MODULAR CONCURRENCY
A New Approach to Manageable Software

Peter Grogono

Computer Science and Software Engineering, Concordia University
Montréal, Québec, Canada
grogono@cse.concordia.ca

Brian Shearing

The Software Factory, Surrey, U.K.
Shearingbh@aol.com

Abstract: Software systems bridge the gap between the information processing needs of the world and computer hardware. As system requirements grow in complexity and hardware evolves, the gap does not necessarily widen, but it undoubtedly changes. Although today's applications require concurrency and today's hardware provides concurrency, programming languages remain predominantly sequential. Concurrent programming is considered too difficult and too risky to be practiced by "ordinary programmers". Software development is moving towards a paradigm shift, following which concurrency will play a fundamental role in programming.

In this paper, we introduce an approach that we believe will reduce the difficulties of developing and maintaining certain kinds of concurrent software. 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 suggest that it may be possible to provide the flexibility and expressiveness of programming with processes while bounding the complexity caused by nondeterminism.

1 INTRODUCTION

Mainstream programming languages are now, and always have been, sequential languages with side-effects. Concurrency, if present, has usually been provided by libraries, although this is known to be unsafe (Boehm, 2005). Many alternatives have been proposed, including functional, logic, single-assignment, dataflow, and process-based languages, but none have achieved widespread, industrial use so far. In particular, concurrent programming has either been confined to particular applications or hidden in operating systems and middleware.

Concurrency has become more important during the last decade for two reasons. The first reason is the transition from single-processor applications to distributed applications. In particular, the rise of network service programming has led to the need for servers handling many concurrent requests. The second reason is that processor clock rates increased until they reached about 3GHz a few years ago, and then stopped increasing, signalling the end of a fifteen-year "free lunch" (Sutter and Larus, 2005). Moore's Law, predicting a steady increase in chip density, has continued to hold. Architects seeking to increase throughput with constant clock rates but more transistors have followed the obvious path, giving us multicore chips. Effective use of multicore chips requires concurrent programming.

Computing communities are addressing the concurrency issue in a variety of ways, ranging from instigating basic research to ignoring it altogether. Much of the software industry has taken the natural approach of incremental addition to tried and tested techniques. Specifically, object oriented programming, the currently dominant paradigm, is working well, and concurrency can be achieved by multi-threading object oriented programs.

Object oriented programing languages, however, are already very complex. Java, for instance, provides fourteen different ways of controlling access to a variable. Limitations of object oriented pro-
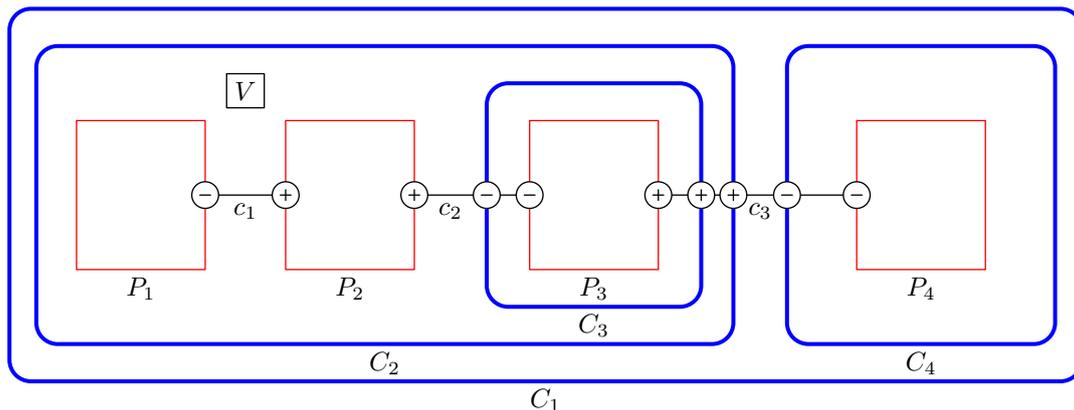
Figure 1: Structure of a μE program.

gramming are being addressed by adding yet more features. Adding concurrency to existing object oriented languages may make them unmanageable (Lee, 2006).

This suggests an approach that is somewhat more radical: we can make use of the valuable lessons that we have learned from object oriented programming, but incorporate them into a *process oriented* language. Our proposal is simple to state: instead of programming with objects that invoke one another's methods, we advocate programming with processes that exchange messages.

There are a number of problems with the process oriented approach. Perhaps the most obvious objection to it is that processes have been tried but have never caught on. Successful languages, such as Ada and Erlang, provide processes but do not use them as the basic abstraction mechanism, and have restricted domains of application. Efficiency is a potential problem: context-switching between processes is typically slower than invoking functions. Most sequential program development relies on informal reasoning techniques that are inadequate for concurrent program development.

Our defence against these arguments is that times and circumstances have changed. Computers have changed radically during the last thirty years, and so have the programs that run on them. In this paper, we examine the implications of a simple idea: the fundamental programming abstraction should be a process rather than an object or a method. Starting with this idea, we introduce a simple process oriented language, with the long-term goal of introducing an industrial-strength, process oriented language. The results reported in this paper are based on our experience with a prototype compiler written for the simple form of the language.

2 PROCESS PROGRAMMING

The language that we are proposing is called Erasmus, after the Dutch humanist Desiderius Erasmus (1466?–1536). The prototype language described in this paper is a subset of Erasmus called microErasmus, abbreviated to μE.

2.1 Program Structure

A μE program consists of *cells* and *processes*, mutually nested to any depth. Cells define the structure of a program and processes determine its behaviour. Cells come in all sizes: a cell might be a single character or a distributed application. Cells are intended to be a first step towards *scale-free programming*.

A cell contains processes and variables. The cell has a single thread of control that is shared by all of its top-level processes (that is, those processes that are not nested in cells). The processes also share access to the variables. Consequently, the processes within a cell are effectively coroutines. Programmers must be aware that the value of a shared variable may change when a process blocks for communication, but the tricky race conditions that may arise when processors access shared variables concurrently cannot occur.

Cells and processes communicate using *ports* and *channels*. A channel is connected to two ports, designated the *client* and the *server*, respectively. These names determine the direction of messages in the channel and have no other significance: a process may have several ports and may be a client with respect to some and a server with respect to others. Communication is symmetric: a demand may come from the sender ("push") or from the receiver ("pull").

Figure 1 illustrates these ideas. A cell is drawn with a thick outline with curved corners; a process is drawn with a thin outline with sharp corners. The
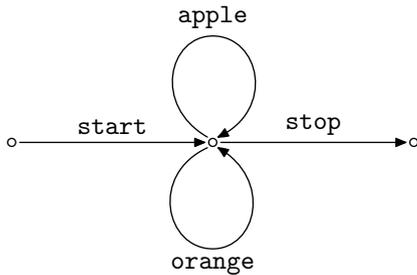
apple

start   stop

orange

Figure 2: LTS corresponding to the protocol `[start;` `*(apple | orange); stop].`

outer cell, C_1, corresponds to the "main program", which consists of two nested cells, C_2 and C_4. Processes P_1 and P_2 share a single thread of control and both have access to variable V. Cell C_2 has a third process, P_3 but, since P_3 is nested inside cell C_3, it has its own thread and cannot access V.

The processes communicate using the channels c_1, c_2, and c_3. With respect to channel c_1, process P_1 is a client (indicated by "$-$") and P_2 is a server (indicated by "$+$"). Process P_3 acts as a client with respect to c_2 and as a server with respect to c_3.

2.2 Protocols

Each channel has a *protocol* that determines the type and sequence of messages that it can carry. Protocols are defined in the same way as regular expressions, with operators for sequence (`;`), alternation (`|`), and repetition (`*`).

The compiler checks that each server *satisfies* its clients. Satisfaction is defined as a partial order on labelled transition systems (LTS). A LTS (Q, L, T) is a labelled, directed graph with nodes (states) Q, edge labels, L, and edges (transitions) T. Formally, $S \sqsupseteq C$ ("S satisfies C") if there is a mapping that preserves labels from the transitions of C to the transitions of S. From a process or protocol, P, the corresponding LTS, $\mathcal{L}(P)$, can be constructed. Given a server process, S, its protocol S_p, a client process C, and its protocol C_p, the compiler constructs the respective LTS and checks that

$$\mathcal{L}(S) \sqsupseteq \mathcal{L}(S_p) \sqsupseteq \mathcal{L}(C_p) \sqsupseteq \mathcal{L}(C).$$

Constructing an LTS from a protocol is straightforward: Figure 2 shows an example. A process has an LTS corresponding to each of its ports. To construct an LTS, the process is sliced with respect to the port. That is, reads and writes to the port and associated control structures are retained, and everything else is discarded. Consequently, the compiler checks the behaviour of each pair of communicating processes. General behaviour checking requires non-local invariants, a topic for future research.

2.3 An Example

We use a simple, textbook example, the bounded buffer, to illustrate these ideas and μE syntax. The protocol `buffProt` specifies a communication consisting of zero or more messages with label `msg` and type `Text`, followed by a signal `stop`:

    buffProt = [*(msg: Text); stop]

The process `buffer`, shown in Figure 3, has two port parameters, `in` and `out`. It acts as both a server (`+buffProt`) with respect to `in` and as a client (`-buffProt`) with respect to `out`. If a process accesses shared variables in its cell (`buffer` does not), these variables would be listed as parameters. The bar (`|`) at the end of the second line indicates the end of its parameter list. The process receives a series of text messages from a client and sends the same messages in the same sequence to a server, buffering up to ten messages in a cyclic array. The last message is the signal, `stop`; process `buffer` forwards this signal and then terminates.

Communication statements may be written anywhere in a process. However, if the process must choose between one of several actions depending on the readiness of its ports, the `select` statement is used. The `loopselect` statement used in Figure 4 is simply a repeated `select` statement; the loop terminates when an `exit` statement is executed.

The body of a `select` statement consists of a set of branches. Each branch consists of a Boolean expression B between bars, a communication statement C, and a sequence S of further statements. A branch is *enabled* if B is `true` and the channel associated with C is waiting to communicate. If no branches are enabled, the statement blocks until a branch becomes ready. If one branch is enabled, it is executed. If more

```
buffer = process
  in: +buffProt; out: -buffProt |
  buff: Integer indexes Text;
  MAX: Integer = 10;
  count, i, j: Integer := 0;
  loopselect fair
    |count < MAX| buff[i] := in.msg;
                  i := (i + 1) % MAX;
                  count += 1
    |count > 0|   out.msg := buff[j];
                  j := (j + 1) % MAX;
                  count -= 1
    || in.stop; out.stop; exit
  end
end
```

Figure 3: A μE process.

```
writer = ...
buffer = // see above
reader = ...
main = ( wp, rp: buffProt;
         writer(wp);
         buffer(wp, rp);
         reader(rp) )
main()
```

Figure 4: A μE program.

than one branch is enabled, one branch is chosen according to a *policy*, which may be `fair`, as in this example, `ordered`, or `random`. The semantics of a `select` statement specifies that, when it terminates, exactly one branch has been executed.

Within the body of the process, the assignment operator, `:=`, is used in both its normal sense of variable assignment and for communication. In the statement `buff[i] := in.msg`, which reads a message into the buffer, `in` is a port name and `msg` is a typed field of the protocol associated with the port. The key point here is that the means of communication (transfer within a memory space, between memory spaces, over a network, etc.) is not specified by the source code, but at a higher level.

Figure 4 outlines the structure of a complete program that uses process `buffer`. There are two other processes, `writer` and `reader`, whose bodies are omitted. The definition of `main` declares channels `wp` and `rp` and the three processes, which are linked by the channels. The last line, `main()`, instantiates the cell and its processes.

Input ports behave like rvalues and output ports behave like lvalues. A statement such as `p.a := q.b + r.c` is allowed and can be paraphrased as "read `b` from `q` and `c` from `r` in either order, add them, and send the result as field `a` of port `p`". This syntax is more concise and flexible than the commonly-used operators such as ! (write) and ? (read).

Given only a source program such as the one in Figure 4, the μE compiler generates code for a single processor. The compiler also accepts a second file that provides a mapping from processes to processors. With this information, it can generate code for a distributed application.

Ports and channels are "first-class citizens" that can be passed from one process to another. Dynamic data structures are built using processes and channels. For example, a tree could be represented with a process at each node and channels as links. The processes can be created at the level of the enclosing cell, in which case they share a thread, or they can be nested within cells, in which case each node has its own thread.

Programming with μE requires a change of attitude on the part of the programmer. Processes are the basic units of computation. Programmers should no more worry about introducing new processes into a program than they currently worry about introducing a new method or a new object into an object oriented program. On the other hand, programmers will have to worry a lot more about correctness and will have to be willing to embrace formal reasoning techniques.

For a more detailed description of μE, see (Grogono and Shearing, 2008).

3 DISCUSSION

The object oriented programming model has many advantages that we do not want to give up. In fact, as a first approximation, programming in μE is somewhat similar to programming in object oriented programming, with cells replacing classes and processes replacing methods. In this section, we look more closely at the differences between the paradigms and give our reasons for preferring processes.

3.1 Adapting to Change

Since process-oriented languages have been available for a long time, but have never become mainstream, it is important to explain what has changed to make process oriented programming important.

There have been three significant changes. First, since 2004, chip manufacturers have been able to offer higher performance by providing more processors rather than higher clock speeds. Second, the prevailing paradigm, object oriented programming, has become increasingly complex. Multithreaded object oriented programs will be very hard to develop, maintain, and refactor (Lee, 2006). Third, we have considerably more experience of programming language design and usage than we had 30 years ago. μE exploits these changes in ways described below.

Coupling. The need for weak coupling between software modules is widely understood but not easy to achieve in object oriented programming. An object has two interfaces: the services it offers and the resources it needs. The first of these, usually called *the* interface, is explicit but the second is implicit. This means, for example, that when an object is moved to another processor, it will drag an unpredictable number of other objects along with it.

The declaration of a μE process includes both the services it offers and those that it requires; there are no

hidden dependencies. A process that requires special resources obtains them through its ports. The condition for moving a cell is just that its ports can be connected in its new environment.

Protocols. Protocols determine not only the messages that the cell can process but also their sequence. Of course, the constraints on sequencing depend on the form of the protocol. A protocol with the form $[*(m_1|m_2|\cdots|m_k)]$ is equivalent to a class with methods $m_1, m_2, \ldots, m_k$. However, protocols can specify more complex interaction patterns and can represent asymmetries: for example, a client typically needs only a subset of the services offered by the server that it is connected to. Moreover, a cell may have several ports, each with its own protocol.

Small Components. The larger a module, the harder it is to reuse. Object oriented programming encourages small methods but not small classes. Processes encourage a "do one thing well" approach that produces small, reusable components. Several related processes can be packed into a cell with a simple interface, analogously to the Façade pattern.

Localized Threads. An object oriented program without concurrency has, by definition, a single thread of control that meanders through the entire program. In multithreaded object oriented programs, threads also tend to be long, typically visiting many objects during their lifetime. This is one of the factors that makes concurrent object oriented programming difficult: an object may be active in multiple threads simultaneously.

In μE, a thread is restricted to a single cell. This gives cells more autonomy than objects; they have full control over their resources. Access to shared resources must still be controlled, of course, but this can be done at a higher level than threads. Reasoning about threads is simplified.

Semantics *versus* Deployment. The language defines the semantics of communication and provides a single operator (:=) for all forms of communication. The *deployment* of the program—that is, the mapping of software components to available hardware—is separated from the source code (Lameed and Grogono, 2008). Static mapping has limitations. For example, it is unsuitable for applications that generate many processes on the fly, or require dynamic load-balancing. We hope to address these kinds of problems in future research.

Abstraction Level. An object oriented program can be transformed to a procedural program, but the converse transformation is generally infeasible. (It would be possible to use heuristics to find implied objects in a procedural program, but the found objects probably would not be the "real" objects.) Similarly, a process oriented program can be transformed to an object oriented program, but not *vice versa*. In this sense, process oriented programming adds a new level of abstraction to the programming language hierarchy.

3.2 Performance Issues

Clearly, performance of process oriented programs will be an important issue. We address performance in several ways.

First, as mentioned, the compiler is given a mapping from cells to processors and memory spaces. Communication within a memory space can be implemented efficiently by memory moves. Processes with simple protocols can be implemented as functions: external communication will be performed by their cells. Small functions can be inlined.

Top-level processes within a cell run as coroutines. In many cases, however, a process can be transformed into sequential code. The transformation was introduced by Jackson (1978), who called it "inversion". When inversion is possible, a cell containing many processes could be implemented as efficient, sequential code. Trade-offs must be considered, however; in a processor-rich environment for which high performance requires concurrency, inversion might actually be counter-productive.

An important question remains. Assuming that many processors are available, how effectively will a process oriented program use them? Some of the optimizations mentioned above translate parallel code to sequential code, which may not help.

We have written and tested many small programs in μE. The results of these experiments suggests that concurrency arises more naturally than might be expected, provided that problems are decomposed into small enough processes. We speculate that this will enable Erasmus programs to execute efficiently on appropriate architectures. More work will be needed with programs of realistic size to verify this hypothesis.

4 RELATED WORK

Work related to ours can be divided into two categories: research within the current framework that aims to extend object oriented languages with abstractions for concurrency; and research directed towards a "paradigm shift" that aims to bring concurrent languages into the mainstream.

Though we now see the need for a paradigm shift, the potential advantages of loosely-coupled concurrent processes were recognized forty years ago:

> We have stipulated that processes should be connected loosely; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes themselves are to be regarded as completely independent of each other. (Dijkstra, 1968)

UNIX® provided an early demonstration of the possibilities of lightweight processes. In particular, pipes (introduced by Douglas McIlroy) made it easy to build complex tasks from simple components. Pipes are useful in spite of their being limited to one input and one output, and character streams.

Hoare (1978) introduced Communicating Sequential Processes (CSP) and Milner (1980) introduced a Calculus of Communicating Systems (CCS), both systems for reasoning about communicating concurrent processes. Erasmus processes communicate synchronously, as do processes in CSP. Synchronous communication is more fundamental than asynchronous communication because we can decouple two communicating processes by inserting a buffer process between them.

CSP was developed into a full-fledged programming language called Occam (May, 1983). Occam's descendant, Occam-π (Barnes and Welch, 2006) has a number of features that are relevant to our project. Occam-π demonstrates the possibility of efficient execution of many small processes; programs with millions of processes can be run on an ordinary laptop. It also provides *mobile processes*, which are processes that may be suspended, sent to another site, and resumed. Occam-π has a formal semantics based on Milner's (1999) π-calculus.

Jackson was an early proponent of concurrent processes. He considered the hardware of the time to be too primitive to actually implement concurrency directly, but he demonstrated the advantages that accrue from modelling a system as a set of processes:

> The basic form of model is a network of processes, one process for each independently active entity in the real world. These processes communicate by writing and reading serial data streams or files, each data stream connecting exactly two processes; there is no process synchronization other than by these data streams. (Jackson, 1980)

In Jackson's method, processes introduced in the design phase were implemented as sequential code. With modern hardware, they could be implemented directly as processes.

Although Simula (Nygaard and Dahl, 1981) introduced many of the features that we now call "object oriented", the first language that used objects as the basic abstraction was Smalltalk. Kay (1993) viewed object-oriented programming as a new paradigm in which each object was an abstraction of a computer. It is interesting that both Simula and Smalltalk provided non-preemptive concurrency, a feature that was not copied in later languages.

There were a number of early efforts to combine objects and concurrency. So many, in fact, that the acronym "COOL" was popular for a while. Inheritance was problematic, however, leading to the identification of the "inheritance anomaly" (Matsuoka and Yonezawa, 1993). More recently, *aspects* have been proposed as a way of avoiding the anomaly (Milicia and Sassone, 2004).

Ada (1995) is one of the few industrial-strength languages that provides secure concurrency. In spite of its advantages, its use has been restricted mainly to the aerospace industry. The `select` statement of μE is very similar to the "selective wait" of Ada. Ada also provides features for real-time programming, an area that we have not considered yet.

Brinch Hansen (1996) designed a number of languages for concurrent programming. Joyce was developed as a programming language for distributed systems (Brinch Hansen, 1987). 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 version of Erasmus is quite similar to Joyce in concept and makes use of several of Brinch Hansen's ideas. For example, `select` statements in Erasmus are similar to `poll` statements in Joyce.

Hermes is an experimental language developed at IBM but never used in production (Strom, 1991; Khorfage and Goldberg, 1995). 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 only with the capabilities that it needs and can obtain additional capabilities only by explicitly requesting them. Erasmus also uses a capability model (not discussed here) to provide protection and to reduce coupling.

The language now known as Erlang started life in an Ericsson laboratory as a Smalltalk program but quickly evolved into a Prolog program (Armstrong,

2007). Erlang is now often cited as a "functional" language because individual processes are coded without side-effects (which is why Erlang is often referred to as a "functional" language). The goals of Erasmus are in many ways similar to those of Erlang but we place greater emphasis on state. Erlang uses asynchronous message passing which eliminates shared references between processes and maximizes process independence (van Roy and Haridi, 2001, page 387). In Erasmus, it is easy to provide buffer processes for asynchronous communication.

The Mozart Programming System (van Roy and Haridi, 2001) is based on the language Oz, which supports "declarative programming, object-oriented programming, constraint programming, and concurrency as part of a coherent whole".[1] Oz 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, and suitable for building highly reliable systems. It is for these very reasons that we have adopted message-passing as the basic communication mechanism in Erasmus.

Microsoft's C# relies on the multi-threaded environment .NET. There are various proposals for extending C# with concurrency primitives at a higher level of abstraction than that provided by .NET. Benton *et al.* (2004) have introduced *asynchronous methods* and *chords* in Polyphonic C#. Active C# enhances C# with concurrency and a new model for object communication (Güntensperger and Gutknecht, 2004). An *activity* is a class member that runs 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.

The Java Application Isolation API allows components of a Java application to run as logically independent virtual machines (Soper, 2002). The motivation for "isolates" is to prevent interference between components and to "simplify construction of obviously secure systems" (Lea et al., 2004).

Henzinger *et al.* (2005) have shown how to synthesize "permissive interfaces" that allow clients to be checked for compatibility with libraries. These interfaces are similar to Erasmus protocols.

The Singularity operating system currently under development at Microsoft Research "consists of three key architectural features: software-isolated processes, contract-based channels, and manifest-based programs" (Hunt and Larus, 2007). Singu-

larity and Erasmus share several common features, notably the emphasis on processes and on checked communications—contract-based channels in Singularity and protocols in Erasmus. Since Singularity defines contracts in terms of general state machines and Erasmus protocols are regular expressions, the two systems are formally equivalent.

UML 2 has responded to the call for concurrency by providing more flexibility than UML 1 for modeling concurrent systems (Object Management Group, 2007). Concurrency is modelled by forks and joins in control flows, as in UML 1, but there is no synchronization following a fork. Following the practice of architecture description languages, system components have ports and communicate via connectors. Many UML 2 models map naturally into Erasmus programs.

5 CONCLUSIONS

Hoare (1974) said that language designers should consolidate, not innovate. We have based μE on concepts that have been used in the past and are well-understood. We have introduced a few new ideas—the cell in which processes run as coroutines, for example—but, on the whole, it is the particular combination of features that is new.

It is widely acknowledged that it will be difficult to write multithreaded object oriented programs that use many concurrent processors efficiently. We suggest that process oriented programs may be better suited to this task. We have several reasons for believing this. First, cells are less tightly coupled than objects because cells exchange data rather than passing control. Second, the fact that control never crosses process or cell boundaries will contribute to simplified reasoning. Third, protocols allow more precise specifications of allowable behaviours than interfaces defined by methods. Moreover, the declaration of a process or cell specifies both the services it provides and those that it needs, not just the services that it provides. Finally, there are no race conditions, because each variable is accessed by a single thread.

There are some trade-offs. Superficially, Erasmus programs look more complex, but this is because all dependencies appear explicitly in the code. (In object oriented programming, if a class C has instance variable $x : D$, then C uses D, although this dependency appears in neither interface.)

Although much remains to be done, we believe that process oriented programming, in some form or another, will prove to be superior to object oriented programming for applications of the future.

---

[1] Quoted from http://www.mozart-oz.org/, accessed 2008/03/15.

ACKNOWLEDGEMENTS

REFERENCES

Ada (1995). Ada 95 Reference Manual. Revised International Standard ISO/IEC 8652:1995. www.adahome.com/rm95. Accessed 2008/03/12.

Armstrong, J. (2007). A history of Erlang. In *HOPL III: Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages*, New York, NY, USA, pp. 6.1–6.26. ACM Press.

Barnes, F. R. and P. H. Welch (2006). Occam-π: blending the best of CSP and the π-calculus. www.cs.kent.ac.uk/projects/ofa/kroc. Accessed 2008/03/13.

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

Boehm, H.-J. (2005). Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pp. 261–268.

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

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

Dijkstra, E. W. (1968). Cooperating sequential processes. In F. Genuys (Ed.), *Programming Languages: NATO Advanced Study Institute*, pp. 43–112. Academic Press.

Grogono, P. and B. Shearing (2008). MEC Reference Manual. http://users.encs.concordia.ca/∼grogono/-Erasmus/erasmus.html.

Güntensperger, R. and J. Gutknecht (2004, May). Active C♯. In *2nd International Workshop .NET Technologies'2004*, pp. 47–59.

Henzinger, T. A., R. Jhala, and R. Majumdar (2005). Permissive interfaces. In *ESEC/FSE-13: Proceedings of the 10th European Software Engineering Conference (held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering)*, pp. 31–40.

Hoare, C. A. R. (1974). Hints on programming language design. In C. Bunyan (Ed.), *Computer Systems Reliability*, Volume 20 of *State of the Art Report*, pp. 505–534. Pergamon. Reprinted in (Hoare and Jones, 1989).

Hoare, C. A. R. (1978, August). Communicating sequential processes. *Communications of the ACM 21*(8), 666–677.

Hoare, C. A. R. and C. Jones (1989). *Essays in Computing Science*. Prentice Hall.

Hunt, G. C. and J. R. Larus (2007). Singularity: rethinking the software stack. *SIGOPS Operating System Review 41*(2), 37–49.

Jackson, M. (1978). Information systems: Modelling, sequencing and transformation. In *Proceedings of the 3rd International Conference on Software Engineering (ICSE 1978)*, pp. 72–81.

Jackson, M. (1980). Information systems: Modelling, sequencing and transformation. In R. McKeag and A. MacNaughten (Eds.), *On the Construction of Programs*. Cambridge University Press.

Kay, A. C. (1993). The early history of Smalltalk. *ACM SIGPLAN Notices 28*(3), 69–95.

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

Lameed, N. and P. Grogono (2008). Separating program semantics from deployment. These proceedings.

Lea, D., P. Soper, and M. Sabin (2004). The Java Isolation API: Introduction, applications and inspiration. bitser.net/isolate-interest/slides.pdf. Accessed 2007/06/14.

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

Matsuoka, S. and A. Yonezawa (1993). Analysis of inheritance anomaly in object-oriented concurrent programming language. In *Research Directions in Concurrent Object-Oriented Programming*, pp. 107–150. MIT Press.

May, D. (1983, April). Occam. *ACM SIGPLAN Notices 18*(4), 69–79.

Milicia, G. and V. Sassone (2004). The inheritance anomaly: ten years after. In *SAC '04: Proceedings of the 2004 ACM Symposium on Applied Computing*, New York, NY, USA, pp. 1267–1274. ACM.

Milner, R. (1980). *A Calculus of Communicating Systems*. Springer.

Milner, R. (1999). *Communicating and Mobile Systems: The π Calculus*. Cambridge University Press.

Nygaard, K. and O.-J. Dahl (1981). The development of the SIMULA language. In R. Wexelblat (Ed.), *History of Programming Languages*, pp. 439–493. Academic Press.

Object Management Group (2007, November). OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. http://www.omg.org/spec/UML/2.1.2/-Superstructure/PDF/. Accessed 2008/03/15.

Soper, P. (2002). JSR 121: Application Isolation API Specification. Java Specification Requests. http://jcp.org/aboutJava/communityprocess/-final/jsr121/index.html. Accessed 2008/03/15.

Strom, R. (1991). *HERMES: A Language for Distributed Computing*. Prentice Hall.

Sutter, H. and J. Larus (2005, September). Software and the concurrency revolution. *ACM Queue 3*(7), 54–62.

van Roy, P. and S. Haridi (2001). *Concepts, Techniques, and Models of Computer Programming*. MIT Press.