

SEPARATING PROGRAM SEMANTICS FROM DEPLOYMENT

Nurudeen Lameed and Peter Grogono

Department of Computer Science and Software Engineering, Concordia University

1455 de Maisonneuve Blvd. W, Montreal, Canada

n_lameed@encs.concordia.ca, grogono@cse.concordia.ca

Keywords: Concurrency, mapping, communication, protocol, abstraction.

Abstract: Designing software to adapt to changes in requirements and environment is a key step for preserving software investment. As time passes, applications often require enhancements as requirements change or hardware environment changes. However, mainstream programming languages lack suitable abstractions that are capable of providing the needed flexibility for the effective implementation, maintenance and refactoring of parallel and distributed systems. Software must be modified to match today's needs but must not place even greater strain on software developers. Hence, software must be specially designed to accommodate future changes. This paper proposes an approach that facilitates software development and maintenance. In particular, it explains how the semantics of a program can be separated from its deployment onto multiprocessor or distributed systems. Through this approach, software investment may be preserved when new features are added or when functionality does not change but the environment does.

1 INTRODUCTION

Advances in software development practices facilitated the programming of many hard problems. Many interesting and crucial problems have been solved. The success however, is generating an increasing demand for ever more challenging computations. Consequently, software is becoming increasingly complex to design. To cope with this challenge, various tools, languages and techniques have been proposed, implemented and utilized. The transition from procedural to object-oriented programming helped to facilitate the construction of software that otherwise would have been inconceivable. However, increase in software complexity as a result of changes in environment and functionality has exposed some significant limitations of object-oriented approach: current practice makes software enhancement and refactoring difficult.

Furthermore, the drive for continued performance gains is causing major processor manufacturers to produce microprocessors with multiple cores on a chip. Multi-core architectures have potentials to boost performance. However, existing applications are largely sequential. To achieve true performance gains software must be carefully written to exploit hardware parallelism (Olukotun and Hammond, 2005; Sutter, 2005a). But concurrent programming is hard. Experts

agree that concurrent programming is hard because mainstream programming languages do not provide suitable abstractions for expressing and controlling concurrency. (Lee, 2006; Sutter, 2005b; Harris and Fraser, 2003).

We have implemented our approach in a new process-oriented language named *Erasmus*. Although in this paper, we describe the approach within the context of Erasmus; we believe due to the nature of process-oriented design, the approach can be implemented in any process-oriented language. The paper is structured into the following sections: section 2 examines the object and the process-oriented models; section 3 briefly describes the Erasmus programming language; section 4 explains how software components may be mapped onto processors; section 5 examines the implementation; section 6 discusses testing and performance evaluation; section 7 discusses the results; section 8 reviews some related work and finally, section 9 concludes the paper.

2 THE OBJECT AND THE PROCESS MODELS

This section reviews the object-oriented and the process-oriented models.

2.1 The Object-Oriented Model

Object-oriented programming has been very successful for general purpose programming tasks for almost two decades. It is hard to imagine another paradigm replacing it. The idea behind OO programming is that an object encapsulates data and provides controlled access to the data via its methods; the behaviour of an object is determined by its class and may be extended incrementally through inheritance. In practice however, OO programming has become more complex than this simple model suggests. For instance, an object does not have control over the sequence in its methods are called. Thus, it cannot ensure that an initialization method is called before any other methods (Grogono et al., 2007). This makes it difficult to design objects. The designer must handle all possible calling sequences or trust clients to call methods in the intended sequence. The compiler cannot enforce correct call sequencing.

Object-oriented languages have extended levels of visibility because the original levels of visibility — private, public and protected — were inadequate. The extensions such as package scope, static method, interfaces, inner classes, anonymous inner classes, simplify programming but make application maintenance difficult.

2.2 The Process-Oriented Model

The process-oriented model represents an alternative to the object-oriented model. A process is a program module that defines a data structure and a sequence of statements that operate on it. The idea of organizing programs into collections of processes is not new. This idea was pioneered at MIT in the CTSS project (Salzer, 1965).

The benefits of the process-oriented model are well known. Processes are more general. They are loosely coupled and have more autonomy as compared to objects. This is because they only have effect on one another during rare and brief communication. In addition, unlike objects that do not have control over the way in which its methods are called, a process owns its thread of control and can choose when to communicate or not communicate. Besides, if a process operates only on its private data, it will be completely independent of other processes. Dijkstra declares:

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 them-

selves are to be regarded as completely independent of each other. (Dijkstra, 1968)

Process-based code can be refactored into object-based code but the converse is difficult, if not impossible. It is easier to map processes onto multiple address spaces than objects. Furthermore, it is often easier to model applications with processes than objects.

Processes have been utilized in operating systems and programming languages. Hoare (Hoare, 1978) proposed communicating sequential processes as a method of structuring programs. Joyce (Brinch Hansen, 1987), Erlang (Armstrong et al., 2004), Occam- π (Barnes and Welch, 2003; Barnes and Welch, 2004) are based on *communicating processes*. The next section describes the Erasmus Programming language.

3 THE LANGUAGE, ERASMUS

The programming language, Erasmus is being developed by Peter Grogono at Concordia University, Canada and Brian Shearing at The Software Factory, UK. Erasmus is based on communicating processes.

Concurrency in Erasmus is based on communicating processes. A *closure* is an autonomous process with its own state and instructions. Closures may have parameters and communicate over synchronous channels satisfying some well-defined protocols. A port serves as an interface for a closure to communicate with another closure.

The basic building block of an Erasmus program are *cells*, *closures* and *protocols*. A cell is a collection of one or more closures. Protocols define constraints on messages that can be transferred with any port that is associated with the protocol. Cells and processes can be created dynamically. Erasmus cells, processes and ports are first-class entities: they can be transmitted from one process to another and may be sent over a network. A program is a sequence of definitions followed by the instantiation of a cell. An example of a simple Erasmus program is

```
sayHello = { | sys.out := "Hello"; };
cell = ( sayHello() );
cell ();
```

The program prints the string “Hello” to the standard output device. It comprises of a process named *sayHello*, followed by a cell definition and an instantiation of the cell. When the cell is instantiated, it instantiates *sayHello*, and *sayHello* starts to execute.

4 MAPPING OF CELLS TO PROCESSORS

Erasmus separates program semantics from its deployment: programmer defines cells and messages that cells exchange; separately, the programmer specifies how cells are assigned to processors — a task commonly referred to as *mapping*. This clear separation of concerns makes it possible to use the same syntax for communications regardless of the environment.

The explicit mapping of cells onto processors has several other benefits. Cells that communicate frequently may be mapped onto processors that are physically close to reduce communication time. Being familiar with the nature of the problem that is solved by the program, programmer may map cells onto processors considering the specific nature of the problem and the characteristics of the individual processors (Bal et al., 1989).

Erasmus program may be compiled to run on a standalone system. The same program may be recompiled to run on a multicomputer — a network of computers with independent memory. Furthermore, the program code may be moved to a multiprocessor system with shared memory. All this is possible without changing the source program. However, compilation of an Erasmus program for a multiprocessor or distributed system requires a separate configuration file that specifies specific detail about mapping of cells onto the participating processors.

A typical configuration file is an XML file; it contains specific properties about processors encapsulated within a pair of `<mapping>` and `</mapping>` tags. A record is defined for each processor within `<processor>` and `</processor>`. A processor record specifies the name of the processor, the port number of its communicating agent called *broker* and, a list of cells mapped onto this processor. The port number is enclosed within a pair of `<port>` and `</port>` tags. A cell is defined within `<cell>` and `</cell>`. A sample configuration file is shown below.

```
<mapping>
  <processor> alpha.encs.concordia.ca
    <port> 5555 </port>
    <cell> squareCell </cell>
  </processor>
  <processor> latvia.encs.concordia.ca
    <port> 5556 </port>
    <cell> clientCell </cell>
  </processor>
</mapping>
```

The configuration file shown above maps *squareCell* onto the processor identified by

alpha.encs.concordia.ca and *clientCell* onto *latvia.encs.concordia.ca*. If the following program is compiled without any configuration file, it generates code for standalone systems. However, the two cells: *squareCell* and *clientCell* may be mapped onto different processors as shown in the configuration file above.

```
sqProt = [(query:Float; ^reply:Text)];
square = { p: +sqProt |
loop
  q: Float := p.query;
  p.reply := text (q * q);
end };
squareCell = (prot: +sqProt |
  square(prot) );

client = { p: -sqProt |
  num : Float := 10;
  p.query := num;
  sys.out := text num + "^2 = ";
  sys.out := p.reply + "\n"; };
clientCell = ( prot: -sqProt |
  client(prot) );

mainCell = ( chan: sqProt;
  clientCell(chan);
  squareCell(chan) );
mainCell();
```

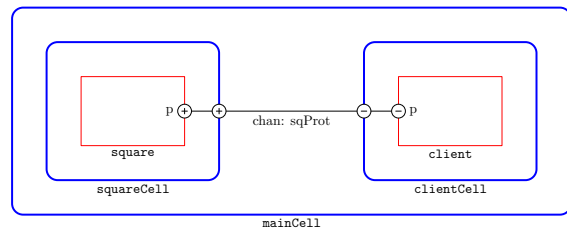


Figure 1: A diagram corresponding to the program above.

In the program shown above, two cells, *clientCell* and *squareCell* are connected to a channel named *chan* through their local ports. Figure 1 shows a diagram corresponding to the program. The ports use the protocol defined by *sqProt*. The processes *square* and *client* have ports that are connected to *chan* as shown in Figure 1. A port that *provides* a service is referred to as a *server* port and is declared using the prefix '+' before the name of the protocol associated with the port. A port that *needs* a service is called a *client* port and is declared using the symbol '-'. A port declaration with no sign creates a channel.

The port *p* of *square* is a server port (i.e. it provides a square service) while *client* has a client port i.e it needs a square service. When *mainCell* is instantiated, *clientCell* and *squareCell* are also instantiated. This causes a concurrent execution of processes in the two cells. The cell *clientCell* sends a number (10) to

the *squareCell* via its local port, *prot*. The *squareCell* sends a reply (i.e. a string containing 100, the square of 10) to the client.

4.1 Compilation Process

Compilation of an Erasmus program that comprises of cells mapped onto some processors requires a configuration file. The compiler reads the XML file, extracts the data and organizes the contents into a table. The compiler generates a unique identification for each cell. Later, it retrieves mapping information for each cell. This is eventually written to a file — *hosts.txt* — in an order determined by cell id; each line of the file contains a record about a cell. If there is no entry for a cell in the configuration file, ‘local-host’ and port number 0 is written for the cell. This indicates that any available processor may execute the cell.

5 IMPLEMENTATION

5.1 Inter-process Communication

Various approaches for implementing communication and synchronization between isolated processes have been proposed and implemented. Gregory Andrews (Andrews, 2000) proposed a centralized message passing implementation that uses a “clearing house”. A clearinghouse process matches pairs of communication requests. A template is a message that describes a communication request. When a process wants to communicate, it sends a template or a set of templates, if several communications are possible. When the clearinghouse receives a template, it checks if it has received before a “matching” template otherwise, it stores the template. If it has a matching template, it sends some synchronization messages to the processes that sent the templates. These processes then use the information received from the clearinghouse to communicate. Subsequently, both processes continue at their next statements after exchanging data.

A disadvantage of a centralized implementation is the inherent potential for the clearinghouse to become a bottleneck in the system. This is likely to be exacerbated where large number of processes is involved. Other implementations can be found in (Silberschatz, 1979; Bernstein, 1980; Schneider, 1982; Buckley and Silberschatz, 1983; Bagrodia, 1989). The implementation described in this section uses a distributed clearinghouse for inter-process communication. Since we require reliable communications and

TCP/IP guarantees that messages are delivered in the order in which they have been sent, our implementation is based on TCP/IP.

5.2 The Broker Process

Each processor runs a special process named *broker*. The broker is responsible for handling all communications by processes. The port address on which the broker attached to a processor runs is specified as a property of the processor in the configuration file. During start-up, a broker loads a table with the data from the file, *hosts.txt* created earlier during compilation by the compiler. As described earlier, a record in *hosts.txt* comprises of a host name and a port number. A broker also maintains a table of valid connections to cells and brokers running on other hosts. Brokers are *daemons* that match communications.

A cell that wishes to communicate with another cell sends the request to its local broker. Cells must first establish connections to their local brokers before executing any instructions. If a connection has been established, the cell activates its processes. Meanwhile, the broker listens for requests from communicating agents (cells or brokers). When a broker receives such request, it stores it until it finds a matching request from another cell. The message transferred in a particular request depends on whether the request is a read (receive) or write (send) operation.

The distributed nature of the broker reduces the likelihood of a broker becoming a bottleneck in system. Messages received by a broker are addressed to cells running on the host.

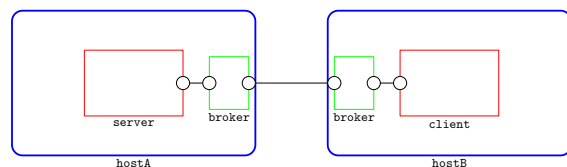


Figure 2: The broker process.

Every cell that communicates with another cell is given a port to do so. A port is connected to a shared channel. A channel connects two or more processes. Ports are shown in the diagram as small circles. Communication takes place when a port is referenced (i.e. a qualified name refers to a field of the associated protocol). When a process in a cell sends a message to another cell, it constructs a message containing some useful headers and sends the message to the broker running on the host. The broker inspects the message and retrieves some of the headers to determine the location of the communicating partner. A message comprises of

1. a port direction (0 or 1) which indicates whether the sending port is a server or a client;
2. the cell id of the server connected to the channel;
3. the cell id of the source of the message;
4. policy type; values include: 0 (ordered), 1 (fair) and 2 (random);
5. input/output direction; 0 for *read* and 1 for *write*;
6. the type of the data carried, if the request is a write operation;
7. the id of the destination cell (client/server);
8. a tag for the beginning of the data;
9. the data to be sent, if it is a write operation; empty if it is a read operation;
10. a tag for the end of the data; a message may carry more than one request.

When a process sends a write request to a broker, it sends along with the message, the data to be written to the other process. If the broker receives a matching request, the data is copied into the variable of the other process and both the sender and the receiver can proceed independently and concurrently. Considering the program and the XML file shown in shown in Section 4, when *squareCell* executes

```
q:Float := p.query;
```

assuming that *squareCell* and *clientCell* have been assigned unique ids, 1 and 2 respectively, the compiler builds a message (used later by the runtime system to send and receive data) of this form:

```

  1   2   3   4   5   6   7   8   9   10
-----
| 0 | 1 | 1 | 0 | 0 | 1 | 2 | $ |   | $ |
-----
```

items at the subscripts 1-7, corresponds to the items 1-7 of the list describing the contents of a message. Items eight and ten (“\$\$”) are tags that demarcate respectively, the beginning and the end of the data in a message. This is a read request, hence item nine — the data — is empty.

This message (“0 1 1 0 0 1 2 \$\$”) is then sent to the broker. When the broker receives the message, it retrieves the first two items in the message and performs one of the following two actions:

1. if the port is a client port, it checks whether the server that is connected to the channel is running on this host or on a remote host. If the server is on a remote host, it forwards the request to the broker running at the host of the server and waits for a reply from the broker. The reply is subsequently forwarded to the client cell.

2. if the port is a server port, the broker queues the message until (hopefully) a matching request is received from the client.

Therefore in this case, the broker running on *alpha.encs.concordia.ca* forwards the message to the broker running on *latvia.encs.concordia.ca*. This message is matched by the receiving broker when it receives a matching request i.e. a message of the form 1 1 2 0 1 1 1 \$d\$ from *clientCell*. The symbol *d* denotes the data which can be of any size. This message is generated when *clientCell* executes the statement

```
p.query := num;
```

Typically, the communication broker *marshals* the data carried in a message before sending the message to a broker on another host. It *unmarshals* the data when it receives a message from another broker. As an optimization, this process, i.e. marshalling/unmarshalling may be avoided if both the source and the destination hosts are known to have the same architecture.

Matching occurs when a broker receives two messages: one from a client and another from a server. The two input/output requests must be opposite. One must be a read operation while the other must be a write operation. The type of the data in the messages must also match. The compiler checks the type and direction of each message, ensuring that communication cannot fail at runtime. During matching, data received is copied and sent to the process executing a read command. The message may contain a set of requests if the process is willing to communicate with more than one process.

6 TESTING

The compiler has been used to compile the program shown in Section 4 for a standalone computer. The same program was also distributed on two computers: the server cell running in one process on a computer while the client runs in another process in a different host. The broker processes on both hosts matched communications between these cells. The program works as a client-server system as specified by protocol *sqProt*. The client process sends a request to compute the square of 10. The server responds by sending the corresponding result (100) to the client. Although the program (Section 4) solves a trivial problem, it nevertheless shows how Erasmus facilitates distribution of programs to different architectures. In prac-

tice, this approach may help preserve software investment where software environment changes.

Tests conducted to evaluate impact of communications on the overall performance of programs written in Erasmus are described next. The two computers used for these tests, i.e. { *latvia*, *lithuania* } .*encs.concordia.ca* have the same specification. The specification is given below:

- Intel(R) Pentium(R) 4 CPU 3.00 GHz, 2.99 GHz, 1.00 GB of RAM; Operating System: Windows XP Professional, version 2002; Service Pack 2.

The tests are grouped into two cases namely:

6.1 Case One

```
sp = [*(ask:Integer; ^answer:Integer)];
square = { p: +sp |
loop
  in: Integer := p.ask;
  x: Integer := in * in;
  --- do more work
  j: Integer := 0;
  loop while j < 1000000 j += 1; end;
  p.answer := x; end };
client = { p: -sp | x: Integer := 1;
loop
  xs: Integer := x * x;
  sys.out := text x + "^2=" + text xs + "\n";
  --- do more work
  j: Integer := 0;
  loop while j < 1000000 j += 1; end;
  x := x + 1;
  if x > 1500 then exit end end };
squareCell = ( p: +sp | square(p) );
clientCell = ( p: -sp | client(p) );
cell = (p: sp; clientCell(p);squareCell(p));
cell();
```

Considering the last program (above), it takes 145s on *latvia.encs.concordia.ca* for *clientCell* to display the squares of the numbers from 1 to 1500. In the program, *clientCell* essentially performs the required computation since no request whatsoever was sent from *clientCell* to *squareCell*.

- **Scenario One:** if *clientCell* in the previous program is replaced with the *clientCell* given below, it takes 289s to display the squares. In this case, *clientCell* communicates frequently with *squareCell* by sending numbers whose squares are computed by *squareCell* through the port *p*.

```
client = { p: -sp | x: Integer := 1;
loop
  p.ask := x;
  xs: Integer := p.answer;
  sys.out := text x + "^2=" + text xs + "\n";
  --- do more work
  j: Integer := 0;
```

```
loop while j < 1000000 j += 1; end;
x := x + 1;
if x > 1500 then exit end end };
```

Both *clientCell* and *squareCell* run in one memory space and therefore the compiler generates code that is executable on a standalone computer. No configuration file is required and none was used in this scenario. The difference in the elapsed-times (145s vs 289s) for *clientCell* to display all the computed results can be attributed to the communication overhead in executing the program above.

- **Scenario Two:** if this program is compiled using the configuration file shown below, it takes 236s on *latvia.encs.concordia.ca* for *clientCell* running in a separate address space to display the squares of the numbers from 1 to 1500. It is interesting to note that *clientCell* runs faster than in the previous scenario. This suggests that the combined overhead of the inter-process communication and the runtime process scheduling overheads is higher in scenario one than the communication delay in scenario two.

```
<Mapping>
<Processor> latvia.encs.concordia.ca
  <Port> 5555 </Port>
  <Cell> squareCell </Cell>
  <Cell> clientCell </Cell>
</Processor>
</Mapping>
```

- **Scenario Three:** if the same program is compiled with the configuration file shown below, it takes *clientCell* running on *latvia.encs.concordia.ca* 241s to completely display the squares of all the numbers from 1-1500 as specified in the program. This is slightly longer than that in scenario two. This means that in this case, *clientCell* runs slower than in scenario two. Frequent network communications between the client process in one host and the server process in another host cause delays, which are likely to offset any performance gains in faster execution of *clientCell* due to the host's processor having fewer processes to execute.

```
<Mapping>
<Processor> latvia.encs.concordia.ca
  <Port> 5555 </Port>
  <Cell> clientCell </Cell>
</Processor>
<Processor> lithuania.encs.concordia.ca
  <Port> 5555 </Port>
  <Cell> squareCell </Cell>
</Processor>
</Mapping>
```

In scenarios two and three, communication brokers add to the communication delays. One broker

is involved in scenario two but two broker processes (one on each host) are involved in scenario three.

6.2 Case Two: Infrequent Inter-process Communications

The test described here is similar to the last three scenarios in case one. The only difference is that *client-Cell* is replaced with the following code.

```
client = { p: -sp | x: Integer := 1;
loop
  xs: Integer;
  if x % 500 = 0 then
    p.ask := x;
    xs := p.answer;
  else xs := x * x; end;
  sys.out := text x + "^2=" + text xs + "\n";
  --- do more work
  j: Integer := 0;
  loop while j < 1000000 j += 1; end;
  x := x + 1;
  if x > 1500 then exit end end };
```

If the processes run in one memory space, *client-Cell* takes 145s to display all the squares. However, if the last two configuration files are used for scenario 2 and scenario 3 respectively as in case 1 above, *client-Cell* takes roughly 116s to finish. Tables 1 and 2 give a summary of the tests in *case one* and *case two* respectively.

Table 1: Summary of tests in Case one (Frequent Communications).

| Scenario | Execution time |
|----------------|----------------|
| Scenario one | 289s |
| Scenario two | 236s |
| Scenario three | 241s |

Table 2: Summary of tests in Case two (Infrequent Communications).

| Scenario | Execution time |
|----------------|----------------|
| Scenario one | 145s |
| Scenario two | 116s |
| Scenario three | 116s |

7 DISCUSSION

Communication granularity concerns the frequency and the size of messages sent from one process to another process within a system. In a system that uses fine-grained communication, small messages are sent by processes. This might be reasonable in environments where the processors are physically close

to one another (closely coupled environments e.g. multi-core architectures). In contrast to fine-grained communication, in a system that uses coarse-grained communication, few large messages are sent from one process to another process. This is most useful in a loosely coupled (processors are physically dispersed) distributed environment where processes perform large computations and send very few but large messages.

The approach described in this paper favours communication at any level of granularity. However, programmers are supposed to match granularity to hardware. Time measurements show that true concurrency improves performances.

8 RELATED WORK

Communicating Sequential Processes (CSP) is a formal language for specifying concurrent systems. Processes in CSP communicate over synchronous channels (Hoare, 2004). Channel simplifies synchronization of interactions between communicating processes. The π -calculus is a model of computation for communicating systems that have changing structures (Milner, 1980). This is especially useful for modeling systems that may be deployed on many different platforms. Channels may be sent from one process to another process to facilitate communication between processes within a system. Our approach has been influenced by CSP and π -calculus models.

Actor model (Hewitt, 1976; Hewitt and Baker, 1977) is an early attempt at formalizing concurrent systems. Here, *actor* represents the basic construct of computation. Actors interact by passing messages over asynchronous channels. An actor include a *continuation* in its message if the receiver of the message needs to send a reply. The receiver uses the continuation to direct its reply. SALSA (Varela and Agha, 2001) is actor-based language for programming dynamically reconfigurable open distributed applications.

Clear separation of program semantics from its distribution has never been the main goal of mainstream languages. Recently, Erlang has become popular for programming concurrent applications. However, network transparency in Erlang requires programmers to address messages to process ids; substantial code would have to be re-written when hardware environment changes. Mozart/Oz (Roy, 2004) separates the functionality of a program from its distribution thereby allowing the same programs to be executed on different architectures. Following Mozart/Oz, we believe the approach described thus

far will facilitate reconfiguration of systems when hardware environment changes.

9 CONCLUSIONS AND FUTURE WORK

It has been shown in this paper that a distributed environment may be more suitable for a system where processes rarely communicate. Frequent communications between the participating nodes in a distributed system may degrade the overall performance of the entire system. Clear separation of semantics and deployment of a program helps to adapt the program to different architectures. In practice, this may help to preserve software investment. As we have seen the feasibility of our approach, we hope implement other aspects of the project and conduct further tests on different architectures, including the multi-core architectures.

ACKNOWLEDGEMENTS

We are grateful to the faculty of Engineering and Computer Science, Concordia University for supporting in part the work described in this paper.

REFERENCES

- Andrews, R. G. (2000). *Foundations of Multithreaded, Parallel, and Distributed Programming*. Addison-Wesley.
- Armstrong, J., Viriding, R., Wikstrom, C., and Williams, M. (2004). *Concurrent Programming in ERLANG*. Prentice Hall, second edition.
- Bagrodia, R. (1989). Synchronization of Asynchronous Processes in CSP. *ACM Transaction on Programming Languages and Systems*, 11(4):585–597.
- Bal, H. E., Steiner, J. G., and Tanenbaum, A. S. (1989). Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322.
- Barnes, F. and Welch, P. (2003). Prioritized Dynamic Communicating and Mobile Processes. *IEE Proceedings - Software*, 150(2):121–136.
- Barnes, F. R. and Welch, P. H. (2004). Communicating mobile processes. In East, I., Martin, J., Welch, P., Duce, D., and Green, M., editors, *Communicating Process Architectures*, pages 201–218. IOS Press.
- Bernstein, A. J. (1980). Output Guards and Non-determinism in CSP. *ACM Transaction on Programming Languages and Systems*, 2(2):234–238.
- Brinch Hansen, P. (1987). Joyce - A Programming Language for Distributed Systems. *Software Practice & Experience*, 17(1):29–50.
- Buckley, G. N. and Silberschatz, A. (1983). An Effective Implementation for The Generalized Input-Output Construct of CSP. *ACM Transaction on Programming Languages and Systems*, 5(2):223–235.
- Dijkstra, E. W. (1968). Cooperating Sequential Processes. In Genuys, F., editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press.
- Grogono, P., Lameed, N., and Shearing, B. (2007). Modularity + Concurrency = Manageability. Technical Report TR E-04, Department of Computer Science and Software Engineering, Concordia University.
- Harris, T. and Fraser, K. (2003). Language Support for Lightweight Transactions. *ACM SIGPLAN Notices*, 38(11):388–402.
- Hewitt, C. (1976). Viewing Control Structures as Pattern of Passing Messages. Technical Report AIM-410, Department of Artificial Intelligence, MIT.
- Hewitt, C. and Baker, H. (1977). Actors and Continuous Functionals. Technical Report MIT/LCS/TR-194, Department of Artificial Intelligence, MIT.
- Hoare, C. A. R. (1978). Communication Sequential Processes. *Communications of the ACM*, 21(8):666–677.
- Hoare, C. A. R. (2004). *Communicating Sequential Processes*. Prentice Hall International, third edition.
- Lee, E. A. (2006). The Problem With Threads. *IEEE Computer*, 39(5):33–42.
- Milner, R. (1980). *A Calculus of Communicating Systems*. Springer.
- Olukotun, K. and Hammond, L. (2005). The Future of Microprocessors. *ACM Queue*, 3(7):26–34.
- Roy, P. V. (2004). General Overview of Mozart/Oz. Slides for a talk given at the Second International Mozart/Oz Conference (MOZ 2004).
- Salzer, J. H. (1965). M. I. T. Project MAC. Technical Report MAC-TR-16, Department of Artificial Intelligence, MIT.
- Schneider, F. B. (1982). Synchronisation in Distributed Programs. *ACM Transaction on Programming Languages and Systems*, 4(2):125–148.
- Silberschatz, A. (1979). Communication and Synchronization in Distributed Programs. *IEEE Transaction on Software Engineering*, 5(6):542–546.
- Sutter, H. (2005a). The Free Lunch Is Over - A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3). Available online at <http://www.gotw.ca/publications/concurrency-ddj.htm>.
- Sutter, H. (2005b). The Trouble With Locks. *Dr. Dobb's Journal*.
- Varela, C. and Agha, G. (2001). Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIG PLAN Notices*, 36(12):20–34.