manner of belt and braces—Figure 4 is a refinement of Figure 3 that might be considered more friendly.

```
process Sum implements Bifilter(int) =
    pre "Only three-digit values allowed":
        0 ≤ inp.x < 1000;
    pre "Only three-digit result allowed":
        0 ≤ out.y < 1000;
    var parms: FilterInit(int); init ? parms;
    out :=alias parms.consumer;
    connect inp to parms.producer;
    init ! ack;
    loop
        var u: int; inp ? u; inp ! ack;
        var v: int; inp ? v;
        if 0 ≤ u+v < 1000 then
            inp ! ack;
            out ! u+v
        else
            inp ! nak "Result must be 3 digits"
        end
    end
end;
```

**Figure 4: Process with better error reporting**

Figure 5 shows how a process sending a value to Process *Sum* might deal with the possibility of a `nak` being returned. The `otherwise` clause is supplied with an port to an exception handling cell. The port is polymorphic, and can respond to a message requesting the reason for the exception, to a message requesting a traceback, and so on. The statement `fail(r)` is assumed to end execution for reason `r`.

Processes *Inc* and *Sum* above are simple processes. Figure 6 is a composite process named *Dinc*. Like Process *Inc*, it is a filter. Process *Dinc* bumps its inputs by two rather than by one. It achieves this not in the obvious way, but by arranging for two instances of Process *Inc* to be placed in a line.

```
var o:  port(int);
o ! 700
    otherwise(exc: port(Exception))
        var query: port(text);
        exc ! ReasonForException(query);
        var reason: text;
        query ? reason;
        fail(reason)
    end
```

**Figure 5: Responding to a `nak`**

```
process Dinc implements Filter(int) =
    var parms: FilterInit(int); init ? parms;
    out :=alias parms.consumer;
    connect inp to parms.producer;
    init ! ack;
    var q: port(int) = new port(int);
    var q: port(int); connect p to q;
    new Inc() ! FilterInit(inp, p);
    new Inc() ! FilterInit(q, out)
end;
```

**Figure 6: A Composite Process**

## 5.9 Coroutines and Aspects

One Erasmus process may implement more than one interface. Each interface provides an *aspect* of a cell's behaviour. Here are examples of aspects that an industrial-quality cell may have.

- client services: perhaps of several kinds;
- logging;
- publication of selected activities to registered listeners;
- communication with services required by the cell, such as databases or email servers;
- control:
  - start a client service;
  - stop a client service;
  - suspend a client service;
  - resume a client service;
  - replace the software for this cell with a new version;
  - move this cell to another address space, whilst retaining all active connections to other cells;
- timing and usage statistics;

- state queries; and
- documentation and meta-queries about the operation of the cell.

Each interface implemented by a process has its own protocol and hence its own flow of control. Flows associated with distinct interfaces execute independently within the cell. Conceptually they execute in parallel but in practice they share a single program counter—they are *coroutines*. This policy ensures that considerations of shared variables (see Section 5.3 above) do not give rise to race conditions and other real-time problems.

The separation of the behaviour of a cell into distinct interfaces contributes to the modularity of an Erasmus system. For example, the interface and ports concerned with control are not visible to clients of a service.

Aspects in Erasmus encompass not only the services a cell provides to its various clients but also the services that the cell itself employs. See the entry 'communication with services required by the cell' in the list of aspects above. This feature of the Erasmus model is a significant contribution to making refactoring-in-the-large a reality. It exposes, in a controlled way, not only the conventional front-end services that a module provides—often dubbed its 'API'—but also the services it depends on. We dub the hiding of services inside the implementation of a model as 'calling out the back'. It is common to the design of all modern software libraries, and we consider it to be a mistaken attempt at providing modularity. Quite the contrary, its pervasiveness is a major contribution to explaining why it is difficult to restructure large systems once their architecture has been decided. Part of the rationale for Erasmus is to try to restore the true meaning of 'information hiding'.

## 5.10 Tests

One improvement in programming practice in recent years is the incorporation into many software development shops of unit testing frameworks such as JUnit [3]. However, rather like design documents expressed in the likes of UML [6] there is a tendency for unit tests not to be maintained, especially under the pressures of product delivery. When a new parameter is added to a method to solve an immediate problem, all too often the associated test methods are not updated, the test suite fails to compile, and testing is set aside—often never to be picked up again. In Erasmus we have tried to reduce the temptation to sideline unit tests by allowing the programmer to place the code of tests not in a separate test script but right at the heart of the code. Every process may have at its head a sequence of test cases. Each test is written as a path-expression satisfying the protocol of the process—a test is a *trace* of execution of the process [15]. Figure 7 illustrates Process *Dinc* embellished with some unit tests.

The Erasmus compiler generates two executables for each process. The first corresponds to the executable code itself, and the second is a program which when run carries out the tests defined for the process. In a strict shop the executable of a process would not be published if execution of any of its tests failed. The test suites defined for Erasmus processes are less likely to be discarded than tests scripts for the likes of JUnit because they are expressed as path expressions rather than in terms of methods and arguments.

```
process Dinc implements Filter(int) =
test "Dinc minimum int OK":
    init(), inp(-2147483648), out(-2147483646);
test "Dinc zero OK":
    init(), inp(0), out(2);
test "Dinc to maximum int OK":
    init(), inp(2147483645), out(2147483647);
    var parms: FilterInit(int); init ? parms;
    ...
end;
```

**Figure 7: Unit Tests within a Process**

# 6. RELATED WORK

As Sutter and Larus point out [33], the movement towards concurrency is not new:

> Concurrency has long been touted as the "next big thing" and "the way of the future," but for the past 30 years, mainstream software development has been able to ignore it. Our parallel future has finally arrived: new machines will be parallel machines, and this will require major changes in the way we develop software.

However, most current approaches are based on adding concurrency to object-oriented programming. Sutter and Larus continue: "What we need is OO for concurrency".

The object-oriented language Eiffel was extended for concurrency by the addition of the single keyword `separate` [27, Chapter 30]. When `separate` is applied to an entity $X$, it means that $X$ "may become attached to objects handled by a different processor". When `separate` is applied to a class, it means that instances of that class will behave as if qualified by `separate`. Normal invocations of methods in an Eiffel object are synchronous; invocations of methods in a `separate` object are asynchronous.

Microsoft's C# relies on the multi-threaded environment .NET. There are various proposals for extending C# with concurrency primitives at a higher level than simple locks. At Microsoft, Benton *et al.* have introduced *asynchronous methods* and *chords* in Polyphonic C#; programs in this language are transformed into C# [4]. Active C# enhances C# with concurrency and a new model for object communication [14]. There is a new kind of class member, called an *activity* and run as a separate thread within an object. Communication is controlled by *formal dialogs*. Activities and dialogs provide an expressive notation that can be used to solve complex concurrent problems in a natural way. The full power of object orientation is retained, with concurrency superimposed.

UML 2 has responded to the call for concurrency by providing more flexibility for modeling concurrent systems than

UML 1.x. Concurrency is still modeled by forks and joins in control flows, but there is no synchronization following a fork [7]. Following the practice of architecture description languages, system components have ports and communicate via connectors [31].

Many projects do not build on existing object oriented languages. For example, Per Brinch Hansen has designed a number of languages for concurrent programming [9]. Joyce was developed as a programming language for distributed systems [8]. The program components are *agents* which exchange messages via synchronous, typed channels. The features of Joyce that distinguished it from earlier message-passing languages such as CSP were: port variables; channel alphabets; output polling; channel sharing; and recursive agents. Static checking ensured a higher degree of security than was provided by earlier concurrent languages. Our prototype language is quite similar to Joyce in concept and makes use of several of Brinch Hansen's ideas. However, Erasmus is considerably richer in features than Joyce.

The language occam-$\pi$ has a number of features that are relevant to our project [2]. First, it demonstrates the possibility of efficient execution of processes: On an 800 MHz P3 processor, context switching or communication require 70ns and process startup/shutdown time takes 20ns. Second, occam-$\pi$ provides *mobile processes*—processes that may be suspended, sent to another site, and resumed. Third, it has a formal semantics based on Milner's $\pi$-calculus [28].

Hermes is an experimental language developed at IBM but never used in production [22]. Since Hermes was designed as a system language for writing applications that might not be protected by hardware and operating system facilities, a new process is provided with the capabilities it needs and cannot obtain any more. We have adapted this idea, and also typestates (Section 5.2), for Erasmus.

The Mozart Programming System [34] is based on the language Oz, which supports "declarative programming, object-oriented programming, constraint programming, and concurrency as part of a coherent whole"[6]. Oz actually provides both message-passing and shared-state concurrency. Its designers state that message-passing concurrency is important because it is: the basic framework for multi-agent systems; the natural style for distributed systems; suitable for building highly reliable systems. It is for these very reasons that we have adopted message-passing as the foundation for concurrency in Erasmus.

The Erasmus idea of a process protocol follows the approach of treating each process of a multi-process model as a finite state machine, an approach that is developed in FSP, *Finite State Processes* [25].

Typical middleware for distributed systems either attempts to hide all of the implementation details of communication (e.g., RPC) or to require programmers to do the dirty work themselves. *Infopipes* [5] provides a different approach, wherein the various aspects of communication are reified, giving programmers a collection of abstractions from which they can build customized communication systems. Infopipes are *compositional*: the properties of a channel can be inferred systematically from the components used to build the channel. For example, the result of joining two infopipes, with latencies $t_1$ and $t_2$, in series, is an infopipe with latency $t_1 + t_2$. We hope to incorporate similar abstractions into Erasmus.

In summary, the related work provides strong evidence that the goals of the Erasmus project are feasible. We are not attempting to do anything that has not been tried before in some form. The trick is to put it all together.

# 7. CONCLUSION

We have described the first, tentative steps towards a programming language and development environment of the kind that will be needed for the next generation of software construction. Although quite different from today's languages, Erasmus builds on well-established theoretical and practical past work. The project combines what we consider to be excellent, but hitherto unexploited, ideas.

Much work remains to be done. The prototype language lacks a number of features. When these features have been added, we will be able to validate our claims. Concurrently, we will work on the development environment that will match our needs and desires as well as, we hope, those of others.

**Acknowledgments** The research described in this paper was funded in part by the Natural Sciences and Engineering Research Council of Canada.

# 8. REFERENCES

[1] Ada. Ada 95 reference manual. Revised International Standard ISO/IEC 8652:1995, 1995. www.adahome.com/rm95.

[2] F. R. Barnes and P. H. Welch. occam-$\pi$: blending the best of CSP and the $\pi$-calculus. www.cs.kent.ac.uk/projects/ofa/kroc, 2006.

[3] K. Beck and E. Gamma. JUnit test infected: Programmers love writing tests. www.junit.org, 2000.

[4] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C♯. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, Sept. 2004.

[5] A. P. Black, J. Huang, R. Koster, J. Walpole, and C. Pu. Infopipes: An abstraction for multimedia streaming. *Multimedia Systems*, 8:406–419, 2002.

[6] C. Bock. UML 2 activity and action models. *Journal of Object Technology (www.jot.fm)*, 2(4):43–53, July–August 2003. www.jot.fm/issues/issue_2003_07/column3.

[7] C. Bock. UML 2 activity and action models part 3: Control nodes. *Journal of Object Technology (www.jot.fm)*, 2(6):7–23, Nov. 2003. www.jot.fm/issues/issue_2003_11/column1.

[6] Quoted from http://www.mozart-oz.org/.

[8] P. Brinch Hansen. Joyce—a programming language for distributed systems. *Software—Practice and Experience*, 17(1):29–50, Jan. 1987.

[9] P. Brinch Hansen. Monitors and Concurrent Pascal: A personal history. In *HOPL-II: The second ACM Conference on the History of Programming Languages*, pages 1–35. ACM Press, Apr. 1993.

[10] P. Brinch Hansen. *The Search for Simplicity—Essays in Parallel Programming*. IEEE Computer Society Press, 1996.

[11] M. E. Conway. Design of a separable transition-diagram compiler. *Communications of the ACM*, 6(7):396–408, 1963.

[12] O.-J. Dahl, E. Dijkstra, and C. Hoare. *Structured Programming*. Academic Press, 1972.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] R. L. Glass. The Standish Report: does it really describe the software crisis? *Communications of the ACM*, 49(8):15–16, Aug. 2006.

[15] R. Güntensperger and J. Gutknecht. Active C♯. In *2nd International Workshop .NET Technologies'2004*, May 2004.

[16] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.

[17] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[18] C. A. R. Hoare. Letter to Per Brinch Hansen, 1991. Reprinted in [**?**].

[19] D. H. Hoffman and D. M. Weiss, editors. *Software Fundamentals: Collected Papers by David L. Parnas*. Addison-Wesley, 2001.

[20] M. Jackson. *Principles of Program Design*. Academic Press, 1975.

[21] M. Jackson. Information systems: Modelling, sequencing and transformation. In R. McKeag and A. MacNaughten, editors, *On the Construction of Programs*. Cambridge University Press, 1980.

[22] A. Kay. The early history of Smalltalk. In T. Bergin Jr. and R. Gibson Jr., editors, *History of Programming Languages–II*. ACM Press, 1996.

[23] B. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall, 1984.

[24] W. Khorfage and A. P. Goldberg. Hermes language experiences. *Software—Practice and Experience*, 25(4):389–402, Apr. 1995.

[25] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[26] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[27] J. Magee and J. Kramer. *Concurency; State Models and Java Programs*. Wiley, 1999.

[28] M. D. McIlroy. Mass produced software components. In *NATO Conference on Software Engineering, NATO Science Committee, Garmisch, Germany*, pages 88–98. Petrocelli-Charter, 1968.

[29] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[30] R. Milner. *Communicating and Mobile Systems: The $\pi$ Calculus*. Cambridge University Press, 1999.

[31] K. Nygaard and O.-J. Dahl. The development of the SIMULA language. In R. Wexelblat, editor, *History of Programming Languages*, pages 439–493. Academic Press, 1981.

[32] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972. Reprinted in [**?**, pp. 145-155].

[33] B. Selic. What's new in UML 2.0? IBM White Paper, Apr. 2005. Available at www-306.ibm.com/software/-rational/uml/resources/uml2/contributions.html.

[34] R. Strom. *HERMES: A Language for Distributed Computing*. Prentice Hall, 1991.

[35] H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, Sept. 2005.

[36] P. van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2001.