

Modularity + Concurrency = Manageability

Peter Grogono¹, Nurudeen Lameed¹, and Brian Shearing²

¹ Department of Computer Science and Software Engineering, Concordia University,
Montreal, QC, Canada

² The Software Factory, Surrey, UK

Abstract. Objects have become such a familiar paradigm for general purpose programming tasks that it is hard to imagine another paradigm replacing them. But software systems are becoming ever more complex and hard to maintain. Adapting to new hardware will create further problems. The current combination of complex scope rules, inheritance, aspects, genericity, and multithreading cannot provide the flexibility needed for the effective implementation, maintenance, and refactoring of parallel and distributed systems.

The time is right for a paradigm shift. There is a need for a methodology of programming that builds on the knowledge that we have acquired while jettisoning much of the baggage that we have accumulated. Software development must be modified to match today's needs but must not place an even greater strain on software developers. We describe an approach that we believe will reduce the difficulties of software development and maintenance. Building on earlier work but applying modern insights, we propose a programming paradigm based on processes that exchange messages. Innovative features include scale-free program structure, extensible modular components with multiple interfaces, protocols that specify the form of messages, and separation of semantics and deployment. We show that it may be possible to provide the flexibility and expressiveness of programming with processes while bounding the complexity caused by nondeterminism.

1 Introduction

Since software is a major investment, there is a strong incentive to keep programs 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, either 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 must be designed specifically to support enhancement and refactoring.

This paper reports work in progress on the Erasmus project, which attempts to deal with the issues raised above.

2 Project Erasmus

The long term goal of our research project, called *Erasmus*, is to demonstrate that the problems described in the introduction can be addressed by an approach to programming that builds on established ideas but avoids the complexity of current approaches. The major goals of the *Erasmus* project are as follows.

- Software should be *fractal*. Instead of a fixed hierarchy of levels (such as statement, method, class, and package), there should be a small number of entities that can be used at all scales.
- Concurrent programming should be safe, simple, and efficient. Tomorrow’s programmer should worry no more about adding a process to a program than today’s programmer worries about adding a function.
- The principle of abstraction should be applied more widely than at present. Details of the ways in which processes communicate with each other and are mapped to processors at run-time should not be part of the program. For example, it should be feasible to change “fat clients” to “thin clients” (i.e., moving code from client to server and modifying communications accordingly) without modifying source code.
- *Modelling* is becoming a popular approach to software design. The notation used for modelling should obviously be more abstract than the programming language but the two languages should not differ in unnecessary ways.
- Software components should provide multiple interfaces. Each interface exposes a single aspect of the component. Taken together, the interfaces provide a *complete specification* of the component, defining both what it provides *and* what it needs.

The sections that follow outline the strategies that we have adopted to meet these goals.

2.1 Processes

Erasmus is based on *communicating processes* rather than functions or objects.

Conventional wisdom says that sequential programming is easier than concurrent programming. Like many generalizations, this one is only partly true. There are simple problems for which it is easier to express a solution with concurrent processes than with sequential functions. Consider the classical example of a data source connected to a data sink via a bounded buffer. For more complex programs, the advantages of modelling are more dramatic. UML 2 encourages modelling with processes even when the implementation is object based [8].

Processes are easier to work with than procedures in many common programming situations. Hoare showed this by example in CSP [4]. Jackson advocated modeling systems as processes and then, for efficiency, transforming the models into procedural implementations [6]. When Jackson proposed his methodology, process switching was slower than procedure invocation by orders of magnitude. Although hardware designers have worked hard to speed up procedure invocation

but little to speed up process switching, we now know how to change contexts quickly; systems with many thousands of processes running on stock hardware are feasible [2, 11].

Of course, there are well-known problems associated with concurrent programming, including race conditions, fairness, deadlock, and livelock. These problems were recognized 40 years ago. Dijkstra, Holt, Hoare, Brinch Hansen, and others proposed programming constructs that enable the compiler to detect many of the potential problems. It is unfortunate that most modern programming languages in widespread use do not have these constructs [3]. However, concurrent programming issues are now understood well enough to be incorporated into the design of a new language.

2.2 Message Passing

Erasmus processes exchange data by passing messages to one another. Message passing is not new but programming languages based on message passing have never become popular. This suggests that we might be flogging a dead horse. We believe not only that the horse is still alive but also that it has the potential to become a powerful engine.

A long time ago, Lauer and Needham showed that message passing and method invocation are formally equivalent [7]. Any comparison of them must therefore be based on properties such as ease of use and reuse, modelling, safety, efficiency, and expressiveness, rather than on formal power.

Without loss of generality, we can base our comparison on clients and servers. This is because any transaction must have an initiator and a responder, which we may label as client and server respectively. A particular software component may be a client, a server, or both.

From the client's point of view, the problem with methods, as normally implemented, is that they inhibit concurrency rather than encouraging it: the client waits for the server method to return. Moreover, the client has no control over what the server is doing while it executes the method. The server can, for example, invoke one of the client's other methods and thereby change the client's private state.

The server also has problems. In particular, it provides an interface consisting of several methods, but has no control over the order in which the methods are called. As mentioned above, the server may have a method called `initialize`, but it cannot ensure that clients call this method first.

Message passing avoids these problems. A client can send a request, continue processing, and collect the results at a convenient time later. While the server is executing, the client retains control and knows that its private data cannot be tampered with.

Meanwhile, the server has full control of its resources. It can, for example, refuse to provide services until it has been correctly initialized, and it can define allowable sequences of requests.

In summary, message passing supports our goals in several ways: it loosens coupling, tightens encapsulation, provides flexible interfaces, and provides the

potential for increased security. message passing is inherently less efficient than method invocation on stock hardware but careful implementation can ensure adequate performance.

2.3 Channels and Protocols

Erasmus processes that communicate are linked by *channels*. The set of processes connected to a channel must have at least two components in order for communication to take place. At least one element must be a server and at least one element must be a client.

A channel is associated with a *protocol*. The protocol determines the ordering and types of messages that can be sent through the channel. Protocols enable the compiler to check the dynamic behaviour of a program in ways that are not feasible for OO programming languages.

Protocol expressions have a similar structure to regular expressions. The basic unit of a protocol expression is a *message* with name v , type T , and default value e , written $v : T := e$. The default value can be omitted, and usually is. If the type is omitted, the message is a *signal*; no data is sent, but signals allow processes to synchronize their activities. Protocol operators provide for options, sequencing, and choice.

There is a *subprotocol* relation on protocols: $P_1 <: P_2$ means that P_1 is a subprotocol of P_2 . Intuitively, P_2 allows all of the sequence allowed by P_1 and may allow others. When a server with protocol P_s and a client with protocol P_c connect to a channel with protocol P , the compiler checks that $P_c <: P <: P_s$. This guarantees that any request sent by the client can be satisfied by the server.

Protocols subsume several features of object oriented programming. A protocol of the query-response form $[q; \hat{r}]$ corresponds to a method that receives an argument q and returns a result r . A protocol with alternatives of the form $[A|B|C|\dots]$ corresponds to a class interface with methods A, B, C, ... that can be called in any order. However, these are only two of many forms of protocol and thus protocols are more general than either parameter lists or interfaces.

A server process that must respond to alternatives in a protocol typically uses a `select` statement. A `select` statement may have any number of branches, and each branch must start with a communication (send or receive) command. The communication is followed by a sequence of commands that may not include `select` commands. The postcondition of the `select` statement is that *exactly one* of its branches has executed.

2.4 Cells

The structural unit of Erasmus is called a *cell*. Cells can be of any size from a few bytes to a complete distributed application. A cell contains at least one process; in general, a cell may contain nested cells, multiple processes, and shared variables. The crucial properties of a cell are: a) *control flow never crosses a cell boundary*, and b) *a cell has a single thread of control*. The first property provides autonomy and the second property avoids race conditions.

Nesting in *Erasmus* is logical, not textual. Thus *Erasmus* avoids the Algol60 trap of monolithic programs. Moreover, nesting of components does not imply nesting of scopes. *Erasmus* provides conventional nested scopes, but only within cell boundaries.

A cell may provide multiple interfaces. Each interface defines one *aspect* of the cell's behaviour. The union of the interfaces specifies the cell completely. This is in contrast with the interface of an object, which defines what the object provides but not what it needs.

Channels, processes, and cells are all “first-class citizens”. A process may instantiate a channel and then pass it to other processes to enable them to communicate. A process may also instantiate other processes and cells.

3 Discussion

We anticipate that *Erasmus* programs will enjoy some of the following characteristics.

- *Erasmus* protocols will play the roles of both method parameter lists and class interfaces in object-oriented programming but will be considerably more expressive than either.
- Cells will have more autonomy than objects. Control flow is restricted to processes and never enters or leaves a cell.
- The greater autonomy of cells will lead to lower coupling between program components. The behaviour of a cell is defined entirely by its protocols; there are no implicit dependencies.
- All of the capabilities of a cell will be provided through its ports.
- Cells, like objects, are created dynamically as needed. More importantly, the processes within a cell are allocated and started only if they are actually needed. This is important for features such as aspects, which may be liberally added to cells in the knowledge that they will create no overhead unless they are actually used.
- Unlike other languages, which provide a finite hierarchy of levels of encapsulation such as expressions, methods, classes, and packages, *Erasmus* provides only one. The cell is the *only* unit of encapsulation. Cells come in all sizes, from a single number to an entire distributed application. The ubiquity of cells opens the path to scale-free programming.

4 Related Work

Since we do not have space for a full discussion of related work, we will say only that many others recognize the importance of new ways of addressing concurrency. Examples include Erlang [1], *occam- π* [2], Singularity [5], UML 2 [8], the Java Isolation API [9], *Hermes* [10], and Mozart/Oz [11].

5 Conclusion

In this paper, we have presented a language paradigm based on message passing but incorporating new ways of structuring programs, specifying protocols, and building in tests.

We have implemented a prototype compiler and a simple run-time system for a subset of the language. Experiments with this system provide supporting evidence for the feasibility of the remainder of the project. The next step is to build a higher quality compiler for the full language.

We have described the first, tentative steps towards a programming language and development environment of the kind that we believe will be needed for the next generation of software construction. Although it is rather different from most of today's languages, **Erasmus** builds on well-established past work, both theoretical and practical.

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 in more detail. Concurrently, we will work on the development environment that will match our needs and desires as well as, we hope, those of others.

References

1. J. Armstrong, M. Williams, C. Wikström, and R. Verding. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
2. F. R. Barnes and P. H. Welch. *occam- π : blending the best of CSP and the π -calculus*. www.cs.kent.ac.uk/projects/ofa/kroc, 2006.
3. P. Brinch Hansen. Java's insecure parallelism. *ACM SIGPLAN Notices*, 34(4):38–45, Apr. 1999.
4. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.
5. G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.
6. M. Jackson. Information systems: Modelling, sequencing and transformation. In R. McKeag and A. MacNaughten, editors, *On the Construction of Programs*. Cambridge University Press, 1980.
7. H. C. Lauer and R. M. Needham. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, 13(2):3–19, Apr. 1979. Originally published in Proc. Second International Symposium on Operating Systems, IRIA, October 1978.
8. 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.
9. P. Soper. JSR 121: Application Isolation API Specification. Java Specification Requests, 2002. jcp.org/en/jsr/detail?id=121.
10. R. Strom. *HERMES: A Language for Distributed Computing*. Prentice Hall, 1991.
11. P. van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2001.