PROCESS-ORIENTED PATTERNS FOR CONCURRENT SOFTWARE ENGINEERING

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY.

By

Adam T. Sampson

2008

Abstract

Concurrency is unavoidable in modern software development, owing to the increasing complexity of computer systems and the widespread use of parallel computer hardware. Conventional approaches to concurrency are fraught with danger: in particular, uncontrolled access to shared resources, poor scalability and the inability of the programmer to reason about the correctness of a program.

Process-oriented programming is a software design approach that offers solutions to many of these problems. A process-oriented program is constructed as a network of isolated, concurrent processes that interact only using synchronisation objects such as channels and barriers. Using techniques drawn from CSP and the π-calculus, design rules can be constructed that enable the programmer to easily build systems with known safety properties. Since process-oriented programs expose by their nature a high degree of explicit concurrency, they can be efficiently distributed across multiple processors and clusters of machines.

This thesis describes a *pattern language* for the engineering of process-oriented programs. Design patterns describe reusable, adaptable solutions to common problems that may arise during the design and development of a system. A pattern language serves the dual purposes of documenting the proven design solutions developed by a community, and providing a common technical vocabulary.

The patterns described in this thesis are drawn from a variety of existing process-oriented real-world applications, and have been used to construct new applications in fields such as embedded systems, multimedia processing, and complex systems simulation. While much of this work has been conducted using the occam-π programming language, the patterns—and the new language and library facilities they inform—are applicable to process-oriented systems built in any language.

Acknowledgements

I would like to thank the members of the Programming Languages and Systems group at the University of Kent, those involved in the TUNA and CoSMoS projects, and the Communicating Process Architectures community at large for the many interesting and fruitful discussions that we have had during the course of this work—and for their willingness to let me use their applications as case studies.

This work was supported by EPSRC: both directly through a research studentship (EP/P50029X/1), and through case studies from the TUNA (EP/C516966/1), CoSMoS (EP/E049419/1) and RMoX (EP/D061822/1) projects.

Contents

List of Figures

Chapter 1

Introduction

Concurrency is a fact of life for today's software developers. Many applications—such as those that communicate over a network, or provide a graphical user interface, or simulate complex systems of interacting agents—involve the modelling of sets of entities that perform actions and respond to events at the same time and in unpredictable ways. Solutions to these *naturally concurrent* problems cannot be expressed effectively using conventional sequential programming techniques.

At the same time, parallel computer hardware has entered the mainstream, with most users and developers now using fast, efficient multicore processors, and ever-growing scientific and business computing problems having moved from supercomputers to clusters of low-cost commodity machines. Programmers can no longer afford to ignore the need to express opportunities for parallel execution in their software—opportunities that writing their software in an inherently concurrent way would provide naturally.

However, concurrency is widely perceived as being fundamentally difficult, and to be avoided whenever possible. Conventional approaches to concurrent programming using the low-level facilities invented at the birth of parallel hardware suffer from well-known problems—such as uncontrolled access to shared resources, poor scalability, unpredictable composition, and the possibility of deadlock—that make it difficult to design, implement and maintain *correct* concurrent programs. As a result, most recent frameworks for programming parallel hardware attempt to hide the fact of concurrent execution behind high-level "data-parallel" interfaces—which make expressing simple problems easy, and complex real-world problems such as those above nearly impossible.

It is hardly surprising that concurrency is perceived as an advanced subject that should not be tackled except by highly-trained experts, and so most programmers never discover the advantages of concurrent design—that expressing the concurrency in a program can often make it *simpler*, while offering greater efficiency and enhanced opportunities for parallel execution.

Process-oriented programming is an approach to concurrent software development in which concurrency is not merely made explicit, but actively encouraged as a desirable property of software systems. Process-oriented programs are built from isolated, lightweight, concurrent processes that share no resources by default, and can be used directly to model entities in a problem. Processes interact only through carefully-defined communication and synchronisation mechanisms—for example, by passing

messages to each other along channels. This "boxes-and-arrows" approach to software design will already be familiar to multimedia programmers, hardware engineers, and others used to thinking in terms of interactions between isolated components.

Preventing processes from interfering with each other, and enforcing their correct provision and use of interfaces, means that processes can be composed in predictable ways, improving modularity and reuse; a process-oriented program is typically built as a hierarchical composition of interacting simple processes. The fundamental mechanisms of process-oriented programming have a basis in process calculi such as CSP, which enables mathematical reasoning about programs—and, more importantly, the construction of simple design rules that allow programmers to write programs that are guaranteed to be safe without needing to use formal methods themselves.

Many environments have been built to explicitly support process-oriented programming across a wide range of programming languages and platforms—including libraries for mainstream languages, and specialised programming languages that offer world-beating performance and enhanced static checking for process-oriented programs. However, many of the fundamental ideas of process-oriented programming can be applied to the construction of concurrent systems using other technologies; many concurrent environments now provide processes, channels, and other fundamentals of process-oriented programming.

It is important to note that process-oriented programming is not intended to be the solution to all programming problems: it is merely one tool in the programmer's toolbox, to be combined with other approaches as appropriate for the problem in hand. However, the process-oriented approach has already demonstrated its utility across a considerable range of applications, and new process-oriented programming environments are under active development.

Process-oriented programming offers the chance of an engineering approach to concurrent software development. The objective of this thesis is to identify and document a set of *design patterns* for process-oriented software: best-practice solutions to common problems that can be applied to a wide range of future concurrent software projects.

1.1 Roadmap

Chapter 2 describes process-oriented programming in more detail, including background material on the process-oriented approach in general, the basic facilities found in process-oriented systems, and a catalogue of environments for process-oriented programming.

Chapter 3 describes a number of real-world examples of process-oriented software, giving details on the contexts of their construction and the more interesting aspects of their design.

The design patterns identified from the case studies are drawn into a pattern language in chapter 4, which describes each pattern in a common format, with a discussion of its applicability and examples of its use in the case studies.

Chapter 5 discusses some possible enhancements to process-oriented environments to support specific patterns.

Some possibilities for the future of process-oriented programming environments—and process-oriented programming in general—are outlined in chapter 6.

Chapter 7 gives a "worked example" of the application of process-oriented design to an embedded system, showing how the pattern language can be used to design and reason about a complete application, and concludes with an overview of the outcomes of this work.

1.2 Conventions

The most common types of cross-reference between sections will be indicated as follows:

- ⋆ **Case Study** is a reference to a case study described in chapter 3;
- ▷ **Pattern** is a reference to a pattern described in chapter 4;
- ○ **Pattern** is a reference to a well-known pattern from a different programming style, many of which are described in a process-oriented context in section 4.5.

The definitions of many of the terms used in this work are listed in the index starting on page 198; the corresponding term will be marked in *italics* on the page referenced.

Most of the example code in this work will be given in an extended version of the occam-π programming language (section 2.3.2), a language designed specifically to support process-oriented programming. However, an attempt has been made to avoid the use of occam-π-specific facilities, in order to make it easier to translate the examples for use in other process-oriented environments. (The programmer will find that many of the examples can be expressed more succinctly in other languages; occam-π was chosen because it was the language of most of the case studies.)

For readers unfamiliar with occam-π, most language features operate as they do in other block-structured languages such as Pascal. The syntactic feature most likely to cause confusion is that "." is used in names as a separator with no special meaning, much as "_" is used in C-like languages. A line prefixed by "..." in a listing is a *fold*, representing a block of code that is only described by a comment rather than being shown; folds have been used to hide sections of code that are not relevant to the general patterns being demonstrated.

occam-π, being a testbed for process-oriented language features, is something of a moving target, and has changed considerably during the course of this work. The version of occam-π used is approximately that implemented by KRoC at the end of 2009, with a few extensions that have not yet been implemented: those described in chapter 5; single-branch `IF` blocks [195]; buffered channels, specified by `BUFFERED(n) CHAN` for `n`-buffering and `BUFFERED CHAN` for infinite buffering [196]; dynamically-sized arrays [243]; and the templating mechanism designed by Jim Moores [141], allowing `PROC`s to be parameterised by type. (Most of the code examples cannot therefore be compiled using currently-available occam-π compilers, although their translation to standard occam-π should generally be straightforward.)

Chapter 2

Background

2.1 Process-Oriented Programming

Process-oriented programming is a concurrent programming style derived from CSP and the occam family of programming languages, where systems are built by composing concurrent processes. The term was in use by 1992 [261, 78], although with varying definitions; process-oriented programming is therefore about twenty years old.

This section attempts to capture the fundamental ideas of process-oriented programming by looking at five important aspects of the process-oriented approach: concurrency, isolation, communication, composition, and reasoning.

2.1.1 Concurrency

Process-oriented programming is sometimes described as a technique for parallel programming, but it is more accurately described as a technique for *concurrent* programming.

Concurrency is a structuring tool. A concurrent programming environment allows a problem to be broken down into a set of activities and the synchronisation relationships between them, with the environment then performing the activities so as to satisfy the relationships.

Parallelism is a performance tool. A parallel program is one in which multiple physical resources (such as CPU cores) are used to perform several activities at the same time. A concurrent program may be executed in parallel—but it does not have to be; a concurrent program can be executed upon a single CPU by serialising or time-slicing parallel activities.

The primary advantage of concurrent programming is that many problems can be simplified by breaking them down into concurrent activities. For example, a webserver must deal with network connections from a large number of clients at the same time, and cannot predict the times at which requests will be received and response fragments acknowledged. A webserver written without concurrency needs to explicitly track the state of each of its network connections, resulting in complex code that is difficult to write and verify. Using concurrent programming, each network connection can be handled by its own activity: a simple program that only needs to deal with a single network connection.

This sort of problem has *irregular concurrency*: there are a number of activities that

must be performed, but the order in which they must be performed and the best way to *schedule* them across physical computing resources must be determined at runtime. In this case, this is because the webserver is responding to unpredictable external events (network requests), but irregular concurrency may also have internal causes—for example, chaotic interactions between agents in a scientific simulation.

This must be contrasted with problems that exhibit *embarrassing parallelism*, where the problem can be broken down immediately into a set of activities that may be performed in any order with no interactions between them. Data-parallel programming environments (such as OpenMP [152]) concentrate upon such problems. Rendering the Mandelbrot set is the classic example of an embarrassingly-parallel problem: each output pixel can be determined independently, so a Mandelbrot renderer with M CPUs can simply assign the Nth pixel to the N mod Mth CPU—this is called *geometric distribution* of work.

Expressing an embarrassingly-parallel problem like this in a concurrent programming environment is very straightforward. Furthermore, because the concurrent environment can make scheduling decisions while the problem is being solved, it can often do a better job than a simple geometric distribution. In this case, since different parts of the Mandelbrot set take different times to render, distributing work geometrically will result in some CPUs having more work to do than others; a concurrent programming environment can balance load more effectively at runtime, producing a result more quickly.

A secondary advantage of concurrent programming is, therefore, that the programmer does not generally have to worry about scheduling: it is up to the concurrent runtime to schedule activities across the available computing resources in the most efficient way possible. Concurrent programs get parallel execution—as far as possible—for free; the best way to improve the parallelism of your program is to express more possibilities for concurrency. Concurrent programming is an especially good fit for problems that have a high degree of *natural concurrency* such as simulations, games, network servers and user interfaces.

In process-oriented programming, these concurrent activities are called *processes* (section 2.2.1)—hence the term *process-oriented programming*, since a problem is decomposed into processes in much the same way that an object-oriented problem is decomposed into objects. "Process" is an awkward name for this concept, owing to confusion with operating system processes, business processes and development processes in many application areas—but we are stuck with it for historical reasons.

2.1.2 Isolation

While concurrent programming offers significant advantages, it is widely considered to be very difficult. Concurrent programs built using the low-level features provided by languages such as C and Java—threads, objects in shared memory, semaphores, monitors, and so on—often suffer from problems such as race hazards and lock order violations. As a result, many programmers try to avoid writing concurrent programs entirely—an approach that is rapidly becoming impractical in the face of the increasing popularity of multicore processors, GPGPUs and other parallel computing resources.

Proponents of process-oriented programming argue that this is wrong: concurrent programming is not fundamentally difficult—after all, humans deal with concurrency

in the real world all the time. The solution is to provide a higher-level set of concurrency facilities that prevent programs with these problems from being expressible.

Process isolation is one aspect of this solution. As many concurrency problems stem from unsafe concurrent access to shared resources, process-oriented programming environments use a *shared-nothing* policy: processes are isolated in memory from each other, only interacting through the synchronisation objects (such as channels; section 2.2.4) that they define explicitly. Isolation may be enforced by the compiler, by the runtime system, or (in environments where neither of the two previous options is possible) by programmer convention. Preventing processes from sharing memory means that the classes of concurrency errors caused by concurrent aliasing (where two processes access a shared object in an unsafe way) and incorrect locking are avoided.

2.1.3 Communication

If processes share no memory, they must instead interact by *message-passing*: when a process wishes to synchronise or share data with another process, it must communicate with it. The fundamental communication mechanism provided in process-oriented programming environments is the channel (section 2.2.2), which provides both synchronisation and communication of data between processes. Other synchronisation objects provide more flexible facilities for coordinating multiple processes.

Writing programs in terms of explicit communication makes process-oriented programming especially useful for distributed and non-uniform-memory systems: the semantics of interprocess communication can be implemented relatively easily over network links as well as through shared memory—although network interactions will have very different performance characteristics. Mobile data (section 2.2.5) allows local communication of data to be performed efficiently, falling back to copying for network links.

The synchronisation objects used by processes must be explicitly defined, and a process may use any number of synchronisation objects, with the ability to wait for events upon multiple objects—that is, a process sends a message to a channel, rather than directly to another process. This distinguishes process-oriented programming from message-passing styles such as the Actor model (section 2.3.4), where each process has a single "inbox" for messages. Distinguishing between processes and synchronisation objects gives process-oriented programming environments considerable flexibility: new types of synchronisation object can be defined, processes can operate without direct knowledge of other processes, and processes can choose which of their synchronisation objects they wish to pay attention to at any particular time.

The messages sent along channels are strictly defined by *protocols*, which describe the formats of messages and which messages are possible in a particular state. This allows the interfaces provided by processes (the sets of synchronisation objects and patterns of synchronisation that they use) to be precisely defined, and means that the code inside each process can be statically checked for conformance with the protocols it uses—preventing another class of programmer errors.

2.1.4 Composition

Process-oriented design is a compositional approach: systems are built by connecting processes together into a *process network*.

Processes may be nested. A process may internally be implemented as a network of subprocesses, meaning that the same design approach can be used at all levels of scale within a program. In general, a process-oriented program starts as a single *top-level process*, which is usually built as a parallel composition of subprocesses, some of which themselves will have subprocesses, and so on. The same approaches to design can therefore be used at multiple scales during the construction of a program.

To make this approach practical, processes' individual behaviours must not change when they are composed with other processes. This is achieved by isolation and process interface enforcement. Where possible, this enforcement is performed at compile time; this avoids the overhead of runtime checking, and the possibility of runtime error.

2.1.5 Reasoning

The primitive features of process-oriented programming environments are based on the facilities provided by process calculi—especially CSP [101, 102], but with increasing influence from the π-calculus [139]. It is therefore possible to automatically (or manually) derive a process-oriented program from a CSP specification, or vice versa, which makes it possible to reason formally about the behaviour of programs—either using manual techniques, or with automated model-checking tools such as FDR [84]. In particular, it is possible to automatically verify (in simple cases, for now) that a process-oriented program is a valid implementation of a CSP specification—which makes process-oriented programming an attractive approach for developing dependable software.

However, this is not to imply that you need to know CSP in order to take advantage of process-oriented design. Formal techniques can be used to derive *design rules* for process-oriented systems: constraints upon process behaviours and compositions that guarantee programs following the rules will have particular formal properties, such as freedom from deadlock (section 4.2.5). This makes design rules—and design patterns—powerful tools for process-oriented software engineering: following design rules allows the straightforward construction of correct concurrent programs. Furthermore, design rules provide further opportunities for static checking—process-oriented tools can ensure that design rules are followed (section 2.5.2).

In addition, the formal basis of process-oriented programming primitives means that carefully-designed process-oriented languages can be shown to have algebraic equivalence laws [188]. These can be exploited by compilers to give greater opportunities for optimisation—for example, by translating them into *fusion* rules that describe how processes may be rewritten to execute more efficiently with the same semantics—and by programming environments to support assisted refactoring of code.

2.1.6 POP and Other Paradigms

It is important to note that process-oriented programming need not be used in isolation: it is another technique for the programmer's toolbox, to be combined with other programming techniques as appropriate for the problem in hand. Process-oriented programming environments are available for object-oriented, functional and multi-paradigm languages (section 2.3). Distributed process-oriented systems can support communication between systems written in different languages (section 3.3.5).

A practical application may therefore use process-oriented programming as a *co-ordination* technique, with different processes implemented in different languages—and even with some processes simply being wrappers around existing non-process-oriented libraries. This allows greater flexibility in choosing the right language for the job, while ensuring conformance to common interfaces. Process-oriented programming is similarly useful as a technique for systems that make use of specialised processing hardware or hardware-software co-design.

2.1.7 Problems with POP

While there are many compelling reasons for using process-oriented programming, the approach has a number of limitations that should be taken into account.

Some of these stem from the imperfection of existing process-oriented programming environments. Many of these implement processes and communication in rather inefficient ways, limiting the degree of concurrency that can practically be expressed (section 2.3.1). Support for the basic process-oriented facilities varies considerably, with no single environment providing a complete set of fully-fleshed-out features (section 2.2). Static checking features are a particularly spotty area at the moment, with few environments providing anything beyond the very basics. Future advances in process-oriented environments should alleviate some of these problems—but this will necessarily be a slow process.

Other problems appear to stem from things that are fundamentally difficult within the framework of process-oriented programming. The most frustrating for beginners to process-oriented programming is usually the prevention—and, failing that, diagnosis—of deadlock, in situations where design rules cannot be applied. (One solution is to disallow designs that do not follow design rules entirely—but, in that case, we need to develop a wider catalogue of design rules!) Error and exception handling remains a weak area of the process-oriented approach, with techniques for dealing gracefully with exceptional conditions still under active development [103]. Tracing, debugging and tuning concurrent programs is awkward simply because of the inherent difficulty of collecting data from and visualising arbitrary process networks.

Nonetheless, the process-oriented style has been extremely successful across a wide variety of application areas—as the examples in chapter 3 demonstrate—and process-oriented languages and techniques are today showing their worth for the programming of multicore and distributed systems.

2.2 Basic Facilities

This section describes the primitive facilities of process-oriented programming: the features that are found in multiple process-oriented programming environments. They are not patterns, since they are always implemented the same way, and are implemented by the programming environment rather than by the user; they are the components out of which patterns are built.

For example, "Channel" is not a pattern in process-oriented programming; it would be a bad idea to implement a channel yourself when your environment provides efficient, correctly-implemented channels as a primitive. However, channels are considered to be a pattern in lower-level concurrent programming environments [156], and

languages that only have channels consider barriers to be a pattern [145]; mobile processes are a more recent example of the same kind of upwards migration [33].

Some of the patterns described later on may likewise become higher-level programming facilities in the future; for example, ▷ **Farm** is currently considered a pattern in process-oriented programming, but libraries such as Python's `multiprocessing` module allow work-distribution problems to be specified at a higher level, with the details of how the farm is implemented hidden entirely from the programmer.

2.2.1 Processes

In process-oriented programming, a *process* represents a flow of control. Processes are isolated: a process's internal state cannot be accessed directly by other processes. Processes usually interact only by using the synchronisation objects (such as channels and barriers) that have been made visible to them. Processes are therefore much like *active objects* in object-oriented programming—objects with their own flow of control, interacting with other objects only through the interfaces they provide [206].

A process differs from the usual object-oriented concept of an object in that it may perform actions on its own, rather than waiting until another object invokes an operation upon it. This kind of passive behaviour can of course be implemented using a process (▷ **Client-Server**)—with the advantage that a process can choose to ignore or defer certain operations if it is not ready to handle them. A process can be used to represent a piece of data that has behaviours and motivations—a "selfish gene" approach to computation [115].

Conceptually, it is useful to think of processes as being able to contain child processes. A process is said to have a child process when the lifetime of the child process is not allowed to exceed the lifetime of the parent (typically by having the parent wait for all its children to exit before exiting itself). Since it is common to implement a process as a sub-network of other processes, thinking of processes as being nested in this way makes it easier to visualise the structure of a system during design and debugging—although process-oriented runtime systems are not usually directly aware of parent-child relationships between processes.

In terms of implementation, a process is a lightweight thread: it has some private storage (called a *workspace* in occam, and often just a section of stack in other environments) and a processor context. Process-oriented runtime systems often make use of cooperative scheduling in order to obtain lower context-switch times and better performance, relying on the programmer to break up long-running computations with synchronisation operations—which, in practice, happens naturally when the use of concurrency is made sufficiently pervasive.

"Process" is a confusing name for reasons previously stated, and as a result different environments use a variety of names for the same idea—for example, Go calls its processes "goroutines"; many other languages call them something like "lightweight threads". In addition, occam uses the term "process" to refer to what here will be called *actions*: the primitive operations out of which an occam program is composed. `ALT`, `:=` et al. are examples of processes.

A process-oriented program typically starts off by running a single top-level process with some predefined connections to the outside world. In order to get useful work done, some mechanism is necessary to start more processes. Two such mechanisms are common in process-oriented environments.

```
PAR
  foo ()
  bar ()
  baz ()

-- foo, bar and baz are now complete
```

Figure 1: Parallel composition

Parallel Composition

One of occam's most striking features is that, unlike most programming languages, it does not assume actions should be performed sequentially by default; the user must explicitly specify `SEQ` or `PAR` for sequential or *parallel composition*.

`SEQ` takes a list of actions and performs each of them in turn; `PAR` takes a list of actions and runs them all at the same time in new parallel processes, waiting for all of them to finish before it finishes itself (figure 1).

`PAR` thus has essentially the semantics of CSP's parallel composition operator. Many other concurrent environments provide a similar primitive; for example, JCSP has a `Parallel` class, CHP has a CSP-like `||` operator, and OpenMP has `#pragma omp sections` and `#pragma omp for` [152].

This approach to parallel composition works well for relatively static systems in which the structure of the process network is known at design time—that is, roughly, those where the process diagram does not change while the program is running. In these cases, the process diagram can be directly translated into a set of nested parallel compositions. Classical occam used `PAR` not just to specify the process network inside a program, but also to specify the static placement of processes across a network of physical processors. `PAR` can also be used to parallelise an existing sequential loop, which is convenient for data-parallel problems.

For more flexibility, parallel composition can be combined with recursion. Recursive algorithms such as Quicksort can often be parallelised simply by replacing sequential composition with parallel composition.

Forking

In classical occam (section 2.3.2), parallel composition was the only mechanism provided for starting new processes. Parallel composition is convenient when parallel processes are run in batches—that is, groups of parallel processes tend to start and end at roughly the same time. In many applications, however, processes start and finish at unpredictable times; for example, network servers that spawn new processes in response to incoming connections, or simulations where processes represent agents that may reproduce and die. These applications can be implemented using a fixed-size pool of worker processes that sit idle until they are needed, but this is both awkward and inefficient. Concurrency in these applications is more easily expressed using *forking*.

occam-π provides forking as an alternative to parallel composition [20]. In occam-π, the `FORK` action takes another action as an argument; it creates a new process, running in parallel with the current process, that executes the provided action. `FORK` is thus a lower-level facility than `PAR`, mapping directly to the runtime system's internal

```
PROC network.server (LISTEN.SOCKET listen)
  WHILE TRUE
    SOCKET sock:
    SEQ
      accept.connection (listen, sock)
      FORK client.worker (sock)
:
```

Figure 2: Forking network server

```
REC PROC forker (CHAN ARGS in?)
  ARGS args:
  SEQ
    in ? args
    PAR
      child.process (req)
      forker (in?)
:
```

Figure 3: Simulating forking using recursion

primitive for starting a process.

FORK allows new processes to be started, but (unlike PAR) does not by itself provide a way of waiting for child processes to finish. This is provided separately by occam-π's FORKING facility. A FORKING block keeps track of processes that are FORKed inside it; all the forked processes must finish before the FORKING block itself finishes. A FORKING block thus provides a *forking context* in which new processes may be started; the context lasts as long as any of the processes contained within it are still running.

In occam-π, the entire program is implicitly wrapped in a forking context (allowing FORK to be used alone), but the user may specify arbitrarily-nested forking contexts of their own. At present, FORK creates the new process in the innermost forking context. For some applications, it would be convenient to specify a context explicitly as part of the fork operation, allowing processes to be forked in a context other than the innermost one [197]. An example would be a simulation in which an agent made use of an internal forking context to consider multiple strategies in parallel, but was also able to spawn new agents in the greater context of the simulation world.

With forking, it becomes possible to write programs that spawn processes at unpredictable times. For example, the "serve one client with each thread" pattern familiar from coarse-grained network server programming can be expressed easily using processes [125, 236] (figure 2).

It is theoretically possible to simulate forking in an environment which only provides parallel composition, provided that it also supports recursion (which classical occam does not), by writing a process which responds to each request sent on a channel by starting a new process in parallel with a recursive call to itself (figure 3). This process is an example of a ▷ **Factory**.

(In practice this would leak stack space, unless the environment were able to tell that the last action this process performed was a parallel recursive call to itself and reuse the process's existing stack; by analogy with tail call elimination, this could be

```
PAR
  foo ()
  bar ()
  baz ()
```

Figure 4: Parallel composition using `PAR`

```
FORKING
  SEQ
    FORK foo ()
    FORK bar ()
    FORK baz ()
```

Figure 5: Parallel composition using `FORKING`

called *tail parallel elimination*. This optimisation may be useful in an environment where parallel composition was widely used to implement recursive parallel algorithms; it is not common enough in occam-π to be worth implementing.)

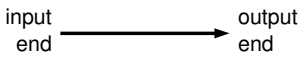
Forking along with forking contexts can be used to implement parallel composition; the programs in figures 4 and 5 are equivalent. Both start three child processes, then wait for them all to finish; in neither case is a guarantee made about which order the processes will actually begin to run in. (The knowledge that all three processes are being started at the same time in the parallel composition example may be useful to a runtime system that attempts to batch processes together at creation time [184], but a smart compiler could transform the second program to extract the same information.)

This duality means that many environments provide forking without parallel composition; for example, the X10 programming language provides `finish` and `async` primitives equivalent to `FORKING` and `FORK` [62]. However, many provide a forking operation without forking contexts; Go has only a `go` operation, with the idiom being to use channel communication to signal completion where necessary (which is awkward when many processes need waiting for) [92, 91].

2.2.2 Channels

In the most general sense, a channel is a connection between processes, along which messages may be sent. Channels are the most common communication and synchronisation mechanism provided in process-oriented environments.

The simplest form of channel is that provided by classical occam, where it is the only supported way for processes to interact. An occam channel is directional, with an input end, to which messages may be sent, and an output end, from which messages may be received. Each end may be used by at most one process; this is enforced at compile time by static checking.



A process attempting to send to a channel will be blocked until a corresponding process attempts to receive from the other end of the channel—and vice versa: a process attempting to read will block until a process writes to the other end. A channel is therefore a mechanism for both *communication* (passing data between two processes)

and *synchronisation* (coordinating the control flow of the two processes).

The messages that may be sent on a channel are defined by a *protocol*; the channel's type indicates what protocol is used for communication on that channel. A *simple protocol* contains a single data type (such as `INT`), meaning that the channel carries values of that type. occam also provides *sequential protocols* which describe messages consisting of several values (for example, a "coordinate" protocol might be defined as `INT; INT`), and *variant protocols* which allow choice between several different types of message based on an initial *tag*. (More flexible ways of defining protocols will be explored later.)

The semantics of occam channels are based on those of CSP events, which have the same synchronisation behaviour. A channel is modelled in CSP as a set of events, with one for each possible message that may be communicated across the channel. To send to a channel, a process engages in only the event corresponding to the value it wishes to send; to receive from a channel, a process offers a choice between all the channel's events, and infers the value that was sent by looking at which event completes. (Barriers (section 2.2.3) provide a more direct equivalent of CSP events in process-oriented languages.)

Because channels are so widely implemented, a number of extensions to this basic behaviour are provided by different implementations.

Buffering

Buffering is the most common extension to channel semantics, provided in some form by nearly every environment *other* than occam-π.

occam channels are *synchronous*: they have no internal buffer. Communication between two processes requires both processes to engage simultaneously upon the channel; once this occurs, the runtime system moves data directly from the sending process to the receiving process.

A *buffered* channel contains an internal buffer which can hold messages during communication. When a process sends a message to a buffered channel, the message is added to the buffer, and the process is immediately able to continue; the blocking behaviour of synchronous channels has been removed. When a process receives from a buffered channel, if a message is available it will be removed from the buffer; if no

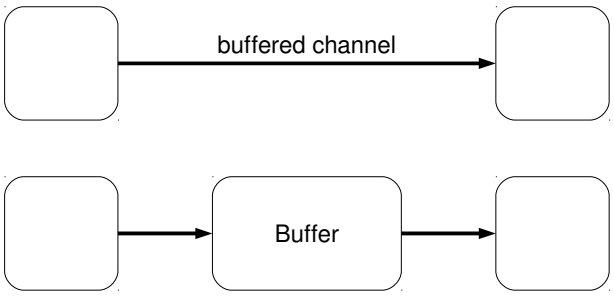


Figure 6: Simulating a buffered channel

```
CHAN SIGNAL uc:
BUFFERED CHAN SIGNAL bc:
PAR
  SEQ
    bc ! SIGNAL
    uc ! SIGNAL

  PRI ALT
    bc ? SIGNAL
      trace.sn ("1")
    uc ? SIGNAL
      trace.sn ("2")
```

Figure 7: Demonstrating buffering delays

message is present, the receiving process will be blocked until a message is sent.

Buffered channels may be simulated in an environment that provides only synchronous channels by using a buffer process (figure 6). See the ▷ **Buffer** pattern for more discussion of the different behaviours that buffers may have; environments with buffering often provide a choice of buffering behaviours for channels.

Note, however, the synchronisation behaviour of a buffer process may be subtly different from a real buffered channel, depending on how the semantics of buffered channel communication are defined. One possibility is that when an output to a buffered channel completes, the sending process can guarantee that the output end of the channel is ready—that is, a receiving process will certainly be able to read from it immediately. This is the *buffered channel guarantee*.

The effects of this are visible in programs that care about the ordering of messages received from different channels. For example, the program in figure 7 will always print `1` owing to the buffered channel guarantee—the buffered channel `bc` becomes ready before the unbuffered channel `uc`. (Even if the second process has not had a chance to run, when it is scheduled as a result of the `uc` communication, it must consider all the channels in the choice and realise that `bc` has become ready.) If the buffered channel is implemented by a conventional process, it may be possible for the message sent to `uc` to be delivered to the receiving process first. In order for a buffer process to provide the guarantee, it would need cooperation from the scheduler (for example, if it could ask to be run at higher priority than all other processes) [203].

For a more practical example, imagine replacing `uc` with a "tick" event that is being used to regulate transitions between timesteps in a simulation. Without the guarantee, the message may be received in a different timestep from the one in which it was sent.

Buffered channels only provide one-way synchronisation: the receiving process knows, when it receives a message, that the sending process sent it at some time in the past, but the sending process cannot tell anything about the control flow of the receiving process. To get two-way synchronisation when only buffered channels are available, it is necessary to use a second buffered communication in the opposite direction as an explicit ▷ **Acknowledgement**.

In practice, one-way synchronisation is often sufficient; for example, in the majority of ▷ **Client-Server** and ▷ **Pipeline** systems. This allows free choice between

synchronous and buffered channels for many applications; this is useful when constructing distributed applications, in which synchronous channels are most efficient for communications between processes on the same shared-memory host, because data does not need to be copied to and from a buffer, but asynchronous channels are more efficient for communications over a network, where the higher latency makes acknowledging receipt of each message very expensive [201].

A process can use a buffered channel to send a message to itself. For example, a process responsible for delivering network messages could accept new messages using a buffered channel, which would then represent the queue of messages which it has to deliver. When a message cannot immediately be delivered, the process could resend it to its input channel, which would place it on the back of the queue. When this technique is used, it is important to ensure that sending a message to the channel can never block (by using a discarding or infinite buffer, for example); otherwise the process would be in danger of deadlock. (This self-signalling technique is used to allow choice over signals in Unix programming, using a signal handler that writes a message to a non-blocking pipe [38].)

Sharing

Many environments allow a channel end to be *shared* between multiple processes. Shared channel ends are one of the extensions to classical occam provided by occam-π [21].

Either or both of the input and output ends of a channel may be shared. When the input end is shared, several processes may send messages to the channel. When the output end is shared, several processes compete to receive messages from the channel; each message will only be delivered to one process. (This is distinct from *broadcast channels*, which deliver the same message to several processes, with all processes engaging in each communication; see section 2.2.3.)

occam-π has *explicit sharing*: when a process wishes to communicate upon a shared channel end, it must first *claim* it using a `CLAIM` block. Inside the `CLAIM` block, the process has exclusive access to the channel end, and can use it for several communications before releasing it. Shared channel ends have a different type (`SHARED CHAN T`) from regular channel ends (`CHAN T`); inside a `CLAIM` block, the `SHARED` qualifier is removed. `CLAIM` blocks may not be nested, in order to prevent lock ordering problems.

Most environments, however, provide *implicit sharing*: any process that has access to a shared channel end may use it for communication at any time, without needing to claim it first. Indeed, in some environments—for example, Go—all channels are implicitly shared; there is no such thing as a point-to-point channel. If preventing sharing through static checking is not feasible, then making all channels implicitly shared allows simpler communication semantics than preventing unsafe sharing using runtime checks.

In practice, both types of sharing are useful in different circumstances. Explicit sharing allows a process to ensure that the communications it performs while the channel is claimed will not be intermingled with communications from other processes on the same shared channel. This often allows very simple channel protocols to be used—for example, the standard output stream in an occam-π program is simply a `SHARED CHAN BYTE`, and mixing of output from multiple processes is prevented by having them claim the channel while printing a message. Similarly, the channel bundles

used to communicate with server processes (▷ **Client-Server**) are explicitly shared; this allows a client to perform multiple operations atomically upon a server without fear of being interrupted.

However, there is frequently no need to prevent intermingling of messages from different processes—for example, when a shared channel is being used to collect the results from several worker processes (▷ **Farm**). In these cases, since there is no need for an explicit claim step before channel communication can take place, implicit sharing allows more concise code, and more efficient runtime implementation (since channel communication algorithms can be designed to allow implicit sharing [182]).

The effects of explicit sharing can be obtained by using an implicitly-shared channel to pass the end of a private communication channel (▷ **Private Line**).

Directionality

Pairs of processes that need to communicate in both directions with each other are very common—for example, clients that make requests to servers, and receive responses (▷ **Client-Server**). As a result, many environments support some form of *bidirectional channels*, which permit messages to pass along the channel in either direction.

While occam-π's channels are unidirectional, it does provide *channel bundles* (or *channel types*), which combine several channels into a single data type, allowing them to be allocated and their ends passed around as a group [34, 20]. Bundling together a pair of channels in opposite directions to allow bidirectional communication is by far the most common use of channel bundles in occam-π programs—although this is somewhat inconvenient since the protocols for the two directions must be specified separately.

If communication may happen in either direction over a channel, it does not really make sense to talk about input and output ends. Sing#'s bidirectional channels have "importing" and "exporting" ends. occam-π's channel bundles have "client" and "server" ends, reflecting their most common use. HTML5's bidirectional channels simply have "1" and "2" ends [238].

Runtime support for bidirectional channels is usually fairly straightforward to implement—channel communication algorithms are often inherently bidirectional—but few languages have really comprehensive support for bidirectional communication; in particular, facilities to enforce communication protocols over bidirectional channels are usually lacking, with Sing# being a notable exception (see section 5.3).

2.2.3 Barriers

Just as channels allow synchronisation between pairs of processes, *barriers* allow synchronisation among arbitrary numbers of processes.

A barrier keeps track of a set of *enrolled* processes. When an enrolled process *synchronises* upon the barrier, it is blocked until *all* the processes enrolled upon the barrier are also trying to synchronise. Once this happens, the barrier *completes*, and all the enrolled processes are allowed to continue. (This is essentially equivalent to a CSP event, with the enrolled set being the processes that have the event in their alphabet.)

Barriers are often used to provide a set of processes with a shared sense of time, by enrolling all the processes upon a barrier and having them synchronise at the end of each timestep (▷ **Phases**, ▷ **Clock**).

```
PROC barrier ([]CHAN SIGNAL users?)
  WHILE TRUE
    SEQ
      PAR i = 0 FOR SIZE users
        users[i] ? SIGNAL
      -- Now all users are blocked.
      PAR i = 0 FOR SIZE users
        users[i] ? SIGNAL
:

PROC sync (CHAN SIGNAL user!)
  SEQ
    user ! SIGNAL
    user ! SIGNAL
:
```

Figure 8: Simulating a barrier with a process

Processes may generally enrol upon and *resign* from the barrier after the barrier has been created, which makes barriers useful for managing dynamic pools of processes. Barriers are also used to implement forking contexts: the parent process creates a barrier and enrols upon it. It then enrols each child process on the barrier as it is started; as each child process finishes, it resigns from the barrier. Finally, the parent process synchronises upon the barrier. When the synchronisation completes, the parent can be sure that all the child processes have finished (since otherwise they would be enrolled on the barrier but not attempting to synchronise).

A very simple barrier can be simulated easily using a set of channels and an "oracle" process (figure 8; an example of ▷ **Acknowledgement**). Adding dynamic enrolment and support for choice between multiple barriers is not much more complicated [255]—although, in practice, real runtime systems implement barriers in a way that allows more efficient scheduling on multicore systems [184, 254].

Conversely, it is possible to implement a channel as a barrier, upon which the two communicating processes are enrolled, and a variable to store the value being communicated. To send to the channel, you write the value into the variable and then synchronise twice on the barrier; to receive from the channel, you synchronise on the barrier, read the value from the variable, then synchronise again. (The second synchronisation ensures that the sender does not overwrite the stored value before the receiver has a chance to read it; this is an example of ▷ **Phases**.)

One advantage of this approach—used by CHP, which in turn implements barriers in terms of transactional memory [51]—is that implementing choice between barriers gets you choice between channels for free, which is not the case for barriers implemented in terms of channels. Another advantage is that this approach can be easily extended to provide *broadcast channels*, in which messages are distributed to multiple receivers, simply by having more than one receiving process enrolled upon the barrier. (This matches the effect of using CSP events to simulate channels.) This approach is used in ⋆ **Occoids** to distribute control messages to multiple processes on the same host of a simulation.

A rarely-implemented extension is *partial barriers*, which complete "early" when

a fixed number of processes enrolled are synchronising, rather than all of them. For example, a 5-partial barrier would block synchronising processes until five processes were attempting to synchronise, at which point it would release just those five [241]. (Whether it immediately continues to collect another five, or waits for the first five to complete, depends on the implementation.) Partial barriers can be used in applications that wish to form fixed-size batches of work, or that have problems that require N resources from a pool—for example, allocating dedicated processing hardware to problems that require N processors to solve.

2.2.4 Synchronisation Objects

Channels and barriers are both types of *synchronisation object*: facilities that can be used to coordinate the control flows of multiple processes. Environments may provide additional synchronisation objects. For example, occam provides `TIMER`s, which can be used to synchronise processes with hardware clocks; JCSP provides buckets, which block a group of processes until another releases them [250], and CREW locks, which mediate access to a shared resource with concurrent-read-exclusive-write semantics [251].

Some common features in process-oriented environments apply to more than one type of synchronisation object.

Choice

The idea of *choice* between multiple events—waiting for one of several synchronisation objects to become ready—is common at the operating system level (for example, Unix `select` or Win32 `WaitForMultipleObjects`), but relatively rare in lightweight concurrency systems, perhaps because it can be difficult to implement in a way that is both correct and efficient. However, choice is one of the basic facilities of CSP, and is nearly ubiquitous in languages derived from it. Choice facilities in process-oriented languages are usually modelled on—or, at least, very similar to—those provided in occam.

occam's choice primitive is `ALT`, which takes a list of events (called *guards*) to choose between, along with actions to perform when one of them becomes ready. If more than one event is ready, the runtime system is free to choose arbitrarily between them. The `PRI ALT` primitive provides *prioritised choice*, which is the same except when more than one event is ready, in which case the events will be considered in the order they were specified.

If no event is immediately ready, the process will block. The primitive `SKIP`, which is always ready, can be specified as the last guard in a `PRI ALT` if the programmer wants to avoid blocking (see ▷ **Polling**).

How this translates into other environments varies; when a process-oriented environment is provided as a library for an existing language, it can be difficult to find a convenient syntax for choice. Many (such as JCSP) follow the `select` model, where a "choice" operation returns an index into a list of guards. CHP, on the other hand, provides a choice operator that can be used to choose between any arbitrary set of processes; everything is a guard.

In practice, most environments implement prioritised and non-prioritised choice the same way (that is, they always use prioritised choice); implementing prioritised

choice is generally more difficult than non-prioritised choice, but their runtime complexities are similar. Environments derived from Newsqueak (section 2.3.3) are exceptions in that they use a random number generator to actually make a random choice when multiple events are ready.

The downside of all choice being (potentially) prioritised is that the programmer needs to carefully consider the order in which events are presented for each choice. A process handling requests from several clients will give priority to clients earliest in its list; this may lead to clients later in the list being *starved* if early clients tend to be ready. A simple cure for this problem is to rotate the list by one place each time the choice happens (move the first element to the end of the list); the "fair `ALT`" idiom is common in occam, and in environments such as JCSP where choice is represented by a long-lived object capable of keeping track of a position in the list, a *fair choice* operation is often provided directly.

An alternative to choice—and another reason why many concurrent environments do not provide a choice operation—is to use shared channels. For example, rather than having N producers connected to a single consumer using N point-to-point channels, with the consumer making a fair choice between them, you could simply have all the producers sending messages to a single shared channel.

With this approach, the runtime system becomes responsible for ensuring fairness. In addition, the code is usually considerably simplified. The simplification is especially striking when the network topology is complex—for example, the grid of cells in a cellular automaton is very awkward to wire up using point-to-point channels (there must be a pair of channels in each of eight directions from each cell), but relatively straightforward when using shared channels (the ▷ **Location** pattern) [203].

Using shared channels is often more efficient, too; competing for a shared channel end is usually cheap compared to making a choice across many events (sending a message becomes $O(1)$ rather than $O(N)$ in CCSP [184]). The downside is that this approach only works for systems where all the producers send messages using the same protocol (i.e. all the input channels have the same type), and where the consumer never needs to refuse messages from any particular consumer; if these conditions are not met, choice will be necessary.

While a process-oriented environment may provide several different kinds of synchronisation object, it may not necessarily support choice across all of them. For example, occam-π provides channels, barriers and timers, but does not allow choice over output ends of channels, nor over barrier synchronisations. This is frequently a cause of frustration for occam-π programmers: there are many situations where having choice over output ends and barriers would be useful (for an example, see ▷ **Polling**).

One reason for this is that it would be possible to specify conflicting prioritised choices if choice could be made over more than one "end" of a synchronisation object (figure 9). The conflict here is over which of the two channels should become ready first; the priorities specified in the two processes are not consistent. This cannot be detected at compile time, and is unfeasibly expensive to detect at runtime. One approach would be to define the semantics of choice such that an arbitrary choice could be made when priorities conflict. An alternative would be to do away with `PRI ALT` and instead attach priorities to the synchronisation objects themselves rather than the choices; this would enable an unambiguous choice to be made in any situation.

Some environments provide *conjunctive choice*, where it is possible to wait for a combination of events to become ready—for example, "both A and B". This appears

```
CHAN SIGNAL c, d:
PAR
  PRI ALT
    c ! SIGNAL
      SKIP
    d ! SIGNAL
      SKIP

  PRI ALT
    d ? SIGNAL
      trace.sn ("1")
    c ? SIGNAL
      trace.sn ("2")
```

Figure 9: Conflicting prioritised choice

to have advantages in terms of composability of processes, but resolving conjunctive choice—especially over multiway events—can be very expensive; the consequences of adding conjunctive choice to a process-oriented environment are not yet fully understood [54].

Poison

Shutting down or resetting a process-oriented system has long been understood to be a problem—how do you ensure that all processes shut down safely, without causing deadlock? Many process-oriented environments provide *poison* as a solution to this problem. Poison was originally proposed as a pattern for use in occam [253], but it was pioneered as a library feature in C++CSP [57], and is now offered as a built-in feature in many process-oriented environments.

In systems supporting poison, any synchronisation object may be poisoned. Whenever a process attempts to use a poisoned synchronisation object, it will instead receive an exception. In response to this, it should poison all the other synchronisation objects it has access to, and then shut down cleanly. Poison thus spreads from process to process: poisoning any synchronisation object will result in the entire system being shut down.

As the programmer may not always want such a drastic response—they may want to only shut down part of the system, prior to restarting it—some mechanism is necessary to limit the spread of poison. For example, JCSP keeps track of the "strength" of poison applied, with each channel having an "immunity level"; this allows a subnetwork to be given a weak poison, isolated from the rest of the network by poison-filtering channels with higher immunity levels [220, 247]. An alternative would be to allow processes to be assigned to process groups, beyond which poison could not travel.

The major problem with implementing poison is choosing an appropriate response to receiving a poison exception, since it is not always straightforward to work out which synchronisation objects a process has access to. At present, programmer invention is usually required for each poisonable process; in the future, with better compiler support, it may be possible to determine poisonable objects automatically.

Extended Synchronisation

If a synchronisation object supports *extended synchronisation*, then it allows the programmer to specify an action that will occur immediately before it becomes ready, but before its readiness is signalled to other processes.

Extended synchronisation is implemented most widely for channels. When an extended input is performed on a channel, the action occurs while the sending and receiving processes are still blocked by the communication; neither can continue until the extended action finishes.

Extended channel communication can be used to write ▷ **Glue** processes that do not introduce an additional place of buffering. It can also be used to avoid the need for ▷ **Acknowledgement**: rather than receiving a request, performing an action and then sending an acknowledgement, a process can perform its action as part of an extended input, the completion of which signals the completion of the action. (This approach halves the number of synchronisations necessary in ▷ **Client-Server** calls that do not return a result.)

When an extended synchronisation is performed upon a barrier, the action runs after all processes are attempting to synchronise, but must finish before any of the processes are released. One use for extended barrier synchronisation is the coordination of multiple barriers in a distributed system: each host has a local barrier, with an extended action that performs a network barrier synchronisation. This allows processes on each host to do a single efficient local synchronisation, with the more expensive network synchronisation only occurring a single time once all the processes have synchronised.

The idea of extended rendezvous comes originally from Ada, where a task (a process) is allowed to execute a procedure in the context of another task while it is held safely in an extended rendezvous [112]—an approach to process interaction similar to the call channels proposed in occam 3 [34].

2.2.5 Mobility

Early process-oriented environments such as classical occam supported a very static programming style, in which the visibility of data and synchronisation objects was decided at the time of process creation. *Mobility* features in more recent environments allow more dynamic programs to be constructed, where the ownership of resources and the structure of the process network can change at runtime.

Many errors in concurrent programs stem from *unsafe aliasing*, where several processes have access to a shared resource that cannot safely be modified by multiple processes in parallel. Preventing unsafe aliasing is an important technique in the engineering of correct concurrent programs, and formal techniques exist to ease reasoning about aliasing; for example, ownership types extend type systems to track the contexts in which objects are visible [63, 42].

Single-Owner Types

Key to mobility is the idea of *single-owner types*, which allow the environment's type system to enforce that a resource may only be visible to one process at a time. The programmer can therefore use a single-owner reference as a token; holding the reference means you have the right to use the resource. If a resource cannot be safely used

by more than one process in parallel, it should be declared with a single-owner type; there will then only ever be at most one reference to the resource, which must be explicitly communicated between processes as required—meaning that creating aliases of the resource becomes impossible.

Channel communication for single-owner resources has special semantics: sending a single-owner resource over a channel results in the sending process losing its reference to it. In some environments, such as occam-π and C++CSP, the type system allows this to be enforced by the compiler. In others, such as JCSP, the programmer must ensure that the reference is not accidentally duplicated; the usual idiom is to set a local reference to `null` after it has been communicated away. (The same semantics apply in other situations; for example, forking off a new concurrent process that takes an argument of a single-owner type is usually semantically equivalent to communicating the argument's value to the new process over a channel.)

As single-owner resources must be able to be communicated between processes—ideally without the cost of actually copying the data they contain—they must be allocated in an area of the heap that is visible to multiple processes. Sing# calls this the *exchange heap*; objects are marked as single-owner by qualifying their type with `in ExHeap` [79]. occam-π calls this *mobile space*; objects are marked as single-owner with the `MOBILE` type qualifier [32].

If information stored in a single-owner resource needs to be used in two processes concurrently, it is necessary to make a copy of the resource. Most environments provide some facility for cloning objects; occam-π has the `CLONE` operator, for example, whereas in Java and Python the usual standard library cloning facilities may be used.

An alternative to copying is to provide a facility to transform (without copying) a read-write single-owner resource into a read-only multiple-reader resource that can safely be used by multiple processes. Similarly, facilities could be provided to divide a large single-owner resource into multiple smaller single-owner resources (and later recombine them). (Separation logic is a formalism that permits reasoning about the correctness of programs with resources that may be divided and combined in this way, and where resources' writability may change at runtime [180].)

Single-owner types permit a number of interesting compiler optimisations. For example, if the compiler knows that a particular process holds the only reference to a shared resource, it can choose to reuse the resource when it knows that it is no longer needed, rather than freeing it and allocating a new resource. Choosing data structures that permit this kind of reuse—for example, cooperating with the memory allocator to allow variable-sized objects to have their sizes rounded up—may offer performance advantages for programs making heavy use of single-owner types.

Mobile Objects

Mobile objects are those that can be communicated by reference between different processes. Process-oriented environments typically support a variety of mobile objects, with the simplest form being *mobile data* [32]—data objects that are only accessible through a single-owner reference. For example, mobile arrays are commonly used as drawing canvases that can be passed around safely among the various processes that make up a concurrent graphical environment, with each process drawing its part of the display before passing the canvas on [201].

Flexibility in the construction of process networks is provided by *mobile channels*,

which make channel ends first-class objects that can be passed between processes [20]. Unshared channel ends must by necessity be single-owner types, whereas shared channel ends can be aliased (since they may be used safely in parallel). Channel end mobility makes it possible to construct and rewire process networks at runtime. This is especially useful where the structure of the process network needs to change to reflect that of a system or data structure it is modelling—for example, the RMoX USB stack dynamically constructs a process network that mirrors the tree of USB devices connected to the computer [24]; process-oriented simulations construct connections between interacting agents using mobile channels [10].

Other synchronisation objects may similarly be made mobile; occam-π provides *mobile barriers*, with each mobile barrier reference representing a single enrolment to a barrier [254]. Enrolment to barriers in occam-π can therefore be controlled by the scoping of references to mobile barriers—a process enrolled on a barrier can pass its enrolment atomically to another process simply by communicating it across a channel. In order to enrol a new process on a barrier, the `CLONE` operator can be applied to a new barrier reference; the barrier will then not be able to complete again until the cloning process has handed off its new (duplicate) reference to another process to use.

occam-π's semantics for mobile barriers are still being refined; in particular, it is difficult at the moment to write a process which safely hands out barrier enrolments—such as a ▷ **Factory** creating enrolled processes—without being enrolled on the barrier itself. One possibility is to have separate types for the barrier and for its enrolments; the barrier could be used to create new enrolments without needing the enrolling process to itself be enrolled.

Some environments provide *mobile processes*, which make processes themselves into first-class single-owner objects [33]. To create a mobile process reference, a running process must *suspend* itself; the reference then represents the complete state of the suspended process, including any nested processes inside it, and references to synchronisation objects outside. An operation is provided to resume the process later. (Mobile processes can usually not be cloned, since they may themselves contain single-owner references.)

There is a duality between mobile process references and mobile channel ends. Rather than choosing to suspend, a process may instead choose to wait for a channel communication before continuing; the other end of the channel can then be passed around to control the process's resumption in the same way that the mobile process is. Mobile processes, however, are currently useful in distributed systems where the runtime system is not itself able to migrate processes between different hosts: a process can be explicitly moved to a different host by sending it there as a mobile process.

Implementing mobile processes requires considerable cooperation from the compiler in order to identify resources and nested processes held inside the process [163]. This is especially the case when mobile process references may be communicated between hosts in a distributed system—a problem comparable in complexity to that of migrating tasks in a distributed operating system [135]. Mobile processes can be simulated in environments that do not support them using the ○ **Memento** pattern [87]: when a process wishes to suspend itself, it should pack up its internal state into a data structure and then exit, and be able to read the data structure on startup to resume from the same point [201].

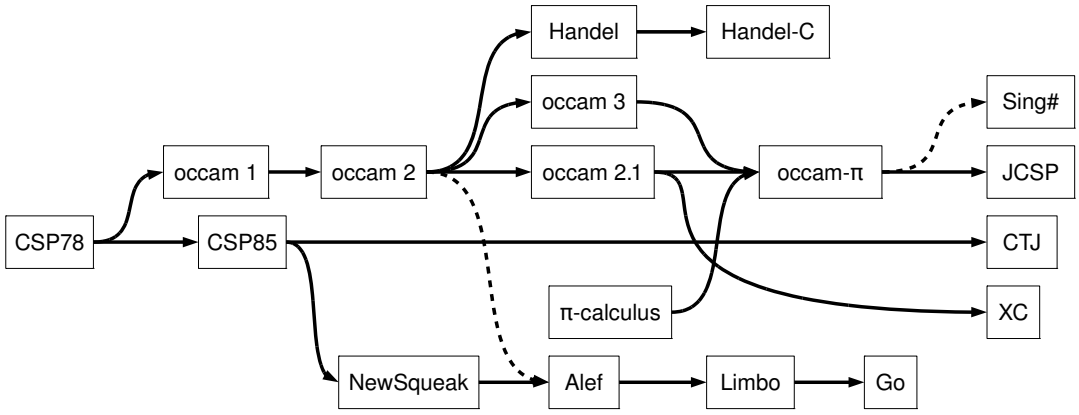


Figure 10: A family tree of process-oriented environments

2.3 Process-Oriented Environments

An *environment* is a context in which process-oriented programs can be constructed, consisting of a programming language, a set of libraries, a compiler, and a concurrent runtime system. The descriptions here will concentrate upon environments with features specifically designed to support the process-oriented style, but some form of process-oriented programming is possible in many concurrent environments owing to the wide availability of basic features such as channels. A variety of such environments exist (see figure 10, which is neither complete nor to scale in terms of time, and figure 11); this section describes several of the most influential or interesting.

A programming language may (occam) or may not (Java) have features that explicitly support process-oriented programming. Some modern languages (such as Haskell) are sufficiently expressive that these features can be provided by a library as an "embedded domain-specific language" [109], with most of the advantages of providing them as language features directly. As such, an additional distinction will not be drawn here between environments with languages that do and do not have process-oriented features, although unusual language features that would be good candidates for EDSL features in more expressive languages will be noted.

Some process-oriented techniques can be applied even in environments that support none of the usual process-oriented synchronisation features. For example, the ▷ **Client-Server** pattern describes a set of formally-proven design rules for interactions between "passive" processes that have OO-object-like behaviour; a concurrent client-server system can be built using objects rather than processes and the design rules still hold (see section 3.9.4 for an example).

2.3.1 Implementing Processes

An ideal process-oriented programming environment would allow the programmer to have as many processes as they like, with the cost of communication between the processes being zero. This is, of course, impossible. A more practical goal is to make process-oriented programming directly comparable to object-oriented programming:

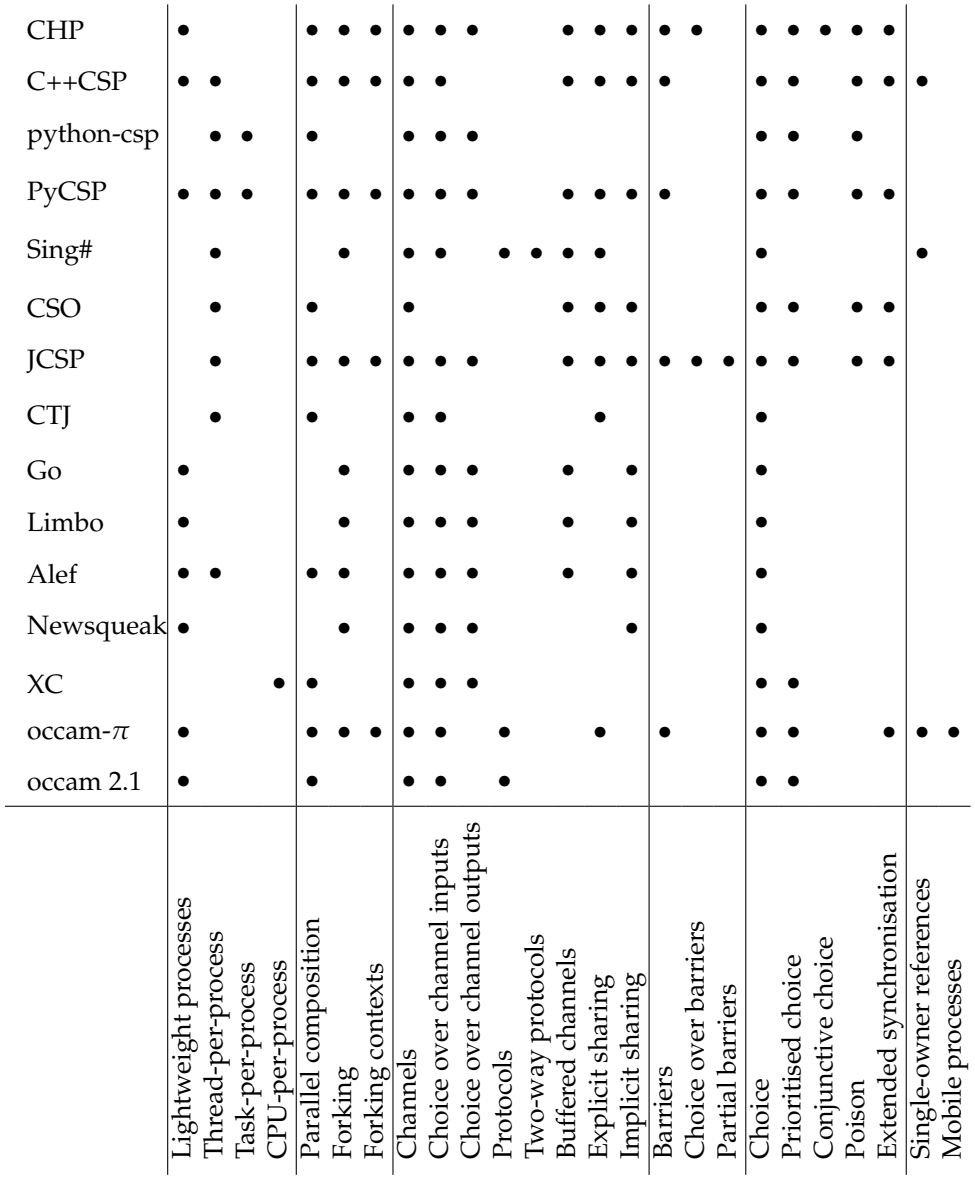


Figure 11: Features provided in process-oriented environments

spawning a process should be as cheap as creating a new object instance, and communicating between two processes should be as cheap as calling a method. Practical process-oriented programs can make use of very large numbers of processes; for example, the ⋆ **Occade** demos use thousands of processes, and some of the ⋆ **TUNA** and ⋆ **CoSMoS** simulations hundreds of thousands.

Process-oriented environments implement processes in a number of different ways. The most common approach is to use operating system threads to represent processes, and allow the operating system to schedule them across CPUs—the *thread-per-process* approach. However, operating systems usually place limits on the number of threads that a program can have (typically no more than a few thousand), the memory overhead per thread is usually on the order of a few kilobytes, and the cost of switching between threads is significant, making communication inefficient.

High-performance process-oriented environments such as occam-π, Go and CHP therefore use *lightweight processes*, where the runtime system manages the state of each process, scheduling processes explicitly across a pool of operating system threads to achieve parallel execution. Lightweight processes usually use *cooperative scheduling*, where a process cannot be interrupted until it calls into the runtime system. (Operating system threads, by contrast, are usually *preemptively scheduled*; the operating system can forcibly deschedule a thread if it has been running for too long.) Cooperating scheduling allows very low per-process memory and CPU overheads; the CCSP runtime system for occam-π leads the pack here, with 8 machine words of memory overhead per process and communication times in the tens of nanoseconds [184]. Communication is cheap because sending a message simply requires a context switch to the receiving process.

A few process-oriented environments use a *task-per-process* approach, where each process is represented by an operating system process. This makes communication very expensive and places severe limits on the number of processes possible—but allows operating system memory protection to be used to isolate processes from each other. This approach is useful when an application makes use of unreliable external libraries, since a library call that causes the operating system process to crash can be detected and contained. It is possible to consider the Unix operating system as an example of a task-per-process environment, since Unix provides processes, pipes (buffered byte channels) and choice via the `select` system call. The python-csp library makes use of Unix facilities in this way to allow robust process-oriented programming [143]. The RMoX and Singularity operating systems can also be considered task-per-process environments, since their tasks *are* lightweight processes.

A final approach is to provide dedicated hardware for each process—the *CPU-per-process* approach. Historically this has been used by systems that compile a process-oriented language down to programmable hardware such as an FPGA, with Handel-C being a notable example; while it offers very high performance—especially in terms of latency—the downside is that even more severe limits are placed on the number of processes possible. The XMOS XS1 architecture (section 2.3.2) is designed specifically to support the CPU-per-process approach: each XS1 CPU core supports eight hardware threads, with multiple cores per die allowing the construction of small (but practical) process-oriented systems that effectively have a dedicated CPU for each process [134]. This architecture is designed to combine the flexibility of process-oriented software design with the exceptionally low latency of FPGA hardware.

2.3.2 occam

The occam family of programming languages have been the most influential process-oriented environments to date. occam development has proceeded in two distinct phases, the first—which is usually referred to as *classical occam* —in the commercial sector, and the second in the research sector.

Classical occam

The occam family consists of block-structured, indentation-delimited procedural languages in the Algol tradition. occam started life as the only supported programming language for the INMOS Transputer, a concurrent microprocessor with a stack-based architecture and hardware support for concurrency. Not only did the Transputer provide dedicated hardware for scheduling, it provided *links* that allowed multiple Transputers to be connected into a larger system, with channels carried efficiently between chips. At launch time, a single Transputer provided world-leading computational performance; by using links, supercomputer performance could be obtained at budget prices.

occam 1, developed with the Transputer in the early 1980s, was primarily inspired by the original 1978 version of CSP [101]. It provided synchronous channels that carry single values (`CHAN`), with input and output communications (`c ? x` and `c ! x`); choice between input communications (`ALT`); and parallel composition (`PAR`) [133]. occam 1 was otherwise a simple, untyped language, with all values and variables being machine words.

occam 2, released in 1987, added static typing of values and variables, compiler-checked protocols for channels, and a safe-but-powerful static reference system ("abbreviations") [77]. occam 2 was therefore one of the first languages to support protocol descriptions for lightweight concurrency; contemporary languages with interprocess communication only supported unstructured messaging. Later versions of the occam 2 compiler introduced static checking for unsafe aliasing of data and channel ends.

occam 3 was a much more ambitious proposal, with a draft specification released in 1992 [34]. Aside from numerous enhancements to the procedural aspects of occam 2, occam 3 provided channel bundles, shared channels, and call channels, which made calls to server processes look and behave like procedure calls. occam 3 was never implemented, but some of its features were later incorporated into occam 2.1 and occam-π.

occam 2.1 was a 1994 incremental update to occam 2, with extensions to the type system to incorporate user-defined and record types, and with minor improvements to the syntax [214]. occam 2.1 was the last version of occam designed by INMOS, by then part of SGS-Thomson (now STMicroelectronics). By this point, the Transputer was no longer competitive in terms of performance with general-purpose processors, and remaining Transputer users were moving to other programming languages for ease of porting.

The Transputer architecture survives today as the ST20 line of microcontrollers, widely used in GPS receivers and set-top boxes—but stripped of most of its concurrency facilities, and programmed using C rather than occam. Further development of occam was to take place in the research sector.

occam-π

occam's formal basis and elegant syntax—and the ability to get serious computing done for little money with the Transputer—aroused considerable interest among computer science researchers.

As the viability of the Transputer waned, occam users in industry and academia started to look for ways to continue using the language upon other hardware. Successful attempts to port occam to new architectures included the Southampton Portable occam Compiler (SPOC), which generated portable C code from occam and used the ω-test to perform static checks much more efficiently than INMOS's compiler [68], and Michael Poole's tranpc, which translated the output of an extended version of INMOS's compiler into IA32 instructions [173].

The "occam For All" project ran from 1995 to 1997, with academic and industrial partners aiming to enhance and portably implement the occam language. This work led, eventually, to the language now called occam-π, which extends occam 2.1 in many ways to make it a more useful modern programming language. Notable additions include: a foreign function interface [264, 72]; user-defined operators [141]; mobile data [141, 246]; channel bundles with mobile channel bundle ends and sharing, process priority, forking and extended rendezvous [27, 20]; result abbreviations, nested protocols and recursion [21]; barrier synchronisation [29]; mobile processes [33]; and transparent networking [210, 209]. Work on occam-π continues; features pioneered in occam-π have since been implemented in other process-oriented environments.

occam-π has multiple implementations. The INMOS occam 2.1 compiler, occ21, has been incrementally extended to support the occam-π features. occ21 generates an extended version of the Transputer instruction set, which is translated into IA32 code by tranx86 [30]. Runtime support for occam-π is provided by the CCSP library, which offers world-class scheduling and communication performance for process-oriented programs on multicore CPUs [184]. (CCSP may also be used by C programs through the CIF interface [31].) For non-IA32 architectures, and especially for mobile and embedded devices, the Transterpreter provides a portable, lightweight, interpretive runtime for extended Transputer code as an alternative to the tranx86-CCSP route [116].

Since the Transputer architecture is a poor match for modern CPUs, the tranx86 approach has performance limitations for straightline code; in addition, occ21 is written in C, and is now unreliable and very difficult to maintain. Development began in 2007 on Tock, a new occam-π compiler, written in Haskell as a replacement for occ21 [202, 199]. Tock uses a nanopass approach for flexibility: new language features can be implemented by slotting new passes into the compiler [204]. At the moment, Tock generates portable C code that uses CCSP for scheduling; in the future, it may use LLVM to output optimised native code directly. Like SPOC, Tock uses the ω-test for efficient usage checking and more helpful error messages. (As a side benefit, the development of Tock has yielded useful new strategies for generic programming in Haskell [55, 56].)

While occam-π has a great number of features to support parallel programming, these have been added to occam in a somewhat ad-hoc fashion over the last twenty years, and as a result are sometimes inconsistent or awkward to use. Section 6.2.1 discusses future options for the development of occam-π.

Embedded Concurrency

While inherently parallel processors—conventional multicore CPUs such as modern IA32 chips, heterogeneous multicore CPUs such as the Cell, massively-parallel graphics processing units, and embedded CPUs such as Parallax's Propeller—have become very common over the last few years, most of these provide little direct support for message-passing concurrency. Two approaches to embedded concurrent processing have low-level support for the process-oriented model, however, both of which are part of the occam tradition (although neither use the occam syntax).

Handel-C is a language for the design of field-programmable gate arrays, aimed at the same market as Verilog or VHDL; now sold as a commercial product by Mentor Graphics, it derives ultimately from Ian Page's work on compiling occam code to FPGAs [161]. Handel-C provides a subset of occam 2's features, with a C-like syntax and a number of extensions to allow access to the built-in features of a wide variety of FPGAs [3]. The advantage of Handel-C over other FPGA design languages is that it is much more of a conventional programming language, which simplifies hardware-software co-design—and the advantage of the occam approach over languages such as ANSI C is that parallelism, which is inherent in FPGAs, can be easily expressed and statically checked to be safe.

The XC programming language is a more recent development. XC also offers extended occam 2 features with a C-like syntax [240]; it is being developed by XMOS, a fabless semiconductor company developing high-performance CPUs for embedded devices. (The name XMOS is a deliberate reference back to INMOS, with a number of ex-INMOS staff being involved with XMOS.)

XMOS's first line of CPUs is the XS1 architecture, with CPU cores that feature support for up to 8 hardware threads along with extremely flexible low-level input-output facilities [134]. The target market is the area where high-end embedded CPUs and FPGAs overlap, with the XS1 combining the flexibility of software with the hardware interfacing abilities of FPGAs. Initial CPUs in this line offer between one and four CPU cores per chip, with fast interconnections between them, and links provided for joining multiple chips together (in much the same way as the Transputer's external links). CPU instructions are provided for asynchronous messaging between threads, with a uniform addressing scheme allowing communication between threads on the same core, on different cores, and on different chips connected by external links. The XC language builds occam-style synchronous channels using these low-level asynchronous operations, with compiler optimisations used to minimise the number of messages sent when multi-step protocols are used.

2.3.3 From Newsqueak to Go

The most visible recent process-oriented environment has undoubtedly been Google's Go language—but Go is actually the latest in a series of CSP-inspired languages that developed in the US, in parallel to (and, sadly, with little influence from) the European occam tradition.

Newsqueak, Alef and Limbo

Newsqueak was developed at Bell Labs in the early 1990s as a concurrent, procedural language for writing graphical applications [168]. Newsqueak's concurrency facilities were inspired by CSP, and are extremely simple. The `begin` statement spawns a new concurrent process. The `chan` type represents a strongly-typed, synchronous channel. The communication operator <- serves for both input and output, depending on whether it is used prefix (`<-c`) or infix (`c<-v`). The `select` statement provides choice between multiple possible communications. When multiple communications are possible, a random number generator is used to select one. Newsqueak's other unusual feature is the `become` statement that allows manual tail recursion elimination—which is useful for writing processes in a CSP style.

Alef was developed as a C-style language for systems programming on Bell Labs' Plan 9 operating system [262]. In terms of concurrency, Alef distinguishes between *processes*, which are preemptively scheduled, and *tasks*, which are cooperatively scheduled. Alef's concurrency facilities derive from Newsqueak's, with some additions and name changes to correspond more closely to occam: choice is performed by `alt`, variant protocols are supported, and a `par` statement provides parallel composition. Later, the `libthread` library for Plan 9 was provided to provide Alef-style concurrency for C programs [36].

Limbo was the standard programming language for the virtual-machine-based distributed operating system Inferno, developed from Plan 9 and Alef [65, 181]. Limbo's concurrency facilities are based on those of Alef's, with the removal of tasks, since Inferno's underlying processes are sufficiently lightweight to use directly, and the removal of the `par` construct. Extensions include implicit choice over arrays of channels when used for input, and support for guards in `alt` statements.

Go

Development on the Go programming language began in 2007, by a team at Google including several researchers who had worked on languages in the Newsqueak tradition. Go attempts to address several perceived problems with existing programming languages, especially the inflexibility of the static type systems commonly used for object-oriented development and the lack of good support for concurrent programming [92].

Go's model of concurrency is derived from the Newsqueak tradition—indeed, of the languages above, it is closest to Newsqueak itself. Go encourages programmers to use mobility to manage access to shared resources: "Do not communicate by sharing memory; instead, share memory by communicating." [91]

Go derives its name from the `go` statement, which spawns a new process. Processes are lightweight and cooperatively scheduled. Supporting hundreds of thousands of processes is an explicit goal of the Go runtime system; processes' stacks start at a minimal size, and grow dynamically as necessary.

Channels carry any Go type, and may be synchronous or N-buffered, with the <- operator used for both input and output. <- may be used in a "non-blocking" polling mode with an extra boolean return value, where it returns immediately with an error value if communication would block. Channels may be explicitly closed, and tested for closed-ness. Once a channel has been closed, sending to it will block, and reading

from it will return a default value defined for the type the channel carries. Channel ends are first-class, allowing channel end mobility, and all channel ends are implicitly shared.

`select` provides choice between multiple communications, with a random choice being made when several channels are ready. The language specification describes this as a "uniform fair choice" [92], which is inaccurate; without control over the order of choice, no guarantee of fair service in processes using multiple `select`s can be made. Both input and output communications may be used. (The occam problem with output guards does not arise, owing to the lack of prioritised choice.)

Go's facilities for error handling are rather limited—much like in occam, with no support for exceptions; a `panic` facility is under development to allow failing processes to be detected. No equivalent of channel poison is provided.

Go does not provide any way to wait for a process to finish, other than by explicitly communicating with it—that is, it has no equivalent of occam-π's `FORKING` or even Alef's `par`. Idioms for waiting for process completion using channels are a common topic of discussion on the Go mailing lists; this suggests that adding an equivalent of `FORKING`—or, at least, support for barriers—would be a useful extension to Go.

Go provides essentially no static checking, other than type safety. The language is designed so that actions that would be errors in an occam-like language (such as having two processes write to a channel) have well-defined effects in Go; along with this, the lack of protocols makes it hard to check the correct use of higher-level concurrent patterns in a Go program.

Nonetheless, Go is an extremely promising language, with many similarities to occam-π. Go—and thus process-oriented programming—is already being applied to a variety of real-world applications.

2.3.4 Erlang

Erlang is not a process-oriented programming language. Instead, it uses the *Actor model* of concurrent computation, which Erlang programmers call *concurrency-oriented programming*. The Actor model is similar to the process-oriented approach in that a program is composed from isolated processes, executing concurrently, and communicating by passing structured messages [80]. However, in the Actor model, messages are sent asynchronously, and addressed directly to particular processes—rather than being sent synchronously along channels. This approach was widely used in concurrent programming environments of the 1970s and 1980s, and was influential upon the development of object-oriented programming through Simula and Smalltalk (section 6.4). More recent languages that support the Actor model include Io [114] and Scala (section 2.3.5)—but Erlang is certainly the most successful contemporary Actor-based language.

The development of the Erlang programming language began in 1985 at Ericsson AB, as a high-level concurrent programming language for the development of reliable telephony systems [14]. Erlang apparently evolved without influence from process-oriented environments; the use of explicit communication channels was considered early on but rejected [15].

Each Erlang process has an incoming message queue (a *mailbox*), to which messages may be asynchronously sent by other processes. The `receive` statement lets a process retrieve messages from its queue that match provided patterns, with messages that

do not match the patterns remaining in the queue for later processing. A process can therefore handle messages in an order different to that in which it received them. The Erlang designers chose this mechanism because it was a close fit to the CCITT standard languages used to specify communication protocols, and it allowed a process to handle messages from several other processes without knowing which order they would be received in. In a process-oriented environment, this kind of problem would be solved using either choice or parallel composition.

For high-availability systems, Erlang provides error-handling facilities that allow programs to be robust against both software and hardware failure. Related processes may be explicitly `link`ed together. When a process encounters an exceptional condition, it sends an "exit" message to any other processes it is linked to. *Normal* processes will themselves exit upon receiving an exit message, causing clean shutdown of all related processes. *System* processes can trap the exit message and recover from the failure—for example, by restarting the failed processes. Erlang designer Joe Armstrong says that "As far as I know, no other programming language has anything remotely like this" [15]—but this is very similar to poison (section 2.2.4), albeit somewhat easier to use; explicit linking would be a useful addition to process-oriented environments.

Erlang-style mailboxes, with out-of-order pattern-matching reception, could be integrated as a new type of synchronisation object into process-oriented environments, with implementation and semantics being very similar to those of buffered channels. This would allow Erlang's patterns for constructing high-reliability systems to be used in process-oriented programs—and allow Erlang programs to take advantage of high-performance process-oriented runtime systems.

2.3.5 Java

Sun's Java programming language is used widely in industry, research and teaching; compiled to bytecode for a virtual machine, considerable research has gone into making it execute efficiently across a wide range of platforms. As a result, there has been considerable interest in supporting the development of concurrent applications using Java. Java's built-in facilities for concurrent programming were, prior to Java 5, primitive and difficult to use: threads (which could be operating system threads or cooperatively-scheduled "green threads", depending on the implementation) and monitors. As a result, even Sun recommended against the use of threads except where absolutely necessary: "If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug." [225]

Nonetheless, several Java packages were built on top of these primitives to provide higher-level concurrency facilities—including the CTJ [98] and JCSP [244, 247] packages for process-oriented programming. In both of these libraries, processes and synchronisation objects are modelled as classes, and the interface attempts to mirror existing Java concurrency practice as far as possible; for example, a process is a subclass of `Process` that implements the `Runnable` interface.

CTJ concentrates upon real-time systems, with extensive support for process and channel priorities, and predictable approaches to scheduling. JCSP provides a wider range of facilities for general-purpose Java programming—including all the process-oriented programming features of occam-π, and a number of further extensions, along with support for the development of distributed applications and a cut-down version

for mobile devices. JCSP has been used successfully to prototype new types of synchronisation objects [242]; while it offers significantly lower performance than the CCSP runtime used for occam-π, experimentation is easier.

The poor state of Java concurrency primitives was rectified somewhat in Java 5, with the introduction of the `java.util.concurrent` package. Its developers intended to address the problems above by standardising a library of well-tested, high-level concurrency facilities [128]. `java.util.concurrent` includes a number of classes that can be used directly as process-oriented synchronisation objects: `SynchronousQueue` is a synchronous channel, `ArrayBlockingQueue` a buffered channel, `CyclicBarrier` a barrier and `CountDownLatch` a bucket. (It also includes other facilities that could be provided as synchronisation objects: for example, `Exchanger`, which allows two processes to atomically swap references, which we would normally implement as a two-process ▷ **Ring**.)

More interesting, though, is the provision of a range of portable atomic operations that can be used to implement new, efficient, lock-free concurrency facilities; these translate into new operations added to the Java virtual machine specification for Java 5. Future versions of JCSP will make use of these operations for more efficient synchronisation. What Java still lacks is an equivalent of coroutines or continuations (a way of "descheduling" a running process); this makes implementing lightweight scheduling in Java very difficult, requiring bytecode manipulation [163, 191].

The popularity of the Java virtual machine means that it is now used as a compilation target for other languages as well; compiling to the JVM means that you can take advantage of the wide range of existing Java packages. Scala is a popular multiparadigm language that runs on the JVM, with a powerful Haskell-like type system and a more succinct, expressive syntax than Java [149]. There is considerable interest in concurrency within the Scala community. Scala comes with an `Actors` library that provides support for lightweight processes using the Actor model, and the Communicating Scala Objects library follows the JCSP model to directly support process-oriented programming within Scala [224].

2.3.6 C#

The C# programming language is Microsoft's competitor to Java, executing through bytecode on the Common Language Runtime. The CLR was designed to address many of the shortcomings of the JVM, including having better support for multiple languages. As with Java, C#'s built-in concurrency facilities are fairly limited, and there is considerable interest in providing higher-level abstractions for the construction of concurrent programs. The Jibu library provides JCSP-style process-oriented programming facilities for C# [121]—but there are several more interesting approaches to concurrent programming that have been prototyped as C# language extensions.

The Cω language (originally Polyphonic C# and X#) extends C# with features from the join-calculus: asynchronous methods and chords [37]. A method declared as asynchronous causes a new process to be spawned to execute its body, running concurrently with the calling process (as if the method call had been forked, in process-oriented terms). A chord is a set of method headers, all of which must be called at the same time (either in different processes, or asynchronously) for the method body to execute. A synchronous channel in the join-calculus can thus be implemented as a chord of `send` and `receive` methods; the processes calling them will only be able to continue once the rendezvous is complete.

If method calls are thought of in terms of message-passing, then an asynchronous method is simply one that does not require an acknowledgement, and a chord is a conjunctive choice across messages. Cω is not a process-oriented language, but it would certainly be possible to implement useful process-oriented facilities in terms of asynchronous methods and chords.

Sing# *is* a process-oriented language: it extends C# with facilities for systems programming on Singularity [79]. Singularity is a process-oriented operating system, with "software isolated processes" communicating over channels [111]. Sing# makes use of mobile data (the "transfer heap") for efficient communication between processes. Interfaces between Sing# processes are defined using contracts, which are bidirectional channel protocols (section 5.3.2); code conformance to contracts is statically checked by the Sing# compiler.

2.3.7 Python

The Python programming language was designed as a practical programming language with sufficiently clean syntax and semantics to be useful as a teaching language. It has a Smalltalk-style dynamic type system, and indentation-based syntax. There are multiple implementations of Python with different performance characteristics and sets of facilities: the CPython implementation, which runs Python programs on top of its own interpretive virtual machine, is the most common, but there is also Jython on top of the JVM, and IronPython on top of the CLR. The Python community makes wide use of shared-nothing, message-passing concurrency, and a number of different facilities are provided to support it.

The Python standard library includes the `threading` module, which gives access to operating system threading facilities in a portable fashion. However, this only enables concurrent programming, not parallel programming: the CPython runtime system serialises the execution of instructions in the Python virtual machine (using the "global interpreter lock"), meaning that multithreading cannot be used to speed up a pure-Python program. In practice this is less of a problem than it seems, since execution of FFI calls *can* occur in parallel, and—since Python is a relatively slow language anyway, even in JITting implementations—Python programs often use FFI-based libraries to perform compute-heavy operations.

Synchronisation and communication between threads can be performed using the low-level operating system facilities exposed by the `threading` module (such as mutexes), but it is more convenient to use the multithreaded queues provided by the `Queue` module, which are designed to provide the facilities of shared, buffered channels for interthread communication. A queue's buffer may be the usual FIFO type, but it may also be last-in-first-out, or priority-ordered. In addition, queues can be used to wait for the completion of other processes: each queue keeps a count of items that have been sent through it, with a `task_done` method that reduces the count (to be called once a receiving process has finished processing), and a `join` method that blocks until the count becomes zero.

Python 2.6 introduced the `multiprocessing` module, which provides interfaces similar to `threading` and `Queue`, but using operating system tasks instead of threads—allowing real parallel execution of Python code, at the cost of slower communication [147].

Python provides Icon-style generators [205]. A generator is a function that is called

in a context that can make use of multiple values (such as a "for each" loop); a generator calls `yield` to return a value, and is then suspended until the next value is required. (In process-oriented terms, a generator is a forked ▷ **Producer** process; `file.get.options` in ⋆ **KRoC**'s `file` module is an example of a generator.) The Kamaelia framework [219] implements generalised message-passing lightweight concurrency for Python in terms of generators. Kamaelia processes are generator functions that are called from a scheduler. Processes in Kamaelia communicate by yielding a description of the communication, causing execution to return to the scheduler, which can then execute the appropriate process to complete the communication. This approach allows reasonably efficient lightweight scheduling—even across multiple threads—but the use of yielding cannot be hidden from the programmer, and only the function representing the process (rather than any function or method it calls) is able to yield, limiting abstraction.

A generator is a restricted form of coroutine, only being able to return execution to the process that called it. Unfortunately, the standard version of Python does not provide support for generalised coroutines, which makes seamless lightweight concurrency (where sending a message is just a method call) impossible to implement in pure Python. The Stackless Python project attempts to rectify this by adding support for generalised coroutines to the Python runtime system, meaning that a modified Python interpreter is required, but a wide variety of concurrent programming facilities can be easily implemented (including processes and synchronous channels) [227]. The Greenlets module implements coroutines as a module that uses FFI calls to change the C stack pointer, which works with the regular interpreter but is unportable and fragile [7]. Neither has really taken off for real Python applications.

Process-oriented programming facilities for Python are provided by two libraries: PyCSP and python-csp.

PyCSP was initially developed in 2007 to support parallel programming of scientific applications in Python; as such, it needed to be friendly enough for use by users with little programming experience [11]. The set of facilities provided by PyCSP were originally based upon those in JCSP—such as `One2One` and `Any2Any` channels—with some simplifications made possible by Python's syntax and type system. The first version of PyCSP was implemented using Python's `threading` module, using algorithms based upon those in JCSP.

PyCSP was extensively revised in 2009 in response to feedback from users [235]. Channel types other than `Any2Any` were removed, bringing PyCSP closer to the Newsqueak model. More convenient facilities for channel poison and network shutdown were provided, with a reference-counting "retirement" mechanism for shared channels. Output guards were supported in choice. Finally, the library interface was redesigned to take better advantage of Python's syntax, resulting in Python concurrency operations comparable in succinctness to occam.

In addition, the 2009 revision of PyCSP provides multiple different implementations of its concurrency primitives [85]. PyCSP processes can now be represented by operating system tasks (using `multiprocessing`), operating system threads (using `threading`), or lightweight processes (using Greenlets), depending on the requirements of the application. To support lightweight processes, support was added for running blocking calls in a separate thread, using an `@io` decorator.

python-csp is the younger of the two libraries [143]. python-csp provides operators for process composition and choice that directly mirror the syntax of CSP. python-csp can use either a thread-per-process or task-per-process model, and in addition can run

on top of Jython, exporting its concurrency facilities to Java programs through a JCSP-like interface.

2.3.8 Haskell

Haskell is a lazily-evaluated, purely-functional programming language [110], with exceptionally good facilities for concurrency. Haskell's type system allows a wide variety of different computational models to be expressed without changes to the language.

Haskell supports the expression of different models of computation within a single program through the use of monads. A monad represents a model of computation: a value in a monad is an action, and the monad defines how actions may be sequenced. For example, input-output operations are performed in Haskell using actions in the `IO` monad; a value of type `IO String` is an action that returns a string (such as reading a line from a file). The Haskell standard library includes a variety of monads for different purposes—for example, the `GenParser` monad represents an action that attempts to parse a value from an input stream, backtracking upon failure.

Values in Haskell are immutable by default, which means that they can be safely and efficiently duplicated in concurrent programs without aliasing concerns. It is possible to construct alias-able reference types within monads that support them, however; for example, the `IO` monad provides an `IORef` type.

There are several implementations of Haskell, with the most widely-used being the Glasgow Haskell Compiler. GHC's runtime system provides a lightweight process scheduler that approaches CCSP in terms of performance, and considerably exceeds it in terms of features. GHC processes may be preempted (at relatively course granularity; it is not a real-time system). The scheduler supports blocking calls to external library functions in much the same way that CCSP does, and additionally includes an I/O scheduler that allows processes to wait efficiently for file descriptors to become ready [130].

The Concurrent Haskell library, `Control.Concurrent`, provides GHC's basic concurrency features—lightweight threads, and I/O scheduling—along with some synchronisation objects [165]. In process-oriented terms, `MVar` is a one-place buffered channel, `Chan` a infinitely-buffered channel, and `SampleVar` a one-place overwriting-buffered channel, all with shared ends. The library does not provide any facilities for choice, however; the usual idiom is to have processes complete for a shared channel, rather than having processes choose from multiple channels.

As Haskell is lazily-evaluated—computations are not actually performed until their results are required, with the runtime system passing around thunks rather than real values—concurrent Haskell programs can have surprising performance characteristics. For example, implementing a ▷ **Farm** in Haskell might not result in any computation being performed in parallel at all! The `Control.Parallel` module provides a `seq` function to force a value to be evaluated, along with a `par` function that lets the runtime precompute a value in the background that will be required later (a *spark*—a purely-functional future).

Transactional Memory

Transactional memory is an interesting and relatively new approach to concurrent programming. The philosophy behind transactional memory is essentially the same as

that of process-oriented programming—concurrency is best achieved by composing isolated processes which interact only at defined interfaces. However, transactional memory attempts to make shared-memory programming safe, rather than discarding it entirely in favour of message-passing.

In a transactional memory system, the interfaces between processes are regions of shared memory that can be updated and read by multiple processes [97]. Safety is achieved using a technique drawn from databases: when a process wishes to access transactional memory, it must first begin a *transaction*. The actions that take place inside the transaction occur atomically, as if the process had acquired a global lock across the whole system—they either all succeed, with the process acquiring a consistent view of all the variables it reads, or all fail, in which case any changes made are invisibly *rolled back*.

This makes transactional memory a highly convenient programming mechanism for algorithms that can be expressed in terms of shared memory [2]. It is particularly attractive as a better mechanism for existing threads-and-locks programs—simply replacing all locked regions with transactions will provide the same semantics, but without the need to worry about acquiring the right locks in the right order; the programmer can write code without needing to explicitly manage access to shared resources.

A transactional memory system gains many of the advantages of a transactional database—in particular, it can execute *optimistically*, meaning that the program assumes it will succeed, and only takes a performance hit when a transaction fails. When contention on shared resources is low, rollbacks are infrequent, and a shared-memory program's performance will usually be improved since the cost of taking locks (especially multiple locks) is higher than that of simply checking that a transaction has succeeded.

However, the converse also applies: when contention is high, many attempted transactions will need to be rolled back. In a system using locking or message-passing, processes will naturally take turns to use a shared resource. In a transactional system, it is possible for processes to "fight", with processes running on different physical processors causing each other's transactions to be rolled back. The result is that it can be difficult to guarantee, in a transactional system, that *any* process is able to make forward progress [158]. Hardware support for transactional memory can help to some degree by reducing the size of the window during which contention is possible [70], but fundamentally the runtime system must be able to detect and prevent it—a difficult problem to solve without full-program static analysis. As a result, most existing STM systems scale poorly as contention and the number of physical processors increase [59].

Transactional memory has been implemented in a number of concurrent programming environments, but one of the most widely-used is the *software transactional memory* implementation that GHC provides for Haskell [96]. In Haskell, actions within a transaction are specified in the `STM` monad; an `atomically` function executes such a transaction from the usual `IO` monad. The `retry` action is a notable feature of Haskell's STM implementation: when executed inside a transaction, it causes the transaction to be immediately rolled back and not executed again until one of the variables that had previously been read changes value. This allows a process to efficiently wait for a value to change, which makes for easy implementation of higher-level concurrency mechanisms, such as channels, in terms of STM. In addition, Haskell STM allows prioritised choice across several possible transactions, with the first to complete succeeding and the rest being rolled back.

Transactional memory could be integrated into a process-oriented programming environment by treating the execution of a transaction as simply another type of synchronisation event, usable in choice and other forms of composition. The use of *nested transactions* could even allow transactions to contain other synchronisations—allowing communications to be rolled back (at a cost) [142].

CHP

The Communicating Haskell Processes library provides an implementation of process-oriented programming facilities for Haskell [51]. CHP is an embedded domain-specific language [109, 222], expressing some of the safety properties of process-oriented programming in Haskell's type system to provide a degree of static checking. Actions in a CHP program are represented as values in the `CHP` monad, which means that CSP-style operators for composing processes can then be provided as functions upon CHP values.

CHP provides both parallel composition and forking operations; a forking context is represented as a monad that extends the regular CHP monad. (Conceptually, an action in the extended monad is one that behaves like a regular CHP action, but in addition will cause the surrounding forking context to block until it finishes.)

CHP provides synchronous channels with implicitly- or explicitly-shared ends. In addition, it provides broadcast and reduce channels that can communicate with several processes simultaneously. Extended input and output operations are provided. The library also provides barriers, which keep track of a phase count, and a prototype implementation of clocks has recently been added. All synchronisation objects in CHP may be poisoned, and smart looping constructs are provided to handle poison in a convenient way.

CHP supports prioritised choice and conjunctive choice—the only environment so far to provide the latter. CHP's choice facilities work over not only arbitrary events (channel inputs and outputs, and barrier synchronisations), but over arbitrary actions in the style of CSP's choice operators. This means that, unlike in most process-oriented environments, the programmer does not have to separate the event from the action that will be performed when it becomes ready when specifying choice. In addition, an event—or a nested choice—can be abstracted into a function, which can itself be used in a choice.

This works because CHP's monadic actions are able to carry extra information about the actions they represent, both at the type level and at the value level—meaning that the next event in the trace of an arbitrary action can be found at compile time. This also permits static analysis of CHP programs by a process of abstract interpretation, and allows the automatic extraction of CSP models from CHP programs, allowing straightforward model-checking of concurrent Haskell programs [47].

As Haskell provides type inferencing, the types of channels (and even their degree of sharing) can usually be automatically inferred in a CHP program. Haskell's support for higher-order programming makes it possible to write higher-order functions for constructing process network fragments—processes and actions are first-class values in Haskell.

The CHP-Plus library uses Haskell's typeclasses to provide very high-level facilities for process composition [48, 50]. Typeclasses are provided that describe how processes may be connected together, automatically matching up compatible channels and

events; using these, operations for constructing regular structures such as rings and meshes are built. CHP-Plus also provides a higher-level EDSL for describing process *behaviours*—such as repeatedly performing an action until an event completes; many common kinds of processes can be built directly using these.

While CHP's channels are strongly-typed, CHP does not yet support protocols. This is not a significant limitation compared to other environments because Haskell's algebraic data types provide all the facilities of occam-π protocols—but it does mean that bidirectional protocols cannot be checked. However, other libraries have experimented with encoding protocols in Haskell's type system using session types [146]; a similar approach could be used in CHP in the future.

CHP's operations are implemented using GHC's STM mechanism, with CHP processes represented by STM's lightweight processes. Internally, CHP has a single CSP event type, which is then used to construct other communication mechanisms. The algorithms used to resolve multiway and conjunctive choice between events in CHP are reasonably complicated—in effect, they are a constraint solver—meaning that CHP has rather lower performance than runtime systems such as CCSP. This was a deliberate design decision to enable experimentation with new process-oriented mechanisms; future work could improve performance at the cost of greater complexity in the CHP library.

2.4 Design Patterns

A *design pattern* captures a reusable solution to a common problem. A *pattern language* is a collection of related, named patterns. Pattern languages serve the dual purposes of documenting best practice, and providing a shared vocabulary for discussing design problems.

2.4.1 "A Pattern Language"

Pattern languages were initially promoted in the mid-1970s by a team led by architect Christopher Alexander. Their books "The Timeless Way of Building" [4] and "A Pattern Language" [5] describe the principles behind design patterns, and give a language of 253 patterns for the construction of towns, neighbourhoods, buildings and rooms. A pattern "describes a problem which occurs over and over again in the environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Patterns in "A Pattern Language" are described in a standard format:

- a section classification by scale;
- a number and title;
- an indication of the team's confidence in the strength of the solution—whether it is a "true invariant" required for any solution of the problem, or merely one possible solution;
- a photo illustrating one application of the pattern;
- a list of larger-scale patterns that this pattern will be involved in implementing;

- a brief statement of the problem;
- a discussion of the problem, with references and examples as "evidence of its validity";
- a brief statement of the solution and its "field of physical and social relationships", as an instruction, often illustrated by a sketch;
- a list of smaller-scale patterns that will be used when implementing this one.

The patterns vary greatly in length and complexity.

"A Pattern Language", and its related work "The Timeless Way of Building" [4], are philosophical works as well as practical ones: a pattern language is "a fundamental view of the world" that provides insight into "the nature of things in the environment". Alexander's view of the world is *compositional* and *generative*: objects are important primarily in terms of how they interact with other objects, and those interactions may themselves spark the creation of new objects. (Design patterns differ from design rules in this respect: patterns are generative, rules are constraining.)

Architecture differs from software engineering in that it has thousands of years of experience behind it, rather than just sixty; Alexander's pattern language can thus aim to be reasonably complete in a way that pattern languages in computer science cannot. Alexander's patterns are inherently interrelated, and operate at multiple levels of scale, with larger patterns requiring smaller patterns for their completion. "No pattern is an isolated entity." Alexander notes that his is but one of many possible pattern languages: "every society which is alive and whole, will have its own unique and distinct pattern language ... and every individual in such a society will have a unique language, shared in part, but which as a totality is unique to the mind of the person who has it."

2.4.2 "Design Patterns"

"Design Patterns: Elements of Reusable Object-Oriented Software" [87], first published in 1995 and often known as the "Gang of Four" book after its authors, was the work that popularised the idea of design patterns for software engineering.

"Design Patterns" presents a catalogue of 23 patterns for object-oriented design drawn from the authors' experience with real-world applications. Unlike "A Pattern Language", the patterns described in the text all apply at roughly the same scale; there are relationships between patterns, but no simple hierachy. This choice of scale is deliberate: "Point of view affects one's interpretation of what is and isn't a pattern. One person's pattern can be another person's primitive building block." Patterns are primarily based upon "common collaborations" among objects.

Patterns represent a higher level of abstraction than is usual in OO decomposition: "Design patterns are not about designs ... that can be encoded in classes and reused as is." A common objection to the use of design patterns is that they do not promote code reuse [94]—but patterns are designed for things that cannot simply be provided as reusable components; if a pattern can be provided as a class, it is not really a pattern. (This implies that pattern catalogues will change over time, as languages and libraries become more expressive; we will revisit this later.)

Patterns are described using a form that is somewhat more regimented than that in "A Pattern Language", resulting in the pattern descriptions all being roughly the same length. Each description includes:

- a name, and "also known as";
- a classification: object or class, creational, structural or behavioural;
- a brief description of the problem;
- a simple example of the pattern in use;
- the situations in which the pattern may be applied;
- a UML-like graphical representation of the pattern;
- a list of the classes that participate, and their collaborations;
- the consequences of using the pattern upon the rest of the system;
- practical advice on implementing the pattern in code, with examples;
- examples of use in real systems;
- a list of patterns "related to" this one.

This catalogue has been enormously influential within the field of object-oriented design; patterns such as ○ **Singleton**, ○ **Iterator** and ○ **Decorator** are familiar to most OO programmers, and several of these are now sufficiently ubiquitous within OO design to have inspired language and library features to support their use. This is partly because "Design Patterns" got there first—it was many programmers' first exposure to design patterns in any context—but the authors' careful choice of scope, elegant style, and focus upon patterns found in real applications lead to a pattern catalogue that was immediately and directly useful to working programmers. The patterns in "Design Patterns" are just as relevant today as they were in 1995.

Much of the advice—such as "Program to an interface, not an implementation", a recommendation familiar to message-passing programmers generally—and many of the patterns in this book may be applied profitably to process-oriented programs as well as object-oriented ones; see section 4.5.

2.4.3 Other Concurrent Patterns

There has, to date, been no comprehensive study of the patterns of process-oriented design, although isolated patterns have been noticed in the course of other work (for example, while using occam-π for teaching [120], during the design of SystemCSP [153], and while building higher-level patterns for simulation [10]). Attempts have been made to describe patterns within other approaches to concurrent design, however.

Doug Lea's book "Concurrent Programming in Java: Design Principles and Patterns" [127], published in 2000, gives an overview of a wide range of contemporary strategies for Java concurrency—including process-oriented programming, presented

using JCSP. It provides excellent discussions of a number of low-level patterns, including copy-on-write data, fork-join, and worker threads. However, these are not presented in a standard form—indeed, it is usually not obvious that a particular section is considered to describe a pattern without consulting the index—and, as a result, it is difficult to use CPiJ as a pattern language.

Jorge L. Ortega Arjona's book "Patterns for Parallel Software Design" [157] collects much of his earlier work on design patterns for concurrent object-oriented programming [156, 155, 154]. While the descriptions are in a standard form, with well-chosen names, and useful descriptions of how to apply them to real-world problems, his patterns are at a very low level—they would generally be considered to be fundamental programming facilities in process-oriented approaches, rather than true patterns. (This does not make them invalid—just not immediately useful for this work.)

The "Pattern-Oriented Software Architecture" series is an ambitious attempt to collect design patterns across a wide range of software engineering applications in a standard form—although the form chosen is even more complex and baroque than that in "Design Patterns". The second volume in the series collects "Patterns for Concurrent and Networked Objects" [206]. Most of the patterns would, again, be considered fundamental features of process-oriented programming ("Thread-Specific Storage"), but others operate at a higher level; as in section 4.5 in this work, the authors show how conventional object-oriented patterns can be adapted to function in a concurrent environment.

The Portland Pattern Repository, a well-known online collection of design patterns and other material on software engineering, has a category for concurrency patterns [174]. Again, these tend to be at a very low level ("Semaphores for Mutual Exclusion"), although there is some discussion of higher-level patterns such as pipelines in the pattern descriptions.

2.5 Process Diagrams

Diagrams are an important tool in process-oriented software design. A *process diagram* is a graphical representation of a process network—the relationships between the process instances in a system—at a particular point in time. For a static network, where the set of processes and the connections between them do not change, a single diagram will suffice. For a dynamic network, a filmstrip or flip-book animation can be used to show the evolution of the network over a sequence of snapshots (for an example, see figure 36).

Process diagrams bear a resemblance to the circuit diagrams used in electronic design. This can be attributed to the widespread use of process-oriented software design in embedded systems, where software designers are likely to already be familiar with circuit diagrams; INMOS and XMOS have encouraged embedded programmers to think of process-oriented design in the same way they would think of connecting electronic components together.

The syntax of process diagrams has never been standardised, although local conventions have emerged in the communities that use them. As a result, the process diagrams in this work are in a variety of different styles. Process diagrams are generally considered to be an *informal* notation, and it is often useful to break the rules of the

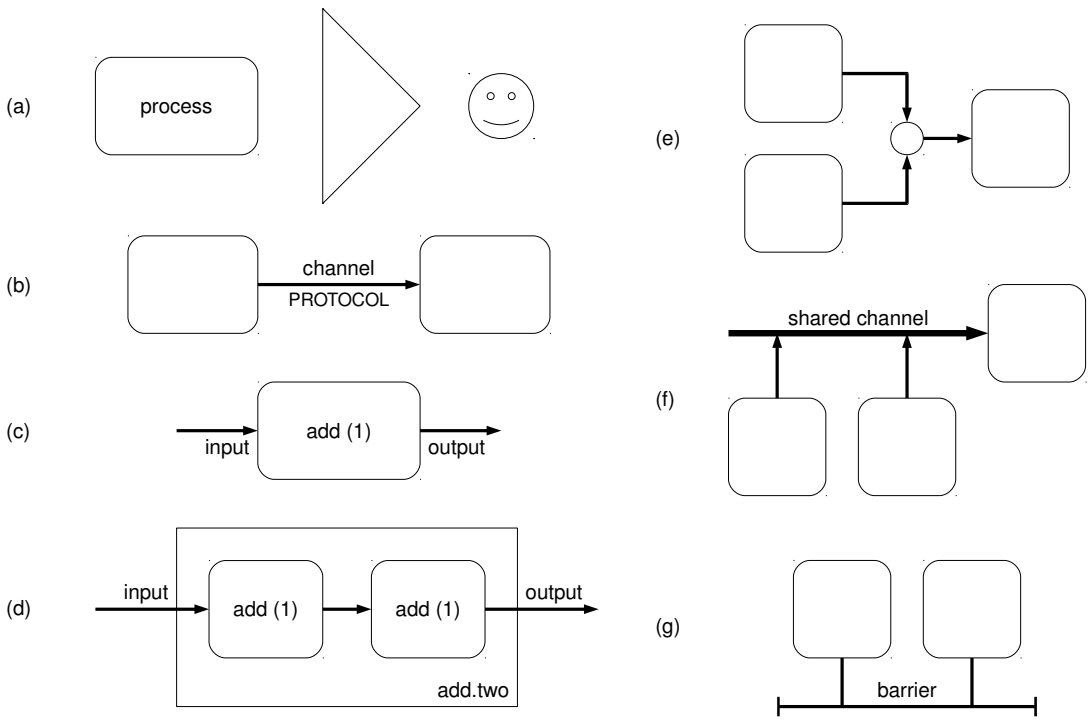


Figure 12: Process diagrams

syntax—or invent domain-specific extensions to the syntax—in order to more effectively describe a system. For example, process diagrams using Trap or other network libraries (figure 39) often show network links as well as channels; hardware devices or external libraries to which particular processes interface can also be shown as if they were processes themselves (figures 17 and 18).

A process diagram is not necessarily a complete representation of the system; it may represent only part of a larger system, with internal details of nested processes elided, or connections running "off the page" to other processes not shown. Complex process networks—especially those with repeated parts, or with both static and dynamic parts—are often best expressed using multiple diagrams.

2.5.1 The Syntax of Process Diagrams

Most of the basic facilities of process-oriented programming described in section 2.2 can be shown in process diagrams.

Processes are usually shown as rectangular boxes, with the name of the process as a label inside the box (figure 12a). Different types of processes can be visually distinguished using different shapes; by convention, ▷ **Merge** and ▷ **Delta** processes are drawn as triangles, ▷ **Black Hole** processes are solid circles, and agents in a simulation are often drawn as "smilies". (Many of the processes in section 4.1 could similarly be distinguished by shape in the future.)

Channels are drawn as arrows that connect processes, with the channel's direction indicated by an arrowhead at the output end (figure 12b). When a channel carries a

two-way protocol (for example, a ▷ **Client-Server** connection), the arrowhead shows the direction of the first communication. Channels may optionally be annotated with their name and the protocol they carry. The protocol can be set in smaller text if necessary to avoid confusion, although naming conventions in most languages make channel names and protocols distinct anyway (for example, protocol and type names may be capitalised).

A process's label may also include the process's parameters that are not synchronisation objects; parameters that are synchronisation objects can be drawn as disconnected stubs when a complete process network is not being shown (figure 12c). When a process is internally implemented as a process network, the internal network may be shown inside the box, with the label moved to the corner, and parameters that correspond to synchronisation objects drawn crossing the edges of the box (figure 12d). Such parameters can be labelled with their names immediately outside the box.

In the case of occam-π, where two-way protocols must be simulated using channel bundles, it is common to draw a ▷ **Client-Server** channel bundle as if it were a two-way channel—that is, just as an arrow. (See section 4.2.5 for more information on drawing client-server relationships in process diagrams.) Mobile channel ends are not usually visually distinguished from static channel ends—except when their reconnection is animated across multiple process diagrams—but where necessary, this can be done by drawing the channel's arrow as a curved line.

A shared channel—one whether either or both of the ends is shared—can be drawn in two ways: either as a small, unfilled circle (figure 12e), or as a heavy arrow (figure 12f). In either case, each individual process using the shared channel has its connection shown as a separate arrow. A channel bundle is drawn as an arrow with a slash across it, optionally with an annotation giving the number of channels contained in the bundle (although this is unnecessary when the bundle's type is given as an annotation); this syntax is the same as used in circuit diagrams for buses.

A barrier is drawn as a line with a T-shaped cap at each end (figure 12g). Each enrolment upon the barrier is shown as a line connecting the enrolled process with the barrier. Other multiway synchronisation objects—partial barriers, buckets, and so on—can be drawn in similar ways, with the line annotated to show the object's type.

When an application is explicitly distributed across multiple physical locations—for example, different processors in an embedded system, or hosts in a distributed system—the locations are drawn as boxes surrounding the processes they contain (figure 29), or divided by lines if the structure is sufficiently regular (figure 39).

2.5.2 Drawing Process Diagrams

As process diagrams are often drawn by hand (on paper, or on a whiteboard), the syntax has evolved to support this. For example, shared channels were originally drawn as arrows with thick lines, but it is difficult to distinguish between thin and thick lines in a hand-drawn diagram; the current syntax with arrows and nodes is more convenient. Similarly, the original INMOS style for process diagrams usually represented processes as circles (figures 16 and 17), using rectangles instead to represent physical processors (figure 15); however, rectangles are easier to draw and make better use of space. Unlike in UML, the styles used for arrowheads have no significance.

The use of process diagrams in automated tools for software design is complicated by their snapshot nature, making them most useful for relatively static systems. The

Strict occam Design Tool [35, 256] was an editor for Transputer process networks, using static design rule checking to allow safe concurrent systems to be constructed graphically; it introduced a number of extensions to the basic syntax of process diagrams, including a "channel weaving" mechanism that allowed several forms of repetitive processing networks (such as FFTs) to be specified graphically. POPed [215] is a process network editor designed specifically for teaching concurrency, allowing students to build complex process-oriented systems using a library of predefined components.

Snapshots can be used to visualise a process network's state during debugging; in these cases, the syntax of process diagrams is often extended to display some representation of the internal state of the processes and synchronisation objects in the network. This is especially useful when dealing with problems such as unexpected deadlock: a visualisation that shows the state of the channels in the network can make the cause of deadlock much easier to determine. GRAIL [221] was an early example of this, showing graphically not just the connections between processes in a Transputer system, but also the load on individual physical processors, and the block structure of the occam code in each process. More recent adaptations to the Transterpreter have allowed an animated representation of the process network in a running occam-π program to be visualised, with interactive features that allow the programmer to explore the nested structure of their program, down to the source code of each individual process [186].

2.5.3 Other Kinds of Diagrams

The lack of a standard definition of process diagram syntax and semantics tends to hinder communication between process-oriented developers. There are several existing forms of diagram upon which a standard process diagram syntax could be based.

There are a wide variety of dataflow diagram syntaxes that resemble process diagrams when the latter are used to describe dataflow systems; these are supported by tools such as The MathWorks' Simulink, Kamaelia [219] and the GNU Radio Companion [41]. These are rarely precisely defined, and often lack abstraction features such as nesting.

Several graphical syntaxes exist for process calculi. These are generally supersets of process diagrams, in that they represent both the relationships between processes and their internal implementations; it is often possible to use a graphical process calculus to represent a process network by simply not showing the internals of the processes. For example, the Philips-Cardelli graphical syntax for the stochastic π-calculus [166] aims to support the modelling of biological systems; it combines the rich semantics and tool support of the π-calculus with the convenience of familiar graphical notations such as Petri nets [164].

GML is a graphical language that supports the design of practical real-time embedded applications, with automated code generation for process-oriented environments such as occam and JCSP [99]. GML includes representations of non-CSP language features such as process priority, and higher-level concepts such as client-server relationships. The gCSP tool provides a graphical editor for GML with automated design rule checking [123]. In contrast to process diagrams, GML shows both communication relationships and compositional (scheduling) relationships between processes; this makes GML more precise and expressive, at the cost of greater complexity.

UML is the best-known family of graphical languages for software engineering,

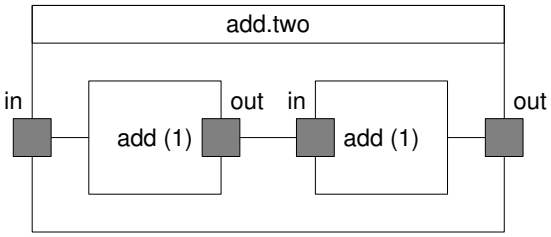


Figure 13: Process network as composite structure diagram

nearly ubiquitous in object-oriented design [148]. Assuming that processes are represented as classes (which is usually the case for process-oriented libraries in OO languages), many types of UML diagram can be used in the design of process-oriented systems; for example, sequence diagrams can be used to show the message-passing behaviour of a set of processes over time. In addition, UML is well-supported by both general-purpose and special-purpose tools, so defining process diagrams as an extension to UML would have considerable advantages. UML's semantics are deliberately somewhat loosely defined, and diagrams can be used in an informal fashion; UML class and activity diagrams are becoming increasingly popular to describe biological models [177].

(Interestingly, class diagrams, while widely used in object-oriented design, seem to be rarely used in process-oriented design—perhaps because process-oriented design rules tend to be stated in terms of relationships in the process network that are not easily visible in a class diagram. Conversely, object diagrams, which show instances of objects and the relationships between them, are relatively rarely used in OO design.)

UML 2.0 introduced *composite structure diagrams*, which represent snapshots of sets of components and the connections between them [151]. Composite structure diagrams support nesting of components, connection ports, and messages ("signals") that are sent between components; they are already a reasonably good match for process diagrams, particularly when used to describe client-server systems. Signals between components are described using a very simple form of protocol; two-way signalling is idiomatically described using matching pairs of ports, in the same way that occam-π simulates two-way protocols using pairs of channels. Figure 13 shows one of the nested process networks from figure 12 redrawn as a composite structure diagram.

Inspired by both GML and UML, SystemCSP is a recent family of graphical languages for engineering CSP-based concurrent systems [153]. SystemCSP's features for describing components and the interfaces between them are similar to UML composite structure diagrams, but with the richer communication semantics of process diagrams.

Chapter 3

Case Studies

The patterns in chapter 4 are all drawn from examples found in real-world process-oriented programs. This chapter describes the projects that were the primary case studies for this work: notable applications of process-oriented programming, all of which contributed multiple instances of patterns. The intention is to give a general overview of each case study, and the context in which it was created. The case studies are listed here roughly in chronological order, proceeding from Transputer applications to projects currently under active development.

The case studies fall into three categories:

- historical projects that this work simply documents: ⋆ **Flightsim**, ⋆ **occam-X11**, ⋆ **occvid**;
- projects that the author already had a significant hand in the development of prior to the commencement of this work: ⋆ **KRoC**, ⋆ **RMoX**;
- projects developed with an explicit goal of developing and applying the patterns described in this work: ⋆ **Life**, ⋆ **LOVE**, ⋆ **Occade**, ⋆ **Occoids**, ⋆ **Plumbing**.

3.1 Flightsim

Flightsim was an INMOS technology demonstrator, first shown off to the public at SIGGRAPH 87 [17]. It is a multi-user flight simulation game, providing a simulated 3D world with a landscape, animated aircraft and air-to-air missiles. Aircraft (and missiles) may be controlled by human players using a joystick, or by a simple AI. Each player's view is rendered in 3D using filled polygons at video frame rates, with a radar screen and a head-up display providing information on their score and their aircraft's attitude and speed (figure 14). Any number of players are supported, provided each has their own display hardware.

Flightsim was built as an example of how INMOS's high-performance Transputer processors could be combined to do in software what would normally require dedicated hardware. The pinnacle of 3D flight simulation gaming on general-purpose computers in 1987 was, perhaps, David Braben's Zarch on the Acorn Archimedes; this achieved much lower resolutions and frame rates, did not support multiple players, and was written in hand-tuned ARM assembly. Flightsim was written with expansion

Figure 14: Flightsim running on KRoC with four players

in mind—for example, for commercial flight training—and its architecture and implementation are generally straightforward and easily-maintainable. The initial version was completed in three weeks [16], and occupies a little under 9,000 lines of occam 2 code.

Flightsim's large-scale architecture has all the players connected in a ▷ **Ring**; each player's connection to the ring is through a "ring control" process (figure 15). The objects in the world are continually sent around the ring, with one message per object. Each player therefore gets to see all the objects in the world as they pass, and can add, modify and remove objects if they choose. For example, when a player fires a missile, they add a new missile object to the ring; and when the player's own aircraft goes past, its position is updated according to its current velocity. Each ring control process contains sufficient buffering for all the objects the player might want to add to the simulation; the overall ring therefore contains enough buffering for the whole simulation, ensuring freedom from deadlock.

The display for each player is rendered using a ▷ **Pipeline** (figure 16). A "database" process maintains the player's model of the world as a BSP tree [86] (simulated in occam 2 using an array), which allows the polygons in the world to be rapidly sorted into depth order for rendering. The database emits a stream of polygons in depth order for each frame. These are first transformed to match the player's viewpoint,

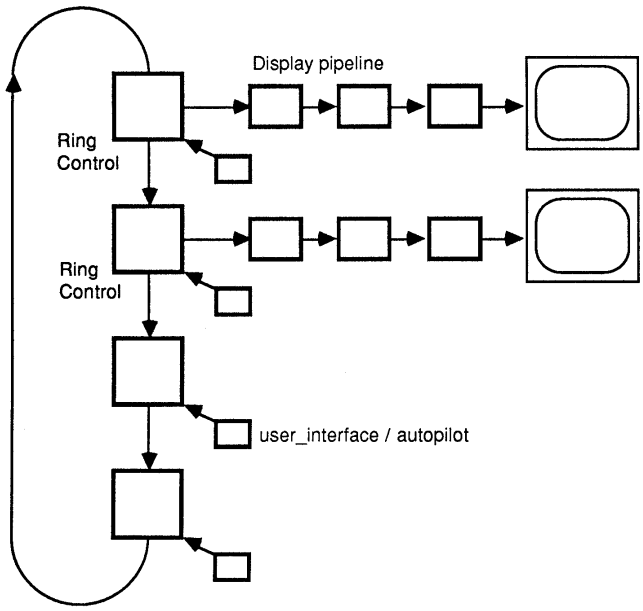


Figure 15: The ring architecture of Flightsim (from [17])

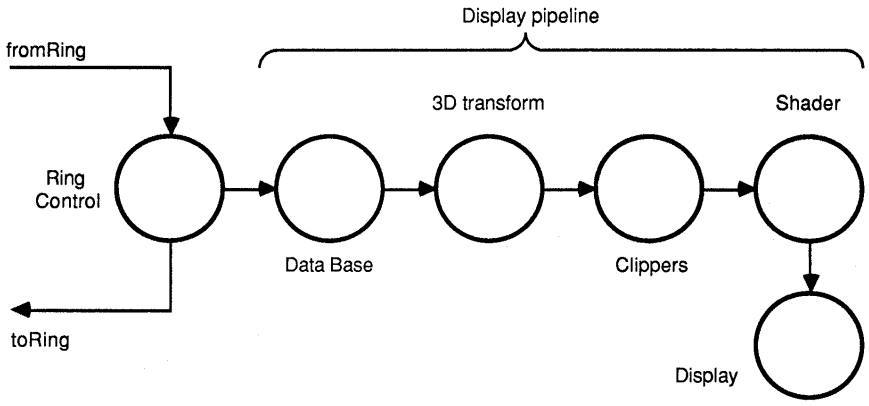


Figure 16: The rendering pipeline for a single player in Flightsim (from [17])

then clipped in all three dimensions to match the visible area. The polygons are then distributed to four shader processes, each of which is responsible for rendering part of the screen (▷ **Fan-Out**); the shaders send partial frames to a final process that merges them together and updates the graphic display.

In addition, each player has a "user interface" process, which reads the player's joystick and buttons and sends appropriate signals to the ring control process. For AI players, this process is responsible for computing the player's flightpath.

The original INMOS system was distributed across eleven Transputer processes per player, with some driving dedicated hardware for the input devices and display. However, it has been recently adapted to run on a regular PC using KRoC, with minor modifications to use the `sdlraster` module for portable video output and keyboard input. Multiple human players are not yet supported under KRoC. It would be relatively straightforward to add network play using Trap (section 3.3.5), although the increased latency of TCP/IP communication over the original Flightsim's Transputer links in the ring would significantly slow down the simulation. Replacing the ring with a central server that only broadcast changes to data would significantly reduce the number of communications necessary, and allow the rendering to be decoupled from the data communication.

3.2 occam-X11

The X Window System is the standard low-level graphics system used on Unix-like machines [266]. Applications ("X clients") communicate with displays ("X servers") using local or network connections, allowing a single X display to contain applications running on several different machines. Version 11 of the X protocol, X11, was standardised in 1987, and provides an extension facility that allows newer features to be defined as protocol extensions that clients and servers may optionally support. As a result, X11—with a number of widely-supported extensions such as GLX, XRender and XVideo—is the low-level protocol used by modern toolkit libraries such as GTK and Qt, and thus the foundation of all modern Unix windowing systems. The reference implementation of the X server and the basic libraries needed by X clients have always been open source software, available under a permissive license.

In 1987, less than two years after the introduction of the Transputer and occam, researchers at the University of Kent at Canterbury found themselves in need of a windowing system for the Meiko Computing Surface [137] and other Transputer-based systems—which provided plenty of computing power and high-performance bitmap graphics for several users at once, but came with no tools to support the construction of the complex GUIs becoming popular on other systems. They quickly settled on the X11 standard being adopted by other scientists and engineers, but found the Meiko C compiler "incapable of making any significant headway when presented with the public-domain X sources" [260].

Instead, they chose to implement an entirely new X server using occam, which would allow them to take advantage of the Transputer's parallelism and high-performance communication facilities. This is an unusual strategy, since most X servers are based on the C reference implementation. The resulting X server was constructed using the ▷ **Client-Server** pattern—the design rules for which were developed during this project, making occam-X11 the first client-server occam program. (It was also one

of the first programs to explicitly describe its design as "process-oriented" [261].)

The X server's functionality is divided among a number of server processes (figure 17). The system could be distributed across up to four Transputer processors to improve performance and make use of distributed memory. A central "window handler" process manages the list of windows on the display, cooperating with a "pixmap handler" process that caches pixmaps (bitmap images) provided by the clients. The outputs from these two processes are combined by a ▷ **Merge** process and fed to a device driver for the graphics display. Separate processes manage the cursor and the list of graphics contexts.

The keyboard and mouse each have a device driver process, which feed input events to the window handler. Timestamps are applied to these events from a shared clock managed by another process.

Client-server links inside the X server follow the approach used in occam-π, with two channels carrying request and response protocols. The X protocol itself is carried over occam channels using variant occam protocols instead of the usual byte-packed network representation of the protocol, improving type safety and allowing occam's usual facilities for processing variant protocols to be used directly upon X messages at a small performance cost. "Input handler" and "output handler" processes adapt between the X protocol, where requests and responses may be interleaved, and the internal client-server interfaces; individual clients are tracked by a "client handler", allowing messages to be reordered and combined as appropriate.

The most recent version of the occam language available at the time of development was occam 2, which does not support the record types introduced in occam 2.1, or the recursion, mobility and dynamic memory allocation features provided in occam-π; the lack of these features considerably increased the complexity of the implementation [261]. The representation and traversal of the X server's tree of windows was a particular problem; in occam-π this would most likely be done using dynamically-forked processes and mobile channel ends, but occam 2 required the use of a "big enough" fixed-size array and carefully-designed traversal algorithms.

The project also produced X client bindings for occam, along with a number of sample occam X applications. While the predicted host of cheap Transputer-based X terminals never surfaced, the ideas and design approaches pioneered in the design of occam-X11 were highly influential on later process-oriented systems.

3.3 KRoC

The KRoC project provides the standard development environment for occam-π.

KRoC is an open-source project with more than twenty-five years' development behind it, derived originally from the tools and libraries provided by INMOS for Transputer development. KRoC now includes a compiler (occ21), runtime systems for large and small devices (CCSP and the Transterpreter), several ancillary tools (occbuild, occamdoc, ilibr, et al.), a collection of testcases and example programs, and the KRoC standard library.

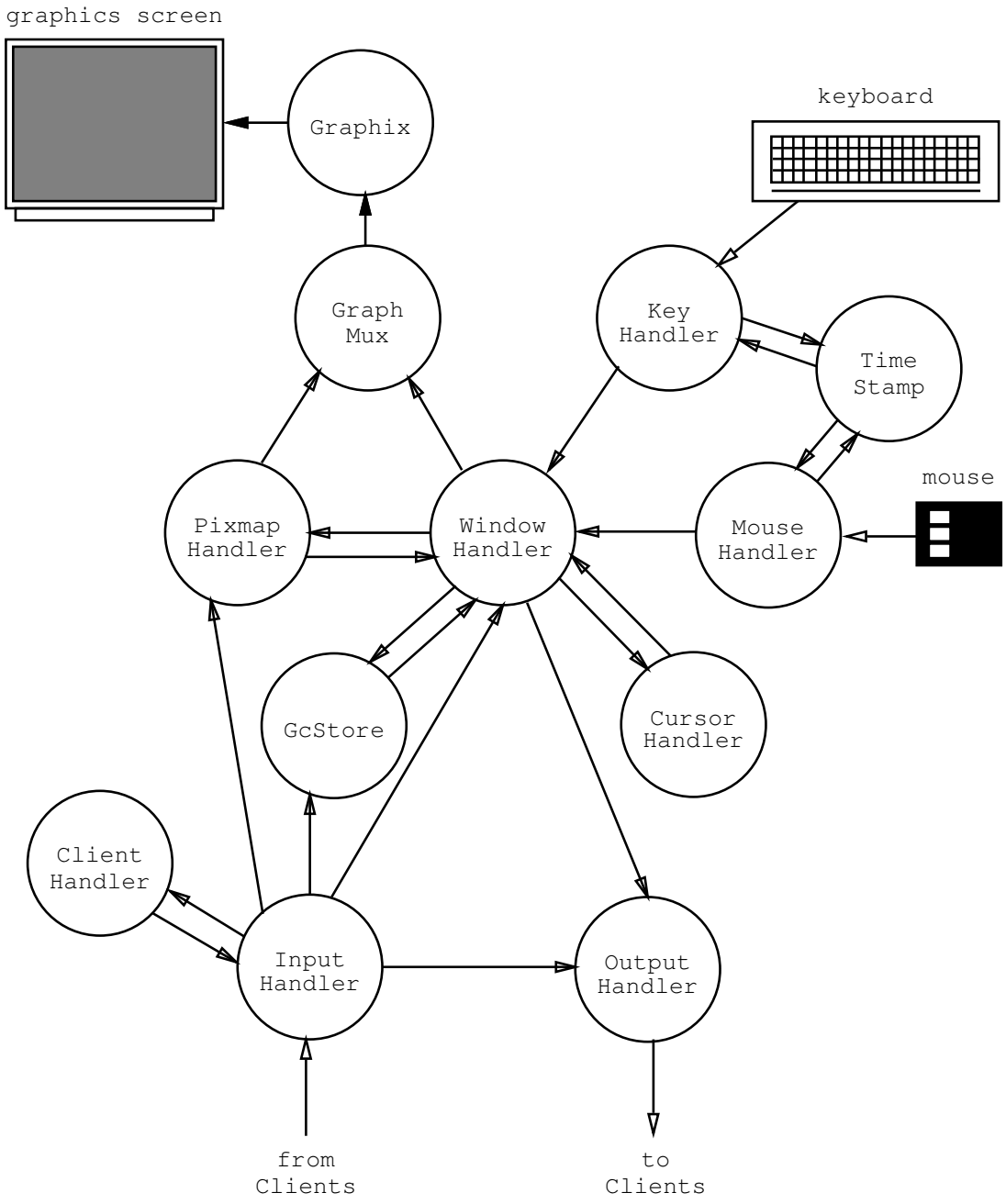


Figure 17: Processes in the occam X server (from [261])

3.3.1 The Module System

Code reuse is achieved in occam-π through the *module system*, primarily supported by the *occbuild* tool. A module is a package of code that may define an arbitrary set of symbols (types, procedures, constants, protocols, and so on); these symbols are made available to an occam-π program when it imports the module. Modules may have dependencies upon other modules and upon external libraries. occbuild allows modules and programs to be built in a uniform way across all occam-π implementations and target platforms, using the appropriate primitive facilities (such as native and bytecode libraries) provided by the different implementations.

The standard library is a collection of occam-π modules of various kinds:

- modules used transparently by occam-π compilers to implement particular data types or wrap primitive operations (such as `forall` and `nocclibs`);
- modules required by the occam language specification (such as the IEEE maths libraries [214]);
- modules that interface to particular external libraries or hardware devices (such as `occGL` and `avr`);
- "pure occam-π" modules, which make up the bulk of the collection.

Modules come from a variety of sources, including several that date back to the original INMOS occam implementation; some of these are still actively used, and others are maintained for backwards compatibility with older occam programs. The occamdoc system is widely used to document occam-π programs and modules [129, 193].

Many modules do not make much use of concurrency—often because they are bindings to or thin wrappers around external non-concurrent libraries, or they provide facilities such as string manipulation that are used inside sequential code: that is, they are used to construct individual processes, rather than process networks. However, several modules do provide processes or protocols for use in applications; the case studies here are drawn from those.

3.3.2 General Utilities

The `course` module is one of the oldest KRoC-specific modules. It provides a set of simple facilities for new occam-π programmers, and is used in most of the exercises in the Kent concurrency course—and in a large number of "real" occam-π applications that need simple I/O, or random numbers, or similar facilities. The `course` module includes "Legoland", which provides simple implementations for teaching of many of the common types of process listed in section 4.1. (Since these are artificially constructed as examples, and are not useful in real applications, they will not be discussed further here.) The Legoland name is fanciful, but KRoC does provide modules for interfacing with actual Lego when the Transterpreter is running on the RCX and NXT controller bricks [217]; in these cases, simple producer and consumer processes provide interfaces to the hardware.

The `useful` module's collection of assorted utilities includes a set of tracing procedures that subvert the usual occam-π resource access management model: they allow the programmer to print a message to the standard error stream without needing the

corresponding top-level channel. While these procedures are unsuitable for use in real applications—for example, they provide no way to prevent the output from multiple processes being interleaved—they have proved extremely useful for "printf debugging" of occam-π programs, since they allow the internal state of processes to be easily printed without having to otherwise modify the structure of the program. A similar tracing facility was provided for Transputer occam programs on Meiko systems: the "supervisor bus", separate from the usual communication links. occam-π's debugging facilities are presently extremely weak; better tools may make these tracing features redundant in the future.

3.3.3 Graphics

As occam-π programs are frequently graphical, several occam-π modules are concerned with bitmap graphics. Bitmap images are usually conveyed around an occam-π program as the `RASTER` data type, which is a mobile 2D array of ARGB pixels; modules provide facilities for loading and saving rasters in various formats, for drawing graphics onto rasters, and for rendering text. The `sdlraster` module allows rasters to be displayed using the cross-platform SDL graphics library [212]. The `raster.display` process exported by this module is designed to be used as part of a ▷ **Ring**, with input and output channels that carry `RASTER`s; for each blank raster it emits, it expects to receive one to display.

3.3.4 Operating System Bindings

The `file` module provides implementations of many of the POSIX library functions. In POSIX, the `getopt` function has some hidden internal state, returning a new argument each time it is called, and so cannot be used safely by multiple threads. The occam-π equivalent, `file.get.options`, is instead implemented as a ▷ **Producer** process that delivers arguments down a channel.

The `selector` module provides a process-oriented I/O scheduler built around the `poll` system call, which allows a program to wait for I/O on multiple file descriptors [40]. In theory, it should never be necessary to use `poll` in occam-π, since the runtime system allows system calls such as `read` to be called in a blocking way from a pool of operating system threads [26]—but in practice, this is highly inefficient owing to thread-switching overhead in programs that need to work with hundreds or thousands of file descriptors at the same time, such as distributed simulations.

`selector` provides a server that allows processes to register with it to receive callbacks when the file descriptors they are interested in become ready for I/O. The server maintains a set of open sockets, and repeatedly calls `poll` across the complete set on the user's behalf. When a socket becomes ready, the I/O scheduler wakes up an appropriate user process to perform the I/O by communicating down a channel. It then waits for confirmation that the I/O has been completed, and returns to the `poll` loop. The user process may opt to keep the socket in the set (if it expects to perform more I/O in the future), or to remove it.

The user may add new sockets to the set at any time. This is achieved using a standard idiom [38]: in addition to the sockets, the `poll` also waits on the reading end of a pipe. Writing a character to the pipe will interrupt the `poll`, allowing it to handle the user's request.

The result is that only the server needs to perform a blocking system call, but the other processes can still be written in a natural process-oriented style, replacing blocking calls with requests to the selector. Processes may block upon socket I/O without needing to worry about accidentally blocking other processes or incurring the overheads of using a separate thread. They may even use choice to wait for any of a set of sockets to become ready. However, the underlying mechanism used to implement communications is the more efficient event-based model supported by poll, which means that very large numbers of sockets can be handled with minimal performance overheads. The efficiency benefits of event-driven I/O [125] are thus obtained without the ▷ **State Machine**-style complexity necessary in most programs written around `poll`.

Ideally this functionality would be built into the occam-π runtime system—as it has been in the GHC runtime for Haskell [130]—but the efficiency gain over `selector` would be minimal.

3.3.5 Distributed Application Support

The `pony` module provides transparent networked mobile channel support for occam-π programs [209]. Pony makes considerable use of internal concurrency, with channels that become extended across the network being transparently replaced with links through processes that themselves have a considerable degree of internal concurrency. The internal processes that parse the occam-π runtime type information values in order to serialise and deserialise occam-π protocols ("protocol handlers") are written in C, using the CIF interface [211, 31]; concurrency considerably simplifies the structure of the processes, which must simultaneously negotiate the structure of the protocols and handle retractions and errors from their partners over the network.

However, with pony, the communication latency over a CSP channel that has been extended over a network is very high compared to local communication. This is because a channel communication must be acknowledged in order to provide the correct CSP blocking semantics, requiring one or more round trips across the network. Many applications—such as the CoSMoS distributed simulations and most ▷ **Client-Server** systems—do not need full CSP synchronisation semantics; the one-way synchronisation provided by buffered channels are adequate.

For distributed applications that do not need full occam-π channel semantics, the `trap` module provides simpler asynchronous messaging between processes in a distributed application [201]. The facilities provided are similar to those of Erlang [19] and MPI's low-level messaging system [144]: a distributed application consists of several *hosts* (equivalent to MPI ranks), each of which has zero or more receiving *ports*. Messages are delivered asynchronously from a host to any port in the system. Messages are guaranteed to arrive in the order that they were delivered, with the Trap implementation batching messages to the same destination together internally to reduce network communication overheads. In process-oriented terms, ports can be considered to be infinitely-buffered shared channels. A receiving process may choose to receive from one port or several; this makes it possible to selectively wait for messages from a particular set of other processes.

Trap aims to provide a reasonably natural interface for the process-oriented programmer, allowing them to mix asynchronous communications with regular local communications and timeouts in choice constructs. The programmer uses a client-server

interface to interact with a *communicator* process on each host, which supports *send* and *receive* operations; *receive* immediately returns a mobile channel down which received messages will be delivered.

Trap uses `selector` to efficiently multiplex input-output operations between many network connections. It has been used to implement distributed simulations for the ⋆ **CoSMoS** project, and to experiment with more efficient MPI-style collective operations for distributed computation [40].

3.4 RMoX

The RMoX operating system has been under development for several years at the University of Kent, as a demonstration of how process-oriented programming can be used to build reliable, scalable systems software [22]. occam and Transputers were widely used for high-performance embedded applications; RMoX aims to allow occam-π and modern embedded systems to be applied to the same sorts of problems.

RMoX runs on IA32 PCs and PC104 embedded devices—the IA32 limitation coming from KRoC, rather than from any aspect of RMoX's design. It includes support for a variety of common hardware, including USB devices and network adaptors.

RMoX is implemented mostly in occam-π, with a small amount of low-level support code in C and assembler; the KRoC suite is used to compile occam-π into native IA32 code. It uses a slightly-modified "standalone" version of the CCSP scheduler and runtime library, with dependencies on a host operating system removed and facilities for interrupt handling added.

RMoX is highly concurrent, and makes heavy use of mobility and the ▷ **Client-Server** pattern: device drivers, filesystems and other system services are server processes, and the client-server design rules are used to guarantee freedom from deadlock and livelock. The RMoX "kernel" processes provide a directory service that allows other servers to be looked up by name. An occam-π language facility designed specifically for RMoX (`MOBILE.CHAN` [27]) allows mobile channel bundle ends of any service-specific type to be returned from the core services to the processes that wish to use them. The directory service is internally hierarchical: when a connection to a driver is requested, the kernel will forward the request on to a `driver.core` process that manages drivers and the hardware resources they correspond to.

When an RMoX application starts up, it is provided with a channel bundle to the kernel; it can then request whatever other services it needs. This is the traditional approach for process-oriented programs: indeed, the top-level process interface in the Transputer implementation of occam provided two channels that were used to communicate with a server process running on a host machine.

RMoX uses a flat memory space for all processes, without hardware memory protection. The onus is therefore on the occam-π compiler—and the programmer—to avoid contention for shared resources. This has been highly successful so far: RMoX is built as (effectively) a single large application, so static checking can be applied just as for any other occam-π program. However, RMoX has no "userspace": it does not yet support dynamic loading of (potentially-faulty) user programs. In order to support this, compiler and runtime extensions will be necessary: options include some form of lightweight memory protection or compiled code verification [25].

RMoX is no longer the only process-oriented operating system: Microsoft's Singularity project uses a very similar approach (section 2.3.6).

3.4.1 Substrates

RMoX was initially built on top of the Flux OSKit [82], a toolkit for IA32 operating systems development that contains pre-written drivers and filesystems drawn from a number of open-source operating systems adapted to provide a standard interface. However, the OSKit has been unmaintained since 2002, and RMoX never made much use of its facilities beyond booting and memory management. To remove the dependency on the OSKit, a Linux 2.4 kernel was stripped down to provide the equivalent facilities ("minlinux"); this made RMoX's build process significantly easier, and improved hardware support during the boot process.

In order to avoid maintaining the cut-down Linux 2.4 tree, a small glue layer was written that enabled RMoX to be built as a kernel module for Linux 2.6 kernels, with a userspace "init" process that simply loaded the module. As with minlinux, the resulting system was essentially running the CCSP scheduler and all of RMoX inside Linux's kernelspace, making use only of the Linux kernel's memory allocator and low-level interrupt handling facilities. This remained unsatisfactory, however, since a Linux kernel source tree was still necessary in order to build RMoX—and even a minimal Linux configuration would still end up building a lot of unnecessary code. In addition, it was difficult to ensure that Linux's interrupt handlers and kernel processes were completely disabled before launching RMoX, which lead to a number of hard-to-diagnose problems.

The final development was a "bare metal substrate" for RMoX: a from-scratch implementation of a Multiboot-compliant [83] bootstrap, and the minimal code necessary to set up interrupt handling and timers before launching the CCSP scheduler. A CPU exception handler is also included, so that the developer gets some "blue screen of death" information out of the system if it crashes. The resulting substrate is less than 2000 lines of mixed C and IA32 assembler code; it has proven more reliable and easier to maintain than the previous approaches.

3.4.2 Network Stack

The RMoX network stack represents packets as mobile data values which are routed around a complex internal network of filters and multiplexers (figure 18). This is a natural and convenient way of expressing packet-processing systems such as network routers, since it mirrors the structure of computer networks at the physical level. A process-oriented router can also be trivially distributed across multiple CPUs or physical machines if necessary. In addition, occam-π's compile-time and runtime safety features are easy to justify in the security- and high-availability-conscious field of networking.

The first version of the RMoX network stack was written by the author as an undergraduate project in 2003 [198]. This version included basic support for the IP, UDP and ICMP protocols. The only network interface devices supported were loopback and SLIP over serial lines.

The original network stack was developed as a standalone program under Linux, and later integrated into the RMoX services framework. It was written at a time when

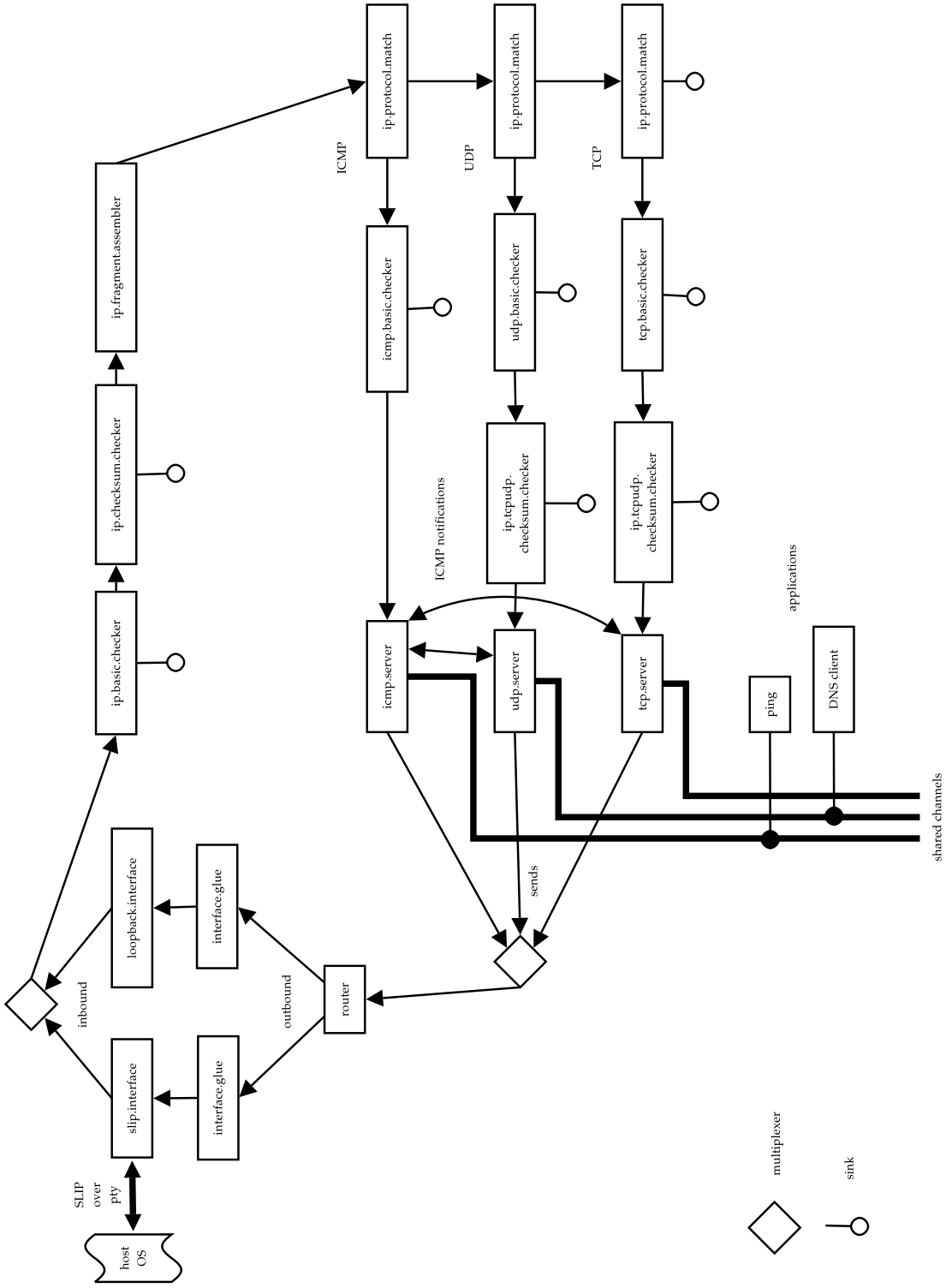


Figure 18: The original design of the RMoX network stack (from [198])

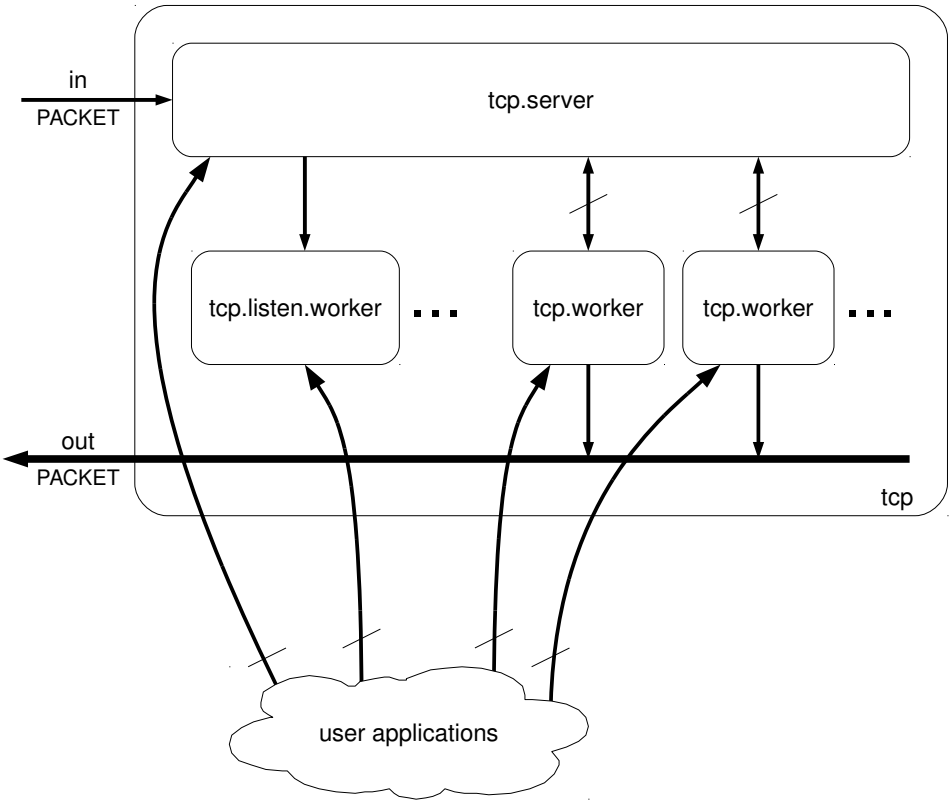


Figure 19: TCP components in the RMoX network stack

KRoC's support for the occam-π language extensions for mobility was relatively immature, and as a result several occam-π features—in particular, forking and dynamic arrays—were used only sparingly in the code.

As part of the present work, the network stack was significantly overhauled to make better use of occam-π, and extended to support Ethernet devices, and the ARP and TCP protocols. This made RMoX's network stack useful for practical applications. The majority of development was performed using the "usermode RMoX" framework [22] in which RMoX runs as a regular process under Linux; an Ethernet device driver was developed for RMoX that used Linux's "tap" emulated Ethernet device.

The TCP implementation was based directly upon the state machine described in the TCP specification. Each active TCP connection is managed by a `tcp.worker` process, the internal state of which corresponds roughly to the per-connection block described in the RFC. A `tcp.server` process handles incoming TCP packets and routes them to the appropriate worker; for each active port, it maintains a channel that connects to the appropriate worker process (figure 19). Workers are forked off by `tcp.server` when a new outgoing connection is requested by a user program; when a user program wishes to bind a port to receive incoming connections, a `tcp.listen.worker` process is forked off that starts new `tcp.workers` in response to inbound connections.

The interface used for bidirectional streams such as network sockets and pipes in

RMoX is by necessity asymmetric in order to conform to the client-server design rules: the client may send a message directly to write, but must send a request (which may block for an indefinite amount of time) in order to read. It is complicated by the ability of communication in either direction to fail at any time: either a read or write may return an error or an end-of-stream indication. Reads and writes are arbitrary-length arrays of bytes rather than individual characters, both for efficiency and to support datagram protocols such as UDP using the same interface.

RMoX provides a command-line interface for starting applications and configuring the system. A server was constructed for the telnet protocol which allows remote access to the RMoX console. The server binds a local socket, and forks off a `telnetd.worker` process for each incoming connection. The worker contains a two-way buffer that maps the client-server protocol used on network sockets into simple channels of bytes as required by the existing console interface, grouping bytes into packets and handling error and end-of-stream messages in either direction.

The Ethernet implementation uses a `eth.adapter` process for each network interface, and a shared `arp.cache` process that maps Ethernet addresses to IP addresses. When an adapter receives an inbound IP packet from the network, it forwards it to the network routing core; when it receives an ARP packet, it stores the information in the ARP cache. When an adapter receives an outbound IP packet from the routing core, it looks up the destination address in the cache. If the corresponding Ethernet address is known, it sends the packet immediately; if it is not known, it sends an ARP request, and forks off a process that waits for a corresponding ARP response before sending the IP packet.

The RMoX network stack is not yet complete. While it has been shown to interoperate successfully with stacks from several other operating systems, its TCP implementation in particular is rather limited: it does not implement any of the extensions to the TCP protocol that more recent stacks use for better throughput and resistance to network congestion, and its retry algorithm is extremely simplistic.

An occam-π-friendly network stack is potentially of interest to programmers building distributed systems in occam-π, since it would reduce the overheads of distributed channel communications—as implemented in pony [210], which was designed with portability in mind. In RMoX, the network stack's internal processes can be scheduled and prioritised along with other occam-π processes. Memory copying overheads can also be avoided, since mobile data can be carried safely from the application right through to the network interface hardware—true "zero-copy".

The network stack at the moment has a number of bottlenecks that prevent it from being fully parallelisable: for example, the IP checksum checker. It would be possible to farm checksum calculation out to a number of worker processes (ideally one per physical CPU). As many programs will operate more efficiently if packet order is maintained, it may be worth introducing some sort of internal sequence number or high-precision timestamp on packets so that they can be reordered before being passed to user processes.

3.4.3 USB Stack

Carl Ritson implemented a USB stack for RMoX [24]. USB presents a number of interesting challenges to operating system developers: USB devices may be connected or disconnected at any time, USB hubs can be used to construct an arbitrarily complex

tree of devices, devices have varying power requirements and communication speeds, and devices can have multiple interfaces which may conform to a standard "device class" or be proprietary. In addition, many common USB devices implement the USB specification poorly—with the result that a host implementation must handle misbehaving devices gracefully.

The RMoX USB implementation provides a low-level USB host controller driver, and a higher-level `usb.driver` process that maintains a tree of individual device processes corresponding to the current physical structure of the connected devices. Class- and device-specific drivers can request various types of connections to USB devices from `usb.driver`. (This description is somewhat simplified; for full details, see [24].)

In addition, a `dnotify` service was added to the RMoX driver core that distributes notifications of devices being connected and disconnected. Any process may register its interest in insertion or removal events involving particular types of device, and `dnotify` will notify it as appropriate. Typically, a driver for a USB device will start a worker process for each device as it appears.

As with the network stack, mobility is used to reduce the need for copying data. The CCSP runtime was extended to support allocation of (selected) mobile data in DMA-capable memory, which allows the host controller driver to transfer data directly from the mobiles allocated by drivers for USB devices—or, in some cases, allocated by user processes.

3.5 TUNA

The TUNA project—Technology Underpinning Nanotech Assemblers—ran from 2004 to 2006, with investigators and students from the Universities of York, Kent and Surrey: it was a pilot study investigating approaches for engineering emergent behaviour. TUNA's primary case study was the design of behaviours and communication strategies for (hypothetical) artificial blood platelets that could clot to heal wounds [230], but a number of simpler complex systems were also considered; in particular, considerable work was done using cellular automata [231, 203]. The project experimented with a variety of techniques for designing, simulating and reasoning about the correctness of complex systems.

The occam-π programming language was used as the primary implementation language for simulations within TUNA. As the CSP process calculus was being used to model and formally verify the behaviours of agents, choosing a process-oriented language allowed direct correspondence between the formal models and the practical implementations in the simulations. occam-π's excellent performance and support for very large numbers of processes made it possible to run large-scale simulations. Several occam-π features were added or enhanced as a result of its use within TUNA; in particular, language support for barriers was added during the course of TUNA, allowing them to be used to directly model CSP events [29].

3.5.1 Life

Cellular automata (CAs) are some of the simplest, most predictable examples of complex systems in which large numbers of autonomous entities exhibit emergent behaviours; as a result, most of the early work done by the TUNA project was carried

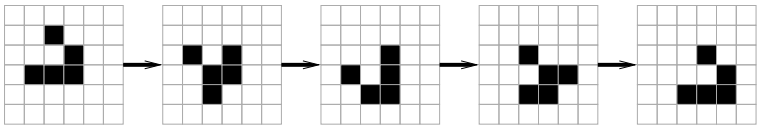


Figure 20: Five generations of a Life glider; black cells are alive

out using CAs.

While CAs are significantly simpler than the sorts of devices TUNA aimed to model—for example, they have little state, usually operate upon a regular grid, and have a common clock—they provided a good starting point for modelling approaches. TUNA examined several sequential and parallel approaches to simulating cellular automata efficiently using process-oriented techniques [203].

The major desirable feature for a CA simulation is that very large scales can be achieved. This means that it should execute as fast as possible and use as little memory as possible—ideally, taking advantage of both multicore processors and distributed clusters of machines.

One of the best-known CAs is John Conway's Game of Life, usually referred to simply as "Life" [88]. First discovered in 1970, Life produces startling emergent behaviour using a simple rule to update the state of a rectangular grid, each cell of which may be either "alive" or "dead". All cells in the grid are updated in a single time step ("generation"). To compute the new state of a cell, its live neighbours are counted, where the cell's neighbours are those cells that are horizontally, vertically or diagonally adjacent to it. If a cell was dead in the previous generation and has exactly three live neighbours, it will become alive; if it was alive in the previous generation and does not have either exactly two or exactly three live neighbours, it will die.

Thirty-five years of research into Life have produced a vast collection of interesting patterns to try. Simple arrangements of cells may repeat a cyclic pattern ("blinkers"), move across the grid by moving through a cyclic pattern that ends up with the original arrangement in a different location ("gliders"—see figure 20), generate a constant stream of other patterns ("guns" and "puffer trains"), constantly expand to occupy more of the grid ("space-fillers"), or display many other emergent behaviours. Life is Turing-complete; it is possible to create logic gates and Turing machines [1].

Life has some features which allow it to be simulated very efficiently. The most important is that cells only change their state in response to changes in the neighbouring cells; this makes it easy to detect when a cell's state must be recalculated. The new state rule is entirely symmetric; it does not make a difference which of a cell's neighbours are alive, just that a given number of them are, so the state that must be propagated between cells does not need to include cell locations. Finally, the new state rule is based on a simple count of live neighbours, which can be incremented and decremented as state change messages are received without needing to compute it from scratch on each cycle. These features are not common to all CAs—and certainly did not hold for some of the models that TUNA investigated—but are nonetheless worth investigating from the implementer's point of view; if such a feature makes a system especially easy to simulate or reason about, it may inform the design of other emergent systems in the future.

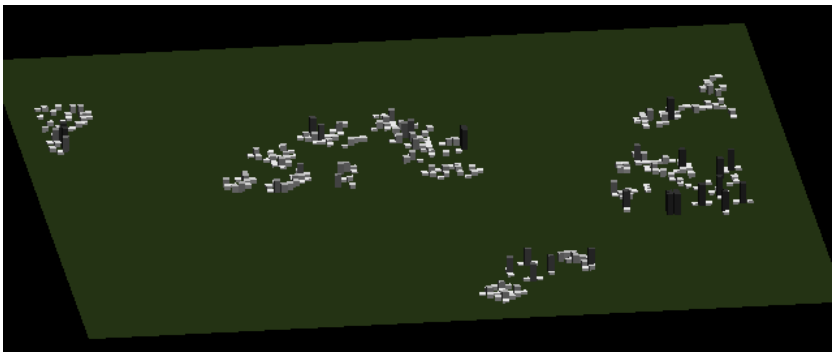


Figure 21: TUNA Life simulation, with OpenGL visualisation

3.5.2 Simulating Life

A CA simulator takes an initial state of the world as input and performs the CA rules on it repeatedly, giving the state of the world at each timestep as output. The TUNA implementation of Life provided a simple simulation framework into which multiple simulation "engines" could be plugged to experiment with different approaches. The framework provided facilities for loading Life patterns from files and generating random patterns, for visualising the state of the Life grid using an OpenGL-based 3D representation (figure 21), and for measuring the performance of the simulation engine.

The simplest approach to simulating Life is to walk over the entire grid for each generation, computing the new state of each cell and writing it into a second copy of the grid, which is later exchanged with the first. This algorithm is $O(N)$, where N is the number of cells in the grid, and has excellent cache locality, so it performs well on a conventional process, or a vector processor or GPU. It can be trivially parallelised by dividing the grid into several fixed-size chunks, and having the new states for each chunk computed by different parallel processes.

As the majority of existing Life implementations are sequential, some techniques have been devised to speed up simulation. One such is Bill Gosper's HashLife algorithm [93], which computes hash functions over sections of the grid in order to spot repeating patterns. The performance depends on the type of pattern being simulated; patterns with many repeating elements will perform very well, but the worst-case behaviour (where the pattern progresses without repetition) is worse than the simple approach, since hash values are being computed for no gain.

The simplest *process-oriented* approach to simulating Life is to break the problem down into the simplest flows of control: have each cell in the grid represented by a process, with channels between them to allow each process to communicate its state to all its neighbours (figure 22). Connections are wrapped around at the edges of the grid, making the world topologically equivalent to a torus. Wiring up this kind of network programmatically is complicated in occam-π, but can be made easier using higher-order programming techniques [48].

Each cell process is written using the ▷ **I/O-PAR** approach: on each timestep, in parallel, it outputs its current state to all its neighbours, and inputs its neighbours'

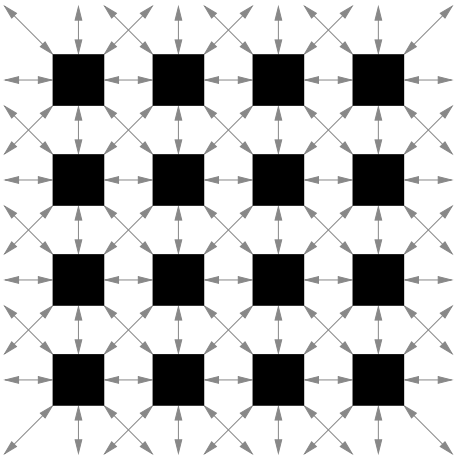


Figure 22: Grid of cell processes with interconnecting channels

states. Once the exchange is complete, the cell can compute its new state using the Life rules. The I/O-PAR approach guarantees liveness [257], and means that there is no need for an external clock to ensure that timesteps do not overlap; the grid is naturally synchronised.

This approach is very inefficient compared to the sequential or trivially-parallel implementations, however: it is doing all the computational work of these simple implementations, and in addition it is performing a huge number of communication operations. In most cases these communications are carrying redundant information since the state of cells (especially empty cells in areas of the grid where there is no activity) has not changed.

We can build a more efficient process-oriented simulation by removing redundant communication: have cells only communicate their state when that state has changed, and have cells only recalculate their state when the state of one of their neighbours has changed—▷ **Lazy Updates**. This means that we cannot use I/O-PAR any more. Furthermore, it is possible that two groups of cells which are active may not be in contact with each other, so the inter-cell communications cannot provide the "generation tick".

Time synchronisation can be provided using the ▷ **Clock** pattern: all cells are enrolled upon a barrier, synchronising once per timestep. As cells' states are only communicated when they change, a cell must be able to tell that it has definitely *not* received a state change from a neighbour. To do this, we can use buffered channels along with the time barrier, allowing sending and receiving of states to be decoupled. At the start of each timestep, each cell checks to see whether a message is waiting on any of its input channels; if none is waiting, the corresponding neighbour's state has not changed. The resulting code is shown in figure 23. (Note that the lazy-updates approach was later refined to not require buffered channels; see figure 59 for the updated approach.)

We have reduced the number of communications to only those necessary, but all processes must still run on every timestep, even if only to check that they do not need to recompute their state. The system as a whole is still ▷ **Polling** rather than event-driven.

To fix this, we make processes resign from the barrier when they detect no changes from the cells around them during a cycle, and sleep until they receive a change notification. Cells therefore truly run only when necessary—changes made to the grid will

```
PROC cell ([8]BUFFERED(1) CHAN BOOL inputs?, outputs!,
           CHAN BOOL changes!, BARRIER bar,
           VAL BOOL initial.state)
  INITIAL BOOL my.state IS initial.state:
  INT live.neighbours:
  SEQ
    ...  do an I/O-PAR exchange to count
         initially-alive neighbours
    WHILE TRUE
      BOOL new.state:
      SEQ
        ...  compute new.state based on live.neighbours
        IF
          new.state <> my.state
            PAR       -- state changed
              my.state := new.state
              PAR i = 0 FOR 8
                outputs[i] ! new.state
              changes ! new.state
          TRUE
            SKIP      -- no change
        SYNC bar
        SEQ i = 0 FOR 8
          PRI ALT
            BOOL b:
            inputs[i] ? b
              ...  adjust live.neighbours
            SKIP
              SKIP    -- just polling
:
```

Figure 23: Code for one lazily-updated Life cell

ripple across the cells that they affect, waking up only those cells necessary to recompute the new state of the grid. One interesting outcome of this approach is that the process network will deadlock if the grid reaches a stable state—one where nothing is able to change—because all processes end up waiting for channel communications that they will never receive.

While this improvement is simple to describe, it is somewhat difficult to implement in practice, since cells must atomically re-enrol upon the barrier when they are woken up by another cell changing state—that is, they must ensure that they always wake up inside the correct timestep. For TUNA, this was achieved using an ad-hoc approach (forcing process priorities) that would not work in more recent versions of KRoC.

Mobile barriers provide a more elegant solution to this problem. When a cell sends a message to another cell to wake it up, the message should include a new enrolment upon the timestep barrier. If the receiving cell is already awake, it is already enrolled upon the barrier and can discard the new enrolment; if it is being woken up, it has been safely enrolled upon the barrier, and can immediately synchronise and poll its inputs to collect any other changes.

This optimisation causes a significant performance improvement, since only active cells occupy CPU time: a small glider moving across a huge grid will only require the cells that the glider touches to run. For typical patterns, performance is now rather better than a sequential simulation of the same grid, and the performance is much better than the first parallel approach described: after fifty generations on a randomly-initialised large grid, this approach was a factor of 15 faster than the original approach, and the relative performance increases further as the number of active cells decreases. However, it still uses far more memory than the sequential approach, as there is a dormant process for each grid cell with a number of channels attached to it.

Since Life cells do not care about which neighbouring cell a change message was received from, we can use shared channels rather than individual channels. The approach is simply to replace the eight channels coming into each cell with a single shared channel; each of the eight neighbouring processes can send to the shared channel. This reduces the simulation's memory usage somewhat—and it significantly simplifies the "wiring" of the grid, making it easier to experiment with alternative topologies such as Penrose tilings [159].

We can do away with the other memory overhead—the dormant processes sitting around—by forking processes ▷ **Just In Time**: rather than resigning from the barrier when nothing is going on, processes would actually exit and hand the end of their input channel off to a ▷ **Factory** "ether" process, representing empty space. The ether could then respawn the cell process when it was next needed. (This approach was never implemented as part of TUNA, although it was applied later in CoSMoS.)

As a final optimisation, we could scale up from individual cell processes to processes that represent groups of cells. The choice of internal implementation within each group could then be made based on the behaviour of the cells—for example, HashLife could be used when repetitive behaviour was detected.

3.5.3 Blood Clotting

Blood clotting (haemostasis) is an emergent behaviour. Wounded tissue emits chemical clotting factors into the bloodstream. Platelets, which are always present in the blood, become sticky in response to encountering clotting factors. Sticky platelets tend

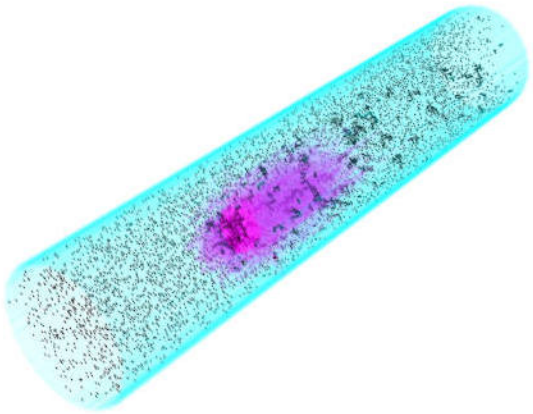


Figure 24: TUNA blood clotting simulation

to clump together and adhere to the tissue around the wound, slowing and eventually staunching blood loss. TUNA's experimentation with haemostasis proceeded in several stages.

Initial efforts focused on platelets as cellular automata in one, two and three dimensions [170], following simple, deterministic rules. The same behaviours were then implemented using CSP||B, an integrated modelling approach where the interactions between B machines are specified using CSP [229]; the CSP elements were model-checked using FDR to guarantee liveness [207], and the simulation then implemented using occam-π [255, 259]. This model was unexpectedly complex, with much of the complexity coming from the rules necessary to ensure that clumps of platelets moved together, with each platelet "dragging" the next along.

The model was then extended to include clots as explicitly-modelled entities [187]. A clot is a *super-agent*, composed of a group of cooperating agents; when two clots collide, they merge to become a single clot. Various approaches were experimented with for this, including a solution using occam-π's mobile channels as "tails" for the clots, allowing clots to communicate directly using a ▷ **Private Line** [254].

Later extensions to the model included the addition of platelet activation, along with diffusion of chemical factors; factors were also represented as agent processes. This final model was implemented in occam-π, making use of the ▷ **Client-Server** pattern, along with ▷ **Phases** to control access to a shared-memory array that represented the contents of 3D discretised space in a compact format. Visualisation and interaction were achieved using the volumetric rendering facilities of the VTK library [208] (figure 24).

This simulation was then distributed across a cluster of PCs using pony, with careful attention paid to batching of communications to reduce latency effects. The shared-memory array was divided into equal-sized cubes, with adjacent cubes overlapping by a few voxels; "border" processes ensured that the duplicated volumes were synchronised correctly, and agents entering the duplicated areas migrated to the correct host. The resulting simulation could support interactive simulation with hundreds of thousands of platelets in a volume of forty million locations.

3.6 occvid

occvid is an occam-π framework for capturing, manipulating and playing back multimedia streams [185]. Media processing frameworks are often structured as a collection of components that can be connected into a ▷ **Pipeline** or an arbitrary graph—GStreamer and DirectShow are widely-used examples—but few make use of concurrent programming techniques to permit efficient scheduling and parallelisation, let alone to ease implementation. The Kamaelia project is a notable exception: a media processing framework originally developed by BBC Research and Development, it is built upon a flexible message-passing concurrent runtime constructed using Python's generator facilities [219]. occvid was built to explore the possibilities of media processing with the mobility facilities provided by occam-π.

Like most multimedia frameworks, occvid consists of a library of processing components, which are modelled as occam-π processes. Components provided include: video and audio codecs (encoders and decoders); readers and writers for various multimedia container formats; input components that read video from DV devices and webcams; output devices that display video frames and play audio frames; and utility processes that synchronise clocks between different processes, process user input, and manage the starting and wiring up of other components.

All multimedia data in occvid is carried as mobile data. When running on top of a conventional operating system, data need only be copied when passing in and out of occvid (from or to a hardware device); if run as part of RMoX, occvid could operate in a true zero-copy mode with processes operating on hardware buffers directly. The performance advantages of this approach are significant given the high data rates typical in video-processing applications—especially so for high-definition video. The low cost of communication in the CCSP runtime means that occvid's performance is comparable with that of the best conventional frameworks when only a single processor is used, and CCSP's parallel scheduling means that better performance is obtained on multicore systems.

Packets of data—which may be raw audio or video frames, or compressed data in a format supported by occvid's codecs—are carried between components using occam-π channels. All packets carry a high-resolution timestamp to allow accurate synchronisation of streams with each other and with output devices. Two models of communication are supported:

- `CT.MM`, a "push" protocol where data flows from input to output;
- `CT.MM.SEEKABLE`, a more complex "pull" two-way protocol where data is explicitly requested by the receiving process, and which allows the receiver to request that the sender seek to a different position in the media stream.

The push protocol is designed for simple applications where data is being processed in bulk; it makes implementing new components very straightforward, and allows processes to be arranged into a ▷ **Pipeline** to process data in parallel with no additional buffering necessary. The pull protocol is intended for interactive applications such as video players (figure 25); it allows efficient movement between different points in the video stream, with the entire processing graph controlled by the component at the output end of the graph. However, since the pull protocol requires that each component obtain an explicit response from its source for each request it makes

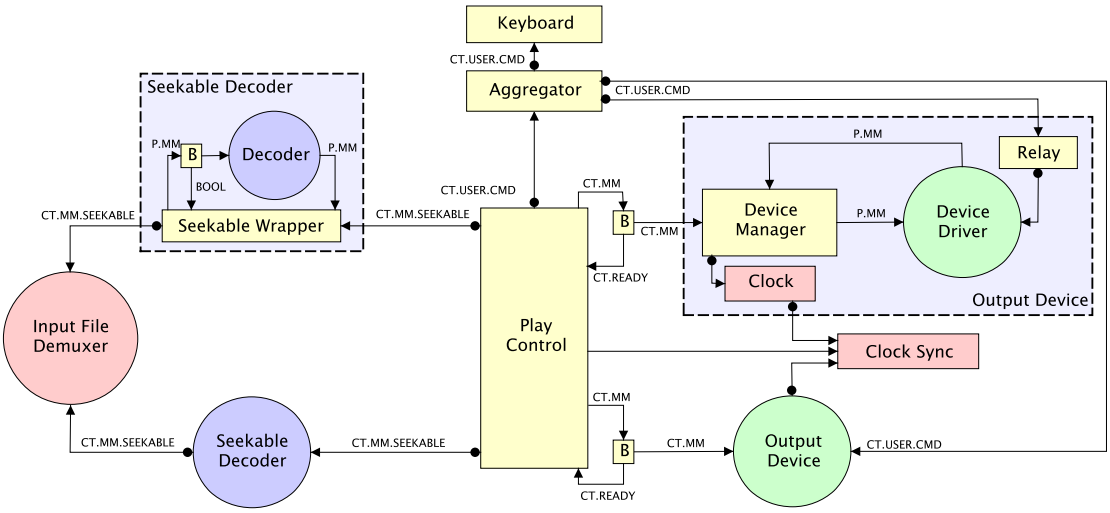


Figure 25: An interactive video player using the pull model (from [185])

(effectively a ▷ **Client-Server** approach), only one component may be processing data at a time: parallel processing requires that explicit ▷ **Buffer** processes be introduced.

For components that follow the ▷ **Filter** pattern, such as codecs, an adapter process, `seekable.wrapper`, is supplied to make a `CT.MM` (push) filter available through a `CT.MM.SEEKABLE` (pull) interface. The push protocol was carefully designed to make this possible, with a "purge" operation that allows the state of a push component to be reset when the stream it is processing is seeking. The result is that most components can be implemented using the push protocol, being wrapped into the pull protocol as necessary for interactive applications.

occvid makes use of "zero-place buffer" processes, which allow one component to efficiently wait for another to be ready to process data. A zero-place buffer sends a do-nothing message using the push protocol and returns a notification once it is accepted; this is effectively simulating choice over output using the "requester" approach often used to implement buffers in occam-π (see figure 47).

occvid makes use of channel end mobility to allow the processing graph to be dynamically configured at runtime, which has a number of applications. For example, most multimedia container formats can contain varying numbers of audio and video streams compressed with a wide variety of codecs. Dynamic reconfiguration allows occvid to start up and connect the appropriate codecs to process a file's streams once it has been opened and its headers read. This allows a video player to be constructed that can play files in any format supported by occvid, with any configuration of channels. Another example application is a video editing system that captures video from cameras; occvid is able to start up the appropriate processes to read video from a camera when it is connected to the computer.

3.7 LOVE

LOVE is a framework for music synthesis, audio processing and live programming in occam-π.

Audio Synthesis and Process Orientation

Electronic music would be more appropriately called *computational* music: generating and processing sound using mathematical operations. Electronic synthesisers date back to the 1940s, with the Hammond Novachord being a notable early example that generated polyphonic sound using oscillators, dividers and filters—all implemented using analogue electronics, in much the same way that contemporary analogue computers were processing abstract data using analogue techniques. Digital techniques were applied to sound synthesis as soon as they became available, with work done on UNIVAC I as early as 1951, and considerable progress made at Bell Labs in the early 1960s. When microprocessors appeared in the 1970s, synthesisers were some of their first applications, initially for control of analogue components and then for increasingly complex processing and direct digital synthesis of sound in systems such as the Synclavier and Fairlight CMI.

Today's electronic musicians primarily work with "soft synth" software running on general-purpose computers, sometimes augmented with DSPs or other specialised processing hardware; today's high-end keyboards are often PCs running a conventional operating system such as Windows or Linux. User interfaces and programming models remain heavily influenced by the conventions of analogue synthesis, however.

A typical synthesiser starts by generating "pure" waveforms using *oscillators* [176]. In theory any sound may be built up simply by combining sine waves—but in practice, it is more convenient to start with a wider range of timbres (sine waves, square waves, triangle waves, white noise, and so on). Alternatively, it may replay samples captured from a recording, or use the sound from another instrument (such as voice or electric guitar) as a source waveform in real time.

The synthesiser then applies *operators* to modify and combine waveforms. Operators may include amplification, filters that affect particular frequency bands, mixing, distortion, modulation, delays, compression, limiting, and any other functions that can be described mathematically. The frequencies and durations of notes, as well as the parameters of the various operators applied to them, are often controlled using a standard protocol such as MIDI [138] or OSC [265], allowing synthesisers to be connected together and orchestrated by a single player or application.

Electronic musicians tend to think of synthesiser configuration in terms of connecting up boxes. For analogue synthesisers, this was literally true: a *modular* synthesiser consists of a collection of different electronic devices, each of which implements one component of a synthesiser, and which may be connected (and expanded, modified, and so on) in any way the musician desires. Modular synthesisers are widely considered to be intuitive and highly flexible; as a result, the configuration interfaces for many soft synths, hardware synthesisers, and general audio processing frameworks—such as Pure Data [175] and Max/MSP—mimic the physical interface of a modular synthesiser.

Electric guitarists are also familiar with this approach: guitar effects boxes follow standard interfaces that allow them to be connected together in arbitrary ways, with

```
PROC amp (CHAN SIGNAL in?, VAL REAL32 factor,
          CHAN SIGNAL out!)
  WHILE TRUE
    SIGNAL s:
    SEQ
      in ? s
      out ! signal ([i = 0 FOR BLOCK.SIZE | s[i] * factor])
:
```

Figure 26: The OAK amplifier process, which multiplies samples by a fixed value

many guitarists using several different effects to provide them with a variety of different tonal qualities.

At a superficial level, this approach to the design of audio processing systems bears a strong resemblance to the graphical techniques used to design process-oriented systems: operators are processes, lines are channels, and so on. This suggests that musicians and audio engineers may be potential users of process-oriented techniques; they would certainly agree that building reliable, scalable systems is important. Furthermore, audio engineers are acutely aware of the problems caused by format and range errors in their "data", and of the effects caused by cycles in their networks.

However, digging deeper into the semantics of these systems reveals ways in which the usual semantics of audio synthesis systems differ from those provided by process-oriented frameworks—especially those provided by synchronous channels. For example, musicians often expect to be able to reconnect the cables between their synthesis operators while the synthesiser is in operation—and without causing their synthesiser to deadlock or even interrupt its audio output. LOVE was built to explore the possibilities of process-oriented software in the field of audio synthesis and manipulation, and to see how these mismatches could be addressed.

OAK

The first prototype of LOVE was OAK: the occam-π Audio Kit. OAK provides a library of processes that correspond to the fundamental components of an audio processing system described above: oscillators, operators, processes to interface to the computer's sound and sequencer hardware and to visualise waveforms graphically, and utility processes (such as ▷ **Buffer** s and ▷ **Delta** s) to help wire up the process network. Channels—which correspond to the patch cords in a modular synthesiser—may carry audio signals, or control signals such as oscillator frequencies. Some operators may usefully be applied to either type of signal—for example, adding the output of a low-frequency oscillator to a frequency control signal results in a vibrato effect. More complex operators can be built from simpler operators: a delay line can be used to implement a comb filter or an echo effect [176].

For simplicity, OAK is entirely deterministic, with channels carrying `SIGNAL`s—fixed-size mobile arrays of samples—and processes written in an ▷ **I/O-SEQ** style. Every process in the system therefore loops S/N times per second, where S is the sample rate (typically 44,100 samples per second) and N is the array size. The resulting processes are usually very simple (figure 26).

The most complex process in OAK is the sine-wave oscillator, since it must handle

smoothly changing frequency while maintaining an accurate frequency and waveform on its output; accurate oscillators are necessary for good-sounding audio synthesis, and there are standard techniques for implementing software oscillators with minimal distortion [223].

To control the synthesis components, OAK includes a simple sequencer. occam-π is a fairly natural language for writing music in, since conventional music notation is primarily concerned with representing sequential and parallel composition, and repetition. A piece of music in OAK is thus an occam-π program which generates a series of MIDI-like note messages down channels. Adaptor processes convert these messages into frequency and trigger messages on control channels, analogous to the CV and Gate voltage inputs on analogue synthesisers. Processes are also provided that convert keypresses and MIDI events to notes, to allow OAK to be played as a conventional synthesiser.

Live Programming

Artists have been creating sounds and music by writing software since the 1950s, and since the early 1980s programming has been an important part of mainstream electronic music. Programming is not normally done as part of a performance, however.

The *live programming* movement aims to change this by making the construction and manipulation of software part of a live musical performance [228]. An audience member does not have to be able to play a conventional instrument to appreciate a conventional live performance. Live programming—encouraging synthesiser performers to do more of their construction on stage, and make what they are doing more visible to the audience, for example by attractive visualisations of their synthesis systems—tries to involve audiences in the same way, breaking down the common image of the electronic musician hidden behind a wall of keyboards. Live programming also offers the musician more opportunities for improvisation in terms of tonal qualities and generated music, and can allow improvised musical performances to be more closely integrated with visual stage effects such as video and lighting.

Live programming presents a number of interesting challenges. Environments for live programming must be highly expressive, allowing significant changes to be implemented rapidly. They must allow incremental changes to be made to a running program, with control over when changes take effect. They must be robust against programmer error. They need to support soft real-time operation, avoiding output glitches and timing problems; in many cases they need to be able to synchronise their output against external tempo and audio clock sources. Finally, they must provide a development environment conducive to live programming.

Several environments have been developed for live programming—both based on existing languages such as Scheme and Perl, and with entirely new syntaxes, both graphical and textual. For example, the ChucK programming language is designed for "on-the-fly" audio processing [239], and provides a variety of highly-sophisticated facilities for concurrent programming. ChucK processes are called "shreds", and are cooperatively scheduled. The syntax is extremely compact, with a single operator `=>` that is used for time synchronisation, assignment, building pipelines and communicating between processes. ChucK uses a ▷ **Clock**-like approach to time management, with a global timer `now` that only advances as processes attempt to synchronise with it.

The challenges of live programming can be met by a process-oriented environment,

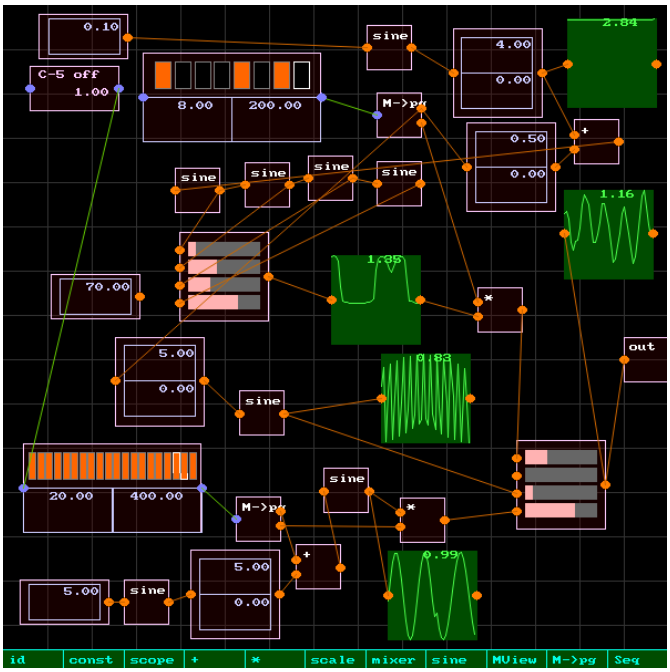


Figure 27: LOVE's user interface, editing a complex synthesis network

but OAK clearly could not be used directly: the occam-π programming language is inappropriate for live programming, because it is not very expressive, it is not visually attractive enough to involve the audience, and it is necessary to recompile and restart the OAK program whenever changes are made. LOVE—the Live occam-π Visual Environment—was an extension and redesign of OAK to support live programming.

From OAK to LOVE

LOVE uses the same approach to modelling synthesis components as OAK—processes connected by channels—but provides a visual environment that allows the programmer to connect up and configure their components using a simplified form of process diagrams (figure 27).

The live programmer picks components from a palette to add to the canvas, and connects them up by drawing channel connections between ports on the components, which are coloured according to the types of data they carry (control messages or audio systems). While connections are being drawn, the user is given immediate visual feedback on which ports may be safely connected. The visual environment applies standard process-oriented design rules in real time to prevent the programmer from constructing incorrect or unsafe programs: for example, the programmer cannot connect two ports of different types, or create an unsafe cycle (see ▷ **I/O-SEQ**) in the network of components.

Ports are implemented using the ▷ **Terminal** pattern. Each output port may be connected to multiple inputs, but each input may only be connected to a single output.

```
PROC amp.component (PROC.CTL? ctl)
  PROC amp (CHAN REAL32 factor.in?, CHAN CHUNK in?, out!)
    ...  conventional Valve process, as in OAK
  :
  STREAM.WIRE? inw:
  STREAM.WIRE! inw.c:
  PORT.CTL? outp:
  PORT.CTL! outp.c:
  PROC.UI? ui:
  PROC.UI! ui.c:
  SEQ
    ctl[resp] ! reg.counts; 1; 1
    inw, inw.c := MOBILE STREAM.WIRE
    ctl[resp] ! reg.stream.in; inw.c
    outp, outp.c := MOBILE PORT.CTL
    ctl[resp] ! reg.stream.out; outp.c
    ui, ui.c := MOBILE PROC.UI
    ctl[resp] ! reg.ui; ui.c
    ctl[resp] ! reg.done

    CHAN REAL32 factor, factor.b:
    CHAN CHUNK thru:
    CHAN VEC.EVENT e0:
    CHAN MOBILE []VEC.ENTRY g0:
    PAR
      value.widget (ui[event]?, e0!, g0?, ui[disp]!,
                    0.1, 0.1, 0.8, 0.8,
                    1.0, factor!)
      overwriting.buf.real32 (factor?, factor.b!)
      dump.events (e0?)
      amp (factor.b?, inw[c]?, thru!)
      stream.port (thru?, outp)        -- Terminal process
:
```

Figure 28: LOVE amplifier component

Input ports are mobile channels, with the input end of the channel registered with a central manager process (an ▷ **Oracle**). Output ports are buffer processes which broadcast the values they receive to a set of channel ends. The manager process has a mobile control channel to each buffer process, through which it can instruct the buffer to connect and disconnect channel ends. The channels themselves are strongly typed at the occam-π level as well as in the user interface. A LOVE component is essentially an OAK component with wrapper processes attached to its input and output channels, and GUI elements added to control parameters (figure 28).

The standard process-oriented design rules can be used to analyse a LOVE network in real time because we know how the LOVE network will be translated into occam-π processes: each output buffer introduces a stage of buffering, but follows the I/O-SEQ rules itself. We have thus engineered a higher-level communication facility that still allows the usual process-oriented approaches to design to be used.

The manager process is also responsible for starting and connecting up components dynamically in response to requests from the LOVE GUI (it is a ▷ **Factory**), and for checking the design rules that determine whether connections are safe. (It does not itself care about the data that channels carry; it only has an abstract idea of channel types.) The manager process is decoupled from the GUI: it would be equally possible to build a "headless" LOVE system that constructed and connected components based on a configuration file, or MIDI or Open Sound Control messages from another application.

LOVE's graphical user interface is itself an interesting piece of process-oriented design. Many of the audio processing components expose some set of adjustable parameters, or provide some sort of visualisation. For example, one component is a simple sequencer that stores a series of notes to play; the component's visual representation includes a display of how many notes it has stored and where it currently is in its playback sequence, and a number entry box that controls the rate at which notes are played back. Another emulates an oscilloscope, displaying audio waveforms graphically.

The GUI consists of a hierarchical process network. The top level (the outermost process) is the whole display; container processes allow this to be divided into different areas, and other processes allow primitive graphics and text to be drawn. A clickable button consists of a container process that combines a rectangle with a text area for the label. Processes are provided for a number of standard GUI elements (buttons, text and numeric entry boxes, sliders), but components such as the oscilloscope can build their own additional GUI elements from the primitives as appropriate.

GUI components are drawn using vector graphics; this allows them to be rendered upon displays of any size and resolution. Lists of vector graphics are propagated up the hierachy from the primitive processes to the top level, with container processes scaling the contained processes' graphics to fit the area they are allocated. Conversely, input events are pushed down the hierachy based on the component the user has selected, until they reach a component that can do something with them. This approach where nested components are connected by channels is also found in the rio window system [167, 169], although rio works with bitmap images rather than vectors, and gives applications direct access to the display.

As a live audio processing system, LOVE has soft real-time requirements: it must minimise the latency between audio or control inputs and audio output. While occam-π does not (yet) provide any facilities for specifying real-time constraints on scheduling and communication, in practice the performance is already adequate; LOVE's I/O-SEQ architecture means that the processor and scheduler load in a LOVE application is fairly predictable. The biggest source of latency is the interaction with operating system facilities for audio input and output, and standard techniques from other audio applications can be applied to minimise (and accurately measure) the processing latency [66].

Synthesis in XC

As an experiment, a cut-down version of OAK was implemented on the XMOS XS1-G4 processor using the XC programming language.

This implementation suffers somewhat from the limitations of the XS1 processor architecture and the XC language. In particular, since XC can only use hardware threads, the entire program must fit into 32 concurrent processes, and the limited amount of

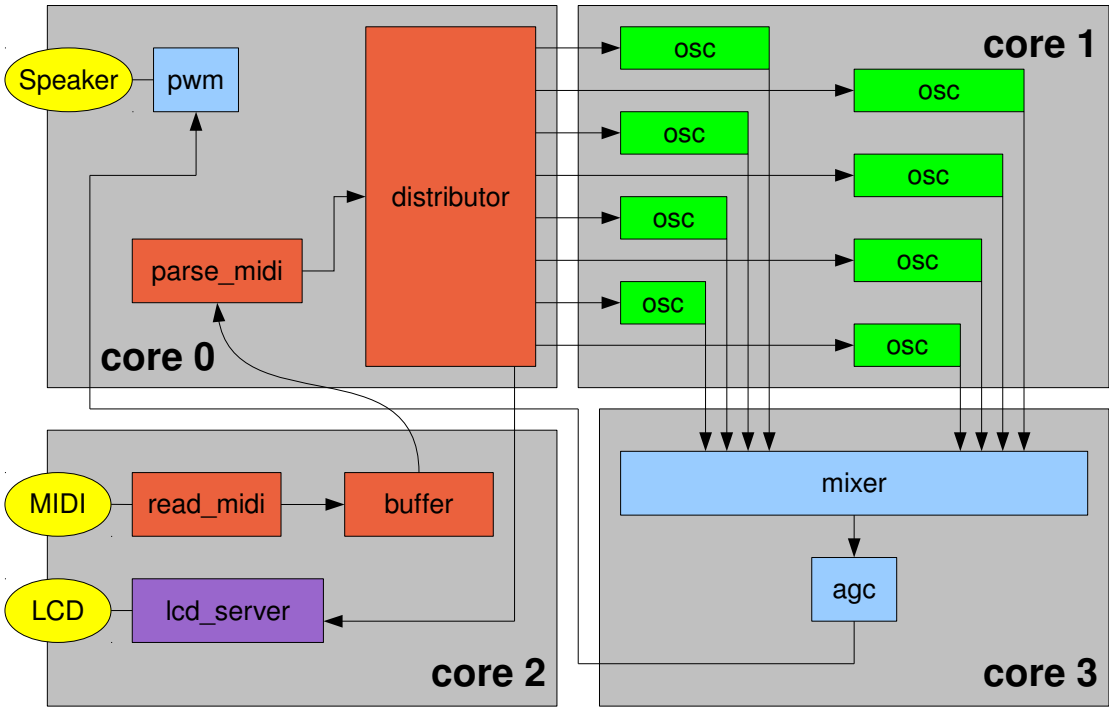


Figure 29: The XC version of OAK

RAM on the XS1-G4 means that large buffers are not possible. The advantage of the XS1 architecture is that everything—including serial input for MIDI, and one-bit digital-to-analogue conversion—can be implemented in software, with safe direct access to hardware moderated by the compiler; overheads are very low, and static guarantees about latency can be made [134, 89].

The XC version of OAK provides MIDI input, eight wavetable oscillators, a mixer, automatic gain control and a 1-bit DAC output. The implementations of the basic components are very similar to those in OAK (figure 30), although the channels in XC carry single samples rather than batches of samples, and samples are integers rather than floating-point values (both owing to language limitations in the early version of XC used).

```
void amp(chanend cin, int mul, int div, chanend cout) {
        while (1) {
                int sample;

                cin :> sample;
                cout <: (sample * mul) / div;
        }
}
```

Figure 30: The XC amplifier process

The XC language does not expose the full flexibility of the XS1-G4's primitive communication operations. It would be possible for a future version of XC to provide a limited form of mobile channel ends, allowing a more dynamic version of LOVE to be implemented. Later processors in the XS1 series are sufficiently cheap and power-efficient for use in synthesisers and guitar effects, so this would have practical applications.

Future Directions

The existing work on compiling occam into DSP code [226] suggests some interesting applications for process-oriented audio processing. Using a graphical tool such as LOVE, a musician could attach components together to create an instrument or effect, test it on their PC, then compile it into DSP code to run on a standalone device. The construction of an occam-π-powered "stomp box" effects unit for guitarists would be an interesting student project.

This project lead to some ideas about transforming process networks for improved efficiency and reduced latency. The "process fusion" rules proposed for CHP [48] could be profitably applied to LOVE applications; fusion could be applied by the manager process automatically.

It would be interesting to consider using the construction of something like this as a tool for teaching process network design, since it shows immediate real-world applications for and analogues of the "Legoland" components. (It would perhaps require students to know some music theory before learning occam, though!)

LOVE's visualisation works well, but does not provide many of the facilities expected in modern GUIs, such as accessibility and internationalisation. It would be possible in the future to build a set of process-oriented wrappers around an existing GUI framework that would present the same convenient interface (as has already been done for some Swing widgets in JCSP). Touch-based interfaces are of particular interest for synthesis, since they allow control elements such as sliders and 2D panners to be easily constructed, and allow several musicians to interact with a single application.

3.8 Occade

Occade is an occam-π module, now included in the KRoC standard library, for programming simple arcade games [192].

As part of their course, students learning concurrent programming at the University of Kent work with a simulation of the *dining philosophers* problem, in which five philosophers compete for access to forks; this problem is widely used as an example of deadlock in concurrent software engineering [71]. Students are asked to add a visualisation to the simulation, and experiment with strategies for preventing deadlock. Before Occade existed, students were encouraged to use simple "ASCII art" character graphics for their visualisation; nonetheless, many students submitted highly elaborate visualisations, adding animation, user interaction and intelligence (such as pathfinding) to the existing simulation. Such extended simulations contain all the elements of simple computer games.

Occade was designed to enable students, even those with a fairly limited command of occam-π, to easily develop programs with animated graphics. The features

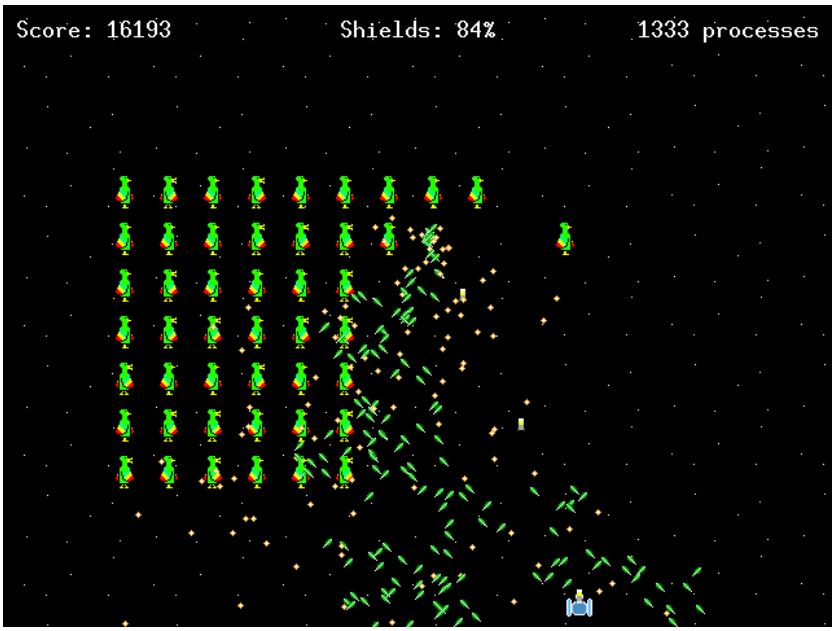


Figure 31: Parrot Attack, one of the Occade examples

it offers were modelled on home computer systems of the early 1980s, especially the Atari 2600 [140]. Occade provides a fixed-size two-dimensional bitmap display with the following features:

- a fixed playfield that is displayed behind all other graphics;
- any number of sprites, shaped graphical images with a transparent background that may be moved anywhere on the screen;
- collision detection between overlapping sprites;
- the ability to load arbitrary graphical images or render text into a sprite;
- delivery of input events from the user in a standard format; and
- some helper functions generally useful for game programmers (such as generating random numbers).

Occade is implemented using the SDL graphics library [212], allowing high-performance graphics on a wide variety of platforms (including all those supported by KRoC). This is achieved through the use of the low-level "occSDL" bindings to the SDL library [72]. Occade does not use the higher-level "sdlraster" module, since this would prevent the use of SDL facilities such as direct access to graphics memory; however, facilities are provided to convert graphics in the occam-π-standard `RASTER` format into sprite images, allowing the existing occam-π graphics libraries to be used for rendering sprite images (such as "life meters") programmatically.

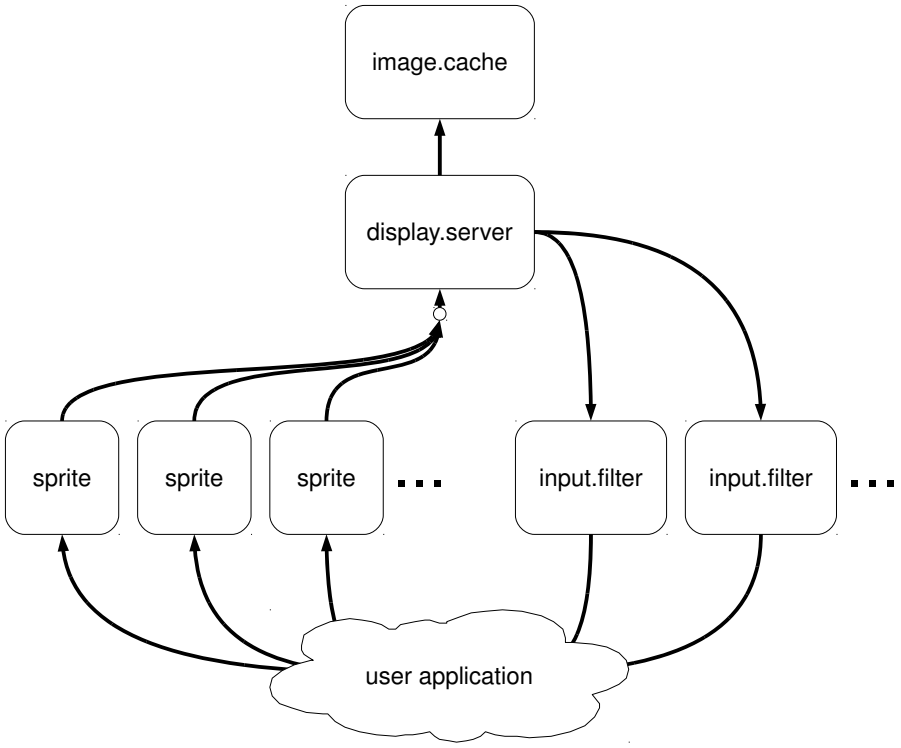


Figure 32: Architecture of Occade

The Occade module comes with several examples, including a couple of complete games: Parrot Attack, a Space Invaders-like game (figure 31), and a clone of Breakout. These illustrate how process-oriented concurrency can be used to easily implement complex animations in a game. For example, when an enemy in Parrot Attack is destroyed, it explodes in a shower of sparks; each spark is modelled as a process controlling a sprite, falling under gravity until it collides with the edge of the screen.

3.8.1 Implementation

Occade's internal architecture and programmer-visible API follow the ▷ **Client-Server** pattern, with a number of different server interfaces being presented to the programmer (figure 32). A "display" server maintains the list of sprites, rendering the graphics display using SDL. Since the display server has complete information about all the objects being rendered, it can tell when only part of the display needs redrawing. The display server is also responsible for computing intersections between sprites' bounding boxes to find collisions. (This can be contrasted with ⋆ **Flightsim**, where collision detection is performed as a distributed activity by all the players in the ring.) A sprite may indicate that it does not participate in collision detection; this is useful for sprites used for things like score displays that should not interact with other sprites.

Each sprite is represented by a process which presents a "sprite" server interface; it packs the sprite's information into a compact form that can be easily rendered and

scanned for collisions. An "image cache" server manages the loading of sprite images indexed by filename, caching each image when it is first loaded so that sprites can change their appearance rapidly.

Input events are handled by "input filter" servers, which connect to the display server and select only the subset of events that the programmer is interested in (for example, only keyboard events). Any number of input filters may be attached to the display. This allows the job of responding events to be easily divided between multiple processes—for example, each player in a multi-user game may have their own input handler.

To allow Occade to be used by student programmers who may not have been taught how to use channel bundles, the details of starting and communicating with server processes are hidden behind convenient `PROC`s. The programmer can therefore think of the channel bundle ends as opaque "handles" to sprites, event filters, and so on.

3.8.2 Sending Events

Occade has two types of server—sprites and input filters—that both present a server interface, and need to be able to deliver events back to the process that would normally act as a client. In client-server terms, this means that they act as both a server (responding to requests from the application) and a client (initiating event reports back to the application). For these servers, the Occade interface uses channel bundles which bind together both a conventional request-response pair for the server interface, and a separate channel for delivering events.

To avoid blocking the display server—which would prevent the display from updating—we must ensure that no interaction with the display server may cause it to block. Requests to the display server are only made by sprite servers, input filter servers and similar processes provided by the Occade library (which are programmed such that they will always complete a client-server conversation with the display server without blocking). However, the delivery of input and collision events could cause the display server to block if the application was unable to immediately accept them.

To prevent this, events are delivered asynchronously through infinite `Buffer` processes; delivering a new event to the buffer will never block, and the application can then process events at leisure. Delivering events over a buffered channel would be equivalent—but occam-π does not provide buffered channels.

3.8.3 Future Directions

In the 2009 and 2010 instances of the Kent concurrency course, several students with varying degrees of programming experience chose to use Occade for their dining philosophers visualisations. Feedback has so far been positive, with all students who attempted an Occade solution having completed at least a basic visualisation, and some developing highly-creative solutions that mimicked games (figure 33). Even students who only implemented the basic visualisation were able to produce a more visually-attractive solution than they could have done using ASCII art, and several drew their own custom artwork. Students will be encouraged to make more use of Occade in future years.

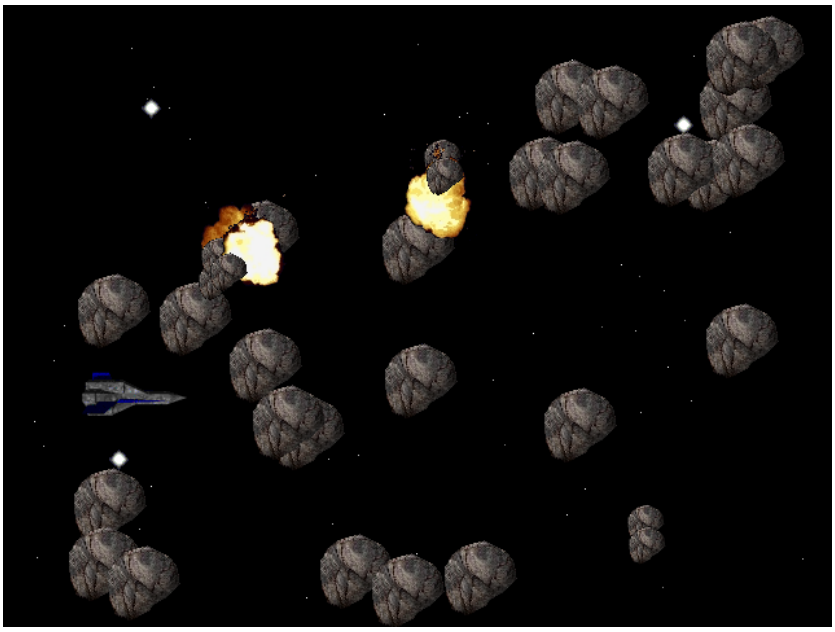


Figure 33: An Occade game written by an undergraduate student

At the moment, Occade only provides facilities for graphical output. Support for sound is an obvious extension: most games include sound effects and background music, both of which could be controlled using process-oriented interfaces. Providing a convenient interface between Occade and the ⋆ **LOVE** audio components would allow the construction of games with generative, reactive music, where the mixing, instrumentation and even the arrangement of the music can change in real time in response to player actions.

Of course, many modern games make use of three-dimensional graphics. OpenGL provides a widely-supported interface for 3D graphics rendering [213], and the occGL module provides OpenGL bindings for occam-π [72]. A process-oriented games framework could provide processes that represent 3D objects in the same way that Occade's sprites represent 2D objects. However, this could be better achieved by providing bindings to an existing 3D game engine such as Cube [233] or OGRE [150], rather than using OpenGL directly; such engines abstract away much of the complexity of modelling a 3D world.

A simpler approach would be to make Occade use OpenGL rather than SDL for rendering its display. This would be straightforward since the use of SDL is entirely encapsulated within Occade, and would allow the use of OpenGL facilities—such as 3D screen transitions, or more attractive transparency and layering effects—within a 2D Occade game.

3.9 CoSMoS

The CoSMoS project—Complex Systems Modelling and Simulation—started in 2008, building on the success of TUNA to "build capacity in generic modelling tools and

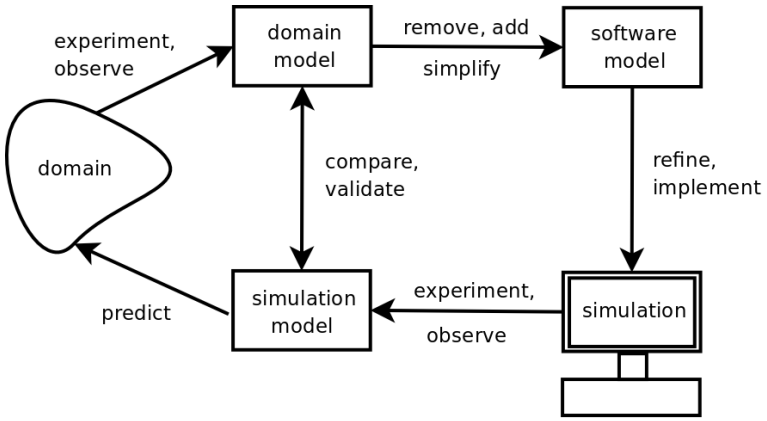


Figure 34: The models defined by the CoSMoS Process

simulation techniques for complex systems, to support the modelling, analysis and prediction of complex systems, and to help design and validate complex systems." [64] CoSMoS aims to develop a development process for the construction and use of complex systems simulations across all fields of scientific endeavour. It is an interdisciplinary, multi-institutional project with researchers and partners from a variety of backgrounds.

For the purposes of CoSMoS, complex systems are those in which consist of many interacting components, and often—but not always—exhibit emergent behaviours as a result of the interactions between the components. The *CoSMoS process* makes use of both models and simulations (figure 34). Models are abstractions used to describe and reason about a complex system; simulations are executable models that can be used to perform experiments.

At the moment, most scientific simulations are built from scratch in an ad-hoc fashion. This causes a variety of problems: considerable duplication of effort between different projects; inefficient implementations; the difficulty of reasoning about simulation validity; and the prevalence of buggy implementations that do not correspond to the models described in the associated publications—a problem that is almost never caught by peer review, and which caused significant difficulties for several CoSMoS case studies.

The CoSMoS process attempts to make use of the practices of agile software development in the development of scientific software, recognising that most researchers do not have the time or inclination to follow a rigid development process. It makes use of design patterns to capture best practices at all stages of development: design, implementation, application, data analysis and validation. It emphasises the consideration and documentation of assumptions and simplifications to aid validation, and defines the roles that the partners in a complex systems project may engage with each other in.

The CoSMoS approach is driven by case studies, with the investigators experimenting with a variety of different complex systems, documenting their efforts, and extracting the common practices that worked well as design patterns. Sources of case studies included: standard textbook examples such as the Boids model of bird flocking [179]; interesting papers such as Martyn Amos's study of annular sorting in ant colonies [8];

and a number of real applications from CoSMoS partners, including models of plant populations [90], lymphocyte rolling in high endothelial venules [172], granuloma formation [81], and social exclusion.

The simulations built by CoSMoS have been primarily agent-based, with agents modelled as processes using process-oriented techniques. The primary aim during implementation has been for scalability rather than straight-line performance, this being a major challenge for complex systems simulations and a strength of the compositional process-oriented approach. Many complex systems require large populations to adequately investigate emergent behaviours (as some emergent behaviours behave differently at different scales). Some overhead is therefore acceptable if it allows the scientist to run bigger simulations by making use of multiple CPUs and distributed systems. Simulations have been implemented for CoSMoS in a number of different environments (occam-π, JCSP, Python, C++, NetLogo, and on FPGAs using VHDL); an explicit aim of the project has been to support multiple implementation languages, even within a single simulation.

3.9.1 Occoids

The first case study considered for CoSMoS was Craig Reynolds' *boids*, a well-known and widely-implemented simulation of bird flocking [179]. Boids move in a two- or three-dimensional continuous space, and can see other boids in a fixed radius around them. They follow three simple rules: move towards the centroid of the other boids they can see, attempt to match the average velocity of the other boids they can see, and avoid getting too close to other boids or obstacles. Bird-like flocking behaviour, including pathfinding and obstacle avoidance, is achieved as an emergent property of these simple rules.

The first implementation of boids for CoSMoS was Occoids, written in occam-π [10]. Boids turned out to be a highly useful case study: it captures many of the important features of other complex systems; a wide variety of interesting behaviours can be obtained by adjusting and extending the rules; and it is very easy to see whether a boids implementation is working correctly just by visual inspection, since even subtle errors result in distinctly different patterns of flock behaviour. Occoids has thus been modified and reimplemented several times during the course of CoSMoS. Furthermore, the space model initially developed for Occoids was abstracted out to form the basis of the simulation framework later used for many of the later CoSMoS case studies.

Most of the complexity in Occoids comes from this model of space, designed to allow efficient, scalable, parallel simulation. From the TUNA Life simulations (section 3.5.2) came the idea of dividing space into a (possibly sparse) network of regions represented by server processes; from the TUNA work on blood clotting (section 3.5.3) came the idea of mobile agents that move from location to location.

In Occoids, space is divided up into regions, with each region represented by a ▷ **Location** process. Each location contains an arbitrary number of *agent* processes (boids and obstacles), and keeps track of a local position for each, relative to the centre of the region. Locations are connected much as cells are in a grid-based model; each location process has a shared channel bundle which its neighbours have access to, and provides a server interface that allows clients to enter a cell, move around, and retrieve a list of agents along with their positions.

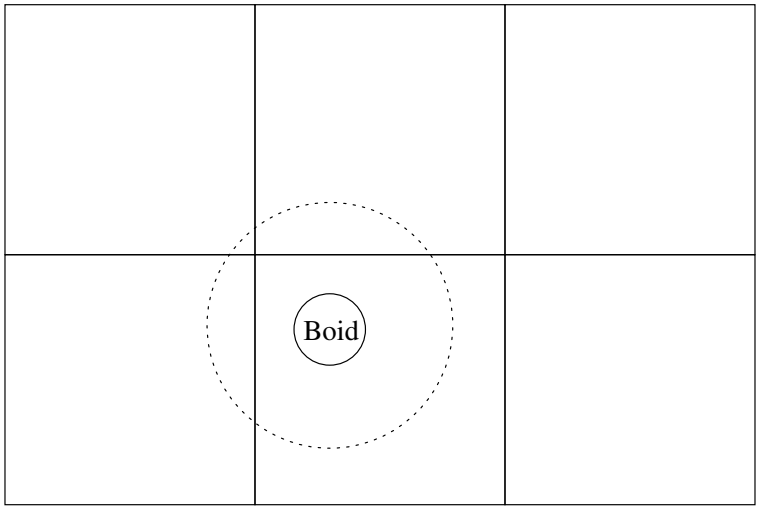


Figure 35: A boid's field of vision

The first thing that each boid must do on each timestep is to "look around" for other agents in its neighbourhood. To do this, the boid needs to gather the contents of all the cells that intersect with the region it can see. The Occoids agents are restricted to seeing a circular region with a diameter of at most one location, which means that it is sufficient to look at the location the agent is in and the eight surrounding locations. Figure 35 shows a boid's field of vision in the partitioned continuous space model.

Since all agents in each location need to look at the same set of nine locations, we can save some effort by delegating this task to a shared *viewer* process. Each location has a viewer process permanently attached to it, and on each time step the viewer updates its view of the surrounding world. The viewer process then provides a server interface to the agents in the corresponding cell which allows the agents to obtain their local view.

In order to guarantee that the agents see a consistent view of the world, we must make sure that all the viewers are updated after the agents have finished moving, but before they look again at the start of the next time step. Each timestep is thus divided into multiple ▷ **Phases**, with a global barrier synchronisation between each phase:

- In phase 1, the viewers request the contents of the surrounding cells.
- In phase 2, the agents request their view from the viewers, compute their new velocity, and send movement messages to their locations.

Once a boid has looked around, it decides in which direction to move by sending a movement vector to its location. The location responds by updating the boid's position. If the boid remains within the same location, no further action is necessary. However, if the boid has moved outside the bounds of the location, it must be moved into the next location in the correct direction. To do this, the location responds with a message instructing the boid to move into the next location. This approach makes it possible to move across multiple locations in one movement step: upon entry, the first new location can respond immediately with another "move" message, and the agent thus reaches the correct target location by an iterative process. This process is illustrated using a filmstrip in figure 36.

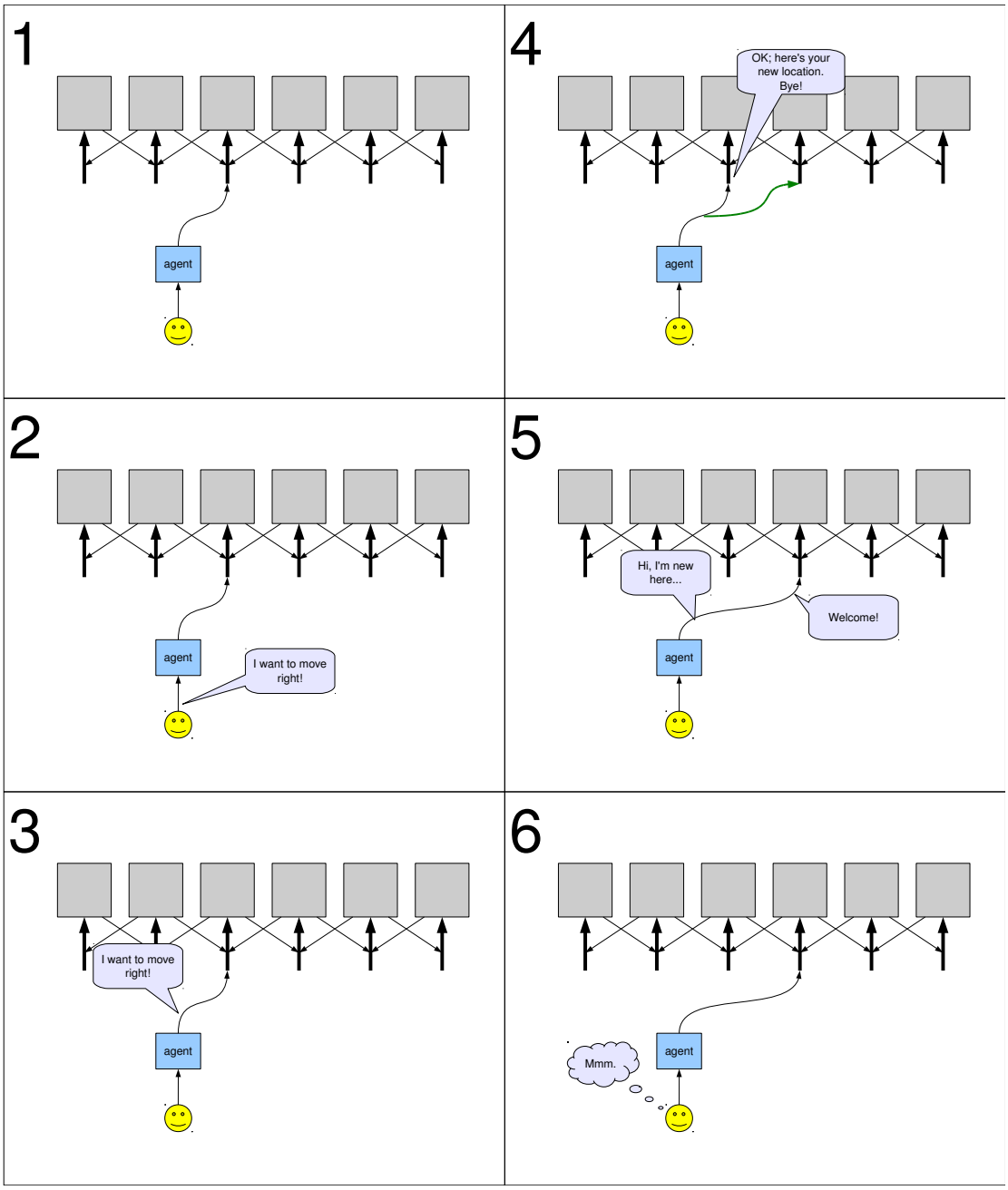


Figure 36: Agent movement in the CoSMoS space model

This was initially solved in a different way: originally, the current location was responsible for handing off the server end of the agent's channel bundle to the new location. However, doing this communication directly would create a cycle in the graph of client-server relationships (since two neighbouring locations would be clients to each other), and could result in deadlock—in particular, if two neighbouring locations tried to send a location to each other at the same time. The ▷ **Messenger** pattern was the solution: when a location wished to relocate an agent, it forked off a short-lived process that was responsible for transporting the channel bundle end to the new location. However, delegating the work to the agent process works equally well, since in the CoSMoS simulations every agent (even one that never moves, such as a tree in Occoids) must have an associated process.

In order to avoid complicating every agent with code to handle movement—especially in simulations with many types of agent—this functionality is abstracted out into a separate *agent manager* process that hides the details of this from the agent itself. The manager provides a simplified interface to the agent, supporting only "move" and "look" requests. Adding this level of indirection simplified later work: it made it possible to have arbitrary behaviour inside the space model that is not visible to the agent itself.

3.9.2 Distributed Simulations

The CoSMoS model of space was extended to allow distributed simulation [201]: simulations running across several computers, connected using message-based network links (rather than sharing memory, as multicore and multiprocessor systems do). Each location upon which programs can run in a distributed system is called a *host*. It is sometimes useful to assign more than one host to a physical machine—for example, to balance load fairly between a cluster of machines of different processor speeds.

The first distributed version of the CoSMoS space model used pony to distribute the boids across several hosts [10], with each host simulating a rectangular region of space modelled by several location processes. pony provides networked channel bundles for occam-π programs, with exactly the same semantics as local channel bundles; the only visible difference is the significantly increased latency compared to local communications. To obtain good performance, it is generally best to engineer a distributed application in such a way as to minimise the number of cross-host communications, and perform cross-host communications in parallel as far as possible.

To start with, the existing simulation was modified to set up the same network of processes across a distributed application. The resulting simulation worked exactly as before, but ran very slowly; furthermore, it got even slower as boids migrated between hosts. There were two major sources of inefficiency:

- Neighbouring viewer processes must request the same view information from a location on the other side of a network link. For local communications this is not a problem, since only a reference is transferred; for network communications the data must be copied.

- More seriously, agent processes continue to run on the host they were started on, so once moved to a new host, every communication they do is across a network link.

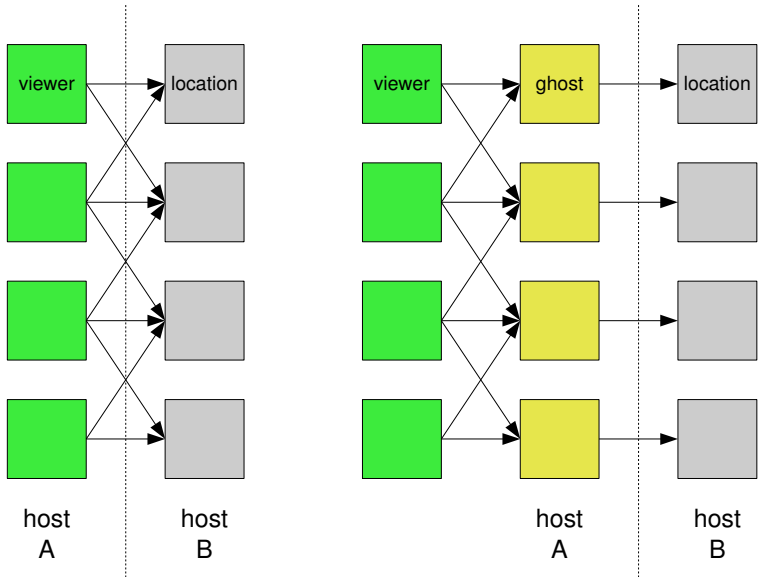


Figure 37: Introducing ghost processes for locations

To solve the viewer problem, the ○ **Proxy** pattern was applied in the form of *ghost* processes, which cache the contents of a location on the other side of a network link (figure 37). Viewer and agent processes that would ordinarily communicate with a remote location are instead given a channel bundle to the corresponding local ghost (which provides the same server interface as the remote location). Since ghost processes must update their cached contents before viewer processes try to read it, an additional phase was introduced to the simulation:

- In phase 1, the ghosts request the contents of their corresponding locations.
- In phase 2, the viewers request the contents of the surrounding cells.
- In phase 3, the agents request their view from the viewers and send movement messages to their locations.

The agent problem was solved using mobile processes. In response to moving to a location on a different host, an agent can be told to suspend itself: pack up its internal state and terminate on the originating host. The state is moved to the destination host, where a new agent process is started using the existing state. This is straightforward to implement: when an agent attempts to move into a ghost (rather than a real location), the ghost replies to the agent with a "suspend" message, and then signals the real target location to spawn a new process (figure 38).

A sample process network at a host boundary in the final model is shown in figure 39. The cycle time of the resulting simulation is approximately equal to that of the single-host simulation plus the network latency. The distributed simulation runs successfully on clusters of networked PCs (such as the Display Wall at the University of Tromsø, for which it was further extended to support user interaction; figure 40).

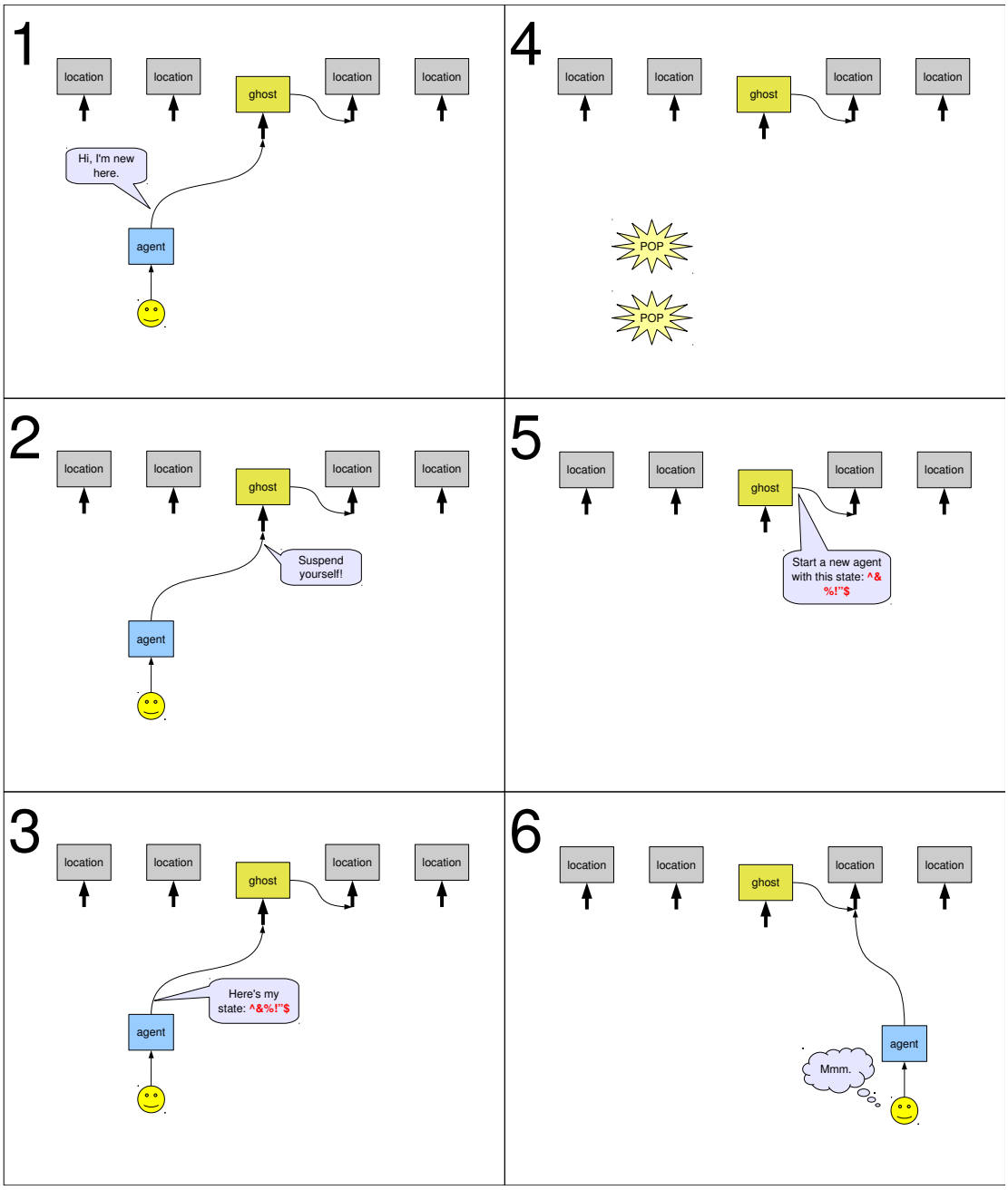


Figure 38: Migrating processes between hosts in Occoids

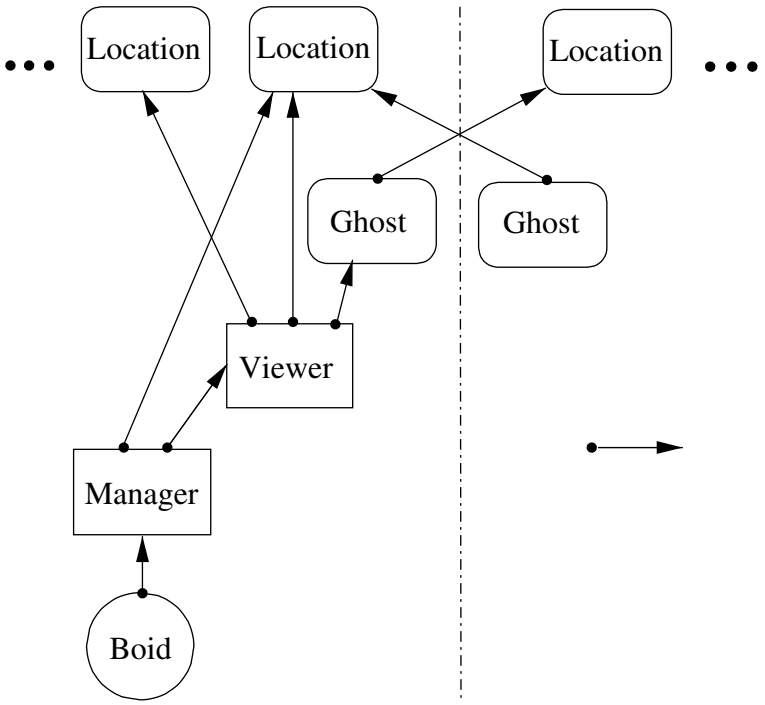


Figure 39: Process network at a host boundary in Occoids

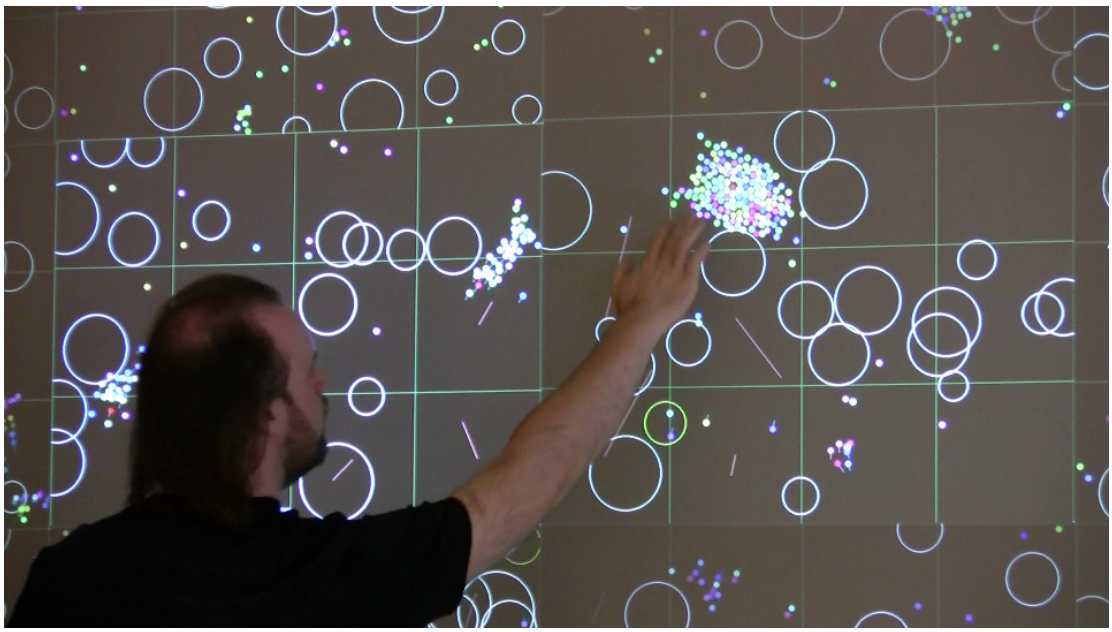


Figure 40: Occoids running on the Display Wall

The cycle time remains approximately constant as the simulation is scaled from two to sixteen hosts. This is as expected, since each host only needs to communicate with its immediate neighbours. The constant factor in the simulation—caused by pony's implementation of CSP communication semantics requiring additional round-trips in network communications—was reduced by using the Trap asynchronous messaging library (section 3.3.5) rather than pony; network channels are only used by the ghost processes, so it was straightforward to adapt the ghost-to-real-location protocol to work with asynchronous messaging, by adding additional acknowledgements where necessary.

3.9.3 Generalised Space Model

The aim of CoSMoS is to develop *reusable* techniques, so the obvious next step was to use the approaches developed in Occoids to implement simulations of different complex systems.

The first reuse was for ant-based annular sorting [8], in which ants sort eggs into rings by size by picking up poorly-placed eggs and dropping them when they find a better location. Ants and eggs are modelled as agent processes. To allow ants to carry eggs, the space model was extended so that agents could pick up other agents (removing them from their locations), and put them down elsewhere. This was the only change necessary to the space model for this use, since the notion of space is exactly the same as in Occoids—two-dimensional continuous space.

More complex requirements were introduced for a simulation of lymphocyte migration [172]. Lymphocytes are white blood cells that pass from the blood system into the lymphatic system inside *high endothelial venules* (HEVs), specially-adapted areas of the network of blood vessels in a lymph node. Lymphocytes are carried through the HEV as part of the normal flow of blood, and can bind to the cell wall; they initially bind weakly, "roll" along the wall until they are sufficiently activated, and pass through the wall into the lymph node. To model this to a reasonable degree of accuracy (especially regarding the interactions between the lymphocytes and the HEV wall) requires 3D space, which meant extending the space model to support three dimensions, with location processes representing thin "slices" of the HEV.

The model was then extended further to treat the incoming bloodstream and outgoing lymphatic drain as locations; since the experiments that were to be performed involved counting the numbers of lymphocytes in each "state", this simplified the collection of results. The position information carried by each agent thus changed from a 3D coordinate to a coordinate plus a state number, with the coordinate only being used when the agent was in the HEV. The lymphocyte simulation was prototyped by copying and modifying the 2D code, with significant changes made to support the more complex position information—clearly not a tenable approach from a software engineering perspective, especially as nearly all complex systems simulations involve agents moving within some kind of coordinate space. The space model was therefore extracted as a reusable component that can be parameterised to support different types of simulation.

Ideas of space in complex systems vary in many ways: space may have varying numbers of dimensions, including very large numbers of dimensions for optimisation problems; it may be continuous or discretised; it may be Cartesian, partially-Cartesian or non-Cartesian; it may be finite, wrap around or infinite; and it may require locations

themselves to have attributes, such as weather or information deposited via stigmergy. A generalised space model must be able to cover as many of these possibilities as possible.

In the CoSMoS space model, space has a three-level structure. Each point in space has a unique *position*. Space is divided into uniquely-numbered *locations*, which are groups of *adjacent* positions. Sets of locations are then assigned to *hosts* in a distributed simulation. For each simulation, the space model is provided with:

- a data type representing a position;
- a data type representing a position update (a change in position);
- a function to apply a position update to a position, giving a new position;
- a two-way mapping between positions and locations;
- a two-way mapping between locations and hosts.

This model has proved to work well for a number of CoSMoS simulations. For example: in Occoids, the position and position update data types are 2D vectors, and the update application function is addition with wraparound at the edge of the space. For the lymphocyte simulation, the position data type is a 3D vector plus a state, and the position update function can recognise when lymphocytes pass out of the HEV into the lymph node.

3.9.4 Ccoids

To demonstrate how process-oriented techniques could be applied to programs constructed in a more conventional style, Occoids was reimplemented in C++ as "Ccoids".

Ccoids uses the CCSP scheduler for concurrency, but replaces the server processes in Occoids with objects, and provides a higher-level C++ interface that hides the details of CCSP. Safe access to objects is provided by a shared reference class parameterised over the interface type that the reference refers to. This class offers the same semantics for access control as the explicitly-shared channel bundles used to mediate access to servers in Occoids, and is implemented using the same low-level CCSP facilities. C++ wrappers are provided for CCSP's barriers and forking facilities, which are used in Ccoids in the same way as Occoids. The code makes no use of channels at all—making a request to a server is done by calling a method through the appropriate interface reference.

The resulting simulation is very close in structure to the original Occoids, with slightly better performance owing to the reduced number of synchronisations necessary, and the better optimisation available in the C++ compiler. It offers the same safety guarantees as a result of the use of the client-server model, while using very few facilities that the average C++ programmer will not be familiar with. This suggests that such a high-level interface to CCSP's facilities may be of use for the construction of concurrency object-oriented programs.

This approach has two disadvantages: no static checking is provided, and only simple request-response conversations (i.e. method calls) between clients and servers can be accommodated, meaning that patterns such as ▷ **Loan** cannot be protocol-checked. In practice such a system is not currently much worse off than it is under occam-π,

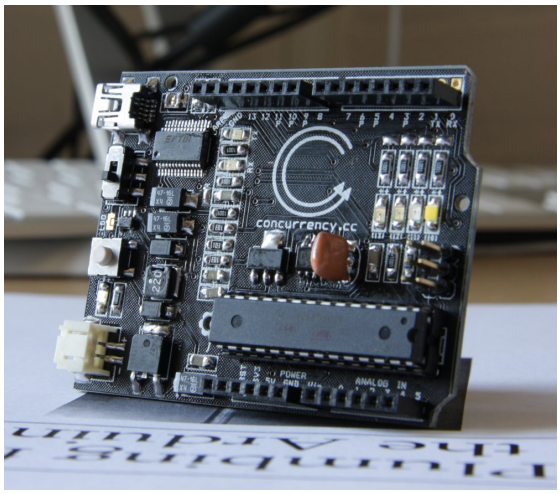


Figure 41: A customised Arduino board (photo by Carl Ritson)

since the existing occam-π compilers do not support checking the client-server pattern or two-way protocols either. (See section 6.3.2 for a discussion of how this could be supported in the future.) These disadvantages must be balanced against all the advantages of using a mainstream programming language for CoSMoS simulations: a much richer set of data types, easier access to external libraries, and more mature, higher-quality compilers and debugging tools.

The Occoids architecture has additionally been reimplemented by CoSMoS partners using JCSP, using CHP, using VHDL on an FPGA, and in a variety of environments as a benchmark for concurrent runtime systems [184].

3.10 Plumbing

Plumbing is a framework for programming Arduino boards using occam-π [118].

The Arduino is a family of development boards for the Atmel AVR microcontroller [13]. Arduino boards provide a USB interface, a power supply, and an AVR processor with all its I/O lines made available through connectors. The Arduino design is open source hardware: it can be freely copied, adapted and built by anybody, and Arduino boards are widely available for around £15 (figure 41). Many variations on the basic Arduino design exist for specific applications, such as robotics or wearable computing.

What really makes an Arduino an Arduino is the bootloader on the AVR chip, which provides a standard way of uploading "sketches" of code. The standard environment provided for Arduino programming is based on Wiring [263], using C++ inside a simple IDE, with a library of standard functions for hardware interfacing. The Arduino and its programming environment have been hugely successful, especially for education, for hobby electronics and for installation art, with a large community of Arduino users and developers and a wealth of documentation available.

However, the Wiring environment provides no support for concurrency—which

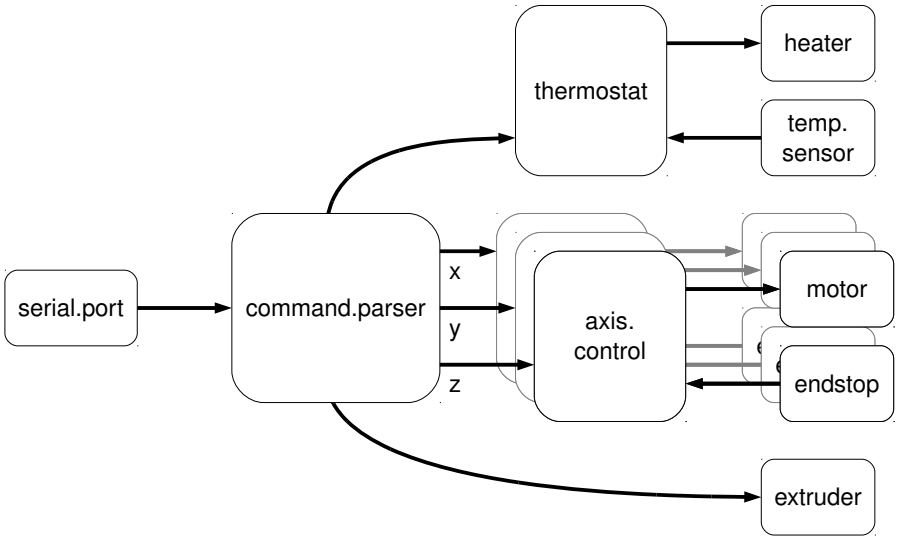


Figure 42: A process-oriented RepRap controller

makes writing many kinds of applications difficult. For example, the control system for the RepRap rapid prototyping system, which prints three-dimensional objects using hot plastic, uses an extended Arduino programmed using Wiring [178]. The RepRap needs to control three servos to position the print head relative to the workpiece, another motor to extrude plastic from the print head, and a heater to maintain the plastic at the correct temperature for printing, while reading from endstop position sensors and a temperature sensor, and responding to commands sent over the USB interface from a host computer. The existing software is essentially written as one very large loop that polls each hardware device in turn. Not only does this approach result in software that is hard to read, it produces a control system that is highly unreliable. If one task takes longer to execute than expected—for example, if an especially hard-to-parse command is sent from the host machine, or a servo fails to reach its intended position—then other tasks can be blocked—such as the heater staying on longer than intended. These kinds of problems are endemic in control systems built using Wiring.

Providing some basic concurrency facilities would make the Arduino programmer's life much easier. In the RepRap, the tasks of responding to sensors and to commands from the host machine should be delegated to concurrent processes (figure 42). The print head can be maintained at temperature by a dedicated process; similarly, each motor can be positioned by a process that controls the motor and reads the corresponding endstop sensors. Indeed, this is one of the kinds of problem that occam and the Transputer were designed to solve: embedded systems that need multiple concurrent flows of control combined with low-level hardware access.

The AVR microcontrollers, however, are a very different environment from the general-purpose PCs upon which the majority of occam-π work has been done over the last ten years. A typical Arduino microcontroller, the ATmega328P, has an 8-bit CPU with 32 KiB of flash memory and 2KiB of static RAM [18]. The KRoC environment supports devices like these through the Transterpreter, a small, portable virtual

Figure 43: A simple Plumbing application (from [118])

machine that interprets extended Transputer bytecode [116, 115].

Running occam-π on the Arduino meant porting the Transterpreter to the AVR architecture. The AVR GCC toolchain makes it appear as a 16-bit CPU, a configuration supported by both the occam-π compiler and the Transterpreter. However, the AVR CPU is a Harvard-architecture device, meaning that it has separate address spaces for program instructions and data—which means that data may need to be copied from flash into RAM, further limiting the space available for the occam-π program's workspace. The Transterpreter was extended to map program memory into the top half of the Transputer memory space, meaning that occam-π programs can be stored directly in flash. Another AVR-specific modification allows processes to wait in an efficient way for an interrupt to fire. The resulting device is roughly equivalent to an INMOS T212 Transputer in terms of computational ability and memory space available, although rather slower owing to the interpretive runtime.

Making occam-π *useful* on the Arduino means providing modules that support the construction of Arduino applications. An `avr` module provides occam-π equivalents of the AVR C headers, allowing programs to support multiple AVR variants easily. Upon this is built the `wiring` module, which mirrors the Wiring API, allowing straightforward ports of existing Arduino applications. However, neither of these provide any features to support concurrent programming.

The `plumbing` module provides a library of processes and protocols that can be used to construct concurrent Arduino applications. Many useful applications can be built simply by connecting together existing processes (figure 43). The processes and protocols were inspired by the electronic components that an Arduino is likely to be attached to; channels carry logic levels, and processes are provided that connect to input and output pins (sending messages only when the value carried changes), debounce switches, detect edges on a waveform, oscillate at a defined rate, act as a flip-flop or an inverter, and so on.

When used on the Arduino, the Transterpreter is simply a program that can be uploaded to the AVR using the Arduino bootloader. This means that Arduino users can easily experiment with occam-π without needing any special hardware to reprogram their chip, and they can move back and forth between occam-π and C++ programs easily. Since the Transterpreter binary is quite large (18 KiB), the occam-π tools allow the user to just upload a new version of their occam-π program, leaving the existing Transterpreter intact in memory, which significantly speeds up uploading.

The Transterpreter does not, however, make use of the existing Arduino C++ library; since hardware access from occam-π is very straightforward, using the library offered no advantage in terms of code simplicity. This also has the advantage that the Transterpreter and Wiring are not tied to the Arduino environment—they can be used to write code for AVR processors embedded in other devices too.

Having the Transterpreter available on a low-cost microcontroller board has distinct advantages for teaching concurrency and computer architecture. The Arduino is sufficiently cheap—and sufficiently well-documented—that it makes sense for an institution to buy a large number of boards for use in class exercises, or even to get students to build their own boards. Plumbing can then be used in compelling, practical exercises that combine concurrent programming with physical computing—and may even feature collaboration between computer science, electronics and art courses. Physical computing with occam-π has already been used successfully in teaching at Allegheny College and the University of Kent [117]; an introductory book is available as a teaching text [118].

Chapter 4

Process-Oriented Patterns

This chapter describes a pattern language for process-oriented software design. The patterns have been drawn from real-world examples of process-oriented software, many of which are described in more detail in chapter 3.

The format in which patterns are described here is based on that of "A Pattern Language" (section 2.4). Each pattern has: a section classification; a name; a brief statement of the problem; a discussion of the solution, the issues surrounding it, and the other patterns to which it is related; a brief statement of the solution (often in the form of a process diagram); and a set of examples that show how the pattern is used.

4.1 Process Patterns

The patterns in this section describe classes of processes that share common features. These include the *interface* the process presents—the synchronisation objects it exposes, and the ways in which it uses them to interact with other processes—and some aspects of its internal behaviour. Many of these processes will be discussed in the context of channels with simple protocols (for example, channels that just carry single integers); these will be called *streams*.

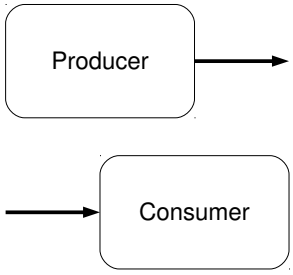
In many environments, it is possible to provide a basic framework for each of these patterns into which the programmer can plug the appropriate custom behaviour for their application. This is especially straightforward in process-oriented frameworks for languages that support type-parametric and higher-order functions. For example, CHP provides a `map` function that implements ▷ **Filter**; as its first argument, it takes the function that should be applied to each data value passing through the filter [50]. There are obvious parallels between such *higher-order processes* and the higher-order functions used for list and data structure processing in functional languages; expressions that operate upon lists can easily be turned into concurrent programs operating upon streams using ▷ **Pipeline**. Similarly, an object-oriented environment may provide a `Buffer` class that provides single-place buffering by default, but can be subclassed to support more complex buffering behaviours.

The occam-π standard library, on the other hand, cannot provide higher-order processes owing to its lack of support for templating, meaning that processes that operate upon an arbitrary type cannot easily be written. Some form of templating or polymorphism greatly eases code reuse in process libraries.

Many of these patterns describe processes that follow predictable patterns of communication—for example, always responding to an input message with an output message. These interfaces may be formally specified using a process calculus such as CSP and checked automatically by a design tool or compiler, as is already done for several process interfaces within RMoX [25]. Future process-oriented frameworks could include a library of communication specifications corresponding to common process patterns against which process interfaces could be automatically checked.

4.1.1 Producer and Consumer

PROBLEM: Values need to be obtained from or delivered to something that is not itself a process.



Producers and consumers are the simplest classes of processes that have any interaction with other processes: a producer process has a single output channel, and a consumer process has a single input channel. A producer connected to a consumer forms the simplest possible process-oriented system.

Producer and consumer processes are common on the edges of a process network as interfaces to non-process-oriented facilities such as hardware devices and operating system streams, and as wrappers around pieces of code that only generate or consume data.

Producer processes are similar to Icon's generators [205]: a producer will usually be blocked, except when a value is required from it.

Examples of Use

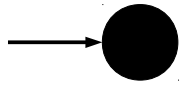
⋆ **Plumbing** provides producer processes that read values from I/O pins, and consumer processes that likewise write values.

⋆ **LOVE** provides a producer process that reads MIDI events from the system sequencer, and a consumer process that writes an audio signal to the soundcard.

In occam-π, the standard way of writing a *print procedure*—that converts some data structure to a human-readable textual form—is as a generator process that writes the string to a `CHAN BYTE` and then exits. The programmer can thus print to the standard output or standard error streams, or to a channel of their choice. ⋆ **KRoC**'s `course` and `useful` modules provide many print procedures that follow this form.

4.1.2 Black Hole

PROBLEM: An unwanted channel must be connected to something.



The most common sort of ▷ **Consumer** process is the *black hole* (or *sink*): a process with a single input channel, which immediately discards any input value it receives [252, 120]. Since writing to a channel from which no other process is reading will block, a black hole process must be used when a process has an output channel that is not needed in the current application.

A "black hole producer" process—that simply never produced any output—would be possible, but in practice it is rarely necessary, since leaving an input channel unconnected has the same effect. Generator processes that emit a fixed sequence of outputs—either a single message, or the same message repeatedly—are useful, though [252]. CHP calls the latter a *white hole* process [50].

Examples of Use

The `raster.display` process in ⋆ **KRoC**'s `sdlraster` module, which provides a bitmap graphics window, has an `events` channel that delivers keyboard and mouse events. Applications that do not need these events must discard them by connecting the channel to a black hole process. This is common enough that the module also provides `raster.display.simple`, which runs `raster.display` in parallel with a black hole process.

When print procedures are used for debugging output, a black hole process can be used to discard the debugging messages when they are not needed; such an approach is used in ⋆ **occam-X11**.

4.1.3 Filter

PROBLEM: Each value in a stream needs to be processed in some way.

Filter processes take a stream of input values, and deliver a stream of output values based on their input. Filter processes fall into two main categories: those that apply a computation to each input value to produce a new output value (like the "map" function in functional programming), and those that examine each input value to decide whether to output it unchanged or to discard it (like the "filter" function).

Filter processes may keep some internal state—for example, integrator and differentiator processes are useful in control engineering [119], and compression filters (such as video codecs) require an internal model of the data stream.

Examples of Use

⋆ **LOVE**, which deals with streams of either MIDI events or audio data, provides a number of stateless filter processes that correspond to standard audio operations, such as amplification and transposition. It also provides a stateful filter that applies an echo effect to an audio stream. Such filters are common in audio processing environments such as Pure Data [175].

In graphical applications that use the ▷ **Ring** pattern to connect rendering processes to a display (such as ⋆ **Occoids**), it is common for the first process in the ring after the display to be a filter that clears the display images as they pass.

The ⋆ **RMoX** network stack uses both types of filter process to handle checksum fields in IP packets. A filter-style filter process rejects incoming packets with incorrect checksum fields; a map-style filter process computes the checksum field on outgoing packets.

```
PROC blocking.buffer <ITEM> (CHAN ITEM in?, out!)
  WHILE TRUE
    ITEM value:
    SEQ
      in ? value
      out ! value
:
```

Figure 44: Blocking buffer process

⋆ **Plumbing** provides a stateful filter process that performs edge detection: it only sends a signal from its output when it receives an input value that differs from the last one it received.

4.1.4 Buffer

PROBLEM: Processes sending data and receiving data need to be decoupled in time.

Buffer processes maintain an internal store of values. In a simple *FIFO buffer*, the buffer process has an internal list of values. Input values are added to the tail of the list, and output values made available immediately from the head of the list. Values are therefore output from a FIFO buffer in exactly the order that they were received, and any input value is output exactly once; no additional output values are fabricated by the buffer.

In practice, most buffers place a limit on the size of the internal list; such a buffer is called an *N-place buffer* (rather than the unlimited *infinite buffer*). Once the buffer is full—for example, N data items have been fed into the buffer with none being removed—the buffer has several choices for what to do when another value is inserted:

- refuse to accept the new value, by blocking the sending process until space is available (a *blocking buffer*);
- discard the new value (a *discarding buffer*);
- discard the oldest value in the list, making space for the new value to be accepted (an *overflowing buffer*);
- discard the newest value in the list, making space for the new value to be accepted (an *overwriting buffer*).

(More complex behaviours are possible, but rare—for example, a smart buffer may examine the values in its list to find one to discard, or may summarise some of the values it holds to reduce the space they take up.)

A one-place blocking buffer is very easy to implement in occam (figure 44). As a result, many ▷ **Filter**-like processes actually act as one-place buffers; a process-oriented program can accrue a considerable degree of buffering by accident.

```
PROC overwriting.buffer <ITEM> (CHAN ITEM in?, out!)
  INITIAL BOOL full IS FALSE:
  ITEM value:
  WHILE TRUE
    ALT
      in ? value
        full := TRUE
      full & out ! value
        full := FALSE
:
```

Figure 45: Overwriting buffer process

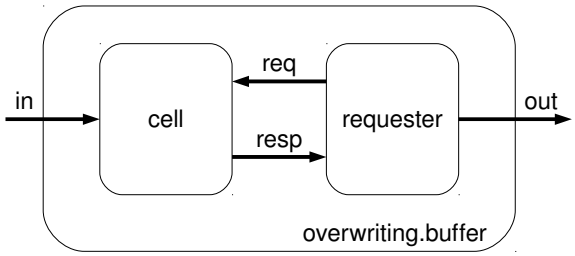


Figure 46: Two-process overwriting buffer

```
PROC overwriting.buffer <ITEM> (CHAN ITEM in?, out!)
  CHAN SIGNAL req:
  CHAN ITEM resp:
  PAR
    INITIAL BOOL full IS FALSE:
    ITEM value:
    WHILE TRUE
      ALT
        in ? value
          full := TRUE
        full & req ? SIGNAL
          SEQ
            resp ! value
            full := FALSE

    WHILE TRUE
      ITEM value:
      SEQ
        req ! SIGNAL
        resp ? value
        out ! value
:
```

Figure 47: Overwriting buffer without choice over outputs

```
PROC holding.buffer <ITEM> (CHAN ITEM in?, out!)
  ITEM value:
  SEQ
    in ? value
    WHILE TRUE
      ITEM new.value:
      SEQ
        in ? new.value
        out ! value
        value := new.value
:
```

Figure 48: One-holding buffer process

A one-place overwriting buffer requires the use of choice (figure 45). Since occam-π does not in practice permit choice over an output guard, this can instead be implemented using a pair of processes, one of which requests values from the other (figures 46 and 47).

So far, we have only seen buffers that block the process reading from them when the internal buffer is empty. Instead, a buffer process may choose to repeat a previous value when it is able to output but its internal buffer is empty. This would be an appropriate choice for a buffer holding data read from an unreliable sensor, for example; it is often better to repeat the previous sample than to create a gap in the output stream.

In addition, these buffers make an output value available immediately when an input value is received. An alternative is to require a buffer process to be holding a certain number of items in its internal buffer before the oldest is made available to output. This type of *holding buffer* is useful when the passage of values through the buffer is being used for synchronisation, and a deliberate delay must be introduced in the stream of values. A one-holding buffer needs to ensure that it has read a new value before the old one is output (figure 48).

Another common option is a buffer process that limits the rate at which values are made available for output, either using real time, or a ▷ **Clock**—a *rate-limiting buffer*.

Some kinds of buffers may reorder the values they contain—for example, if they hold messages with priorities attached, they may use a priority queue as the internal store in order to have high-priority messages be made available at the output more quickly than low-priority messages.

Examples of Use

Buffer processes are widely used to simulate buffered channels (section 2.2.2) in environments that only provide synchronous channels.

Many graphical programs, including ⋆ **Occade** and ⋆ **Occoids**, use rate-limiting buffer processes to control the rate at which the display is updated. Graphical program such as ⋆ **LOVE** that provide GUI widgets to adjust parameters often decouple each widget from the process it is controlling using an overwriting buffer, so that only the latest value provided by the user is delivered.

Sort pumps make use of specialised buffer processes; see ▷ **Pipeline**.

```
PROC glue <ITEM> (CHAN ITEM in?, out!)
  WHILE TRUE
    ITEM value:
    in ?? value
      out ! value
:
```

Figure 49: Glue process

4.1.5 Glue

PROBLEM: Two channels must be spliced together.

A glue process has the interface of a buffer, but actually provides no buffering at all—it simply connects two channels together, providing the usual synchronisation semantics transparently. The usual approach to implementing a glue process is to use extended input to ensure that the process on the input side is blocked until the process on the output side has taken the value, by doing the output as the extended action (figure 49).

Note that this only works in one direction, though; in an environment where choice over output guards is permitted, the process is no longer transparent, since the output can complete before the input side is unblocked; this can be solved using conjunctive choice in environments that provide it [52]. The same technique and problem applies to making invisible ▷ **Delta** and ▷ **Terminal** processes.

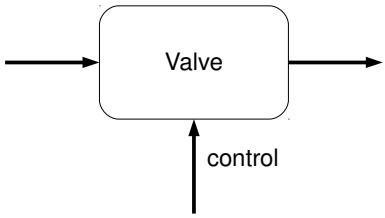
Examples of Use

The static implementations of ⋆ **LOVE** provide glue processes that can be used to replace removed components.

A ▷ **Terminal** process is a glue process that joins a mobile channel end to a static channel, with facilities to switch channel ends at runtime.

4.1.6 Valve

PROBLEM: A stream of values needs to be manipulated under the control of another process.



Like a ▷ **Buffer** or ▷ **Filter**, a valve process has an input channel and an output channel, between which it normally passes values—but it also has one or more control inputs that allow other processes to control its behaviour during operation. A valve process may simply be a buffer or filter with some parameter that may be adjusted at runtime—for example, a rate-limiting buffer where the rate is adjustable, or a buffer with an input that allows the buffer to be emptied—but there are other possibilities.

The most common type of valve is the *pause process*, which usually passes values unchanged from its input to its output, until it receives a message on its control input; it then waits to receive a second control message before it resumes its usual behaviour. This is usually implemented using the ▷ **Acknowledgement** double-communication

```
PROC pause <ITEM> (CHAN ITEM in?, out!, CHAN SIGNAL control?)
  WHILE TRUE
    PRI ALT
      control ? SIGNAL
        control ? SIGNAL

      ITEM value:
      in ? value
        out ! value
:
```

Figure 50: Pause process

```
PROC reset <ITEM> (CHAN ITEM in?, out!, replace?)
  WHILE TRUE
    ITEM value, dummy:
    PRI ALT
      replace ? value
        SEQ
          in ? dummy
          out ! value

      in ? value
        out ! value
:
```

Figure 51: Reset process

pattern in occam (figure 50). This kind of pause process blocks its input channel while paused; an alternative would be to discard messages from its input channel while paused.

The *reset process* is another type of valve. When a reset process receives a value on its control input, it uses it to *replace* the next value read from its regular input (figure 51).

Reset processes are especially useful as part of a ▷ **Ring**, where the ring is being used to share a value among a group of processes that usually just read the value and pass it on. A reset process in the ring can be used to replace the value with a new one.

Examples of Use

The digital circuit simulator built in occam at Kent in the late 1980s modelled components as processes and wires as channels, using the ▷ **I/O-PAR** pattern [252]. It provided "probe-point" valve processes that allowed the user to inspect the logic levels on particular wires by attaching a virtual logic probe to them.

⋆ **LOVE** provides a number of controllable filters that follow this pattern: for example, it has a volume control process that multiplies each of a stream of audio samples by a scaling factor, with the control input allowing the scaling factor to be adjusted.

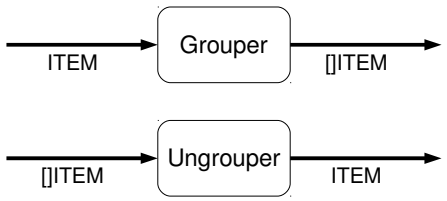
⋆ **Flightsim** uses a ring to distribute the locations of objects in the world among the players; for each object, there is a corresponding data value circulating in the ring. Each player's connection to the ring includes a ▷ **Delta** to deliver a copy of the object

information to the player's graphics system, and a smart reset process to replace the data value corresponding to the player's aircraft when it next passes.

Subsumption architectures for control of autonomic systems—where the system's behaviour is described using a network of possible behaviours connected by elements that allow behaviours to promote and override actions—can be implemented in a natural way using process-oriented techniques [216]. Suppressor and inhibitor elements in a subsumptive control system are represented as valve processes that allow messages from one behaviour to override those from another.

4.1.7 Grouper and Ungrouper

PROBLEM: A stream of aggregate values needs to be split into individual values; or vice versa.



Groupers and ungroupers are specialised kinds of ▷ **Filter**.

Grouper processes collect several input items to produce a single larger output item; ungrouper processes break a single input item into several smaller output items. For each input item, a grouper will therefore produce a sequence of outputs; an ungrouper will accept a number of inputs before producing an output.

Groupers and ungroupers can be used to enable greater parallelism when processing streams of arrays: a ▷ **Farm** or ▷ **Fan-Out** can be surrounded by an ungrouper-grouper pair to allow each element in an array to be processed individually.

Examples of Use

Process-oriented programs that need to work with text files—for example, to read configuration information or input data—generally use grouper and ungrouper processes to produce streams of tokens or lines at an appropriate level. Typically, a ▷ **Producer** process makes the bytes of a file available over a channel; grouper processes can then be used to:

- convert encoded bytes into Unicode characters;
- tokenise a character stream into tokens;
- collect characters into words or lines; and
- join continuation lines to produce flattened lines.

Upon output, an ungrouper process can be used to break lines into Unicode characters, and another to encode characters into an external multibyte representation. Several of these processes are provided in ⋆ **KRoC** modules; others are generally constructed to suit the application's requirements.

Aspects of network protocols can often be specified in terms of grouping and ungrouping. The ⋆ **RMoX** network stack implements the SLIP protocol as a grouper-ungrouper pair [198]: incoming bytes from the serial device (terminated by a marker value defined by the SLIP protocol) are grouped into packets, and outgoing packets are

```
PROC merge <ITEM> ([]CHAN ITEM ins?, CHAN ITEM out!)
  VAL INT n IS SIZE ins:
  WHILE TRUE
    SEQ i = 0 FOR n   -- fair choice idiom
      PRI ALT j = 0 FOR n
        ITEM value:
        ins[(i + j) \ n] ? value
          out ! value
:
```

Figure 52: Merge process using fair choice

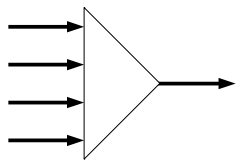
```
PROC merge <ITEM> ([]CHAN ITEM ins?, CHAN ITEM out!)
  SHARED! CHAN ITEM shared:
  PAR
    PAR i = 0 FOR SIZE ins
      WHILE TRUE
        ITEM value;
        SEQ
          ins[i] ? value
          CLAIM shared!
            shared ! value

    blocking.buffer (shared?, out!)
:
```

Figure 53: Merge process using a shared channel

ungrouped into bytes. Similarly, RMoX's implementation of the telnet protocol uses a grouper-ungrouper pair to convert between the streams of bytes used by RMoX's terminal interface, and the streams of arrays of bytes expected by its TCP socket interface.

4.1.8 Merge

PROBLEM: Values from several streams must be interleaved into a single stream.



A merge process takes values from several input channels and writes them all to a single output channel. Each input will result in a single output.

The most common form of merge process simply makes a fair choice across its input channels and writes any value it receives to its output (figure 52). The effect of this is to simulate a channel with a shared input end. Indeed, a merge process can be implemented using only buffer processes and a shared channel (figure 53).

Simple merge processes like the example above are relatively rare in environments such as occam-π that support shared channel ends, because using shared channels makes it easier to wire up the process network. Even where an existing process interface provides messages over unshared channel ends, ▷ **Glue** processes can be used to connect them to a shared channel end. This is therefore a partial antipattern: before

writing a merge process, you should consider whether a shared channel can be used instead.

However, it is sometimes useful to know from which input a particular value came; in these cases, a merge process that tags values with the identifier of the channel from which they were received can be used.

Examples of Use

Many occam programs used merge processes where shared channel ends are now used in occam-π: for example, the INMOS flight simulator can be controlled using either keyboard or joystick commands, and this is implemented by having keyboard and joystick events fed from two different processes into a merge process, to produce a single stream of input events.

The ⋆ **RMoX** network stack uses a merge process to handle packets coming from several network interfaces in a uniform way; packets are tagged with the name of the source interface. (Since network interfaces may be created and destroyed at runtime, this merge process is also a ▷ **Terminal**, allowing new interfaces to be connected.)

4.1.9 Collector

PROBLEM: Sets of values that arrive upon several streams need to be collected together.

Like a ▷ **Merge** process, a collector process has several input channels and one output channel—but rather than passing individual values through to the output, a collector process collects one value from each of the input channels, and computes a single value to output from the complete list of values.

A collector process may simply output the list of input values as the output value; such a collector would properly be called a *multiplexer*, by analogy with the equivalent component used in telecommunications systems [252]. (However, this term is also sometimes used for ▷ **Merge** processes.) A more common use is to summarise the list in some way—for example, to compute the sum of the input values.

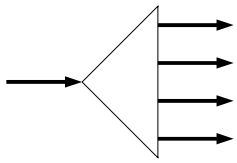
Examples of Use

Collector processes are often used to implement ▷ **Fan-Out**; for example, in ⋆ **Occoids**.

In audio processing frameworks such as ⋆ **LOVE**, mixer processes are collectors: they read an audio sample from each of their input channels, then perform a weighted sum operation to compute an output sample. In addition, LOVE's mixer component handles unconnected input channels by generating a synthetic stream of silent samples, allowing new components to be connected to unused channels while the network is in operation.

4.1.10 Delta

PROBLEM: A stream of values must be distributed to several processes, with each value going to all processes.



A delta process has one input channel and several output channels. Each value input will result in one value being written to each of the output channels.

The most common form of delta process simply sends its input value to all of its output channels [252, 120]. Where the data type in question is a single-owner type, this will require the data value to be copied. As a result, delta processes may be a source of inefficiency in systems that make use of the communication semantics of single-owner types. (See ▷ **Fan-Out** for how some uses of delta processes may be refactored into pipelines to avoid this problem; a ▷ **Ring** could also be used to distribute data among processes.)

Just as a ▷ **Merge** process emulates a shared channel using unshared channels, a delta process emulates a broadcast channel using regular channels. Delta processes are therefore widely used to distribute information in environments where broadcast channels are not available.

Delta processes can be introduced into a process network to aid debugging, by providing a second copy of the information being sent along a particular channel. In this case, the ▷ **Glue** process technique is useful to avoid introducing an extra place of buffering inside the delta process.

Two strategies are possible for performing outputs in delta processes. *Sequential delta* processes write to their output channels one at a time, in order; *parallel delta* processes perform all of their outputs in parallel. While a sequential delta is simpler to implement, it can be blocked if one of the processes that it is sending to is not willing to receive; parallel deltas are usually preferred since they will deliver messages to receiving processes as soon as they become available. Parallel deltas are often slightly less efficient though; a parallel delta must either perform a choice across a group of output guards, or fork and wait for a number of child processes.

A *demultiplexer* is a delta process that takes an array of values as input, and sends a different element from the array to each of its output channels. A delta process with no output channels is equivalent to a ▷ **Black Hole**.

Examples of Use

A delta process is often used to implement ▷ **Fan-Out**; for example, in ⋆ **Occoids**. The distributed version of Occoids also makes use of delta processes to distribute parameter change messages to all the hosts in the simulation.

In ⋆ **LOVE**, delta processes are often used as analogues for distribution amplifiers that send the same audio or MIDI event stream to multiple destinations: for example, to process the same oscillator through several different delays to produce a reverberation effect, or to turn a monophonic audio stream into a stereophonic one.

4.1.11 Distributor

PROBLEM: A stream of values must be distributed among several processes, with each value going to one process.

A distributor process has a single input and several outputs, like a ▷ **Delta**, but it follows a different pattern of communication: each input value is delivered to only one of the output channels. Distributor processes are used to route messages to an appropriate receiver.

Load-balancing is a common use of distributor processes. A distributor may simply deliver values to each output in turn (like a car engine's distributor), or select an output at random. More usefully, a distributor could perform a choice across all the outputs to select one that is ready to receive. This technique is commonly used to distribute work to worker processes in a ▷ **Farm**. Just as a ▷ **Merge** process can be replaced with a channel with a shared input end, this kind of distributor can be replaced with a channel with a shared output end.

Alternatively, the output may be chosen based on a function applied to the value; such a distributor is called a *router*. This is useful when messages must be distributed to the appropriate one of a pool of specialised worker processes.

Examples of Use

The ⋆ **RMoX** network stack uses several levels of distributor processes to route incoming network packets to the correct destination: first, packets are distributed based on their destination address, then their protocol, and finally their port number (so a packet to a different machine may be routed out another network interface, whereas a packet to a local TCP port would be routed to the process that manages that port).

Elsewhere in RMoX, router processes are used to pass mobile channel ends to the appropriate subsystem when devices are opened; see ▷ **Hand-Off**.

A particular kind of load-balancing process is provided in ⋆ **LOVE** for use when building synthesisers: one that distributes MIDI notes to one of the several oscillators in a polyphonic synthesiser (a "voice allocation" process) [176]. The problem comes when more notes are being played than there are oscillators available—the voice allocator must make a decision that will make the least difference to the overall sound of the synthesiser (generally with no knowledge about how long the notes will last). A standard approach is "voice stealing": replace the note that has been playing the longest.

4.1.12 Factory

PROBLEM: Processes need to be started in a common context.

A process that wishes to start another process may not necessarily have access to all of the context that the new process needs. For example, it is common to provide a process with connections to the system services that it will need on startup—but a process that wants to open a new graphical window may not itself have the direct access to the display hardware that the window process requires. Similarly, a new process may need to be assigned a unique ID (from an ▷ **Oracle**), or registered in a global registry, or safely enrolled upon a barrier, or started in a particular forking context so that it can be safely terminated later.

We use a factory process to solve this problem. A factory process forks new processes in response to requests. Factory processes are often able to start many different kinds of processes in response to different requests. The factory process can hold any context that the new process might need; another process that wants to be able to start new processes only needs to have a connection to the factory to do so.

Factories are used in object-oriented programming as well—for example, using the ○ **Abstract Factory** pattern [87]. However, OO factories return a reference to the object that has been created; process-oriented factories do not necessarily provide any way of communicating with the new process, only acknowledging that the process has been started.

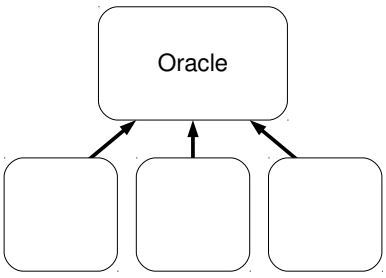
Examples of Use

In CoSMoS simulations, new agents may be spawned into the simulation at any time, either as a result of other agents' actions—such as ants leaving pheromone trails—or as a result of user interaction. ⋆ **Occoids** allows the user to add and control predators that interact with the other agents in the simulation. An agent process needs a considerable amount of context when it starts up: it must be enrolled upon the simulation ▷ **Clock**, registered with the appropriate location process, and so on. This context is provided by a factory process that spawns new agent processes based on an agent information structure. The same mechanism is used for migrating agents between hosts using the standard approach for simulating mobile processes: an agent being migrated packs up its internal state into an agent information structure and then exits, and the structure is forwarded to the agent factory on the destination host.

⋆ **RMoX** makes similar use of factory processes to spawn appropriate driver processes when new hardware is detected: the `driver.core` process implements both the factory and ▷ **Oracle** patterns, in that it keeps a list of drivers that are running, and is able to start a requested driver if it is not already running.

4.1.13 Oracle

PROBLEM: Several processes need to make decisions based on shared information; or a single resource must be shared among several processes.



An oracle process provides a single location that collects information on behalf of a group of other processes. An oracle therefore mediates access to a shared resource: it can ensure that the shared information is updated and queried atomically, and can automatically expire out-of-date information.

Oracle processes are usually passive servers (▷ **Client-Server**); processes must push information to them to keep them up-to-date. The interface provided by an oracle will depend on the information it stores and the queries that it must be able to perform upon it; oracles do not necessarily report all the information they hold for each query.

Caution is necessary in the application of this pattern: an oracle process may be a ▷ **Bottleneck**.

Examples of Use

Choice over barrier guards can be simulated in environments that do not support it directly by the use of an oracle process that has complete information about all barriers in the system [245]. This is certainly a bottleneck since all synchronisations must be serialised through the oracle, but the implementation is straightforward and easy to formally verify [6], so it may be an appropriate implementation choice in embedded systems where parallel performance is not a concern.

⋆ **Occade** implements collision detection between sprites using an oracle process that knows the locations of all of the sprites on the display. When a sprite's position changes, it reports this to the oracle; the oracle will then immediately return a list of any other sprites with which it is now colliding (with the exception of those for which collision detection is disabled).

The simulation framework used in ⋆ **Occoids** and other CoSMoS simulations divides space into regions, each of which is managed by a ▷ **Location** oracle process that knows about all the agents in that region of space [10]. This distribution of knowledge reduces the bottleneck effect, although it does mean that agents may need to consult multiple location processes to find out about all the other agents within the region of space they can see. Occoids also uses an oracle process to map location IDs to the corresponding mobile channel ends, allowing rapid access to remote locations.

Oracle processes are often used as caches: for example, the cache of loaded sprite images in ⋆ **Occade**, and the X server's pixmap cache in ⋆ **occam-X11**.

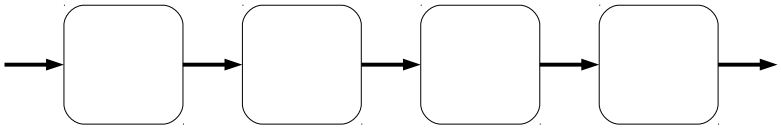
Oracle processes are used for real-time synchronisation in both ⋆ **occvid** (for multiple media streams) and ⋆ **occam-X11** (for multiple sources of input events). In both cases, two or more streams must carry consistent timestamps; this is achieved by filtering each stream through a process that explicitly synchronises with another process that keeps track of time.

An oracle process may also be used to serialise access to a shared external resource. For example, the OpenGL graphics library requires that it is called from only one system thread; in a process-oriented system where activity happens in many threads in an unpredictable manner, this can be ensured by having a process that wraps the OpenGL interface and provides a process-oriented interface to it.

4.2 Patterns of Structure

Patterns in this section describe features of the structure of the process network within a program.

4.2.1 Pipeline



PROBLEM: Each value in a stream needs to be processed by several ▷ Filter s in turn.

A pipeline is a chain of processes that follow the ▷ **Filter** pattern, with the output of each process connected to the input of the next [120, 155]. A pipeline is thus a ▷ **Filter** itself; values sent through the pipeline are processed by each of the processes in turn.

Because more than one value may be passing along a pipeline at a time, pipelines can be used to enable parallel processing of a stream of values, provided sufficient values are available to keep all the filter processes busy.

Pipelines are widely used, not just in process-oriented programming but also in other approaches to software and hardware design, especially in "dataflow" applications [39]. The Unix shell has always supported pipelines—making this a pattern of concurrent programming that has been around since the late 1960s. (Unix pipelines are simple streams of bytes with fixed-size buffers; more recent systems designed along the same lines, such as Microsoft's PowerShell, support more expressive protocols.) The ○ **Chain of Responsibility** pattern describes an object-oriented analogue of a pipeline.

Each process in a pipeline will have a number of places of internal buffering—typically one place for a simple filter (see ▷ **Buffer**). The pipeline as a whole will have as many places of buffering as the sum of the number of places of buffering in each of its components. Pipelines can be used to construct larger buffers from smaller buffers; the simplest way to build an N-place blocking buffer is to construct a pipeline of N one-place blocking buffers.

The compositional property of process-oriented design allows the filter processes in a pipeline to be implemented in any way, including as networks of subprocesses, provided they present a filter-style interface. Processes may have additional input and output channels; for example, they may be ▷ **Valve** processes, allowing control of the values flowing through the pipeline, or ▷ **Delta** processes, allowing the pipeline to split. It is relatively common for a pipeline to include instances of the ▷ **Fan-Out** pattern, allowing internal parallelism for individual values; a generalised pipeline that contains fan-out is sometimes called a *spaceline* [132].

In a high-throughput pipeline, the overhead of communication between the processes may be significant. The use of buffered channels or buffer processes between the processes in a pipeline can significantly improve the performance of the pipeline, by allowing each process to process several input items before its output buffer fills and context switching is necessary.

A *expanding pipeline* may be constructed recursively, starting with a single process. To expand the pipeline, the last process executes as a parallel composition of itself and a new process, connected by a channel. The standard example of this is a pipeline that generates prime numbers by starting a new filter process to remove multiples of each prime as it is found (figure 54).

Pipelines are an example of a type of regular process network that can be built automatically using higher-order process constructors [48].

SOLUTION: Connect the processes together into a pipeline.

Examples of Use

Expressing a problem as a composition of filters is a common technique in most approaches to programming; a problem solved by composing functions in a functional language can be translated straightforwardly into a pipeline of processes in a process-oriented environment, introducing opportunities for parallel execution.

```
PROC generate (CHAN INT out!)
  INITIAL INT i IS 2:
  WHILE TRUE
    SEQ
      out ! i
      i := i + 1
:

REC PROC filter (CHAN INT in?, out!)
  INT prime:
  SEQ
    -- First number received must be prime
    in ? prime
    out ! prime

    CHAN INT thru:
    PAR
      -- Remove multiples of it
      WHILE TRUE
        INT n:
        SEQ
          in ? n
          IF
            (n \ prime) = 0
              SKIP
            TRUE
              thru ! n

      -- and recurse
      filter (thru?, out!)
:

PROC primes (CHAN INT out!)
  CHAN INT all:
  PAR
    generate (all!)
    filter (all?, out!)
:
```

Figure 54: Expanding pipeline that generates prime numbers

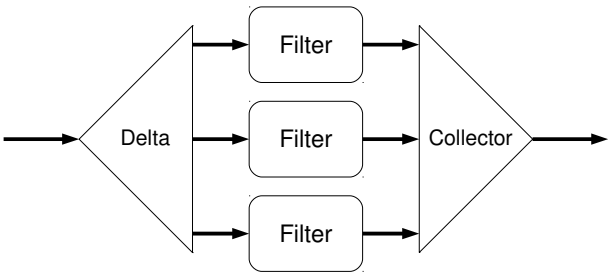
Parsers are often constructed in this way (see also ▷ **Grouper**). The command-line option parser in ⋆ **KRoC**'s `file` module uses a pipeline internally (for simplicity rather than parallelism, since performance is not a concern): a ▷ **Producer** process emits each raw command-line argument in turn, then a ▷ **Grouper** groups options with their arguments according to the option definitions, and a final process translates options into the correct form for the application program to handle.

INMOS's original occam 1 compiler was itself an occam 1 program, implemented as a pipeline that performed tokenisation, parsing, name resolution, code generation and output as separate processes. (Today's equivalent is the nanopass approach used by Tock [204].)

The rendering of 3D graphics is traditionally expressed as a pipeline—which is implemented in hardware by all modern graphics cards, and in software by ⋆ **Flightsim** (figure 16).

The sort pump is an especially interesting example of a pipeline: a parallel sorting algorithm that sorts streams of n values in $O(n)$ time using a pipeline of n processes [53]. Each process in a sort pump holds on to the highest value it sees, passing others through to the remainder of the pipeline; when it sees the end of the list of values, it releases the value it is holding, meaning that values emerge from the other end of the pipeline in order. Since communication is typically more expensive than comparison, sort pumps are rarely used in practical programs, although sorting *networks* (which operate upon several values in parallel, effectively unrolling a conventional sorting algorithm in time) are widely used for sorting small numbers of items in hardware designs.

4.2.2 Fan-Out



PROBLEM: Each value in a stream needs to be processed in several different ways.

Some problems can be broken down into several parallel activities, all of which require access to parts of the same input data value to produce a result. For example, implementing a reverberation effect in an audio processing system is usually done by combining the outputs of separate simulations of direct transmission of the signal, early reflections from walls and other objects, and damped tail resonance—all of which require the complete input signal.

In a process-oriented system, we can do this by distributing the input values to several different ▷ **Filter** processes using a ▷ **Delta**, and then recombining the results using a ▷ **Collector**, resulting in one output value for each input value. The delta may simply distribute the same data to each filter, or it may extract only the parts required by each filter; in an environment such as CHP where data values are immutable (so

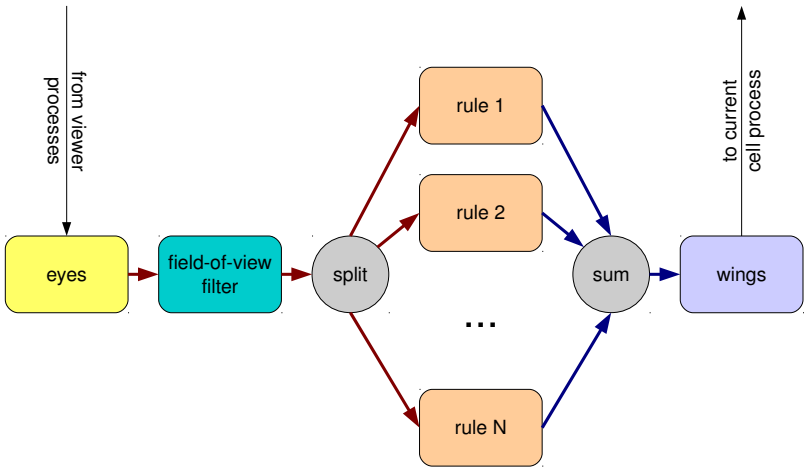


Figure 55: A boid process in Occoids

copying a value is free), the first option may be more efficient. The overall network has a ▷ **Filter** interface.

Fan-out problems can usually also be solved using ▷ **Pipeline**, with different performance characteristics. To transform a fan-out problem into a pipeline, the messages sent along the pipeline should include both the input data and the results so far. If the final result cannot be computed piecemeal, add an extra process to the end of the pipeline to do it. The advantage of pipelining is that the input data does not need to be copied, as it can be transferred from process to process as a single-owner reference—but the disadvantages are that each piece of data must traverse each process in the pipeline in turn, increasing latency, and the pipeline must be kept full in order to obtain parallel execution. Pipeline is more appropriate when the cost of copying data in the delta is significant, latency does not matter, and there is sufficient work to keep the pipeline full.

The MapReduce framework for distributed computation—where problems are divided up into smaller work units, each of which is processed individually, and the results recombined—is essentially an implementation of this pattern [67]. MPI's scatter and gather operations can be used in much the same way [144]—although process-oriented implementations of these operations are often more efficient than their MPI equivalents [40].

This pattern is for problems where the same data has to be processed in several different ways; where different pieces of data need to be processed in the same way, use ▷ **Farm**.

SOLUTION: Use a ▷ Delta to split the stream into several parallel streams, process all in parallel, and combine them using a ▷ Collector.

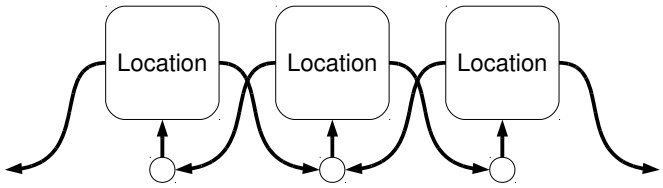
Examples of Use

The agents in ⋆ **Occoids** implement the boids rules for flocking: agents look for other agents around them, and apply rules to the list of other agents to compute an acceleration vector. In the original version of Occoids, the rules were implemented in parallel using fan-out (figure 55).

Since the list of agents is a mobile array, this meant agents were doing a significant number of unnecessary memory allocations and copies each time the rules were applied. This was refactored as described above into a pipeline, passing the list of visible agents and the acceleration vector (initially zero) along the pipeline, and having each rule add its own contribution to the vector.

In ⋆ **Flightsim**, polygons are rendered by breaking them down into lines, then distributing the set of lines to four processes to render. Each rendering process clips the lines to the area of the screen that it is responsible for, and delivers complete rendered screen images to another process that writes them into screen RAM.

4.2.3 Location



PROBLEM: Processes need to communicate based on their proximity in a modelled space.

In a simulation or game, processes may be used to model regions or locations in space: either a physical space, or some more abstract idea of space such as a state space. Processes in such a system may need to communicate with other processes that are near to them in space—for example, when agents within the space, which are connected to their current locations, move around or observe their neighbourhood. We can achieve this by using channels to connect each process to its immediate neighbours: using processes to model space, and channels to model proximity.

Modelling a space in this way has several advantages. The simulation of space can be parallelised: different regions of space can perform computations in parallel, only synchronising where necessary. In addition, there is no need for agents within the space to have an absolute sense of position (such as a global coordinate scheme); they only need to have a *relative* idea of their position within the current location. This allows the modelling of spaces where the locations cannot easily be numbered—for example, Penrose tiles, or arbitrary graphs—or with unusual connectivity, such as Cartesian space with occasional wormholes.

When an agent moves across a boundary between locations, it "follows" the appropriate channel to its new location, updating its relative position to match. If an agent moves across more than one location within a single step, it may need to follow several links before it finds its new location. (A location therefore needs to be able to select a link based on a relative position—which in Cartesian space is straightforward.)

One approach to modelling proximity is to represent each neighbour relationship as a point-to-point channel (figure 22). This requires a very large number of channels—especially if diagonal connectivity needs to be represented, or more than two dimensions are required—and is complicated to "wire up" correctly. A better solution is to give each location a shared input channel, to which all its neighbours have access; this way, only one channel per location is required, and arbitrary neighbour relationships are easy to construct.

SOLUTION: Give each location process a shared input channel, and references to the input channels of its neighbours.

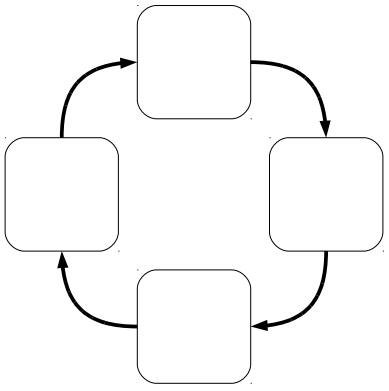
Examples of Use

The TUNA ⋆ **Life** simulation experimented with this pattern for simulating cellular automata efficiently: since Life cells only need to know the state of their neighbours, the shared input channels are just used to receive state change messages on each timestep. This work led to the ⋆ **CoSMoS** generalised space model, which uses locations to represent spaces described by arbitrary functions, with explicit migration of agents between hosts in a distributed simulation of space, and to later simulations which take a similar approach to simulating a graph-based space (such as a system of blood vessels) as a set of connected location processes.

The multiplayer adventure game used to teach the use of mobile channel ends in the Kent concurrency modules uses this pattern to represent rooms [28]: each room in the game has a shared input channel, with neighbouring rooms having references to the channel labelled with movement directions. This pattern is especially useful for adventure games because it is possible to only allow movement in one direction—for example, a tall cliff that the player can fall off but not climb up.

4.2.4 Ring

PROBLEM: Information must be distributed among several processes; or several processes must take turns to access a shared resource.



A ring is a loop of ▷ **Filter** processes: a ▷ **Pipeline** with its ends connected together. Rings are used to distribute values among the processes in the ring, with each process getting a chance to examine or discard each value in turn. A value introduced into the ring by one of the processes will circulate around the ring until a process discards it. Rings have a number of uses.

In classical occam—as in older physical network topologies—rings were often used for routing data. Packets labelled with addresses can be used to route data between any of the processes in the ring: when a router sees a packet with its address on, it removes it from the ring.

A ring can also be used for broadcasting data to all of the processes in the ring, by having them examine each message and then pass it on. To prevent messages circulating forever, the process that introduced each message to the ring is responsible for

recognising it when it returns and removing or replacing it.

Finally, a ring can be used to share a resource fairly among multiple processes, with each process getting a turn to use the resource—represented by a mobile reference, or an abstract token—before passing it on. This approach also works for a fixed-size pool of resources, all of which can be allowed to circulate in the ring; when a process needs to use a resource, it waits to receive one and then removes it from the ring for as long as it is needed, reintroducing it when it is no longer required.

Many applications that would have used rings in older process-oriented systems can now be implemented using shared channels, or shared interfaces to server processes. These newer approaches are often more efficient: claiming a shared channel to send a message directly to a server is usually cheaper than doing several communications to send the message round a ring. However, rings used for token passing are still useful, especially since mobile references can now be passed around a ring. A ring can be used to share a writable resource efficiently among several processes, ensuring that they access it in a predictable order—which is very useful for graphics rendering. In addition, the rate at which items pass around a ring can be easily controlled by introducing a rate-limiting ▷ **Buffer** process.

To a programmer used to working inside the constraints of the ▷ **Client-Server** design rules, the use of a ring will always be accompanied by a vague sense of unease—it looks like a cycle! However, the rule for constructing a safe ring is relatively simple: for a ring around which N items will circulate, ensure that the ring contains *at least* N places of buffering [189]. This can be achieved, if necessary, by adding buffer processes to the ring. The easiest way to ensure that this invariant holds is to make each process responsible for providing enough buffering for all the items that it may add to the ring. If the number of items will vary, the programmer can specify enough buffering for the maximum number of items that will ever be present—or dynamically introduce more buffering as more items are added to the ring.

It is important not to block the flow of data around a ring: if a value needs to be processed, it is usually better to remove it from the ring and reintroduce it later, passing on other values in the meantime, than to block the flow of other values during processing. To support this, programs using rings around which multiple values pass often make use of *ring router processes*, which handle interaction with the ring, removing interesting values and presenting them for processing to a different process.

To shut down a ring safely (in an environment without poison), a shutdown message must be circulated around all the processes in the ring. When a process decides that it wishes to shut down the ring, it should send a shutdown message (ensuring that there is enough buffering to do so—for example, by discarding an existing message from the ring), then wait until it receives one back, discarding any other messages received in the meantime; it can then exit. When a process receives a shutdown message that it did not send, it must pass it on the next process and then immediately exit. If more than one process may initiate shutdown, this approach will still allow the entire ring to shut down safely—but a process initiating shutdown may get a shutdown message back before all the processes in the ring have exited.

SOLUTION: Connect the processes into a ring, and circulate data between them.

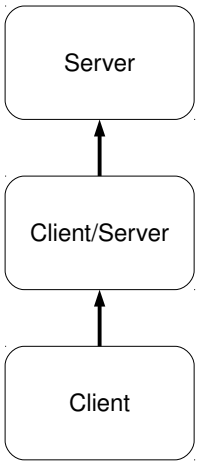
Examples of Use

⋆ **Flightsim** is a good example of the classical-occam use of rings, with each player having a ring router process. The messages circulating in the ring represent the objects in the flight simulator; they are updated as they pass each player's router. Since Flightsim was distributed over several Transputer chips, the ring was a physical one between several different processors.

By far the most common use of rings in modern process-oriented systems is for graphics, with a mobile reference to a graphics display being circulated between several processes that draw on it in turn. This use is supported by ⋆ **KRoC**'s `raster` module; applications that use this strategy include all the ⋆ **TUNA** and ⋆ **CoSMoS** visualisations. The `raster` module's protocols include support for shutting down the ring, and provide a procedure to perform the shutdown process above in the correct way. Standard processes are provided that can be used in the ring—for example, to clear the display to a solid colour before other processes draw over it, and to limit the rate at which the display is updated.

4.2.5 Client-Server

PROBLEM: A network of processes communicating using request-response protocols needs to be arranged so as to avoid deadlock and livelock.



The idea of a server process that performs actions in response to requests from other processes is very common in concurrent software design. An exchange between a client and server, which may include messages in both directions, is called a *conversation*; the two-way channel (or channel bundle, etc.) over which a client and server converse is called an *interface*. A server process may itself act as a client—for example, when fulfilling a request requires making a request to another server—but connecting up a complex network of clients and servers in an uncontrolled way can result in undesirable behaviours. The client-server design rules describe how client and server processes can be implemented and connected together in such a way as to guarantee freedom from deadlock and livelock. This pattern therefore describes both the structure of client-server process networks, and the behaviour of the processes within them; the safety of the resulting system relies on both sets of rules being followed.

The client-server design rules, as originally stated, as are follows [257]:

- A conversation must be initiated by a client process. (This rule was written for output-guard-less occam 2, where having the first message in a conversation sent by a server process would result in the server being committed to that particular client; in a more modern system a client may safely "initiate" a conversation by taking an output from the server, provided the server is not blocked if the output is *not* taken.)

- A conversation can include several communications in either direction, provided both client and server agree. (The directions can be specified as a two-way protocol.)

- Each client process may communicate with only one server at a time. (A client-server component may be implemented as several processes internally to allow it to converse with several servers concurrently, one per client process.)
- A server may serve multiple clients, but it may only serve one client at a time. (Servers may also provide different interfaces to different clients.)
- A server process must accept a request from any client it serves within a finite time—one client cannot block another.
- A server process may act as a client to other servers during a conversation.
- Cycles within the directed graph of client-server relationships in a process network are forbidden.

These rules have been shown to generate systems that are free of deadlock and livelock—both through informal reasoning [257] and using CSP proof techniques [131]. (The no-cycles rule in particular can be intuitively understood once a cycle has been encountered; designers of distributed systems use the same rule.) The client-server pattern is widely used in process-oriented design, to the point where programming languages have been designed explicitly to support it—for example, Honeysuckle, which provides client-server interfaces as the primary mechanism for process interaction, and statically enforces the client-server design rules [73].

As a simple protocol (one that carries single data values) can be regarded as a client-server protocol, many process networks can be reasoned about using client-server rules. In a ▷ **Pipeline**, for example, each process is a server to the preceding process and a client to the process that follows it. The parallelism in a pipeline comes from the client-server pattern allowing a server process to continue to perform work on behalf of a client after the conversation with it has finished, provided that it does eventually return to service another request—something that would not be as straightforward if a "pipeline" were built from OO objects. Similarly, a server can perform other work between processing requests—for example, a cache server may periodically perform cache expiry when it is not otherwise busy.

Since every conversation is guaranteed to complete in a finite time, it is always safe (that is, it will not cause deadlock) to make a request to a server from a process that does not itself act as a server. This allows client-server systems to be "driven" by other kinds of system at the lowest level; a program may use ▷ **Phases** to regulate the overall behaviour of a client-server system, or be structured as a ▷ **Ring** that processes individual items using client-server requests.

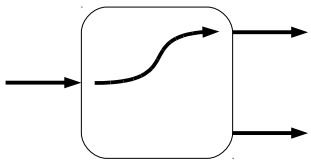
Cycles can arise in client-server systems as a result of feedback loops in control systems, or (more commonly) in callbacks—where a request does not generate a response immediately, but at some later time, requiring the server to make a client request back to the originator in order to deliver the response. When a cycle cannot be avoided, it is necessary to introduce a new process to break the cycle and maintain the design rules: the ▷ **Intermediary** and ▷ **Messenger** patterns are two possible approaches.

An example of a cycle-breaking intermediary process is a component that acts as both a client and a server, without actually having a client-server relationship between the two interfaces—such as the overwriting buffer process shown in figure 45 [257]. This process works by using internal concurrency to ensure that its two interfaces

never block each other. Note that the communications between the two "halves" of the overwriting buffer do *not* follow the client-server rules, since the input side can refuse requests from the output side when it is empty. Another approach would be to use a process that forked a new messenger process to forward each request it received. Such buffer processes are only really practical for relatively simple client-server protocols (since a protocol that did not proceed in a predictable fashion could not easily be buffered); in practice this is not a serious limitation.

In an environment without poison, shutting down a client-server network can be awkward: a server process needs to know when other processes no longer need to make requests to it. This can be done by providing an explicit "close" request (for example, using the ▷ **Snap-Back** pattern), but this relies on every server maintaining its own reference count. In systems with garbage collection (such as most modern object-oriented languages, and Haskell's STM system), making the server end of the interface a weak reference allows the runtime system to tell when it is no longer used by any clients; future environments could allow servers to be notified by the runtime when they can safely shut down. An alternative is to use some other mechanism—for example, a barrier synchronisation, or a second server interface that only includes a "close" request—to encourage all the clients and servers in the network to shut down simultaneously—but wiring this up is awkward!

When reasoning about the safety of a client-server network, it is important to know about all the client-server relationships within the network. Process abstraction can hinder this, since a subnetwork of client-server processes that exposes client interfaces cannot be treated simply as a single server: the user needs to know about the client-server relationships that exist between the different interfaces in order to tell whether using it would create a cycle. These are called *client-server dependencies*, and can be indicated on a process diagram using arrows inside abstracted client-server processes [75].



The client-server rules were originally designed for occam 2, where client-server relationships were implemented as pairs of channels, and the process network was unable to change at runtime. The introduction of channel end mobility has allowed greater flexibility in the construction of client-server systems.

For the purposes of the no-cycles rule, a client and server only have a relationship if they actually communicate. Simply holding a reference to a server's interface does not cause a cycle if the holding process does not use the interface itself. This allows a server process to hold an interface reference that it cannot use itself, but can provide to clients for them to use; this is useful, for example, when implementing the ▷ **Location** pattern, with locations represented as server processes that hold references to the interfaces of their neighbours. It can be useful to show references that are held but not used on process diagrams; for example, using dotted rather than solid arrows.

The client-server relationships within a network may change dynamically, provided that the client-server rules hold at any point in time. For example, the client-server relationship between a pair of processes may reverse at runtime—provided that they both agree to the reversal at the same time (for example, by sending a special message), and doing so does not introduce a cycle into the process network.

Current process-oriented environments support only limited static checking facilities for client-server programming. Two-way protocols (section 5.3) would allow the

protocols of client-server conversations to be specified and checked; more advanced systems could also check that the client-server design rules are followed.

SOLUTION: Use the client-server design rules.

Examples of Use

The client-server design rules were first explored during the development of ⋆ **occam-X11**, a static client-server system implemented in classical occam. The pattern was already in use before that, however, even within process-oriented programming: for example, the `iserver` interface used by Transputer processes to communicate with a host machine uses a client-server protocol over a pair of channels.

⋆ **Occoids** (like most of the CoSMoS simulations) consists of a client-server core that is driven from the agent processes by a system of ▷ **Phases**, using phase synchronisation to ensure that the information stored in servers is updated in the correct sequence. "Phase adapter" processes are used that enrol upon a phase system and make a request to a server in a given phase, hiding the details of phase interaction behind a client-server interface. The visualisation processes in Occoids are similarly driven by a ▷ **Ring**.

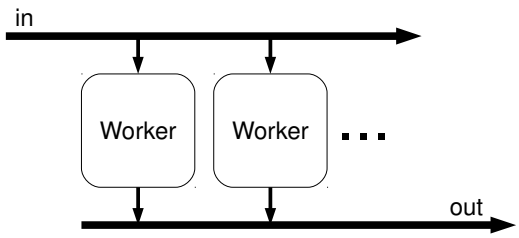
⋆ **KRoC**'s `selector` module provides a client-server interface, with application processes acting as servers to the selector. When a file descriptor becomes ready for I/O, the selector makes a request to the corresponding process; the response from the request indicates whether to wait for the file descriptor again, or to remove it from the set. (See ▷ **Messenger** for how client-server cycles are broken when using `selector`.)

⋆ **Occade** is a client-server system, with the application processes usually being clients to Occade's servers. To deliver input and collision report events from Occade to the application, buffer processes are used to prevent client-server cycles.

⋆ **RMoX** largely follows the client-server pattern. Its implementation of TCP provides an example of dynamic reversal of a client-server relationship. Each connection is represented by a `tcp.worker` process, which acts as a client registered with the `tcp.server` process while the connection is active. Once the connection is closed, and the worker can no longer communicate with application processes, the relationship reverses to allow the worker to unregister itself.

4.2.6 Farm

PROBLEM: Each value in a stream needs to be processed in the same way, with work divided between several processors.



Many applications can be expressed in terms of a stream of values being passed through a ▷ **Filter** process. Using a single filter causes serialisation; when the filter process has no internal state, it is straightforward to parallelise the work across a *farm* of *worker* processes, with a process that has the shape of a single filter [95, 122].

Distributing work across a farm of worker processes is a common approach in concurrent programming: many frameworks provide tools for managing worker farms or thread pools [154]. Embarrassingly-parallelisable problems are easily implemented using farms. Network server software is often written in such a way that each incoming connection is assigned to one of a pool of workers [125].

In process-oriented systems, a farm can be implemented using a ▷ **Distributor** process to assign each input value to one of a pool of workers, and a ▷ **Merge** process to collect the outputs from the workers back into a single channel. When shared channel ends are available, this is even simpler: worker processes can read values from a shared output end, and write their results back to a shared input end. However, in distributed applications, the cost of claiming explicitly-shared channels may be significant; implicitly-shared or unshared channels can offer significantly better performance for distributed farming applications [211].

The downside of this simple approach is that it may reorder the values in the stream after processing. For many applications this does not matter, but if the order must be maintained, then one approach would be for the distributor and merge processes to agree upon the order in which values should be delivered to and accepted from worker processes. This may result in processes blocking if different values take different times to process. An alternative would be to use a filter process to add sequence numbers to values as they are received by the farm, and then reassemble the values into the correct sequence (using a sorting buffer process) upon output; this would allow values to be processed out of order.

Distributing work fairly among the workers requires some intelligence in the distributor process (see ▷ **Distributor**); to maximise throughput, the distributor should avoid ever blocking on an output by using choice to select a worker that is able to accept a new value immediately. Using a shared output end instead of a distributor solves this problem. As with ▷ **Pipeline**, introducing buffering before and after the worker processes can reduce communication overheads by allowing worker processes to collect several values to work on at a time, not having to context switch until their output buffer is full or input buffer empty.

Farms normally use a fixed-size pool of workers—often corresponding to the physical resources available for performing the work, such as the CPUs in a multicore system, or more specialised hardware resources. However, there is nothing to prevent the size of the pool from varying at runtime—often within defined bounds, to minimise the overhead of keeping useless worker processes around while providing sufficient parallel workers for the workload. If the cost of forking new processes is low compared to the cost of processing an individual value, then farming can be done entirely dynamically by forking a new worker process for each value received.

Just as a ▷ **Fan-Out** network can be refactored into a pipeline, so can a farm, with the primary benefit being easier wiring of the process network [160]; with the increased availability of shared channels and mobile channel ends, this refactoring is rare in modern applications.

SOLUTION: Use a ▷ Distributor or a shared-output channel to distribute work to several worker processes, and a ▷ Merge or shared-input channel to collect the results.

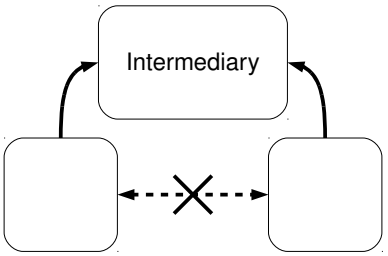
Examples of Use

Some early work on the ⋆ **TUNA** project experimented with different approaches to farming for distributed applications [211].

The various implementations of ⋆ **LOVE**—like most software synthesisers—implement polyphony by assigning notes to one of a farm of oscillator pipelines (see ▷ **Distributor**).

4.2.7 Intermediary

PROBLEM: Two processes need to communicate, but there is no clear directional relationship between them.



Many process-oriented systems contain large numbers of processes that can be considered peers: they need to exchange information on equal terms. However, it can sometimes be inconvenient to design processes so that they may at any time respond to a request from another process: for example, in a system following the ▷ **Client-Server** design rules, connecting peers directly to each other would cause a violation of the no-cycles rule (and potential deadlock).

This problem rarely arises in the real world because real-world communication is always mediated by some kind of intermediary. For example, communication by speech can be considered as both parties interacting with the air between them [107]. This pattern solves the problem in a similar way: by introducing an intermediary process that both processes can safely interact with. This is distinct from providing an ▷ **Oracle** process that *all* the peers share: using individual intermediaries allows greater concurrency, at the cost of requiring a greater number of processes.

An intermediary process is often a kind of ▷ **Buffer**, or a pair of buffers to allow two-way communication. Overwriting buffers are often used to allow decoupling in time without needing to provide large amounts of storage.

SOLUTION: Introduce a new process that can have the same kind of relationship to both parties.

Examples of Use

In some of the TUNA ⋆ **Blood Clotting** simulations, clots are represented by processes that span several locations. When two clots collide, they must merge together into a single larger clot. Clots can communicate using channel ends that they place in locations, but there is no clear client-server relationship between clots: they may encounter each other from any direction and at any time. To allow orderly communication, each agent carries around an intermediary process that can handle requests from itself and from other agents [187].

In the ⋆ **CoSMoS** implementation of a small-world network, edges in the network are occam-π channel bundles with one channel in each direction, to allow arbitrary

communication between vertices. However, channel bundles have an inherent direction (with client and server ends), which meant that the operations that changed connections within the network needed to be able to handle both types of ends. The solution was to represent each edge as two channel bundles connected by an intermediary (actually a parallel pair of ▷ **Buffer** s), meaning that vertices only ever see client ends of channel bundles.

4.3 Patterns of Cooperation

Patterns in this section describe patterns of communication or synchronisation between multiple processes.

4.3.1 Acknowledgement

PROBLEM: Processes need to synchronise for an extended period: for example, the execution of a request.

The usual synchronisation objects only provide a way for processes to synchronise at a point in time—for example, a synchronous channel communication or barrier synchronisation will only complete once all processes are attempting to synchronise. However, it is sometimes useful for processes to synchronise for a *period* of time. The pause process (▷ **Valve**) is an example of this: it is used to block messages passing on a stream for a period of time. Server processes (▷ **Client-Server**) sometimes need to do this as well: a client making a request to a server may want to wait for the server's corresponding action to complete (rather than allowing the server to perform its action at a later time—an asynchronous request).

The conventional solution in process-oriented programs is for the processes to use conventional synchronisation objects, but synchronise twice—once at the start of the period, and again at the end. This ensures that all processes enter and leave the synchronisation period together. When channels are used, the second communication may be in the same direction as the first (as in figure 50), or it may be in the opposite direction; in the latter case it can be used to return the result of an operation.

The second synchronisation must always follow the first: it is thus part of the protocol used on the synchronisation object. However, most existing implementations of protocols do not allow actions to be performed between individual messages in a protocol—which argues for a more powerful idea of protocols based upon session types (section 5.3).

Extended synchronisation is an environment feature designed to support this kind of synchronisation, allowing actions to be performed while the processes are synchronised. However, not all environments provide extended synchronisation, and some (such as occam-π) only provide extended input, meaning that only one party in a synchronisation can perform an action. Call channels allow extended synchronisation with data passing in both directions between the processes, but similarly only allow an action in one process (the callee).

SOLUTION: Synchronise twice: once at the start of the period, and again at the end.

```
PROC io.par <VALUE> (VAL VALUE initial.value,
                     []CHAN VALUE inputs?,
                     []CHAN VALUE outputs!)
  INTIIAL VALUE value IS initial.value:
  [SIZE inputs]VALUE input.values:
  WHILE TRUE
    SEQ
      PAR
        PAR i = 0 FOR SIZE inputs
          inputs[i] ? input.values[i]
        PAR i = 0 FOR SIZE outputs
          outputs[i] ! value
      ...  compute new value from input.values
:
```

Figure 56: I/O-PAR process

Examples of Use

Several kinds of ▷ **Valve** process can usefully be implemented using this pattern, where the effect on the stream must last for a period of time: for example, suppressor processes in a subsumptive control system.

This pattern can be used to simulate barrier synchronisation using only point-to-point channels (figure 8); the barrier process achieves point-in-time synchronisation of a group of processes by overlapping their individual synchronisation periods.

4.3.2 I/O-SEQ and I/O-PAR

PROBLEM: A network of processes that periodically exchange data needs to be arranged so as to avoid deadlock and livelock.

In a network of processes where the state of each process must be computed periodically based upon the states of other processes—such as a physical simulator, or a cellular automaton—the simplest way to arrange for the processes' states to be communicated to those they depend upon is to connect them with channels. Processes can then periodically perform an *exchange*, where they receive the state of all the processes they depend upon, and communicate their state to all their dependents [257].

When the graph of dependencies may contain cycles, processes perform an *I/O-PAR* exchange: each process performs all its input and output communications in parallel (figure 56). When cycles are not present, the *I/O-SEQ* pattern can be used, where each process first performs all its input communications in parallel, then performs all its output communications in parallel (figure 57). The advantage of the I/O-SEQ approach is that the process's new state can be computed between the input and output communications—so when the network has connections to the outside world, each set of values fed in will result in a new set of values coming out immediately, whereas in the I/O-PAR approach the new state will be delayed by one exchange. I/O-SEQ can thus be considered a generalisation of the ▷ **Pipeline** approach to arbitrary cycle-free networks.

Since an I/O-PAR or I/O-SEQ exchange involves all the processes in the network

```
PROC io.seq <VALUE> (VAL VALUE initial.value,
                     []CHAN VALUE inputs?,
                     []CHAN VALUE outputs!)
  INTIIAL VALUE value IS initial.value:
  [SIZE inputs]VALUE input.values:
  WHILE TRUE
    SEQ
      PAR i = 0 FOR SIZE inputs
        inputs[i] ? input.values[i]
      ...  compute new value from input.values
      PAR i = 0 FOR SIZE outputs
        outputs[i] ! value
:
```

Figure 57: I/O-SEQ process

(although not necessarily at the same time) and the pattern of communications is identical for each exchange, these patterns give processes a shared sense of time without the need for global barrier synchronisation. The lack of global synchronisation means that this pattern can offer greater opportunity for parallel execution than regulating computation using ▷ **Phases** or a ▷ **Clock**: the computations in different timesteps can safely overlap.

I/O-SEQ components can be included in an I/O-PAR network provided that there are no cycles consisting solely of I/O-SEQ components; furthermore, as I/O-SEQ exchanges, I/O-PAR exchanges, and ▷ **Client-Server** conversations are all guaranteed to complete in a finite time period, it is safe to combine all three patterns of communication within a single program [257]. A server process may offer a choice between responding to a request on one of its interfaces, and any communication in an exchange; once it begins an exchange, it must complete it before returning to act as a server. In a language without output guards—making choice over an exchange impossible—this can instead be implemented by ▷ **Polling** the server interfaces between exchanges.

Using this pattern involves doing several communications for each process on each timestep, and requires all processes to be involved in every exchange. If the cost of communication is a significant overhead, and the state of few processes in the network change on each iteration, the ▷ **Lazy Updates** pattern can allow significantly more efficient communication at the cost of greater complexity in the individual processes.

SOLUTION: Use the I/O-SEQ or I/O-PAR design rules.

Examples of Use

These patterns are generally used in network simulation and dataflow systems where every process needs to recompute its state on each timestep. The Kent digital circuit simulator [252], several Transputer "butterfly" FFT implementations [258], and the early implementations of ⋆ **Life** are structured as I/O-PAR systems; the various implementations of ⋆ **LOVE** are I/O-SEQ systems.

It is often useful to perform a single I/O-PAR exchange between a group of processes when they start up to synchronise their initial states, before proceeding with a more complex pattern of interactions such as ▷ **Lazy Updates**.

4.3.3 Phases

PROBLEM: Several processes need to safely update and read from shared storage.

Process-oriented designers are strongly discouraged from having processes share memory, instead using single-owner references to communicate the ownership of resources around a process network. However, for applications where data values need to be broadcast to many processes, this can result in significant overheads from the communication—in terms of both design complexity and execution efficiency. The use of phases can provide a way for processes to share memory safely and efficiently.

When using this pattern, the execution of a program in divided into several phases, with a barrier or ▷ **Clock** used to ensure that all processes are synchronised at the end of each phase. Phases often repeat in a cyclic fashion. Each resource (such as a region of shared memory) has a set of access permissions defined for it during each phase; the permissions change only when the phase changes. The permissions are defined so that CREW conditions are maintained: each resource may only be written to by one process at a time, but may be read by any number of processes when it is not being written to.

As an example, consider a simulation in which each agent needs to observe the states of its neighbours during each timestep. We can solve this problem by providing all the agents with access to a phase-protected shared array, and dividing each timestep into two phases. In the first phase, each location in the array is writable to just one agent, which writes its state into its location within the array. In the second phase, all locations are read-only, and each agent can read the states of all its neighbours from the array. The execution of the simulation only requires two barrier synchronisations per timestep—a considerable improvement over access control techniques such as locking or transactional memory, where each access potentially requires a synchronisation.

The same approach can be used to control access to other kinds of shared resources. For example, in a ▷ **Client-Server** system, clients may exchange information with server processes. Phase transitions can be used to guarantee ordering of client-server operations; for example, to ensure that all processes storing data have done so before any processes that are retrieving data get to run. Similarly, phases may be used to regulate changes to client-server relationships: one way of dealing with two-way relationships between peers is to make them clients and servers, but swap the direction of the client-server relationship in different phases.

A disadvantage of the phased approach over directly representing data dependencies using channels (▷ **Lazy Updates**) is that no processes can proceed to the next phase until the previous one is complete—even if the data they need has already been written. Using a single phase system in a large program may therefore lead to poor performance; it can be better to break it up into smaller phase systems protecting smaller sets of shared resources, with appropriately-sloppy synchronisation between them (see ▷ **Clock**). However, this can be mitigated somewhat by using parallel composition to overlap computation with phase synchronisation. Figure 58 shows an agent process that can immediately start computing its new state once it has examined its neighbours, while requiring phase synchronisation to complete in parallel before it publishes its new state.

This style of programming—where many processes operate upon shared resources, synchronising periodically to change access permissions—is known as the Bulk-Synchronous Parallel approach [232]; the BSP literature contains many examples of applications using this synchronisation model.

```
PROC agent (BARRIER bar)
  WHILE TRUE
    SEQ
      SYNC bar  -- phase 0: read from world

      ...  read state of neighbours

      PAR
        ...  compute new state
        SYNC bar  -- phase 1: write to world

      ...  write my state
:
```

Figure 58: Overlapping computation with phase synchronisation

At the moment, no process-oriented environment offers direct support for phases; indeed, in occam-π it is necessary to disable static checking for phase-protected memory in order to allow it to be shared. Section 5.4.5 discusses possible language bindings for phase-protected resources.

SOLUTION: Divide the work into several phases, with different access permissions for the shared resources in each phase.

Examples of Use

This technique was first explored for efficient inter-cell communications in the ⋆ **TUNA** Life simulations, with a shared array and a barrier synchronisation [203]. Further thought was required when this pattern was later combined with ▷ **Lazy Updates**, since to achieve true laziness the cell processes had to resign from the barrier when they were not being updated. Getting the processes to re-enrol atomically without skipping phases was eventually solved through the use of explicit ▷ **Acknowledgement**. In addition, the Life simulation experimented with combining phases and atomic operations, to allow processes to "vote" upon actions within a single phase.

The ⋆ **CoSMoS** generic space model uses a phase system to regulate access to server processes for two purposes: to ensure that all agents get a consistent view of the world by ensuring that all changes to the world have completed before observation begins, and to ensure that local copies of remote locations in a distributed simulation are updated before they are observed [201].

4.3.4 Clock

PROBLEM: Several processes need to share a sense of time.

Applications in which processes must share a sense of time are common—for near-real time in games, for simulated time in simulations, and for more abstract ideas of time such as ▷ **Phases**. We can implement this in a simple way by using a barrier, upon which all the processes synchronise at the end of each timestep. A clock is therefore the combination of a barrier and a variable that records the current timestep (the number of barrier synchronisations); the terminology used for clocks (enrolment, synchronisation, and so on) is the same as that used for barriers.

Many process-oriented environments already provide a way of synchronising processes to a real-time timer (such as occam's TIMERs). It is frequently useful to limit the real-time rate at which a clock can synchronise—for example, to ensure that a game does not run so fast it is unplayable, while allowing it to slow down gracefully if the CPU cannot keep up, or to slow down a simulation to make the visualisation more comprehensible. In these cases, all that is necessary is a process synchronising upon the clock that does a fixed real-time delay between synchronisations.

A distributed program may need to synchronise clocks between several different hosts. To do this, have a process on each host enrolled upon the clock which performs a network synchronisation (for example, both sending and receiving an asynchronous message) with the corresponding processes on its neighbouring hosts during each timestep. This may represent an unacceptable overhead if network latency is much higher than the time that it would normally take the clock to cycle. In these cases, it may be possible to use *sloppy synchronisation*, where the processes only perform the network synchronisation every N ticks, allowing clocks to drift apart slightly. (For example, if each simulation timestep is divided up into multiple phases, there may be no need to synchronise the phase changes between the hosts; the network synchronisation only needs to happen once per simulation timestep.)

The simplistic synchronisation approach described here is inefficient when a process does not need to perform an action on every timestep; see section 5.4 for how clocks can be bound into process-oriented programming environments, allowing more efficient synchronisation.

SOLUTION: Have the processes synchronise upon a barrier at the end of each timestep.

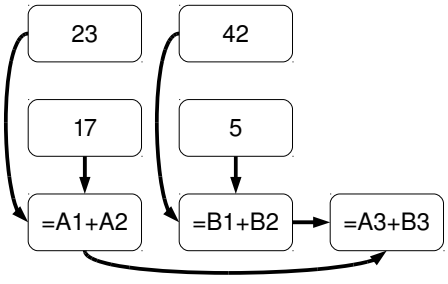
Examples of Use

The term "clock" comes from the X10 programming language, which provides clocks—barriers with counters—as the primary mechanism for synchronising processes [62]. Many X10 programs provide examples of the use of this pattern.

All the ⋆ **CoSMoS** simulations use this pattern to simulate time, with distributed simulations using sloppy synchronisation in the way described above.

4.3.5 Lazy Updates

PROBLEM: Several values with arbitrary dependencies between them need to be updated efficiently.



Process-oriented programming provides a good way of modelling systems of data dependencies. A spreadsheet is an example of this kind of problem: each cell has a value that may be entered by the user, or computed from the values of other cells. When the user changes the value of a manually-entered cell, the spreadsheet should only recompute the values of the other cells that need to change—and it should do so with as great a degree of concurrency as possible, to take advantage of parallel hardware.

We can do this by representing values as processes, and data dependencies as channels. When the value represented by a process changes, it should send the new values to all of the processes that depend upon it in parallel; they can then recompute and send out their new value, and so on, with the changes "rippling out" through the system, resulting in only the values that need to change being recomputed. Note that if a computation returns the same result that it did before, there is no need for a change notification to be sent out.

When the graph of dependencies contains cycles, this approach will not terminate unless the system converges upon a steady state. If this behaviour is desired—for example, when simulating a cellular automaton—then some additional mechanism is necessary to ensure that new values are propagated out fairly between processes. To do this, enrol all value processes upon a barrier, and allow one propagation step per barrier synchronisation (figure 59). On each step, the barrier cannot complete until all processes have had a chance to send out a new value; as communications are synchronous, each process only needs to wait for new input values until the barrier completes.

The downside of this approach is that every cell process must engage in every barrier synchronisation, even if its value does not need to be recomputed. To make updates truly lazy again, ▷ **Just In Time** can be used to dynamically spawn value processes only when the value actually needs to be updated.

Shared memory protected by ▷ **Phases** may be used to avoid the need to transfer values between processes, with the channels only used for synchronisation.

SOLUTION: Represent each value by a process, and each dependency by a channel; when a value changes, have its process communicate the new value to its dependents.

Examples of Use

The ⋆ **Life** simulations (and other cellular-automata simulations built for TUNA) used this approach. Since a typical Life grid is mostly static, only performing recomputations where necessary improved performance by approximately a factor of 15 on a randomly-initialised fixed-size grid [203].

4.3.6 Just In Time

PROBLEM: Entities are modelled by processes—but the overhead of actually having a process for each entity all the time is excessive.

Modelling things as processes is a powerful technique, but even in an environment with lightweight processes, it may be impractical to always have a process running for every modelled entity when the entities are very numerous. Fortunately, it is often unnecessary to actually run the processes; for example, regions of space in a simulation can be modelled by processes (see ▷ **Location**), but the space may only be sparsely occupied, so regions spend most of their time sitting empty.

In these situations, we can get away with delaying the construction of these processes until they are actually required, while making this transparent to the other processes that communicate with them—that is, we construct processes "just in time". We do this by having the interface of the unconstructed processes be provided by an *ether* process (figure 60); when the ether receives a request on one of these interfaces, it can

```
PROC cell <VALUE> (BARRIER bar, VAL VALUE initial.value,
                   []CHAN VALUE inputs?,
                   []CHAN VALUE outputs!)
  INITIAL VALUE value IS initial.value:
  INITIAL VALUE prev.value IS value:
  [SIZE inputs]VALUE input.values:
  SEQ
    ...  do an I/O-PAR exchange of initial values

    WHILE TRUE
      INITIAL BOOL recompute IS FALSE:
      SEQ
        CHAN SIGNAL synced:
        PAR
          SEQ
            -- If value has changed, send to dependents
            IF value <> prev.value
              SEQ
                PAR i = 0 FOR SIZE outputs
                  outputs[i] ! value
                prev.value := value

            SYNC bar
            synced ! SIGNAL

          -- Read input values until the barrier completes
          INITIAL BOOL reading IS TRUE:
          WHILE reading
            ALT
              ALT i = 0 FOR SIZE inputs
                inputs[i] ? input.values[i]
                  recompute := TRUE
              synced ? SIGNAL
                reading := FALSE

        IF recompute
          ...  recompute value from input.values
:
```

Figure 59: Cell process using lazy updates

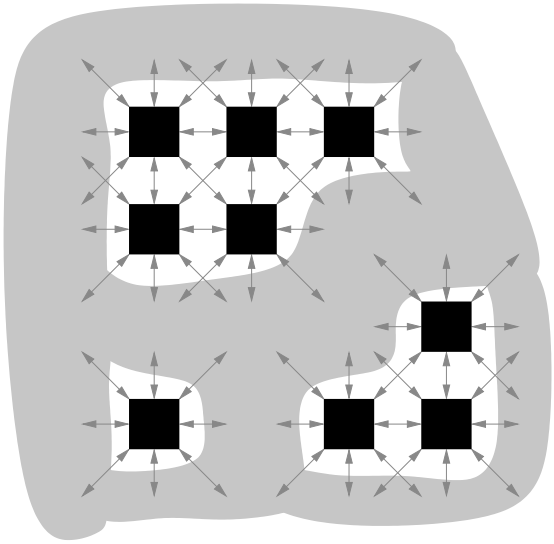


Figure 60: Ether processes model as-yet-unused space

start up the real process and ▷ **Hand-Off** the interface to it. The ether is therefore a kind of ▷ **Factory**; it must store enough information to be able to construct the appropriate processes as required.

If processes are constructed just in time as they are required, it is also generally useful to have them be destroyed when they are no longer required. This can be achieved by having a process be able to shut down and store a representation of its internal state in the ether—an example of process mobility used in time, rather than space. It may not be necessary for a process to store any state at all in the ether if it is sufficiently simple or its state is stored elsewhere—for example, if it is a ▷ **Lazy Updates** process updating memory protected by ▷ **Phases**.

If the cost of forking a new process is significant, the programmer may prefer to have processes only shut down once they have not been used for an appropriate period of time. If processes can be forked cheaply, a simple approach would be to write processes that only handle one request and then immediately suspend again—such a process is effectively a passive object.

An advantage of this approach is that the ether does not always have to fork the same kind of process for each request; it only has to provide a process with the correct interface. Where the same functionality can be implemented in different ways, the ether can pick an implementation based on its own criteria; for example, if it is running on a heavily-loaded host, it may choose an implementation that uses fewer resources. The ether may even choose to fork a process that can provide several interfaces at once—for example, simulating space at different granularities.

The ether may be a ▷ **Bottleneck**; this can be mitigated by dividing it into several coordinated processes.

The ○ **Flyweight** pattern—where an object serves different purposes through different references to it—is related to this [87], but it is generally harder in OO environments

to replace the flyweight with a use-specific object when it is required (unless each reference is via a proxy object). The natural decoupling of synchronisation objects from process identity in process-oriented programming allows a "flyweight" space process to be replaced with a real space process.

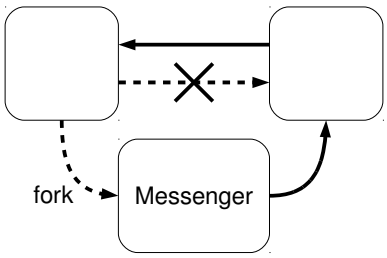
SOLUTION: Have channels to unused resources be connected to an "ether" process; when a resource is used for the first time, have the ether process fork a real process for it and ▷ Hand-Off the channel end.

Examples of Use

The ⋆ **TUNA** Life simulation modelled cells this way [203]. The choice of algorithm used to implement Life can be made based on the pattern of cells in the grid (for example, HashLife works well for repeating patterns); the ether can make this choice when forking new processes.

4.3.7 Messenger

PROBLEM: A server process needs to make a request to a client.



In a ▷ **Client-Server** system, cycles in the graph of client-server relationships must be avoided—but the programmer will occasionally encounter situations where this is unavoidable: a server process needs to make a request to a process that usually acts as a client to it, or two server processes need to act as clients to each other. Sometimes this can be solved by using ▷ **Phases** to dynamically rearrange the client-server relationships in a network, but this is difficult in cases where communication does not happen in regular patterns.

A simpler solution is to fork off a new process that makes the request and then exits: a messenger. Since the new process is only a client, it cannot create a cycle in the graph of client-server relationships.

In a system with ▷ **Phases**, it may be necessary to ensure that the messenger makes its request in the same phase it was forked in. This can be achieved by enrolling the messenger upon the phase barrier; the barrier will not be able to complete until the messenger has exited. (This is an example of a case where ownership of a reference—the barrier enrolment—is all that is required; a compiler for a process-oriented language cannot warn about the reference being unused in this case.)

A messenger is the embodiment of an asynchronous message. Messengers can be used to simulate infinitely-buffered channels—but they can also be used to perform more complicated actions; for example, a messenger can engage in a complicated protocol, making several requests, or making requests to multiple processes.

A messenger process may communicate with the process that spawned it, provided it does so in a way that does not violate design rules. For example, a server process may spawn a messenger to make a request to another server process; once the messenger has finished, it can then make a request to the original server to indicate completion. Because the messenger only ever acts as a client, the client-server design rules are not violated. A server process may even use a messenger as a way of sending a request to

itself—knowing that the messenger will be blocked until the server process is able to accept more requests.

Messenger and ▷ **Intermediary** are both options in cases where server processes are peers but need to communicate. Since using a messenger involves the cost of forking a new process, an intermediary is a better option when many messages need to be sent and the cost of forking would be prohibitive; but a messenger is a better option when communication is sporadic or to unpredictable destinations, and maintaining a permanent intermediary would be an unacceptable expense.

SOLUTION: Fork off a temporary worker process to do the communication.

Examples of Use

In early versions of the ⋆ **CoSMoS** space model, locations were directly responsible for the migration of agents; a location had to ▷ **Hand-Off** the agent's channel end to the destination location when it crossed the boundary. This violated the client-server design rules, since locations were both clients and servers to each other. To solve this, when an agent had to be migrated, a messenger process was forked to introduce the corresponding channel end to the new location. (Later versions were refactored so that agents were responsible for migration, removing the need for the extra process.)

The ⋆ **RMoX** network stack uses messengers to implement ARP. When the hardware address of a machine on the Ethernet to which a packet must be sent is not known, the network stack sends an ARP request packet instead, and forks a messenger process to resend the original packet after a delay, by which time an ARP reply will probably have been received.

In the `selector` module in ⋆ **KRoC**, processes that want to block on I/O operations act as servers to the selector process. As a server would not be able to make a request to the selector itself in order to change the set of file descriptors it is performing I/O upon, the module provides a `selector.delayed.add` procedure which starts a messenger process that acts as a client to the selector to request a change safely.

4.4 Patterns of Mobility

Patterns in this section describe how the structure of the process network—or, more generally, the allocation of resources to processes—may change at runtime.

4.4.1 Private Line

PROBLEM: Processes need to meet in public, but have private conversations.

Channels with shared ends can be used to allow several processes to communicate with each other, without needing to set up point-to-point channels corresponding to every possible communication, or requiring processes to route messages. However, shared channels—like shared radio frequencies or telephone party lines—do not cope gracefully with long conversations: explicitly claiming a shared channel for a long period of time will block other processes from using it.

The solution is to make use of channel mobility. A process that wishes to engage in a conversation over a shared channel should first allocate a new mobile channel, and then use the shared channel to communicate one of the ends of the mobile channel to the process with which it wishes to communicate. The two processes are then

connected by the mobile channel—a private line, over which they can have as long a conversation as they like without tying up the shared channel.

The use of private lines can be used to simulate explicit sharing of channels in environments that only have implicitly-shared channels (or do not have protocols that would allow protracted communication over a shared channel): rather than claiming a shared channel, you send a private line. This emulation could be implemented transparently by a process-oriented environment—and if implicitly-shared channel ends can be implemented more cheaply than explicitly-shared channel ends, doing so would have performance advantages. The cost of allocating the private line could be reduced by smart management of references: allocate a new private line whenever a new shared channel end reference is created, and reuse it for all subsequent "claims" of that channel end.

SOLUTION: Use public communication to pass a mobile channel end that can be used for a private conversation.

Examples of Use

Early ⋆ **TUNA** experiments with farming made use of private lines to communicate between the farmer and worker processes, with a significant performance advantage in distributed farms owing to the high cost of distributed shared channels.

⋆ **RMoX** makes heavy use of private lines for connections between application processes and hardware drivers (see ▷ **Snap-Back**).

4.4.2 Snap-Back

PROBLEM: Processes need to keep track of how many other processes they are communicating with.

In applications where processes need to control how many other processes they communicate with—typically, where a process is managing a fixed-size pool of resources—it is useful to treat the "user" end of a ▷ **Private Line** as a single-owner token, the ownership of which confers the right to use the resource. When a user is finished with the resource, they should return the channel end to the issuing process—which can be done as a communication through the channel itself. Figure 61 shows a two-way protocol in occam-π with a `close` message that uses this approach.

It is safe for a user process to ▷ **Hand-Off** its end of the private line to another process, provided it does eventually get returned in the correct way.

One possible problem with this approach is that protocols in process-oriented environments typically can only provide type safety; they cannot guarantee that the reference returned is actually the one that is being used to make the request. (The same problem applies to ▷ **Loan**.) This pattern could be explicitly supported in protocol definitions by allowing a "return self" operation to be specified; given a sufficiently expressive two-way protocol facility (section 5.3), this could be done automatically when the protocol reached a state where no further requests were possible.

SOLUTION: Require mobile channel ends to be returned to the originating process as the final communication on the channel.

```
CHAN TYPE CONNECTION:

PROTOCOL CONNECTION.REQ
  CASE
    ...  other requests
    close; CONNECTION!   -- snap-back message; no response
:

PROTOCOL CONNECTION.RESP
  CASE
    ...  responses
:

CHAN TYPE CONNECTION
  MOBILE RECORD
    CHAN CONNECTION.REQ req?:
    CHAN CONNECTION.RESP resp!:
:
```

Figure 61: A client-server connection using snap-back

Examples of Use

⋆ **RMoX** makes extensive use of this pattern to control access to hardware devices. For example, it does not make sense for more than one process to make use of a USB device; to prevent this, connections to the low-level USB driver require the client end of the connection to be returned through the connection when no longer needed.

4.4.3 Loan

PROBLEM: Several processes must compete for individual access to a resource.

When a server process is used to encapsulate a single-owner resource, one way to allow client processes to make use of it is to provide requests that cover all the possible operations that the client might want to perform. This may require data to be copied, though, particularly if the common operations involve modifying or reading parts of a large resource.

An alternative is to allow a client to borrow the resource: move the reference into the client process, which can then perform whatever operations it likes—which might include reading or modifying the resource, or even lending it to another process in turn. When the client is finished, it returns the reference to the server. This is implemented using a three-step protocol (figure 62):

$$\mathit{borrow}; \mathit{RESOURCE}! \rightarrow \mathit{lend}; \mathit{RESOURCE}? \rightarrow \mathit{return}; \mathit{RESOURCE}!$$

This pattern is an example (in fact, the most common example) of a two-way protocol that includes more than two messages: it cannot be safely implemented using a protocol system that only allows request-response interactions (such as call channels). In such a system, an alternative approach would be to allow clients to send the server a closure that the server can execute with the shared resource as a parameter.

The downside of this approach is that it serialises execution: the processes that wish

```
CHAN TYPE CONNECTION:

PROTOCOL CONNECTION.REQ
  CASE
    ...  other requests
    borrow; RESOURCE      -- (1), replies "lend"
    return; RESOURCE      -- (3), done
:

PROTOCOL CONNECTION.RESP
  CASE
    ...  other responses
    lend; RESOURCE        -- (2), replies "return"
:

CHAN TYPE CONNECTION
  MOBILE RECORD
    CHAN CONNECTION.REQ req?:
    CHAN CONNECTION.RESP resp!:
:
```

Figure 62: A client-server connection that loans a resource

to use the resource must queue up to use the server process, and they must copy out the data that they want from the resource. If this pattern is being used to distribute the same data to many processes that do not want to modify it, then using shared memory protected by ▷ **Phases** will probably allow greater concurrency.

As with ▷ **Snap-Back**, there is no guarantee that the reference returned is the reference that was originally loaned. For many applications this would be acceptable: a user could reasonably allocate a new resource and copy the contents over (for example, if an array was being resized), secure in the knowledge that nothing else can have a reference to it. For other applications—for example, where the resource being loaned is a piece of hardware—it would be helpful to ensure that the same reference is returned; this could be achieved through more expressive protocols (section 5.3).

SOLUTION: Use a three-step protocol to request, lend and return a single-owner reference to the resource.

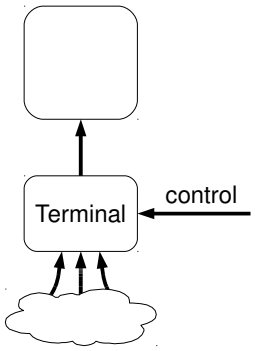
Examples of Use

▷ **Snap-Back** is a specialisation of this pattern, where the resource is a channel that is used to return itself.

The ⋆ **CoSMoS** space model uses this approach to distribute views of space to multiple processes. A view is a single-owner reference to a list of other agents within a ▷ **Location**, normally held by the location; other processes (agents, network proxies, visualisation processes) can borrow the view, returning it once they have retrieved the data they need.

4.4.4 Terminal

PROBLEM: Connections between processes need to be dynamically rearranged under the control of another process.



Mobile channel ends allow a process network to be dynamically reconnected at runtime. In process-oriented environments, it is not possible for one process to directly reconnect the channels of another process, since that would mean breaking encapsulation; instead, channel ends must be managed by the processes to which they are connected.

A terminal process manages mobile channel ends on behalf of another process, connecting them internally to static channels. This allows processes written as if they use only static channels to be dynamically reconnected. A terminal process provides a server interface that allows other processes to provide it with a new channel end, or to take away an existing channel end.

The typical applications for terminal processes are interactive or adaptive systems, where process networks need to be constructed by a *controller process*. The controller can start up new component processes (along with their associated terminals), allocate new channels, and pass the ends of the channels to the terminal processes. Both terminal processes and controllers follow fairly regular forms; future process-oriented environments may be able to provide standard components for constructing process networks described by data structures.

Central control is not necessary, however; dynamic process networks can also be managed in a decentralised manner, where larger components reconfigure their internal processes dynamically. This kind of architecture is useful for dataflow systems, for example, where components can reconfigure themselves to suit the data passing through them.

Mobile channel ends can be connected to static channel ends inside the terminal in several different ways. When each static channel end may only be connected to one mobile channel end, the terminal can use ▷ **Glue** or ▷ **Buffer** processes. Alternatively, a terminal may allow several mobile channel ends to be connected, in which case it could act as a ▷ **Delta**, ▷ **Distributor**, ▷ **Merge** or ▷ **Collector** process where the set of connected channels can vary at runtime.

More complex communication semantics are also possible. For example, it may be preferable that an application should not block when some of its terminals do not have channels connected to them; in this case, an output terminal should behave as a ▷ **Black Hole** when no mobile channel is connected.

SOLUTION: Use terminal processes that connect a set of mobile channel ends to a static channel end, with a control interface to allow channel ends to be connected and disconnected.

Examples of Use

⋆ **LOVE**'s components are connected using terminal processes; a LOVE component consists of a process written using static channels, with a controllable terminal process attached to each channel, linked back to a central controller process that accepts commands from the LOVE GUI. LOVE's output terminals can have multiple channels

connected to them, behaving as ▷ **Delta** processes.

⋆ **occvid** provides several kinds of terminal process that wrap a subnetwork of components such as an input file reader or a stream filter.

The ⋆ **CoSMoS** implementation of L-systems models the string of symbols as a pipeline of processes, using terminals to allow new symbols to be inserted into the pipeline as evolution rules are applied [10].

A terminal process with delta semantics can be used to implement the ○ **Observer** pattern in a process-oriented system, where the terminal can be used to send notifications to all the processes connected to it, with new processes registering themselves at runtime. The `dnotify` process in ⋆ **RMoX** works this way, providing other processes with notifications of new hardware devices being connected to the machine. (As described in ▷ **Delta**, this is effectively an emulation of a broadcast channel—but one where new references to it can be created on-the-fly.)

Another example of a delta terminal is found in ⋆ **Occade**: the event filter processes used to receive input events from the user are internally registered with a terminal that delivers all events to them, which they then filter for delivery to the application processes.

A terminal process with distributor semantics can be used to implement dynamic routing based on some aspect of the messages received on the static channel—that is, when a process registers a new channel end, it also specifies how to identify the messages that should be delivered to it. The RMoX network stack's implementation of TCP has a worker process for each open TCP port; worker processes register a channel end with a terminal to receive messages addressed to that port.

4.4.5 Hand-Off

PROBLEM: Once a server has established communications with a client, it needs to delegate dealing with the client to another server.

A server that is communicating with a client may wish to delegate the task of dealing with the client to a different server, without the client being aware that this has happened. This is called *handing off* the client to the new server.

The approach is to use a ▷ **Private Line** for the client-server connection, and to communicate the mobile server end of the private line to the destination server. This is completely transparent to the client; in an environment with two-way protocols (section 5.3), hand-off can even happen during a client-server conversation with the correct state in the new server being verified by the compiler. Hand-off cannot be performed when an explicitly-shared channel is used to provide the server interface (unless *all* future clients need to be handed off).

Hand-off has several applications. It can be used to determine what services the client needs before starting (or restoring from a suspended state—see ▷ **Just In Time**) a more specific server process to handle the request. It may be used for load balancing, with a front-end process distributing clients among a ▷ **Farm** of servers (an approach familiar from the world of high-performance network servers). It may be used when a client does not know which server it should communicate with; a directory server—or even a hierachy of directory servers, following the model of telephone exchanges—can hand it off to the correct server.

The easy transparency of hand-off is made possible by the separation of synchronisation objects from processes in process-oriented environments; in an object-oriented

system, it would be necessary to introduce a proxy object for each handed-off interface that could hide the details of hand-off from the client.

Hand-off is similar to but distinct from the ○ **Delegate** pattern in object-oriented programming, where an object implements some of its operations by forwarding them to another object (although this can of course be implemented using processes—and often is, for example in ⋆ **CoSMoS**'s space model's agent-abstraction processes). When hand-off is used, the original process is no longer involved with the conversation after it has been handed off, and can proceed with other work.

Where the server processes involved are peers, it may be necessary to use a pattern such as ▷ **Messenger** to avoid hand-off causing a cycle in the client-server graph.

SOLUTION: Establish a ▷ Private Line to the client, and pass the end of the private line to the new server.

Examples of Use

⋆ **KRoC**'s `pony` module uses hand-off to implement transparent networking for mobile channel bundle ends. When a mobile channel bundle end is communicated across a network link, a notification message is sent to the pony system on the destination host, and the local channel bundle end handed off to a pony process that serialises and deserialises the data communicated and sends it across the network, using the ▷ **Glue** technique to maintain normal occam-π channel communication semantics.

Some ⋆ **CoSMoS** simulations use hand-off to allow agents to move between different ▷ **Location** processes. Each agent has a private line to its current location; when it sends a message to the location indicating that it has moved out of the location's region of space, the location hands it off to the next location in the direction of movement, passing its new relative location (meaning that it will eventually reach the correct destination, even if it has moved across several locations in a single step).

⋆ **RMoX** uses a hierachy of directory processes as described above to make operating system services, implemented as server processes, available to user programs.

4.5 Adapting Existing Patterns

Existing patterns from object-oriented design, describing the relationships between objects in terms of references, can often be used in process-oriented programs to describe relationships between processes in terms of channels; this section gives some examples, but many more are possible. The translations are especially direct when an object-oriented pattern is expressed in terms of ▷ **Client-Server** interfaces.

The ○ **Facade** pattern encapsulates a complex subsystem of objects inside a wrapper with a simpler interface [87]. Rather than having references to the internal objects directly, external objects may only reference the facade. Similarly, a process may be internally composed of a complex network of subprocesses—but rather than giving those subprocesses direct access to external channels, communications can be mediated through a facade process. ⋆ **RMoX**'s kernel is an example of a facade, presenting a standardised interface to application programs.

The ○ **Adapter** pattern uses a wrapper to translate one interface into another [87].

In a process-oriented system, adapters transform the protocols used between two processes—for example, converting one data type into another, attaching tags to data values, or providing a simple protocol in terms of a more complex one. For a simple unidirectional protocol, an adapter is a type of ▷ **Filter**.

For example, in ⋆ **KRoC**'s `file` module, the option parser `file.get.options` was replaced by `file.get.long.options`, which supports more complicated styles of option and exposes a more powerful interface. For backwards compatibility, the original process was rewritten in terms of the new process, using an adapter to transform the old interface into the new one.

A more complicated interface between processes may involve bidirectional protocols, and multiple channels, with the adapter translating some and leaving others unchanged. The `seekable.wrapper` process in ⋆ **occvid** is an example of this kind of complex adapter.

The ○ **Proxy** pattern involves an object providing the interface of another object by forwarding requests to it—for example, over a network link [190]. Proxy processes are used to create efficient distributed client-server applications; for example, the ⋆ **CoSMoS** space model uses proxies for remote ▷ **Location** processes which cache the contents of the location and migrate agents moving into the location.

The ○ **Memento** pattern involves an object constructing an opaque representation of its internal state that can later be used to reconstruct it [87]. The equivalent in process-oriented programming is a mobile process; mementos are used to simulate mobile processes in environments that do not directly support them (section 2.2.5).

The ○ **Observer** pattern involves objects registering to receive later notifications; this can be implemented using a ▷ **Delta**-style ▷ **Terminal** process (section 4.4.4).

4.6 Antipatterns

Antipatterns are the opposite of patterns: identifiable, recurring features of a system's design that are known *not* to work well [58]. Antipatterns represent design approaches that should be avoided wherever possible.

A general piece of advice is to avoid inventing new kinds of process networks where possible: if a problem can be expressed in terms of one of the well-understood structures described in section 4.2, this will usually make it easier to understand and reason about. This may require introducing additional processes.

4.6.1 Bottleneck

A bottleneck is a single process or synchronisation object that causes other processes to block: for example, a shared server that every other process in the program communicates with would be a bottleneck. Not only do bottlenecks tend to serialise the execution of the program, reducing opportunities for parallelism, they can also increase communication overheads, since contended synchronisations are often more expensive than uncontended synchronisations.

Bottlenecks are sometimes unavoidable or even desirable—a barrier is an *intentional* bottleneck—but often they are simply caused by failing to express potential concurrency in a problem. When trying to solve the problem of a heavily-contended resource, think about how it can be divided up into multiple resources—can a process's

```
PROC receive <ITEM> ([]CHAN ITEM in?, RESULT []ITEM values)
  INITIAL []BOOL waiting IS [i = 0 FOR SIZE in | TRUE]:
  WHILE any (waiting)
    ALT i = 0 FOR SIZE in
      waiting[i] & in[i] ? values[i]
        waiting[i] := FALSE
:
```

Figure 63: Receiving from several channels using a state machine

```
PROC receive <ITEM> ([]CHAN ITEM in?, RESULT []ITEM values)
  PAR i = 0 FOR SIZE in
    in[i] ? values[i]
:
```

Figure 64: Receiving from several channels using processes

responsibilities be divided among several smaller processes, or duplicated among several identical processes? Does a barrier really need to synchronise every process in the system, or can you get away with smaller groups of processes being tightly synchronised, with looser periodic synchronisations between them?

4.6.2 State Machine

Solving a problem that has a high degree of natural concurrency in an environment without support for concurrency usually involves building a state machine. Programmers who are used to this approach will often attempt to build state machines in process-oriented programs—but this is almost always the wrong approach, resulting in code that is harder to write and harder to understand, and exposes less potential for parallel execution.

The correct approach to tracking state in a process-oriented program is to use processes to do so—that is, to use the position of execution inside a process to represent the state of the entity being modelled. For example, in a network server that must deal with many clients communicating using a complex network protocol, a state machine solution would deal with several clients in one process, with variables to indicate which messages had been received. A process-oriented solution would use one process per client, with the structure of the process corresponding to the structure of the network protocol.

Multiple processes are useful even for very simple operations. Figure 63 shows an example of a procedure that reads one value from each of several channels into an array, keeping track of which channels have already provided a value. This can be expressed much more concisely using multiple processes: figure 64 shows how parallel composition can solve the problem. The latter solution allows the runtime system to decide how best to schedule the inputs, rather than serialising them within a single process; it offers more opportunities for parallelism.

Similarly, the pause process in figure 50 could instead have been written with a variable tracking whether the process is paused or not—figure 65. This results in longer code with fewer invariants—it may be either paused or not at the top of the loop. This

```
PROC pause <ITEM> (CHAN ITEM in?, out!, CHAN SIGNAL control?)
  INITIAL BOOL paused IS FALSE:
  WHILE TRUE
    PRI ALT
      control ? SIGNAL
        paused := NOT paused

      ITEM value:
      (NOT paused) & in ? value
        out ! value
:
```

Figure 65: Pause process using a state machine

```
PRI ALT
  channel ? value
    ...  channel is ready
  SKIP
    ...  channel is not ready
```

Figure 66: Polling idiom

is not always the cause—the buffer process in figure 45 is more concise with the `full` state variable—but the use of variables that track process state should generally be cause for concern. A process-oriented programmer should not hesitate to communicate or present a server interface in more than one place in a loop, if it results in clearer code.

4.6.3 Polling

Process-oriented programs are inherently event-driven, and many process-oriented runtime systems make use of cooperative scheduling for greater efficiency. If your program needs to wait for something to happen, it is always better to get the runtime system to do the waiting for you; it is more efficient, and you will not accidentally block something else that is trying to run. In general, you should avoid busy-waiting and polling loops in process-oriented languages.

This is not always possible, of course; sometimes you need to wait for something that the runtime system does not know how to wait for, such as a change in a hardware register or a long-running call in a third-party library. In these cases, encapsulating the polling within a process that presents a channel-based interface will allow choice over the event being polled (and allow the process to be replaced with a more efficient implementation in the future). Where using choice is not possible, you need to ensure that you give the runtime system a chance to run other processes while you are waiting—for example, by explicitly requesting rescheduling during the polling loop.

The "polling" idiom in occam-π is sometimes used to check whether a channel is ready without blocking (figure 66). For example, a process may have a main loop that synchronises on a timestep barrier, but also needs to respond to events coming in on a channel. Since occam-π does not (yet) support choice on barriers, the programmer's only option is to synchronise unconditionally on the barrier and poll the channel on

each cycle. This is easy to get wrong, however: you must make sure that you collect all events coming in on the channel (that is, keep polling until the input channel is not ready). Furthermore, you must ensure that the process sending to the channel does not mind that you are ignoring it until after the barrier has completed—for example, if you have two processes both enrolled on the barrier that also send messages to each other at random times, this will lead to deadlock unless buffered channels are used. You can reason about the safety of this sort of situation by thinking of the barrier as a client process that drives both the others as servers—and, indeed, the solution to this problem used in the more recent ⋆ **CoSMoS** simulations is to have a process enrolled upon the barrier that invokes the agents through a server interface.

Chapter 5

Language Enhancements

This chapter describes several proposals for language enhancements to better support process-oriented programming, based upon the design patterns in section 4. While these features are described in terms of extensions to occam-π, they could equally well be integrated into other process-oriented environments.

5.1 The Unit Protocol

It is often useful to use a channel purely for synchronisation, rather than for communication: the programmer does not care what value has been sent, just that the communication has happened. For example, a process representing a button in a GUI will typically send a message down a channel when it is clicked—but the message contains no meaningful content; a process waiting for a button click only needs to know that a message has been received.

In occam, it is necessary for the programmer to use a conventional protocol and send a "dummy" value (by convention, `CHAN BOOL`, sending the value `TRUE`). This is awkward—not least because it is necessary for the receiver to define a variable to hold the value being received. A better solution is to define a variant protocol, but it is awkward to include such a protocol definition in every program that uses this approach, especially as there is no one-line shortcut syntax for variant protocols with a single message.

In programming languages that provide a "unit type" with only a single value—for example, Haskell's `()`—a channel carrying the unit type can be used for this purpose. However, occam-π offers no such type.

occam-π has been extended to allow channels carrying the protocol `SIGNAL` to be defined [118]. This protocol has only a single message: `SIGNAL` (figure 67).

The implementation is simple and fully compatible with existing occam-π code.

```
CHAN SIGNAL channel:
PAR
  channel ! SIGNAL
  channel ? SIGNAL
```

Figure 67: Using `CHAN SIGNAL`

```
PROTOCOL SIGNAL
  CASE
    SIGNAL
:
```

Figure 68: The definition of the SIGNAL protocol

```
MOBILE CHAN FOO out!:
MOBILE CHAN FOO in?:
SEQ
  out, in := MOBILE CHAN FOO
  PAR
    out ! some.foo
    in ? other.foo
```

Figure 69: Mobile channel ends

SIGNAL is a built-in variant protocol, defined exactly as if the user had started their program with the conventional definition in figure 68 [122].

The only change necessary to the syntax of occam-π was to make the CASE keyword optional in variant input communications such as `channel ? CASE tag`. This also has the effect of making the syntax for ! and ? more symmetrical.

As no new keywords are defined, backwards compatibility is preserved with programs that already use SIGNAL as a name.

5.2 Mobile Channel Ends

At present, occam-π allows ends of channel bundles to be mobile, but not individual channel ends—meaning that channels must be wrapped inside channel bundles to make their ends mobile.

The existing syntax for channel end abbreviations in occam-π appends the ! and ? decorators to the name of the abbreviation. The same syntax could be used for mobile channel ends (figure 69).

However, this makes it impossible to write the type of a channel end on its own—for example, if you wanted to declare a protocol carrying channel ends, or define a type alias. Where does the decorator go? It would be simpler to always include the direction as part of the type of a channel end (figure 70).

```
MOBILE CHAN! FOO out:
MOBILE CHAN? FOO in:
```

Figure 70: Simpler syntax for channel ends

```
PROTOCOL DIE.REQ
  CASE
    roll; INT
    quit
:
PROTOCOL DIE.RESP
  CASE
    rolled; INT
    dropped
:
CHAN TYPE DIE
  MOBILE RECORD
    CHAN DIE.REQ req?:
    CHAN DIE.RESP resp!:
:
```

Figure 71: Die interface using a channel bundle

5.3 Two-Way Protocols

This section proposes *two-way protocols* as a language extension for occam-π. (A version of this section was presented by the author at CPA 2008 [200].)

In a process-oriented system, it is very common to have client-server relationships between processes: a server process answers requests from one or more clients, and may itself act as a client to other servers while processing those requests. The ▷ **Client-Server** pattern allows the construction of client-server systems of processes that are guaranteed to be free from deadlock and livelock, and has proved extremely useful when building complex process-oriented systems. Process-oriented servers have expressive interfaces: a client-server communication can be a *conversation* containing several messages in both directions, not just a single request-response pair.

Most non-trivial occam-π programs today make some use of the client-server pattern, with communication implemented using channels. However, while occam-π allows the protocol carried over an individual channel to be specified and checked by the compiler, it does not yet provide any facilities for checking the protocols used across two-way communication links such as client-server connections.

5.3.1 Client-Server Communication

Client-server communications are currently implemented in occam-π using a pair of channels: one carries *requests* from the client to the server, and the other carries *responses* from the server to the client. The two channels are usually packaged inside a channel bundle. We can use this approach to specify a client-server interface to a random-number generator, which will attempt to roll an N-sided die for you, and either succeed or drop it on the floor (figure 71).

The `req` and `resp` channels carry requests and responses respectively, each with their own protocol. In this case, a `roll` message from the client would provoke a `rolled` or `dropped` response from the server; a `quit` message would cause the server to exit with no response. To use this process, a client need only send the appropriate messages over the channels in the bundle (figure 72).

```
PROC roll.die (DIE! die)
  SEQ
    ...  obtain die

    die[req] ! roll; 6
    die[resp] ? CASE
      INT n:
      rolled; n
        ...  rolled an 'n'
      dropped
        ...  dropped the die

    die[req] ! quit
:
```

Figure 72: Using the die interface

```
SEQ
  die[req] ! roll; 6
  die[req] ! roll; 6
```

Figure 73: Using the die interface incorrectly

The syntax for defining and using client-server interfaces is rather clumsy. Each client-server interface requires two protocols and a channel bundle type to be declared. The protocol names—`DIE.REQ` and `DIE.RESP` in this case—are usually only used within the channel bundle definition. When sending messages over a client-server interface, the name of the channel being used must always be specified, even though it is unambiguous from the direction of communication whether the `req` or `resp` channel should be used.

More seriously, occam-π provides no facility for specifying the relationship between the two protocols in a client-server interface. By convention, the programmer writes a comment saying "replies rolled or dropped" next to the definition of `roll`, but this is only useful to humans. The compiler cannot check that the processes using the channel bundle are correctly ordering messages between channels. For example, the process in figure 73 correctly follows the protocol on each individual channel—but it will deadlock because the server expects to only receive a single `roll` message before sending a response. At the moment, it will be accepted by the compiler without complaint.

The vast majority of channel bundle definitions in existing occam-π code are client-server interfaces like `DIE`. Providing a more convenient language binding for client-server interfaces would not only simplify many programs, but also allow the compiler to detect more programmer errors at compile time.

5.3.2 Related Work

Facilities for specifying two-way communication are present in other concurrent languages.

```
PAR
  -- Client
  CALL cosine (RESULT REAL32 result, VAL REAL32 x):

  -- Server
  ACCEPT cosine (RESULT REAL32 result, VAL REAL32 x)
    result := COS (x)
```

Figure 74: occam 3 call channel syntax

```
service class Console :
{
  ...

  sequence
    receive command
    if command
      write
        acquire String
      read
        sequence
          receive Cardinal
          transfer String
}
```

Figure 75: Honeysuckle compound services

The draft occam 3 language specification [34] described a *call channels* mechanism built on top of channel bundles; this provided a way of declaring channel bundles that were used for call-response communications. The declaration of a call channel therefore implicitly defined protocols to carry the parameters and results of a procedure. The suggested syntax made clients look like procedure calls, and servers look like procedure declarations (figure 74).

Since ACCEPT is implemented as a channel input for the parameters, followed by a channel output for the results after the block is complete, it is possible to use it as a guard in an ALT. The same idea has been implemented in other process-oriented frameworks such as JCSP [244].

occam 3-style call channels are a useful abstraction for programmers transitioning from the object-oriented world, since they make calls to a server look like method calls upon an object. However, they only allow a single request and response; they do not provide the richer conversations afforded by protocols.

The Honeysuckle language provides facilities for easily composing client-server systems, with interfaces being defined as *services* [74]. Of particular interest here are *compound services*, which allow a server's behaviour to be specified using a subset of Honeysuckle including communication, choice and repetition constructs (figure 75).

This notation is very powerful; it allows arbitrary conversations between a client and server to be precisely specified. It is, however, possible to specify a protocol that

```
contract Die {
  in  message Roll();
  out message Rolled(int value);
  out message Dropped();
  in  message Quit();

  state START: one {
    Roll? -> (Rolled! or Dropped!) -> START;
    Quit? -> FINISH;
  }
}
```

Figure 76: The die interface as a Sing# contract

cannot be statically verified by using repetition with a count obtained from a channel communication. Such protocols may require runtime checks to be inserted by the compiler if the repetition counts cannot be statically determined.

The most flexible implementation of two-way protocols is in Sing#, where channel interfaces are defined using *channel contracts* [79]. A contract defines a set of states that the channel may be in and a list of structured messages that may be sent. Each state defines the set of messages permitted in that state, and the state transitions resulting from each message (figure 76).

5.3.3 Session Types

Session types [106] provide a formal approach to the problem of specifying the interactions between multiple processes, by allowing communication protocols to be specified as types. The type of a communication channel therefore describes the sequence of messages that may be sent across it. For example, a channel with the session type

$$foo! \,.\, bar?$$

can be used to send ("!") the message foo, then receive ("?") the message *bar*; the "." operator sequences communications.

Session types can also specify choice between several labelled variants using the "|" operator. For example,

$$(\mathit{left}!\ .\ \mathit{INT}!)\ |\ (\mathit{right}!\ .\ \mathit{BYTE}!)$$

can be used to either send *left* followed by an integer, or *right* followed by a byte.

When checking the correctness of a process, a session-type-aware compiler will update the type of each channel as communications are performed using it. For example, if a channel's session type is initially $\mathit{foo}!\ .\ \mathit{bar}?\ .\ \mathit{baz}?$, after it is used to send the message *foo*, its session type will be updated to $\mathit{bar}?\ .\ \mathit{baz}?$.

Session types were originally defined in terms of the π-calculus, but can also be applied to network protocols, operations in distributed systems, and—most interestingly for our purposes—communications between threads in concurrent programming languages.

Neubauer and Thiemann [146] describe an encoding of session types in Haskell's type system, representing communication operations using a continuation-passing approach. Session types may be defined recursively, which is convenient for specifying protocols containing repetition or state progression—for example, a type may be defined as several operations followed by itself again. The specifications are applied to sequences of I/O operations, such as communications on a network socket; there is no discussion of their application to local communication, although the same approach could be used to sequence communication between threads.

The L_{doos} language [69] integrates object-oriented programming and session types. Its session type specifications cannot contain branching or selection, but they support arbitrary sequences of communications in both directions, making them more flexible than simple method calls.

Vasconcelos, Ravara and Gay [234] give operational semantics and type-checking rules for a simple functional language with lightweight processes and π-calculus-style channels, where channel protocols are specified using session types. Its session types may be defined recursively, and may include choice between several labelled options. It notes that aliasing of channels can introduce consistency problems, since operations may affect one alias and not update the session type of the others. It demonstrates that session types can be applied effectively to communication between local concurrent processes.

The SJ language [108] extends Java with *session sockets* that are conceptually similar to TCP sockets (and are implemented using TCP), but over which communication takes place according to protocols which are defined using session types. It supports conditional and iteration constructs in which the branch taken is implicitly communicated across the socket by the sending process; this ensures that the two ends cannot get out of step.

Connected session sockets can be passed around between processes, and the SJ system tracks their session types correctly even when they are in mid-communication; this makes it possible to hand off a connected socket to another process to continue the conversation.

```
PROC roll.die (CHAN DIE die!)
  SEQ
    die ! roll; 6
    die ? CASE
      INT n:
      rolled; n
        ...  rolled an n
      dropped
        ...  dropped the die
:
```

Figure 77: Using the die interface with two-way protocols

5.3.4 Two-Way Protocols

occam-π's protocols could be generalised so that they can specify two-way conversations rather than just sequences of one-way communications.

occam-π's unidirectional protocol specifications can be viewed as a restricted form of session types: all the communications must be in the same direction, only a single choice is permitted at the start of the protocol, and no facilities are provided for iteration or recursion within a protocol (although the same protocol can be used multiple times across the same channel). In order to specify two-way communications, we must relax some of these restrictions.

For example, the `DIE` interface above could be expressed as a single two-way protocol between the client and the server. In this protocol, a client starts a conversation by sending a `roll` or `quit` message; the server will reply to `roll` only with `rolled` or `dropped`. We can specify this as a session type from the client's perspective:

$$(\mathit{roll}! \,.\, \mathit{INT}! \,.\, (\mathit{rolled}? \,.\, \mathit{INT}? \mid \mathit{dropped}?)) \mid \mathit{quit}?$$

(occam-π programmers would not need to use this syntax for protocol definitions; see section 5.3.6.)

Protocols can contain multiple direction changes; for example:

$$\mathit{move}! \,.\, (\mathit{moved}? \mid (\mathit{suspend}? \,.\, \mathit{suspended}!))$$

One valid conversation using this protocol would be *move*!, *suspend*?, *suspended*!; another would be *move*!, *moved*?. The conversation *suspended*! would not be valid.

We constrain the first communication in a two-way protocol specification to always be an output; this makes it possible for the compiler to always be able to tell in which direction the next communication is expected to come.

A client-server connection can now just appear as a channel to the occam-π programmer; there is no need to specify whether a particular communication is a request or a response, since that is implicit in the operation being used. Our `DIE` client can now be written as in figure 77—and the compiler now has enough information to be able to tell that the equivalent of the process in figure 73 does not conform to the protocol.

Since the session type is now tracked between multiple communications on the same channel, we could allow sequential communications to be split up over multiple communication processes: that is, `die ! roll; 6` would be merely syntactic sugar for

`die ! roll` followed immediately by `die ! 6`.

5.3.5 Implementation

Two-Way Channels

Two-way channels could be implemented by the occam-π compiler using a pair of regular channels inside a channel bundle. The transformation required would be very straightforward, simply selecting the "request" or "response" channel inside the bundle based on the direction of communication. This approach would allow the translation of occam-π code using two-way channels into code that would be accepted by the existing compiler.

However, all existing implementations of occam-π channels on a single machine allow communication in either directions, provided both users of the channel always agree about the direction of communication they expect. Since session types allow the compiler to reason about the direction of communication whenever a channel is used, we can use the existing occam-π runtime's channels as two-way channels with no additional overhead.

This also offers a small memory saving: one channel can now be used where two and a channel bundle were previously necessary. This may be useful in programs with very large numbers of channels.

Protocol Checking

Checking that one-way protocols are used correctly is simple: each input or output operation must always perform the complete sequence of communications that the protocol describes, so the compiler can tell what communications should happen from the type of the channel alone. Two-way protocols complicate this somewhat because the protocol may take place across multiple operations. We can solve this problem by representing protocols as session types, and attaching a session type to each channel end.

A common representation for a session type is a finite state machine, with each message being an edge in the state machine's graph (see figure 5.3.5). The compiler will translate each protocol definition it sees into a state machine; a session type can then be represented as a pair of a state machine and a state identifier within that machine. Given a channel's current state, this makes it possible to tell whether an operation upon it is valid, and if so what the resulting state is. The same approach is already used in Honeysuckle and in implementations of session types in other languages.

Each protocol has an *initial state* for the start of a conversation. Since occam-π protocols may be repeated arbitrarily as a whole, a message with nothing following it in the protocol specification is recorded as a transition back to the initial state.

To check a program for protocol compliance, it is first transformed into a control flow graph (as Tock already does, in order to perform other sorts of static checks). Each channel end variable is tagged with a state, starting in the initial state when a channel is first allocated. The control flow graph is traversed; when a communication operation is performed upon a channel end, the current state and the message are checked against the appropriate state machine, and the state is updated. If the communication is not

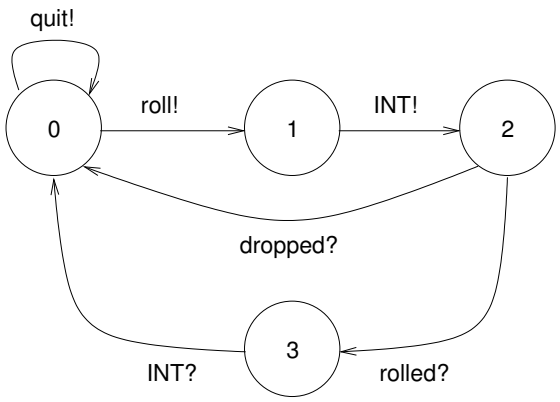


Figure 78: Finite state machine representing the `DIE` protocol

valid for the present state, the compiler can report not only that it is invalid, but also what communications *would* have been valid at that stage of the protocol.

When two flows of control rejoin, the compiler must check that the state of each channel end is the same in both flows; this ensures that conditionals and loops do not leave channels in an inconsistent state. When a channel is abbreviated—either via an explicit abbreviation, or in a procedure definition—it must be left in the same state at the end of the abbreviation that it was in at the start. (This rule may need to be adjusted to support mid-conversation handoff; see section 5.3.6.)

Similarly, when a channel end is sent between processes (for example, as part of a channel bundle), its state must be preserved by the communication. As with SJ, it would be perfectly reasonable to hand off a channel end in the middle of a conversation to another process, provided the receiving process agrees what session type it should have. This allows the process network to be dynamically reconfigured without consistency problems.

A `CLAIM` upon a shared channel must start and end with the channel in its initial state, since the channel must be left in a predictable state for its next user. Shared channels are the only case in which channel ends may be aliased in occam-π, so this restriction avoids consistency problems caused by aliasing of session-typed channels.

Note that the channel's state is not tracked at runtime, as with many other session types systems. Two-way protocols incur no runtime overheads.

5.3.6 Protocol Specifications

The syntax used so far for session types is hard to read and write, particularly for complex protocols with many choices and direction changes; something more convenient is needed for use in occam-π. An adaption of the state-based syntax used in Sing# is one possibility; this section outlines some others.

Starting Small

We must preserve the existing syntax for unidirectional protocols in order to avoid breaking existing occam-π code, but since a unidirectional protocol is just a special

```
PROTOCOL DIE
  CASE
    roll; INT
      CASE
        rolled; INT
        dropped
    quit
:
```

Figure 79: Using nesting for direction changes

```
PROTOCOL DIE
  CASE
    ! roll; INT
      CASE
        ? rolled; INT
        ? dropped
    ! quit
:
```

Figure 80: Nesting with directions explicitly labelled

case of a two-way protocol, we can deprecate the existing syntax in favour of a new one. However, there are some advantages to basing our new syntax on the existing one: the existing syntax has worked well for over twenty years, and occam-π programmers are already familiar with it.

We could keep the existing syntax for unidirectional protocols—so communications in the same direction are still sequenced using ;—but allow an indented block inside a protocol specification to mean a change of direction. We could then write our `DIE` protocol as in figure 79.

This protocol is very simple, but if it had more changes of direction (and therefore deeper nesting), it would be harder to tell the direction of each communication. We could require the user to explicitly specify the direction of each communication (figure 80).

The directions are specified from the perspective of the client. This is more useful for documentation purposes; a programmer is more likely to be writing a client to somebody else's server than a server to somebody else's client. Since all the communications within a single `CASE` must be in the same direction, it is somewhat redundant to specify the direction on all of them; it would be possible to apply the direction to the `CASE` itself instead.

This syntax does not allow the full power of session types, though. It is only possible to have choice at the start of a communication or after a change of direction, which means you cannot send some identifying information followed by a command. Furthermore, there is no way to name and reuse parts of the protocol; you cannot write a protocol containing repetition, or share a response (such as a set of error messages) between several possible commands.

```
PROTOCOL ERROR
  CASE
    ! ok
    ! file.not.found
    ! disk.full
:
PROTOCOL FILE
  CASE
    ! open; FILENAME
      ERROR
    ! write; STRING
      ERROR
:
```

Figure 81: Two-way protocol inheritance

Protocol Inheritance

How would protocol inheritance (provided in occam-π [27]) work with these simple two-way protocols? We could allow the inclusion of an existing protocol in a new one by giving the existing protocol's name (figure 81). The effect would be as if the existing protocol's specification were textually included in the new protocol.

Note that the `ERROR` protocol's direction has been implicitly reversed when it is included in `FILE`. In combination with occam-π's existing `RECURSIVE` keyword, which brings a name into scope for its own definition, this approach would allow protocols containing repetition to be defined recursively.

This approach has several downsides, though. It is only possible to recurse back to the "outside" of a protocol. Mutually recursive protocols—which may be useful when you have a protocol that switches between two or more stable states—cannot be written. It is also difficult to describe the type of a channel in mid-conversation. The session type of a `CHAN FILE` after an `open` or `write` message has been sent is the same: the expected messages are those from `ERROR`. However, it is not a `CHAN ERROR`, because after the error message has been sent the next communication will be one from `FILE`. This makes it impossible to write a reusable error-handling process.

Named Subprotocols

A more flexible approach would be to allow the user to define named subprotocols within a single protocol definition—which the compiler will eventually translate into named states within the protocol's state machine. Our `FILE` protocol with errors can now be written using a subprotocol for error reporting (figure 82).

Note that the message directions are now written consistently between the top-level protocol and its subprotocol.

We can now refer to a particular state within a protocol when describing a channel's type, which lets us write abbreviations and procedures expecting a channel in a particular state—for example, `CHAN FILE[ERROR] c?`.

One problem with this is that, by the checking rules described earlier, the use of `c` as a procedure parameter would require it to have the same type when the procedure exited—which will not be the case if it has handled the error. To solve this, we

```
PROTOCOL FILE
  SUBPROTOCOL ERROR
    CASE
      ? ok
      ? file.not.found
      ? disk.full
  :

  CASE
    ! open; FILENAME
      ERROR
    ! write; STRING
      ERROR
:
```

Figure 82: Named subprotocols

could allow the input and output states of a protocol to be specified in an abbreviation's type—for example, `CHAN FILE[ERROR, FILE] c?`. Another option is to just specify `CHAN FILE` and have the compiler infer the input and output states.

The top-level protocol's name is still made available if `RECURSIVE` is used. This could be generalised to permit mutual recursion between subprotocols, which is rather unusual for occam-π; its scoping rules usually forbid mutual recursion. Mutually-recursive subprotocols would allow us to specify a protocol with multiple "stable states": for example, a network socket that may be either connected or disconnected, and supports different sorts of requests in different states.

5.4 Clocks

This section proposes *clocks* as a language feature for occam-π.

5.4.1 Motivation

The ▷ **Clock** pattern is found in many process-oriented simulations: a clock allows several processes to share an idea of virtual time. The virtual timesteps provided by a clock are often subdivided into ▷ **Phases** (using the same barrier) to control access to shared resources. We can combine the two by counting phases as "smaller" timesteps. Both of these patterns assign higher-level semantics to barrier synchronisation; abstracting these into library or language features would considerably ease the implementation of the patterns.

While these patterns both achieve the desired goal of maximal concurrency within the synchronisation constraints of the problem, they are both usually implemented in an inefficient way. In a simulation, it is rare that every process using a clock has an action to perform on every phase of every timestep—yet when the patterns are implemented using barriers, every process enrolled on the barrier must wake up each time the barrier synchronises, even if it has no work to do. (For example, in the ⋆ **CoSMoS** lymphocyte rolling simulation, the lymphocyte behaviour processes only perform an action in one of the three phases of the barrier; they therefore do three times as many

```
CLOCK INT c:                -- time type
PAR i = 0 FOR num.agents    -- process enrolled on clock
  INITIAL INT time IS 0:
  WHILE TRUE
    SEQ
      SYNC c, (time * 10) + 0
      -- Phase 0 actions

      SYNC c, (time * 10) + 1
      -- Phase 1 actions

      -- etc.

      time := time + 1
```

Figure 83: Example of clock synchronisation

synchronisations than they need to.) This can be mitigated somewhat by using multiple barriers for different phases or granularities of time, but this is awkward to program.

5.4.2 Using Clocks

Clocks with efficient synchronisation could be provided as a synchronisation object in process-oriented environments. A clock, as in the design pattern, is a barrier with a sense of time: each time it completes a synchronisation, the time advances. When a process offers to synchronise upon the clock, its offer includes the time at which it wishes to be woken up. Once all the processes enrolled upon a clock are attempting to synchronise, the clock blocks those processes and completes: it examines its set of offers to find the next timestep that needs to run, and releases only the processes that want to be woken during that timestep, leaving the others blocked.

An example of how clocks might look in occam-π is shown in figure 83. The syntax and semantics of clock references are the same as those of barriers, with the exception that synchronisation takes an extra argument for the time offer. Dealing with clock references is similar to dealing with barrier references—for example, they can be generated by a factory or synchronised between hosts in a distributed application in much the same way.

Choice over clocks also makes sense: just like a barrier, a clock guard becomes ready when all processes enrolled upon the clock are offering to synchronise. Once the guard is chosen, however, the process is blocked until the time it requests comes around. A process cannot speculatively wait for a later time and then back off before that time is reached, because changing its time offer retrospectively could result in it having to run before a process that made an offer between the old and new times.

For example, suppose we have a clock with two processes enrolled upon it. The first process offers to synchronise at time 2. The second process offers to synchronise at time 3—so the clock completes, and the first process runs. It is now not safe for the second process to withdraw its original offer, because if it offered time 1 then it would not be possible to run it.

5.4.3 Time Representations

In occam-π, it is traditional to represent real time as an `INT`—but there is no particular reason to constrain clocks to integers.

While in theory time could be represented by a "bignum" data type that could grow to an infinite value, providing a *monotonic clock* where time only ever increased, it is often more practical for clocks to use a fixed-size data type (for example, wrapping at 2^{32}) or to be forced to wrap at a lower value—a *periodic clock*. A program that only used phased synchronisation could use a periodic clock with the period set to the number of phases.

The only requirement for the type used by a clock to represent time is that it must be possible for the clock to identify the *next* time to run from its set of offers. For a monotonic clock, the next time can be identified simply by choosing the lowest time offered. (In this case, a process offering a time in the past would run before other processes—but this could reasonably be detected as a runtime error.) For a periodic clock, the clock needs to keep track of the last time that ran, and then select the offered time that is "least ahead" of the last time (i.e. minimise $(\text{offer} - \text{lastTime}) \bmod \text{period}$). A periodic clock must thus work out an initial "last time" when it first synchronises—which it can do by taking the lowest of its initial offers.

A wide variety of time representations are therefore possible. For many applications, an integer type will suffice. In a simulation with both timesteps and phases, it might be convenient to have time be a pair $(\text{timestep}, \text{phase})$. Using a Haskell-style abstract data type to represent the phase would allow phases to be named, which would ease readability (for example, a time might be $(42, \text{Move})$).

Some problems must proceed in phases, but with "subcycles" of phases being repeated an unknown number of times. For example, an implementation of an evolutionary algorithm could use a clock to represent the generations of evolution, but inside each generation a simulation would need to repeatedly evaluate the candidate solutions until the fitness function returns a stable value. One way to achieve this would be to make the time representation a rational number, and number the evaluation steps inside each generation as a geometric series converging on the next generation (for example, proceeding $1, 1\frac{1}{2}, 1\frac{3}{4}, 2\ldots$).

A clock where the time representation is the unit type, or where all enrolled processes always synchronise with the same time, is equivalent to a barrier. Verifying this mechanically would be a useful check upon algorithms for implementing clocks.

5.4.4 Implementing Clocks

To implement clocks in a process-oriented runtime system, it is instructive to look at how event-based simulations represent time. A common approach is to use a priority queue of sets of events, ordered by time. The simulation then only needs to pull the next set off the queue in order to find both the time that it should run and the set of events that need processing.

We can combine this approach with the usual implementation of barriers to provide clocks in a process-oriented system. A trivial barrier has an enrolment count and a set of blocked processes; when the size of the set reaches the enrolment count, all the processes in the set are scheduled and the set is emptied. A trivial clock would have an enrolment count and a priority queue of sets of blocked processes; when the sum of

```
PROC clock.server ([]CHAN INT reqs?, []CHAN SIGNAL resps!)
  VAL INT N IS SIZE reqs:
  [N]INT wake.at:
  INITIAL INT time IS 0:
  SEQ
    -- get initial requests from all clients
    PAR i = 0 FOR N
      reqs[i] ? wake.at[i]

    WHILE TRUE
      INT min.diff, next.time:
      SEQ
        -- find the next time being waited for
        min.diff := MOSTPOS INT
        SEQ i = 0 FOR N
          VAL INT diff IS wake.at[i] MINUS time:
          IF diff < min.diff
            min.diff, next.time := diff, wake.at[i]
        time := next.time

        -- run clients waiting for that time
        PAR i = 0 FOR N
          IF wake.at[i] = next.time
            SEQ
              resps[i] ! SIGNAL
              reqs[i] ? wake.at[i]
:

PROC clock.sync (VAL INT time,
                 CHAN INT req!, CHAN SIGNAL resp?)
  SEQ
    req ! time
    resp ? SIGNAL
:
```

Figure 84: A simple clock implementation

the size of all the sets in the queue reaches the enrolment count, the first set is popped off the queue and all the processes in it scheduled.

In practice we would not implement clocks this way in CCSP, just as we do not implement barriers using sets [184]. A practical CCSP clock might store blocked processes in a balanced tree of lists of batches, with a batch list for each time offer.

A prototype implementation of clocks is already available in the CHP library for Haskell [49]. Alternatively, a clock can be simulated using a server process (figure 84).

5.4.5 Static Checking for Clocks

Static checking features could be used to detect errors in clock usage at compile time.

In particular, it would be useful to detect when a process is using the phase sequence represented by a clock in an incorrect way; a process that accidentally synchronised in the order A, C, B, D rather than A, B, C, D would advance two timesteps for

```
CLOCK PHASES c:
PHASED(c) READ(LOOK) WRITE(UPDATE) INFO info:
SEQ
  ...
  CLAIM info, c   -- c must be in LOOK phase
    local := info
  ...
  CLAIM info, c   -- c must be in UPDATE phase
    info := local
  ...
```

Figure 85: Phase-protected resources

each cycle rather than one. (This kind of error might be introduced when a new phase was added to a clock in existing code.) To detect this, the compiler would need to be able to tell that the programmer only intended the process to advance by one timestep.

This suggests that providing a richer structure for clock times than just arbitrary types may be useful—just as occam-π provides protocols for channel communication. A *clock protocol* could explicitly specify features such as timestep counts, phases and repetition of subcycles represented inside a clock. The compiler could then attach session types to clock variables (section 5.3.5) to track the possible offers that could be made using each clock, and warn about unsafe advances.

Another use for session types in combination with clocks would be to allow resources to be protected by clock-implemented phases. A shared resource could be declared as "phased" to a particular clock, with an access list indicating what types of access were available in which phase. The compiler could then enforce CREW rules [251] to ensure that processes can safely read from the resource in parallel but only write to it one at a time, ensuring that the clock presented as a credential when it is used is in the correct phase (figure 85).

An alternative to the use of session types would be to provide a `PHASE` block that forced the structure of the code to mirror that of the phase structure—but session types allow the same degree of static checking with more succinct syntax.

Chapter 6

Future Work

This chapter discusses some possible approaches to the future development of process-oriented programming.

With the rise of parallel hardware and the increasing complexity of computer systems, concurrent programming certainly does appear to *have* a future. The interest in the Go programming language and the widespread reinvention of process-oriented synchronisation features in other concurrent programming environments both indicate that process-orientation can play a part in that future. To make this a reality, process-oriented programming must become a practical approach to software engineering that can be used in real-world applications; the following sections will discuss some of the steps that must be taken towards that goal.

6.1 A Standard Interface

As we saw in section 2.3, there are currently a wide variety of concurrent programming environments that offer some degree of support for process-oriented programming. There is, however, no single environment that offers a *complete* best-of-breed set of process-oriented programming features, and the features provided by different environments often differ in subtle ways. Furthermore, there is no standardisation in the vocabulary used to describe process-oriented facilities. This makes it difficult—as earlier chapters have shown—to write about process-oriented programming without including a good deal of implementation-specific detail.

Having a definition of a core set of process-oriented programming features, described by a common vocabulary and provided through a standard interface (moderated by the requirements of the particular environment), would make promoting, teaching and applying process-oriented techniques much more straightforward—in the same way that the standard vocabulary and interface of the Document Object Model have made it easier to work with XML in different languages, and to define other systems that make use of XML [237].

The definition of such an interface is a job for a standards committee—but the following features from existing process-oriented environments should certainly be considered for inclusion:

- lightweight processes;
- blocking system calls;

- parallel composition;
- forking, with named forking contexts;
- channels, with messages specified using bidirectional protocols, featuring optional buffering, and optional explicitly- and implicitly-shared ends;
- barriers;
- real-time timers;
- clocks;
- choice over arbitrary synchronisation objects, including prioritised choice and fair choice, and perhaps conjunctive choice;
- extended synchronisation;
- poison, with automatic poisoning of "linked" synchronisation objects (following the Erlang model), and the spread of poison limited by process groups or poison filtering;
- single-owner references to data and synchronisation object ends;
- a library of common processes (from section 4.1);
- higher-order process network constructors;
- higher-level features for client-server interactions, including automatic termination of unused servers.

Several of the more popular process-oriented environments now support the construction of distributed applications that extend synchronisation objects across multiple hosts—for example, through pony for occam-π and the `jcsp.net` module for JCSP. Providing a standardised, extensible network protocol for communication between different environments would allow the construction of process-oriented systems spanning different languages and types of hardware [61].

It would also be useful to standardise a lower-level concurrency interface—not channels and barriers, but the primitives from which such synchronisation objects are built. This would allow the prototyping and implementation of new types of synchronisation object without concern for portability problems. The interface would include portable atomic and memory-barrier operations (as the `java.util.concurrent` package provides), facilities for explicitly scheduling and descheduling lightweight processes, and a standard interface for choice. (CHP already permits this by using Haskell's STM implementation as its low-level interface—but STM operations are arguably too high-level an abstraction for implementing high-performance synchronisation objects in general.)

In addition, providing a standard low-level interface would allow experimentation with different implementations of the primitives to be provided—much as different MPI implementations can take advantage of different hardware communication facilities, or different Java virtual machines can experiment with new approaches to memory management. Reference implementations could be constructed from the existing CCSP and Transterpreter runtimes.

6.2 Process-Oriented Languages

One approach to the continued development of process-oriented programming is to further develop programming languages that are specifically designed to support process-oriented software development, such as occam-π, Go and XC.

Having precise control over the semantics and implementation of a programming language offers several advantages for process-oriented development. Static checking is far easier if dangerous and hard-to-analyse language features can be omitted or restricted. Interfacing with the runtime system becomes more predictable and incurs lower overheads, allowing advanced features such as mobile processes to be implemented efficiently. Designing a language from scratch allows the entire environment—not just the language, but the standard library, development tools, and so on—to be constructed in a way consistent with the process-oriented programming model.

However, maintaining a specialised programming language (of any kind) has significant disadvantages. Designing a *good* programming language that people will want to use is a very difficult problem—the field of computer science research is littered with programming languages that have not gained widespread acceptance. Developing a language specification, compiler and standard library is a lot of work, requiring a dedicated team of programmers and authors willing to engage in what is largely uninteresting donkey-work with little research value. Developing documentation and teaching material for new users is similarly time-consuming. A new language will have difficulty taking advantage of existing tools and libraries; "glue" must be developed, and ways found to map existing conventions into new ones. Worst of all, however, is the difficulty of persuading people to use a new language: programmers must invest significant time and effort in becoming proficient with the syntax, semantics, idioms and libraries of the programming languages they use, and they will not do so unless the payoff from learning a new language is considerable.

Process-oriented programming languages may have advantages in specific applications—in particular, for embedded programming with constrained resources and real-time requirements, where programs are relatively small and language features can be limited. Specialised programming languages have been successful in other programming styles, where development impetus has been available—for example, Haskell was designed to replace a large family of existing lazily-evaluated functional languages [110], and Erlang to support concurrent programming for Ericsson's telephony applications [15]. While alternative approaches are available—see section 6.3—the author believes that it is at least worth considering the prospect of further development of process-oriented languages.

6.2.1 occam 4

The occam family of languages has not produced a new general-purpose programming language in several years. occam-π is explicitly maintained as a research language, constantly changing, lacking a formal language definition or a really solid implementation, and thus unsuitable for developing real-world applications. However, occam-π's basis was in occam 2.1, a stable, well-defined language that was—and, as the core of occam-π, still is—used successfully in a wide variety of applications. We could, therefore, consider developing an "occam 4": a new language in the occam tradition with a stable definition, drawing on the features from occam-π that have been found to be

successful, while correcting the shortcomings of the existing language.

The applications in which occam-π has been used (many of which have been described in chapters 3 and 4) should be supported at least as well by a new language. occam-π has also been used successfully at several institutions to teach concurrent programming in the process-oriented style—which implies that a new language should carefully consider how design choices would affect new programmers. Haskell and Python are both examples of successful languages that have been designed to support both practical applications and teaching; this is, by now, a reasonably well-understood problem.

The development of occam-π has, for the last few years, been documented in the form of *occam enhancement proposals*, which collect both proposed and implemented language features [194]. OEPs describe incremental changes to the existing language that generally avoid breaking backwards compatibility with existing programs, and cover everything from trivial syntactic changes (making `OF` optional in `CHAN OF`) to wide-ranging proposals (such as support for exceptions). As of early 2010 there were some seventy OEPs representing possible changes to the occam 2.1 language, a little over thirty of which had been implemented.

However, greater opportunities for improvements to the language are possible if backwards compatibility does not need to be maintained. (In fact, there is relatively little occam-π code in the real world—certainly far less than Python or Perl, both of which languages are currently undergoing non-backwards-compatible revisions.) The author recently conducted a survey of occam 2 and occam-π users to collect suggestions for changes that could be made to occam-π to make it a more generally-useful programming language, assuming that compatibility with existing code was not a concern. Several responses were received, many of which made similar suggestions.

When asked to identify the general strengths of the existing language that should be maintained in a new design, those surveyed suggested:

- support for the process-oriented model in general;
- the provision of usage checking and language features that support safe programming;
- efficient implementations upon embedded, desktop and distributed platforms;
- clear semantics, with direct correspondence to CSP and pi-calculus models, and nothing "hidden" from the programmer.

Suggestions for improvements included:

- most features found in other process-oriented environments, such as buffered channels, implicitly-claimed channel ends, poison, extended outputs, and choice over all synchronisations rather than just inputs;
- replication over arbitrary sequences (`SEQ i IN list`), with the replicator variable being a writable abbreviation;
- type inference;
- Haskell-style type classes;

- exception handling;
- bidirectional protocols, with protocol delegation;
- templating, allowing processes and protocols to be parameterised by type;
- a proper module system, with support for hiding private definitions;
- simpler scoping rules, since occam's ":" convention is very unusual by the standards of modern languages and frequently confuses new users;
- general recursion and tail call elimination, allowing processes to be written in a more CSP-like style;
- extended barrier synchronisations;
- first-class `PROC`s, allowing higher-order programming;
- more flexible timers, supporting different resolutions and virtual times;
- enhancements to the abbreviation system, with claiming and enrolling being explicitly performed as abbreviations;
- making all data mobile by default, with static analysis used to optimise away movement where not necessary [46];
- clearer semantics for mobile barriers;
- better support for low-level hardware and memory access, including interrupts;
- allowing arbitrary blocks of code to be forked, with names optionally given to forking blocks;
- more powerful, arbitrarily-nested basic data types, such as dynamically-sized arrays, algebraic types and union types;
- a wide variety of minor changes to make the syntax more concise, with a closer resemblance to more recent indentation-structured languages (for example, using C-style string escape sequences and lowercase keywords);
- changing the name of the language, since "occam" is strongly associated with INMOS and the Transputer in the minds of many potential users.

While some of these suggestions—such as automatic mobility—require further research, most represent clear and uncontroversial improvements to the language, making occam-π into a friendlier, more widely-applicable programming language while maintaining its general character.

6.2.2 Go

Many of the features proposed for occam above are already implemented in the Go programming language. Go offers only a very basic set of process-oriented features—channels, forking, choice—but is otherwise a relatively rich programming language. An alternative to adding conventional programming features to occam is therefore to add the missing process-oriented features to Go.

From the point of view of a process-oriented programmer, the most obvious missing features in the Go language definition are:

- parallel composition;
- forking contexts;
- protocols;
- prioritised selection;
- some mechanism for shutting down process networks cleanly (such as poison, or an extension of the existing Go error-handling mechanism).

All but the last of these would be trivial additions to the Go language. Parallel composition and forking contexts would be particularly useful, and could be implemented very straightforwardly using trivial barriers (which do not even need to support choice).

In addition, the current prototype implementations of Go are rather primitive compared to other process-oriented environments. The Go runtime emphasises simplicity over efficiently, using a "big lock" approach, and thus scales poorly on multiprocessor systems. Rehosting Go on top of CCSP—with CCSP's primitives extended to support Go features such as implicitly-shared channels—would allow significantly better performance for Go applications. In addition, the Go compilers provide very limited static checking; again, occam-π approaches could be easily adapted.

6.3 Embedding Process-Orientation

An alternative to developing a new language specifically for process-orientation is to *embed* process-oriented programming within another language. This has stronger implications than simply providing a library of process-oriented facilities, as many existing process-oriented environments do: it means making use of the process-oriented style both convenient and safe, *as if* the facilities had been directly integrated into the language.

6.3.1 EDSLs

One approach to this is to use a programming language that is sufficiently expressive to allow the syntax and semantics of the concurrency features to be defined in the language itself. When this approach is used, the concurrency features form an *embedded domain-specific language* within the host language [109]. Languages that support this tend to have very simple native syntax, and are built from simple primitives, so that most of the conventional programming facilities are already described in

terms of the language itself. Older examples are either untyped or dynamically-typed; Forth and Smalltalk are examples. Newer examples have extremely powerful—and frequently very complicated—type systems that blur the distinction between value-level and type-level programming, allowing the specification of different models of computation through types.

Haskell [222] and Scala [104] exemplify this style of programming, both having been widely used for EDSL implementation. As described in section 2.3, both programming languages have built-in support for lightweight concurrency, and both already have reasonably mature EDSLs for process-oriented programming—CHP for Haskell, and CSO for Scala. These EDSLs use the host language's type system to provide some of the static checking facilities that compilers for dedicated process-oriented languages would implement. Improved facilities for type-level programming—such as dependent types—are under active development within the functional programming world; the static checking facilities available to process-oriented EDSLs within functional languages will only improve given time.

The monadic model of computation used by Haskell is especially interesting, in that it allows the programmer to distinguish between three types of code in their program. A typical Haskell program will combine *purely-functional* code—with no side-effects—with code written in the `IO` monad, which can interact with the real world. A CHP program adds a `CHP` monad for concurrent code, within which `IO` actions and purely-functional code can be embedded—but a CHP process knows that it cannot be rescheduled while executing such embedded code. (The same distinction appears in generator-based Python approaches to concurrency, where calls that may reschedule must be `yield`ed, and in C libraries that must explicitly pass a concurrency context around. The distinction between purely-functional vs. purely-imperative vs. fully-concurrent code could be made explicit in other languages.)

6.3.2 Extending Languages

Rather than using an EDSL, we could instead start from an existing conventional programming language, and *extend* it with facilities that allow our programming style to be embedded in it. This approach involves more work than using an EDSL, but gives us a much wider choice of languages to start from: we can pick a language that is already in use across a range of fields, with existing high-quality open-source implementations, and with a large userbase who are already familiar with it.

For example, we could start with C++. The C++CSP library already provides process-oriented programming facilities with lightweight processes for C++, with a convenient STL-style interface [45]—but it does so by using stack-switching tricks that are both memory-inefficient and highly unportable, and it provides no static checking.

LLVM, a portable framework for building compilers, may provide a solution to both of these problems [126]. Clang, LLVM's frontend for C and C++, is designed to support static analysis of programs; we could modify C++CSP to provide hints to Clang about the processes and references present in the program, and then write static analysis tools using LLVM's facilities that not only perform the usual occam-π-style static checks, but also calculate the exact stack requirements of each process. In this case, the only modification necessary to the C++ language would be the addition of some LLVM annotations that would never be visible to application programmers.

Given this analysis, C++CSP could then be built on top of an existing process-oriented scheduler such as that from CCSP, with a new LLVM intrinsic provided to switch stacks between lightweight threads in a portable way; this would allow lightweight concurrency on any platform supported by LLVM. Some work has already been done on this to support compiling occam-π using LLVM, with CCSP as a scheduler [183].

The result would be a system that offered the safety and most of the convenience of a dedicated process-oriented language, with the portability, familiarity, low maintenance requirements and easy access to existing libraries of a mainstream programming language. Compared to the approach taken by Handel-C and XC—where a process-oriented language was extensively modified to achieve the appearance and behaviour of a C-style language, for a target audience familiar with C—this approach requires less work and will achieve better performance.

Other approaches to concurrency—notably Apple's Grand Central Dispatch [12], and Intel's Threading Building Blocks [113]—have similarly been implemented as libraries on top of C++, with minor extensions to the language to support better integration with the compiler and runtime system. C++0x provides some extensions—anonymous functions, for example—that make embedded languages such as these more convenient.

C++ is not the only option here; other possibilities include Java and C#, both of which already run on virtual machines that interact with operating system threading facilities. In both cases, the virtual machine could be modified to use an existing lightweight threading system instead, allowing languages running on top of the VM to take advantage of lightweight threads. (The Python virtual machine has already been modified to run on top of CCSP with some success.)

6.4 Revisiting Object-Orientation

Object-oriented programming is currently the most widely-used approach to software design; it is sufficiently ubiquitous that it is usually the first approach taught to new programmers, and—in many cases—the only one they will encounter in the real world. Earlier chapters have drawn strong parallels between patterns in object-oriented and process-oriented software engineering. To understand the reasons for this, we must examine the history of object-oriented programming.

Object-oriented programming began with Simula, a language developed to support event-based simulation. Simula is a concurrent language, providing coroutines as a primitive and processes and queues as library features [218]. Entities in a simulation are modelled by processes, which can be suspended and later reactivated, and which can synchronise in simulated time using a clock-like mechanism. Simula objects have methods that can be invoked by other objects—but they also have a flow of control, and can be either active or passive as required.

Object-orientation was really popularised by Alan Kay's group at Xerox, who, inspired by Simula, had developed FLEX—a personal computer system where "persistent objects like files and documents were treated as suspended processes" [124]. During the development of FLEX, Kay "realized that the bridge to an object-based system could be in terms of each object as a syntax directed interpreter of messages sent to it.

...The mental image was one of separate computers sending requests to other computers that had to be accepted and understood by the receivers before anything could happen."

The group went on to develop Smalltalk: an object-oriented language designed to be simple enough for children to use, in which "everything is an object", "objects communicate by sending and receiving messages" and "objects have their own memory". The idea of Smalltalk objects as active entities with their own motivations was paramount: "Since control is passed to the class before any of the rest of the message is considered—the class can decide not to receive at its discretion—complete protection is retained. Smalltalk-72 objects are "shiny" and impervious to attack."

The original Smalltalk idea of an object is therefore far closer to the process-oriented idea of a server process than the passive C++-style objects now common in object-oriented languages; "active objects" are a reinvention of the original ideas about how objects should behave. It should therefore be no surprise that patterns that work for objects can also work for processes—and vice versa. We can consider how some of the features of process-oriented environments may be applied to object-oriented environments.

Making objects into active server processes with a flow of control has distinct advantages, as seen in the ▷ **Client-Server** pattern: an object responds to messages it receives by executing methods. Objects could continue to do work in between accepting requests, giving greater opportunities for parallelism, and objects would have control over when they chose to accept requests. (Method execution need not be entirely serialised, as in the ○ **Monitor Object** pattern [206]—an object could respond to a request by forking a worker process to execute a method.) Treating method calls as really being message-passing between objects means that self-messaging needs to be rethought—but this need only be a conceptual change, since one method can call another without a message needing to be sent. Inheritance can be implemented either by delegation (late binding) or by importing methods from the namespace of another class (early binding).

Single-owner references should be as useful in an object-oriented system as they are in a process-oriented system: in a concurrent system of objects, tracking ownership of resources and preventing unsafe aliasing is equally important [105]. In systems that use garbage collection, single-owner references may provide performance advantages: when a reference is destroyed (rather than passed to another object), the corresponding object can immediately be considered dead, and discarded or recycled.

Interface abstraction—separating communication links from processes—is a powerful tool under certain circumstances (see ▷ **Hand-Off** and ▷ **Just In Time**). In most object-oriented systems, however, an interface to an object is simply a reference to an object; you cannot create the interface separately from the object itself, making it awkward to construct systems of objects containing circular references. An object-oriented language could offer interface abstraction as an optional facility, creating a lightweight proxy for an object that would be created later.

Method calls in object-oriented systems only permit very simple forms of interaction between objects—in process-oriented terms, just simple request-response pairs. More expressive interactions—such as the three-step exchange in the ▷ **Loan** pattern—can only take place using multiple method calls, in which case the type system cannot enforce the correct order of interactions. These kind of coroutine-like interactions could

be specified using an object-oriented equivalent of stateful bidirectional protocols (really just sequences of requests and responses): "co-operations"?

As servers are so common in process-oriented systems, and so close in conceptual scope to objects, it would make sense to provide syntax for defining and using servers in process-oriented environments that mirrors the syntax for defining classes and calling methods upon objects in object-oriented languages. Given this, it would be possible for future programming languages to provide a seamless integration of process-oriented and object-oriented features—concurrency for all!

Chapter 7

Conclusions

This section demonstrates how the pattern language described in chapter 4 can be used to design, discuss, and reason about a complete, if small, piece of real-world embedded software—a scrolling LED matrix display controlled by an Arduino—and reflects upon the outcomes of this work.

7.1 A Worked Example

The Arduino is just about the smallest machine that occam-π is currently useful on, with only two kilobytes of RAM and an 8-bit AVR CPU; nonetheless, it can run a couple of dozen concurrent processes using the Transterpreter runtime. occam-π development on the Arduino is supported by the Plumbing environment (section 3.10).

A light-emitting diode consists of a semiconductor element that emits light when a current is passed through it. LED matrix displays are widely used for changeable signs and advertising applications; they consist of a tightly-spaced grid of LEDs, each of which can be individually controlled, allowing messages and images to be displayed. Modern matrix displays are built from LED matrix modules, which provide standard-sized rectangular sections of matrix that can be slotted together; 8×8 single-colour modules are cheap and readily available.

The simplest way to control such a display would be to connect each LED to an output pin on a microcontroller. However, this would require an impractically huge number of output pins—and an equally complex circuit board layout—to drive even a small display. Instead, an LED matrix is constructed as a grid of horizontal and vertical conductors, with an LED connected between the conductors at each intersection. To turn on an individual LED, a voltage must be applied across the corresponding horizontal and vertical conductors. However, this makes it impossible to display arbitrary patterns on the display: you cannot, for example, turn on just two LEDs that are diagonally adjacent to each other, because the voltages applied would result in other LEDs adjacent to those also turning on.

Matrix displays get around this problem by taking advantage of persistence of vision. A picture is displayed on a matrix by lighting up only one column of the matrix at a time, cycling between columns at a rate sufficiently high that the human eye cannot perceive the flickering of the image. To compensate for each column only being turned on for a fraction of the time available, the LEDs are normally lit to a much higher brightness than would be used for a static display.

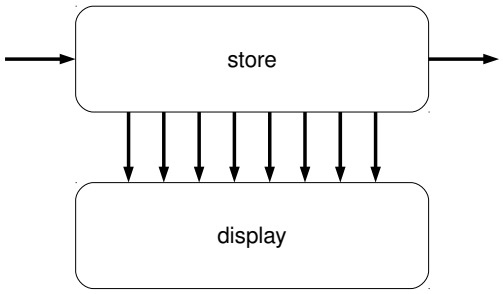


Figure 86: Outline process network for the matrix display

```
PROC store.column (CHAN COLUMN in?, out!, to.display!)
  CHAN COLUMN thru:
  PAR
    delta (in?, [to.display!, thru!])
    holding.buffer (thru?, out!)
:
```

Figure 87: Matrix column store process

Our objective is use an Arduino to control an LED matrix display, displaying messages that scroll smoothly across the matrix. The Arduino has 18 input-output pins, which is sufficient to drive an 8×8 matrix module, and the AVR microcontroller has just enough output current drive capability to allow the module to be directly connected to it.

The program running on the Arduino has two major responsibilities: it it must store the message being displayed, scrolling it along the display a column at a time, and at the same time it must light up the columns of the display one by one as rapidly as possible. These clearly need to interact, but they are separate concerns with different timing requirements, and can therefore be most easily represented as separate processes, `store` and `display`, connected by channels through which each new set of columns can be delivered to the display when necessary (figure 86). The `COLUMN` data type is simply an 8-bit byte, with a 1 bit indicating that the corresponding LED should be turned on.

The `store` process is a kind of ▷ **Buffer**, holding eight column values. We can use the ▷ **Pipeline** pattern to construct an eight-place buffer from eight single-place buffers; `store` is therefore a pipeline of eight `store.column` processes.

When a `store.column` process receives a new column value, it must do two things: pass it on to the `display` process to show, and send the previous column value that it received on to the next `store.column`, so the columns scroll along the display. Again, these are two separate responsibilities, and can be implemented as processes—both of which are standard process patterns: distributing a value to multiple output channels can be done using a ▷ **Delta**, and holding on to a value until a new one is received is the job of a one-holding ▷ **Buffer**. This means that `store.column` is simply a network of standard processes (figure 87).

```
PROC display.column (CHAN SIGNAL in?, out!,
                     CHAN COLUMN from.store?,
                     VAL BOOL first)
  SEQ
    IF first
      out ! SIGNAL

    INITIAL COLUMN v IS 0:
    WHILE TRUE
      ALT
        from.store ? v
          SKIP

        in ? SIGNAL
          SEQ
            ...  light up column briefly
            out ! SIGNAL
:
```

Figure 88: Matrix column display process

The `display` process can similarly be constructed from eight simpler processes—but now these processes must take turns to light up their column of the display. We can implement this sharing using a ▷ **Ring**, with a token that permits the process holding it to use the display. `display` is a ring of eight `display.column` processes. Each `display.column` process makes a choice between receiving a new column to display, and receiving the token; when it receives the token, it lights up its own column briefly, then passes the token on (figure 88). The only special arrangement that must be made is for one of the processes in the ring (it does not matter which) to introduce the token. Note that there is no absolute synchronisation to real time here: the token can circulate at whatever speed the rest of the system permits.

Most of our process network is in place. We need a ▷ **Producer** to generate the columns that will appear on the display at a fixed rate. (This is simple enough to write as a single process, as we did in practice—but it could be decomposed further for greater flexibility if necessary: for example, it could be a short pipeline, with a generator that emitted ASCII characters, followed by a ▷ **Filter** to convert characters into lists of columns, and finally a ▷ **Valve** to allow the speed to be controlled.) We need one final process to make the system work: the unused output channel from the last `store.column` process must be connected to a ▷ **Black Hole**, to discard columns that "fall off the side of the display".

The complete process network is shown in figure 89. Is this system safe? First, we must look at the ring: it has one token and eight places of buffering, so it will circulate without deadlock, and the code executed when the token is received is guaranteed to complete in a finite time, so it will not block the `display.column` processes' other communications. The rest of the system can now be considered using the ▷ **Client-Server** design rules: each channel communication is a simple request, and all the processes follow the rules for clients and servers. So yes—we can be certain that this system will not deadlock.

While this system works well, it is only driving a very small display, wide enough

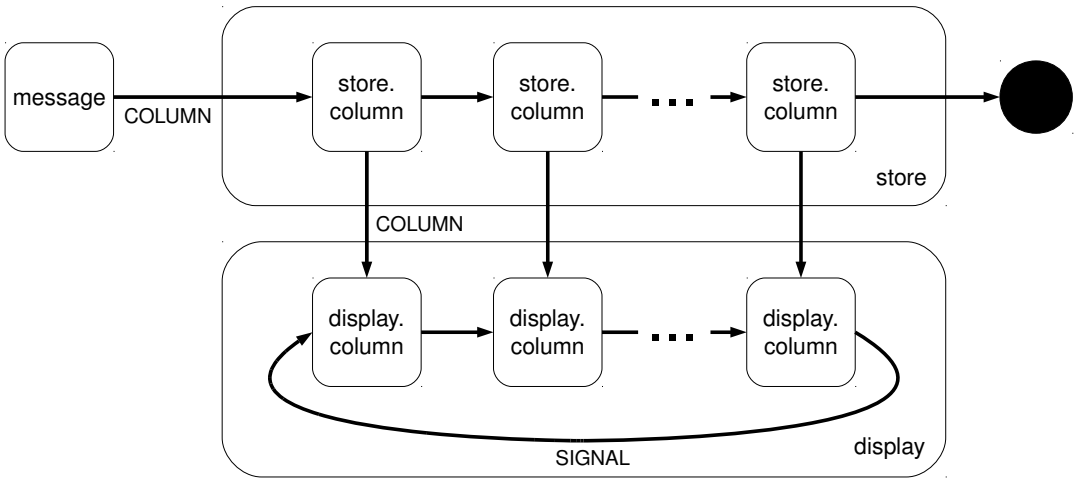


Figure 89: Complete process network for the matrix display

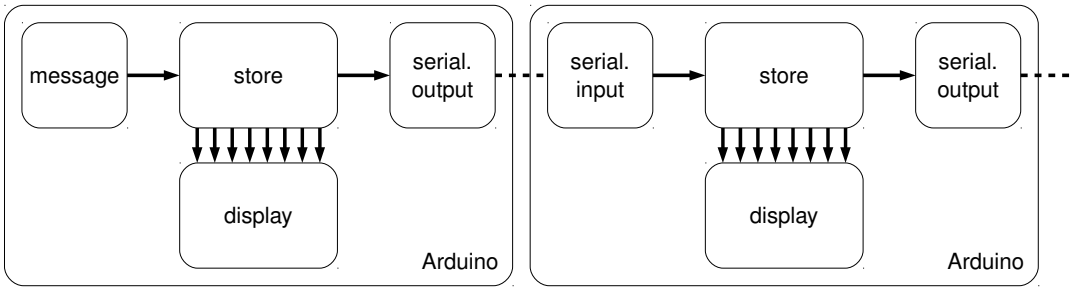


Figure 90: Distributed matrix display

for only one or two characters—not yet useful for practical applications. To support a larger display, we need some more hardware. We could connect the rows of several display modules together, and use an external decoder chip to drive more columns (since we know that only one column need be turned on at once), or we could upgrade to a more expensive AVR processor with more I/O pins. In either case, we can simply expand our existing process network: the same processes can be used and the same safety guarantees hold.

But if we want to stick with the standard Arduino hardware, we must take a different approach. Recall that we have used 16 of the Arduino's 18 pins. The AVR chip supports serial communication on two of the pins. These are normally used to communicate with a host computer when developing software—but we can use them instead to communicate between different Arduinos. If we chain the serial output of one Arduino to the serial input of another, we can construct a pipeline of Arduino boards, each of which drives one matrix display.

Taking advantage of this requires only a very simple modification to the process

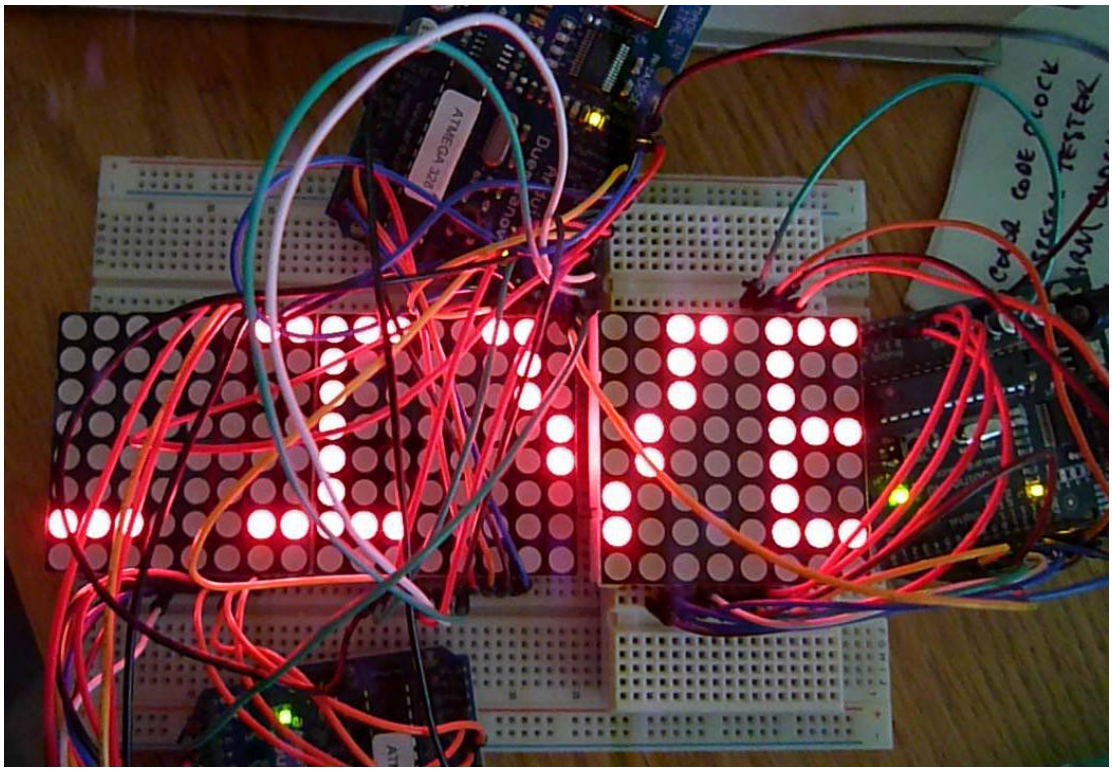


Figure 91: Three running matrix displays (photo by Christian Jacobsen)

network. We remove the black hole process, and replace it with a serial output process (a standard Plumbing component) that writes each column's value to the serial port as it is received. On every Arduino but the first in the pipeline, we remove the message generator and replace it with a serial input process that receives column values from the serial port and outputs them to a channel. As a result, each column that scrolls off the end of the first Arduino's display will appear at the start of the second one—we have distributed our pipeline across multiple Arduino boards, with a separate ring for each display (figures 90 and 91). This is an example of the ○ **Proxy** pattern, used to extend a channel interface over a "network" link.

(The complete source code for this program is included as the `ledmatrix` example in the Plumbing distribution.)

7.2 Reflections upon Confidence

The pattern descriptions in "A Pattern Language" (section 2.4.1) included an indication of the authors' confidence in their patterns: did they think they had found the best solution to a problem, or merely *a* solution? We can consider the patterns described here in the same way.

Some patterns clearly deserve a high level of confidence. For example, ▷ **Pipeline** and ▷ **Client-Server** are not only widely used in process-oriented programming, but have been discussed in the literature of other programming styles as well. Other patterns are more specifically tied to the programming style, such as ▷ **Acknowledgement** and ▷ **Private Line**, but similarly have been successfully applied in a wide range of applications. A few patterns are more recent inventions, and been used only a handful of times so far—▷ **Clock**, ▷ **Loan** and ▷ **Messenger** being examples—but are so clearly the correct (and non-obvious) approach to use in the existing applications that their future utility seems assured.

Some patterns—notably ▷ **Just In Time** and ▷ **Snap-Back**—are not so well-tested. They have been successfully applied several times, but often within a closely-related set of applications; perhaps they will not prove useful elsewhere, in which case they should not remain in a catalogue of *general* patterns, or better patterns will be found to replace them. However, in these less certain cases, it seems better to document the patterns as they currently exist than to omit them entirely. (Predicting the future is difficult—it is always possible that some future process-oriented designer will invent an improved replacement for the client-server pattern.)

In "Design Patterns" (section 2.4.2), we saw that "patterns are not about designs . . . that can be encoded in classes and reused as is" [87]. As process-oriented environments become more capable, with more expressive type systems and more powerful static checking features, it will become possible to provide more of these patterns as reusable code libraries—and where a distinction cannot be drawn between the abstract idea of a pattern and its concrete implementation any more, the pattern will arguably no longer be valid. Patterns such as ▷ **Black Hole** and ▷ **Clock** seem likely to disappear in this way (whereas, for example, the general pattern ▷ **Consumer** that black holes are a specialisation of will remain). This is not an indication of weakness in the patterns themselves; merely an example of how patterns in software engineering tend to influence the development of programming languages—after all, channels and isolated processes started life as patterns.

7.3 Contributions

Several contributions have been made during the course of this work.

The primary aim of this work was to establish a pattern language for process-oriented design, documenting best-practice solutions to the challenges of concurrent software design in the process-oriented style, and providing a common technical vocabulary that can be used to discuss and reason about process-oriented software. This aim has been achieved (chapter 4). It should be noted that this is an *initial* pattern language; it should be expected—and allowed—to develop and expand over time, as facilities in process-oriented environments develop and further experience is gained with process-oriented design.

The patterns have been successfully applied during the design and implementation of several new pieces of process-oriented software, including LOVE, a family of frameworks for audio processing and music synthesis using process-oriented techniques (section 3.7), and Occade, a module for programming arcade games in a process-oriented style, now in use for teaching concurrent programming at Kent (section 3.8). Significant contributions were also made to the RMoX project, a process-oriented operating system (section 3.4), and the TUNA project, investigating strategies for engineering and simulating emergent systems (section 3.5). The patterns described here are being used today as the basis for the development of higher-level patterns for complex systems simulation using process-oriented techniques as part of the CoSMoS project (section 3.9) [171, 9].

Based upon these experiences, proposals have been made for new facilities in process-oriented programming environments to better support the use of the ▷ **Client-Server**, ▷ **Phases** and ▷ **Clock** patterns (chapter 5). In addition, important contributions have been made to infrastructure for the development of process-oriented software: the first version of Tock, a new compiler for occam-π and other concurrent languages, written using a functional nanopass approach (section 2.3.2); a set of standard tools for packaging and documenting occam-π modules (section 3.3.1); and numerous enhancements to the KRoC standard library (section 3.3).

Applications have so far been identified for process-oriented programming in the fields of systems programming, multimedia processing, games development, embedded software and complex systems simulation. The author is confident that the process-oriented approach will, in the future, enable the engineering of complex, scalable, efficient and correct concurrent software in an expanding range of application areas throughout the world of software development.

Bibliography

[1] A. Adamatzky, editor. *Collision-Based Computing*. Springer-Verlag, 2001. ISBN 1-85233-540-8.

[2] Ali-Reza Adl-Tabatabai, Christos Kozyrakis, and Bratin Saha. Unlocking Concurrency. *ACM Queue*, 4(10):24–33, 2007. ISSN 1542-7730.

[3] Agility Design Solutions, Inc. *Handel-C Language Reference Manual*, 2007. RM-1003-4.4.

[4] C. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979. ISBN 0-19-502402-8.

[5] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977. ISBN 0-19-501919-3.

[6] Alastair R. Allen, Oliver Faust, and Bernhard Sputh. Transfer Request Broker: Resolving Input-Output Choice. In Welch et al. [249], pages 163–177.

[7] Kyle Ambroff. greenlet.
http://pypi.python.org/pypi/greenlet

[8] M. Amos and O. Don. An Ant-Based Algorithm for Annular Sorting. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC)*, pages 142–148. IEEE Press, 2007.

[9] Paul S. Andrews, Fiona A. C. Polack, Adam T. Sampson, Susan Stepney, and Jon Timmis. The CoSMoS Process Version 0.1: A Process for the Modelling and Simulation of Complex Systems. Technical Report YCS-2010-453, Department of Computer Science, University of York, 2010.
http://www.cs.york.ac.uk/ftpdir/reports/2010/YCS/453/YCS-2010-453.pdf

[10] Paul S. Andrews, Adam T. Sampson, John Markus Bjørndalen, Susan Stepney, Jon Timmis, Douglas N. Warren, and Peter H. Welch. Investigating patterns for the process-oriented modelling and simulation of space in complex systems. In S. Bullock, J. Noble, R. Watson, and M. A. Bedau, editors, *Artificial Life XI: Proceedings of the Eleventh International Conference on the Simulation and Synthesis of Living Systems*, pages 17–24. MIT Press, Cambridge, MA, 2008.

[11] Otto J. Anshus, John Markus Bjørndalen, and Brian Vinter. PyCSP - Communicating Sequential Processes for Python. In McEwan et al. [136], pages 229–248.

[12] Apple, Inc. *Grand Central Dispatch (GCD) Reference*, February 2010.

[13] Arduino.
http://arduino.cc/

[14] Joe Armstrong. The development of Erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN International Conference on Functional Programming*, pages 196–203. ACM Press, 1997. ISBN 0-89791-918-1.

[15] Joe Armstrong. A history of Erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of Programming Languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X.

[16] Phil Atkin and Stephen Ghee. High performance graphics with the IMS T800. Technical Report 72-TCH-037-00, INMOS Limited, March 1988.

[17] Phil Atkin and Stephen Ghee. A transputer based multi-user flight simulator. Technical Report 72-TCH-036-01, INMOS Limited, March 1988.

[18] Atmel. *Atmel 8-bit AVR Microcontroller Data Sheet (ATmega328P et al.)*, May 2009.
http://www.atmel.com/

[19] Jonas Barklund and Robert Virding. *Erlang 4.7.3 Reference Manual*, February 1999.
http://www.erlang.org/download/erl_spec47.ps.gz

[20] Fred Barnes and Peter Welch. Prioritised Dynamic Communicating Processes: Part 1. In Pascoe et al. [162], pages 321–352.

[21] Fred Barnes and Peter Welch. Prioritised Dynamic Communicating Processes: Part 2. In Pascoe et al. [162], pages 353–370.

[22] Frederick R. M. Barnes, Christian Jacobson, and Brian Vinter. RMoX: A raw-metal occam Experiment. In Broenink and Hilderink [43], pages 269–288.

[23] Frederick R. M. Barnes, Jon M. Kerridge, and Peter H. Welch, editors. *Communicating Process Architectures 2006*, volume 64 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, 2006. WoTUG, IOS Press. ISBN 978-1-58603-671-3.

[24] Frederick R. M. Barnes and Carl G. Ritson. A Process Oriented Approach to USB Driver Development. In McEwan et al. [136], pages 323–338.

[25] Frederick R. M. Barnes and Carl G. Ritson. Checking Process-Oriented Operating System Behaviour using CSP and Refinement. *SIGOPS Operating Systems Review*, 43(4):45–49, 2009. ISSN 0163-5980.

[26] Frederick R.M. Barnes. Blocking System Calls in KRoC/Linux. In Peter H. Welch and Andre W.P. Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering*, pages 155–178, Amsterdam, The Netherlands, 2000. WoTUG, IOS Press. ISBN 978-1-58603-077-3.

[27] Frederick R.M. Barnes. *Dynamics and Pragmatics for High Performance Concurrency*. PhD thesis, University of Kent at Canterbury, June 2003.

[28] Frederick R.M. Barnes. The New occam Adventure, 2006.
http://frmb.org/occam.html

[29] Frederick R.M. Barnes, Peter H. Welch, and Adam T. Sampson. Barrier Synchronisation for occam-pi. In Hamid R. Arabnia, editor, *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '05)*, pages 173–179, Las Vegas, Nevada, USA, June 2005. CSREA Press. ISBN 1-932415-58-0.

[30] F.R.M. Barnes. tranx86 – an Optimising ETC to IA32 Translator. In Chalmers et al. [60], pages 265–282.

[31] F.R.M. Barnes. Interfacing C and occam-pi. In Broenink et al. [44], pages 249–260.

[32] F.R.M. Barnes and P.H. Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In Chalmers et al. [60], pages 243–264.

[33] F.R.M. Barnes and P.H. Welch. Communicating Mobile Processes. In East et al. [76], pages 201–218.

[34] Geoff Barrett. *occam 3 Reference Manual*. INMOS Limited, March 1992.

[35] D.J. Beckett and P.H. Welch. A Strict occam Design Tool. In C.R. Jesshope and A. Shafarenko, editors, *Proceedings of UK Parallel '96*, pages 53–69, Guildford, UK, July 1996. Springer-Verlag. ISBN 3-540-76068-7.

[36] Bell Labs. *Plan 9 libthread(2) manual page*.
http://plan9.bell-labs.com/magic/man2html/2/thread

[37] Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems*, 26(5):769–804, 2004. ISSN 0164-0925.

[38] D.J. Bernstein. The self-pipe trick.
http://cr.yp.to/docs/selfpipe.html

[39] Micah J. Best, Alexandra Fedorova, Ryan Dickie, Andrea Tagliasacchi, Alex Couture-Beil, Craig Mustard, Shane Mottishaw, Aron Brown, Zhi Feng Huang, Xiaoyuan Xu, Nasser Ghazali, and Andrew Brownsword. Searching for Concurrent Design Patterns in Video Games. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 912–923, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-03868-6.

[40] John Markus Bjørndalen and Adam T. Sampson. Process-Oriented Collective Operations. In Welch et al. [249], pages 309–328.

[41] Josh Blum. The GNU Radio Companion.
http://www.joshknows.com/grc

[42] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–230, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1.

[43] Jan F. Broenink and Gerald H. Hilderink, editors. *Communicating Process Architectures 2003*, volume 61 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, 2003. WoTUG, IOS Press. ISBN 978-1-58603-381-1.

[44] Jan F. Broenink, Herman W. Roebbers, Johan P.E. Sunter, Peter H. Welch, and David C. Wood, editors. *Communicating Process Architectures 2005*, volume 63 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, 2005. WoTUG, IOS Press. ISBN 978-1-58603-561-7.

[45] Neil C. C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures,. In McEwan et al. [136], pages 183–205.

[46] Neil C. C. Brown. Auto-Mobiles: Optimised Message-Passing. In Welch et al. [248], pages 225–238.

[47] Neil C. C. Brown. Automatic Model-Checking for Communicating Haskell Processes. In Markus Roggenbach, editor, *Proceedings of the Ninth International Workshop on Automated Verification of Critical Systems (AVOCS 2009)*, volume 23 of *Electronic Communications of the EASST*, 2009. ISSN 1863-2122.

[48] Neil C.C. Brown. Abstractions for Message-Passing: what can be learned from functional programming. To appear.

[49] Neil C.C. Brown. *CHP API documentation*.
http://hackage.haskell.org/package/chp

[50] Neil C.C. Brown. *CHP-Plus API documentation*.
http://hackage.haskell.org/package/chp-plus

[51] Neil C.C. Brown. Communicating Haskell Processes: Composable Explicit Concurrency Using Monads. In Welch et al. [249], pages 67–83.

[52] Neil C.C. Brown. How to Make Your Process Invisible, a.k.a. The Dialing Philosophers. In Welch et al. [249].

[53] Neil C.C. Brown. Concurrent Pearl: The Sort Pump, 2009.
http://chplib.wordpress.com/2009/09/18/the-sort-pump/

[54] Neil C.C. Brown. Multiway Synchronisations with Disjunctive and Conjunctive Choice using Software Transactional Memory, 2009. To appear.

[55] Neil C.C. Brown and Adam T. Sampson. Matching and Modifying with Generics. In *Trends in Functional Programming (TFP) 2008*, pages 304–318, May 2008.

[56] Neil C.C. Brown and Adam T. Sampson. Alloy: Fast Generic Transformations for Haskell. In *Haskell '09: Proceedings of the 2009 ACM SIGPLAN Haskell Symposium*, pages 105–116, 2009. ISBN 978-1-60558-508-6.

[57] Neil C.C. Brown and Peter H. Welch. An Introduction to the Kent C++CSP Library. In Broenink and Hilderink [43], pages 139–156.

[58] W. J. Brown, R. C. Malveau, H. W. McCormick III, and T. J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Ltd., 1998.

[59] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *ACM Queue*, 6(5):46–58, 2008. ISSN 1542-7730.

[60] Alan Chalmers, Majid Mirmehdi, and Henk Muller, editors. *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, 2001. WoTUG, IOS Press. ISBN 978-1-58603-202-9.

[61] Kevin Chalmers, Jon M. Kerridge, and Imed Romdhani. A Critique of JCSP Networking. In Welch et al. [249], pages 271–291.

[62] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 519–538, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0.

[63] David G. Clarke, John M. Potter, and James Noble. Ownership Types for Flexible Alias Protection. In *OOPSLA '98: Proceedings of the 13th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64. ACM Press, 1998.

[64] CoSMoS: Complex Systems Modelling and Simulation infrastructure. http://www.cosmos-research.org/

[65] Russ Cox. Bell Labs and CSP Threads. http://swtch.com/ rsc/thread/

[66] Paul Davis. The JACK Audio Connection Kit. In *Linux Audio Developers' Conference 2003*, 2003.

[67] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008. ISSN 0001-0782.

[68] Mark Debbage, Mark Hill, S. Wykes, and Denis A. Nicole. Southampton's Portable Occam Compiler (SPOC). In Roger Miles and Alan G. Chalmers, editors, *Proceedings of WoTUG-17: Progress in Transputer and Occam Research*, pages 40–55, March 1994. ISBN 90-5199-163-0.

[69] Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, Alex Ahern, and Sophia Drossopoulou. L_{doos}: a Distributed Object-Oriented Language with Session Types. In Rocco De Nicola and Davide Sangiorgi, editors, *TGC 2005*, volume 3705 of *LNCS*, pages 299–318. Springer-Verlag, 2005. ISBN 3-540-30007-4.

[70] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *ASPLOS '09: Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, pages 157–168. ACM, 2009. ISBN 978-1-60558-406-5.

[71] Edsger W. Dijkstra. Hierarchical ordering of sequential processes. In *The origin of concurrent programming: from semaphores to remote procedure calls*, pages 198–227. Springer-Verlag, New York, NY, USA, 2002. ISBN 0-387-95401-5.

[72] D.J. Dimmich and C.L. Jacobsen. A foreign function interface generator for occam-pi. In Broenink et al. [44], pages 235–248.

[73] Ian R. East. The Honeysuckle Programming Language: Event and Process. In Pascoe et al. [162], pages 285–300.

[74] Ian R. East. Interfacing with Honeysuckle by Formal Contract. In Broenink et al. [44], pages 1–11.

[75] Ian R. East. Concurrent/Reactive System Design with Honeysuckle. In McEwan et al. [136], pages 109–118.

[76] Ian R. East, David Duce, Mark Green, Jeremy M.R. Martin, and Peter H. Welch, editors. *Communicating Process Architectures 2004*, volume 62 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, 2004. WoTUG, IOS Press. ISBN 978-1-58603-458-0.

[77] Steven Ericsson-Zenith. *occam 2 Reference Manual*. INMOS Limited, 1988.

[78] Steven Ericsson-Zenith. *Process Interaction Models*. PhD thesis, Universite Pierre et Marie Curie, PARIS (VI), 1992.

[79] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R. Larus, and Steven Levi. Language support for fast and reliable message-based communication in Singularity OS. *SIGOPS Operating Systems Review*, 40(4):177–190, 2006. ISSN 0163-5980.

[80] Robert E. Filman and Daniel P. Friedman. *Coordinated Computing: Tools and Techniques for Distributed Software*. McGraw Hill, New York, 1984. ISBN 0-07-022439-0.

[81] Anton Jakob Flügge, Jon Timmis, Paul Andrews, John Moore, and Paul Kaye. Modelling and Simulation of Granuloma Formation in Visceral Leishmaniasis. In *2009 IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 3052–3059. IEEE Press, 2009. ISBN 978-1-4244-2959-2.

[82] Bryan Ford, Godmar Back, Greg Benson, Jay Lepreau, Albert Lin, and Olin Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Symposium on Operating Systems Principles*, pages 38–51, 1997.

[83] Bryan Ford and Erich Stefan Boleyn. *Multiboot Specification version 0.6.96*. Free Software Foundation, Inc., 2009.

[84] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement: FDR2 Manual*, 1997.

[85] Rune Møllegaard Friborg, John Markus Bjørndalen, and Brian Vinter. Three Unique Implementations of Processes for PyCSP. In Welch et al. [248], pages 277–292.

[86] Henry Fuchs, Zvi M. Kedem, and Bruce F. Naylor. On visible surface generation by a priori tree structures. In *SIGGRAPH '80: Proceedings of the 7th annual conference on Computer Graphics and Interactive Techniques*, pages 124–133, New York, NY, USA, 1980. ACM. ISBN 0-89791-021-4.

[87] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995. ISBN 0-201-63361-2.

[88] Martin Gardner. The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223:120–123, October 1970.

[89] Huw Geddes. *XMOS Tools User Guide, Version 9.7*. XMOS Ltd., July 2009.

[90] Teodor Ghetiu, Robert D. Alexander, Paul S. Andrews, Fiona A. C. Polack, and James Bown. Equivalence Arguments for Complex Systems Simulations—A Case-Study. In Susan Stepney, Peter H. Welch, Paul S. Andrews, and Jon Timmis, editors, *Proceedings of the 2009 Workshop on Complex Systems Modelling and Simulation, York, UK, August 2009*, pages 101–140. Luniver Press, 2009. ISBN 978-1-905986-22-4.

[91] *Effective Go*.
http://golang.org/doc/effective_go.html

[92] *The Go Programming Language Specification (Version of March 25, 2010)*.
http://golang.org/doc/go_spec.html

[93] R. Wm. Gosper. Exploiting regularities in large cellular spaces. *Physica D*, 10:75–80, 1984.

[94] Paul Graham. Revenge of the Nerds.
http://www.paulgraham.com/icad.html

[95] Andy Hamilton. Some issues in scientific-language application porting and farming using transputers. Technical Report 72-TCH-053-01, INMOS Limited, July 1989.

[96] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, pages 48–60, New York, NY, USA, 2005. ACM. ISBN 1-59593-080-9.

[97] Maurice Herlihy and J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, pages 289–300, May 1993.

[98] Gerald Hilderink, André Bakkers, and Jan Broenink. A Distributed Real-Time Java System Based on CSP. In *The Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, ISORC 2000*, pages 400–407, March 2000.

[99] Gerald H. Hilderink. Graphical modelling language for specifying concurrency based on CSP. *IEE Proceedings: Software*, 150(2):108–120, April 2003. ISSN 1462-5970.

[100] Michael G Hinchey, editor. *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, Stanford, California, August 2006. IEEE Computer Society. ISBN 0-7695-2530-X.

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

[102] C.A.R. Hoare. *Communicating sequential processes.* Prentice-Hall, 1985.

[103] Tony Hoare. Fine-grain Concurrency. In McEwan et al. [136], pages 1–20.

[104] Christian Hofer, Klaus Ostermann, Tillmann Rendel, and Adriaan Moors. Polymorphic embedding of DSLs. In *GPCE '08: Proceedings of the 7th international conference on Generative Programming and Component Engineering*, pages 137–148, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-267-2.

[105] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Mess.*, 3(2):11–16, 1992. ISSN 1055-6400.

[106] Kohei Honda. Types for Dyadic Interaction. In *Proceedings of CONCUR '93*, number 715 in LNCS, pages 509–523. Springer-Verlag, 1993.

[107] Tim Hoverd and Adam T. Sampson. A Transactional Architecture for Simulation. In *ICECCS 2010: Fifteenth IEEE International Conference on Engineering of Complex Computer Systems*, pages 286–290. IEEE Press. ISBN 978-0-7695-4015-3.

[108] Raymond Hu, Nobuko Yoshida, and Kohei Honda. Language and Runtime Implementation of Sessions for Java. In Olivier Zendra, Eric Jul, and Michael Cebulla, editors, *ICOOOLPS'2007*, July 2007. ISSN 1436-9915.

[109] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, December 1996. ISSN 0360-0300.

[110] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A History of Haskell: being lazy with class. In *The Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, June 2007.

[111] Galen Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. An Overview of the Singularity Project. Technical Report MSR-TR-2005-135, Microsoft Research, October 2005.

[112] Jean D. Ichbiah, John G.P. Barnes, Robert J. Firth, and Mike Woodger. *Rationale for the Design of the Ada Programming Language*. Cambridge University Press, 1991. ISBN 0-521-39267-5.

[113] Intel. *Threading Building Blocks*, April 2010.

[114] The Io programming language.
http://iolanguage.com/

[115] Christian L. Jacobsen. *A portable runtime for concurrency research and application*. PhD thesis, University of Kent at Canterbury, December 2006.

[116] Christian Jacobson and Matthew C. Jadud. The Transterpreter: A Transputer Interpreter. In East et al. [76], pages 99–106.

[117] Matthew C. Jadud. Cool Stuff in Computer Science.
http://www.cs-ed.org/blogs/cscs/

[118] Matthew C. Jadud, Christian L. Jacobsen, and Adam T. Sampson. Plumbing for the Arduino, March 2010.
http://concurrency.cc/book/

[119] Matthew C. Jadud, Christian L. Jacobsen, Jon Simpson, and Carl G. Ritson. Safe Parallelism for Behavioral Control. In *2008 IEEE Conference on Technologies for Practical Robot Applications*, pages 137–142. IEEE, November 2008.

[120] Matthew C. Jadud, Jonathan Simpson, and Christian L. Jacobsen. Patterns for programming in parallel, pedagogically. In *SIGCSE '08: Proceedings of the 39th SIGCSE technical symposium on Computer Science Education*, pages 231–235, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-799-5.

[121] Jibu for .NET.
http://www.axon7.com/

[122] Geraint Jones and Michael Goldsmith. *Programming in occam 2*. Prentice Hall, 1988. ISBN 0-13-730334-3.
http://www.comlab.ox.ac.uk/geraint.jones/publications/book/Pio2/

[123] Dusko S. Jovanovic, Bojan Orlic, Geert K. Liet, and Jan F. Broenink. gCSP: A Graphical Tool for Designing CSP systems. In East et al. [76], pages 233–251.

[124] Alan C. Kay. The early history of Smalltalk. In *History of programming languages—II*, pages 511–598. ACM, New York, NY, USA, 1996. ISBN 0-201-89502-1.

[125] Dan Kegel. The C10K Problem.
http://www.kegel.com/c10k.html

[126] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[127] Doug Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 2000. ISBN 0-201-31009-0.

[128] Doug Lea. JSR 166: Concurrency Utilities, September 2004.
http://www.jcp.org/en/jsr/detail?id=166

[129] Peter Maley and Adam T. Sampson. *OccamDoc*.
https://www.cs.kent.ac.uk/research/groups/plas/wiki/OccamDoc

[130] Simon Marlow, Simon Peyton Jones, and Wolfgang Thaller. Extending the Haskell foreign function interface with concurrency. In *Haskell '04: Proceedings of the ACM SIGPLAN workshop on Haskell*, pages 22–32, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-850-4.

[131] J.M.R. Martin and P.H. Welch. A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications*, 3(4), 1997. ISSN 1070-454X.

[132] Philip Mattos. Program design for concurrent systems. Technical Report 72-TCH-005-01, INMOS Limited, February 1987.

[133] David May. OCCAM. *SIGPLAN Notices*, 18(4):69–79, 1983. ISSN 0362-1340.

[134] David May. *The XMOS XS1 Architecture*. XMOS Ltd., October 2009.

[135] Steve McClure and Richard Wheeler. MOSIX: how Linux clusters solve real world problems. In *ATEC '00: Proceedings of the USENIX Annual Technical Conference*, pages 49–56, Berkeley, CA, USA, June 2000. The USENIX Association.

[136] Alistair A. McEwan, Wilson Ifill, and Peter H. Welch, editors. *Communicating Process Architectures 2007*, volume 65 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, 2007. WoTUG, IOS Press. ISBN 978-1-58603-767-3.

[137] The Meiko computing surface: an example of a massively parallel system. In *C3P: Proceedings of the third conference on Hypercube Concurrent Computers and Applications*, pages 852–859, New York, NY, USA, 1988. ACM. ISBN 0-89791-278-0.

[138] MIDI Manufacturers Association. *Complete MIDI 1.0 Detailed Specification*. ISBN 0-9728831-0-X.

[139] Robin Milner. *Communicating and mobile systems: the π-calculus*. Cambridge University Press, New York, NY, USA, 1999. ISBN 0-521-65869-1.

[140] Nick Montfort. *Racing the Beam: The Atari Video Computer System*. MIT Press, 2009. ISBN 0-262-01257-X.

[141] James Moores. *The design and implementation of occam/CSP support for a range of languages and platforms*. PhD thesis, University of Kent at Canterbury, December 2000.

[142] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006. ISSN 0167-6423.

[143] Sarah Mount, Mohammad Hammoudeh, Sam Wilson, and Robert Newman. CSP as a Domain-Specific Language Embedded in Python and Jython. In Welch et al. [248], pages 293–309.

[144] MPI Forum. MPI-2: Extensions to the Message-Passing Interface. Technical report, MPI Forum, July 1997.

[145] Kedar S. Namjoshi. Are Concurrent Programs That Are Easier to Write Also Easier to Check? In *Exploiting Concurrency Efficiently and Correctly – CAV 2008 Workshop*, 2008.

[146] Matthias Neubauer and Peter Thiemann. An Implementation of Session Types. In Bharat Jayaraman, editor, *PADL*, volume 3057 of *Lecture Notes in Computer Science*, pages 56–70. Springer-Verlag, 2004. ISBN 3-540-22253-7.

[147] Jesse Noller and Richard Oudkerk. PEP 371: Addition of the multiprocessing package to the standard library, 2008.
http://www.python.org/dev/peps/pep-0371/

[148] Object Management Group, Inc. *Unified Modeling Language (Version 2.2): Infrastructure and Superstructure*, 2009.
http://www.omg.org/spec/UML/2.2/

[149] Martin Odersky. The Scala Language Specification, Version 2.7. Technical report, Programming Methods Laboratory, EPFL, March 2009.

[150] OGRE — Open Source 3D Graphics Engine.
http://www.ogre3d.org/

[151] Ian Oliver and Vesa Luukkala. On UML's Composite Structure Diagram. In *5th Workshop on System Analysis and Modelling*, 2006.

[152] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, May 2008.

[153] Bojan Orlic. *SystemCSP: a graphical language for designing concurrent component-based embedded control systems*. PhD thesis, University of Twente, September 2007.

[154] Jorge L. Ortega-Arjona. The Manager-Workers Pattern: An Activity Parallelism Architectural Pattern for Parallel Programming. In *9th European Conference on Pattern Languages of Programs (EuroPLoP 2004)*, Irsee, Germany, July 2004.

[155] Jorge L. Ortega-Arjona. The Pipes and Filters Pattern: A Functional Parallelism Architectural Pattern for Parallel Programming. In *10th European Conference on Pattern Languages of Programs (EuroPLoP 2005)*, Irsee, Germany, July 2005.

[156] Jorge L. Ortega-Arjona. Design Patterns for Communication Components of Parallel Programs. In *12th European Conference on Pattern Languages of Programs (EuroPLoP 2007)*, Irsee, Germany, July 2007.

[157] Jorge Luis Ortega-Arjona. *Patterns for Parallel Software Design*. John Wiley & Sons, Ltd., 2010. ISBN 0-470-69734-2.

[158] Bryan O'Sullivan, Don Stewart, and John Goerzen. *Real World Haskell*. O'Reilly Media, November 2008. ISBN 0-596-51498-0.
http://book.realworldhaskell.org/

[159] Nick Owens and Susan Stepney. The Game of Life on Penrose Tilings: still life and oscillators. In Andy Adamatsky, editor, *Game of Life Cellular Automata*. Springer-Verlag, 2010. ISBN 1-849-96216-2.

[160] Jamie Packer. Exploiting concurrency: a ray tracing example. Technical Report 72-TCH-007-01, INMOS Limited, October 1987.

[161] Ian Page and Wayne Luk. Compiling occam into Field-Programmable Gate Arrays. Technical report, Oxford University Computing Laboratory, October 1991.

[162] James Pascoe, Roger Loader, and Vaidy Sunderam, editors. *Communicating Process Architectures 2002*, volume 60 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, 2002. WoTUG, IOS Press. ISBN 978-1-58603-268-5.

[163] Jan Bækgaard Pedersen and Brian Kauke. Resumable Java Bytecode – Process Mobility for the JVM. In Welch et al. [248], pages 159–172.

[164] Carl Adam Petri. *Kommunikation mit Automaten*. PhD thesis, Institut für Instrumentelle Mathematik, 1962.

[165] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 295–308, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3.

[166] Andrew Phillips and Luca Cardelli. A Graphical Representation for the Stochastic Pi-Calculus. In *Proceedings of BioConcur 2005*, San Francisco, California, August 2005.

[167] Rob Pike. A Concurrent Window System. *Computing Systems*, 2(2):133–153, 1989.

[168] Rob Pike. Newsqueak: A Language for Communicating with Mice. Technical Report 143, Bell Labs, April 1993.

[169] Rob Pike. Rio: Design of a Concurrent Window System, February 2000. http://doc.cat-v.org/plan_9/3rd_edition/rio/

[170] Fiona Polack, Susan Stepney, Heather Turner, Peter Welch, and Fred Barnes. An Architecture for Modelling Emergence in CA-Like Systems. In Mathieu S. Capcarrère, Alex Alves Freitas, Peter J. Bentley, Colin G. Johnson, and Jon Timmis, editors, *Advances in Artificial Life, 8th European Conference on Artificial Life (ECAL 2005)*, volume 3630 of *Lecture Notes in Computer Science*, pages 427–436, Canterbury, UK, September 2005. Springer-Verlag. ISBN 3-540-28848-1.

[171] Fiona A. C. Polack, Paul S. Andrews, Teodor Ghetiu, Mark Read, Susan Stepney, Jon Timmis, and Adam T. Sampson. Reflections on the Simulation of Complex Systems for Science. In *ICECCS 2010: Fifteenth IEEE International Conference on Engineering of Complex Computer Systems*, pages 276–285. IEEE Press. ISBN 978-0-7695-4015-3.

[172] Fiona A.C. Polack, Paul S. Andrews, and Adam T. Sampson. The engineering of concurrent simulations of complex systems. In *2009 IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 217–224. IEEE Press, 2009. ISBN 978-1-4244-2959-2.

[173] Michael D. Poole. Extended Transputer Code – a Target-Independent Representation of Parallel Programs. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 187–198, March 1998. ISBN 90-5199-391-9.

[174] Portland Pattern Repository: Concurrency Patterns.
http://c2.com/cgi/wiki?CategoryConcurrencyPatterns

[175] M. Puckette. Pure Data: another integrated computer music environment. In *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41, 1996.

[176] Miller Puckette. *The Theory and Techniques of Electronic Music*. World Scientific, 2007. ISBN 981-270-077-3.

[177] Mark Read, Paul S. Andrews, Jon Timmis, and Vipin Kumar. Using UML to Model EAE and its Regulatory Network. In *8th International Conference on Artificial Immune Systems (ICARIS)*, volume 5666 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[178] RepRap.
http://reprap.org/

[179] Craig W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer Graphics and Interactive Techniques*, pages 25–34, New York, NY, USA, 1987. ACM. ISBN 0-89791-227-6.

[180] John Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *17th Annual IEEE Symposium on Logic in Computer Science (2002)*, pages 55–74. IEEE Computer Society, 2002.

[181] Dennis M. Ritchie. The Limbo Programming Language. In *Inferno Programmer's Manual, Volume 2*. Vita Nuova Holdings Ltd., 2000.

[182] Carl G. Ritson. Implicitly shared channels, 2009. Private communication.

[183] Carl G. Ritson. Translating ETC to LLVM Assembly. In Welch et al. [248], pages 145–158.

[184] Carl G. Ritson, Adam T. Sampson, and Frederick R. M. Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Thudichum Vasconcelos, editors, *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5521 of *Lecture Notes in Computer Science*, pages 163–183. Springer-Verlag, 2009. ISBN 978-3-642-02052-0.

[185] Carl G. Ritson, Adam T. Sampson, and Frederick R.M. Barnes. Video Processing in occam-pi. In Barnes et al. [23], pages 311–329.

[186] Carl G. Ritson and Jonathan Simpson. Virtual Machine Based Debugging for occam-π. In Welch et al. [249], pages 293–307.

[187] Carl G. Ritson and Peter H. Welch. A Process-Oriented Architecture for Complex System Modelling. In McEwan et al. [136], pages 249–266.

[188] A. W. Roscoe and C. A. R. Hoare. The laws of Occam programming. *Theoretical Computer Science*, 60(2):177–229, 1988. ISSN 0304-3975.

[189] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.

[190] Jörg Roth. Patterns of Mobile Interaction. *Personal Ubiquitous Computing*, 6(4):282–289, 2002.

[191] Marcin Rzeznicki. coroutines: fully-functional coroutines for Java. http://code.google.com/p/coroutines/

[192] Adam T. Sampson. *Module occade: a library for programming simple arcade games in occam-pi*. http://occam-pi.org/occamdoc/occade.html

[193] Adam T. Sampson. *OccamDoc index: the KRoC API*. http://occam-pi.org/occamdoc/

[194] Adam T. Sampson. OEP/1: occam enhancement proposals. https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/1

[195] Adam T. Sampson. OEP/136: Compact IF and expanded WHILE. https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/136

[196] Adam T. Sampson. OEP/138: Type attribute syntax. https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/138

[197] Adam T. Sampson. OEP/161: Named FORKING blocks. https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/161

[198] Adam T. Sampson. The design and development of occamnet, 2003. http://offog.org/publications/occamnet.pdf

[199] Adam T. Sampson. Compiling occam to C with Tock. In McEwan et al. [136], pages 511–512.

[200] Adam T. Sampson. Two-Way Protocols for occam-π. In Welch et al. [249], pages 85–97.

[201] Adam T. Sampson, John Markus Bjørndalen, and Paul S. Andrews. Birds on the Wall: Distributing a Process-Oriented Simulation. In *2009 IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 225–231. IEEE Press, 2009. ISBN 978-1-4244-2959-2.

[202] Adam T. Sampson and Neil C.C. Brown. Tock (translator from occam to C from Kent). http://projects.cs.kent.ac.uk/projects/tock/

[203] Adam T. Sampson, Peter H. Welch, and Frederick R.M. Barnes. Lazy Cellular Automata with Communicating Processes. In Broenink et al. [44], pages 165–175.

[204] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A Nanopass Infrastructure for Compiler Education. In *Proceedings of ICFP '04, September 19–21, 2004, Snowbird, Utah, USA.*, pages 201–212. ACM Press, 2004.

[205] Neil Schemenauer, Tim Peters, and Magnus Lie Hetland. PEP 255: Simple Generators, 2001.
http://www.python.org/dev/peps/pep-0255/

[206] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Ltd., 2000. ISBN 0-471-60695-2.

[207] Steve Schneider, Helen Treharne, Ana Cavalcanti, and Jim Woodcock. A Layered Behavioural Model of Platelets. In Hinchey [100], pages 98–106.

[208] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics*. Kitware, Inc. ISBN 1-930934-19-X.

[209] Mario Schweigler. *A Unified Model for Inter- and Intra-processor Concurrency*. PhD thesis, Computing Laboratory, University of Kent, Canterbury, UK, August 2006.

[210] Mario Schweigler, Fred Barnes, and Peter Welch. Flexible, Transparent and Dynamic occam Networking with KRoC.net. In Broenink and Hilderink [43], pages 199–224.

[211] Mario Schweigler and Adam T. Sampson. pony - The occam-pi Network Environment. In Barnes et al. [23], pages 77–108.

[212] Simple DirectMedia Layer.
http://www.libsdl.org/

[213] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification*. The Khronos Group Inc., 2010.

[214] SGS-THOMSON Microelectronics Limited. *occam 2.1 reference manual*, May 1995.

[215] Jonathan Simpson and Christian L. Jacobsen. Visual Process-oriented Programming for Robotics. In Welch et al. [249], pages 365–380.

[216] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. Mobile Robot Control: The Subsumption Architecture and occam-π. In Barnes et al. [23], pages 225–236.

[217] Jonathan Simpson, Christian L. Jacobsen, and Matthew C. Jadud. A Native Transterpreter for the LEGO Mindstorms RCX. In McEwan et al. [136], pages 339–348.

[218] Jaroslav Sklenar. An Introduction to OOP in Simula. In *30 Years of Object Oriented Programming*. University of Malta, 1997.

[219] M. Sparks. Kamaelia: highly concurrent and network systems tamed. Technical Report WHP 113, BBC Research and Development, June 2005.

[220] Bernhard H.C. Sputh and Alastair R. Allen. JCSP-Poison: Safe Termination of CSP Process Networks. In Broenink et al. [44], pages 71–108.

[221] Susan Stepney. GRAIL: Graphical Representation of Activity, Interconnection and Loading. In Traian Muntean, editor, *7th Technical meeting of the occam User Group, Grenoble, France*. IOS Amsterdam, 1987.

[222] Don Stewart. Domain Specific Languages for Domain Specific Problems. In *Workshop on Non-Traditional Programming Models for High-Performance Computing, LACSS*, 2009.
http://www.galois.com/ dons/talks/edsl-haskell-talk.pdf

[223] Tim Stilson and Julius Smith. Alias-Free Digital Synthesis of Classic Analog Waveforms. In *Proceedings of the 1996 International Computer Music Conference*. The International Computer Music Association, 1996. ISBN 962-85092-1-7.

[224] Bernard Sufrin. Communicating Scala Objects. In Welch et al. [249], pages 35–54.

[225] Threads and Swing, September 2000.
http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html

[226] Oyvind Teig. PAR and STARTP Take the Tanks. In Peter H. Welch and Andrè W. P. Bakkers, editors, *Proceedings of WoTUG-21: Architectures, Languages and Patterns for Parallel and Distributed Applications*, pages 1–18, 1998. ISBN 90-5199-391-9.

[227] Christian Tismer. Continuations and Stackless Python, 1999.
http://www.stackless.com/spcpaper.htm

[228] The TOPLAP Manifesto.
http://www.toplap.org/index.php/ManifestoDraft

[229] Helen Treharne and Steve Schneider. Using a Process Algebra to control B OPERATIONS. In *Integated Formal Methods*, pages 437–456. Springer-Verlag, 1999.

[230] TUNA: Final Report, 2007.
http://www.cs.york.ac.uk/nature/tuna/outputs/finalreport.pdf

[231] Heather Turner, Susan Stepney, and Fiona Polack. Rule Migration: Exploring a design framework for emergence. *International Journal of Unconventional Computing*, 3(1):49–66, 2007.

[232] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, 1990. ISSN 0001-0782.

[233] Wouter van Oortmerssen. The Cube Engine.
http://cubeengine.com/

[234] V. T. Vasconcelos, António Ravara, and Simon Gay. Session types for functional multithreading. In *CONCUR'04*, number 3170 in LNCS, pages 497–511. Springer-Verlag, 2004.

[235] Brian Vinter, John Markus Bjørndalen, and Rune Møllegaard Friborg. PyCSP Revisited. In Welch et al. [248], pages 263–276.

[236] Rob von Behren, Jeremy Condit, and Eric Brewer. Why Events Are A Bad Idea (for High-concurrency Servers). In *Proceedings of HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*. The USENIX Association, May 2003.

[237] W3C. *Document Object Model (DOM) Level 2 Core Specification*. http://www.w3.org/TR/DOM-Level-2-Core/

[238] W3C. *HTML5: A vocabulary and associated APIs for HTML and XHTML (Working Draft 4 March 2010)*. http://www.w3.org/TR/html5/

[239] Ge Wang and Perry R. Cook. ChucK: a concurrent, on-the-fly audio programming language. In *Proceedings of the International Computer Music Conference (ICMC)*, pages 219–226, September 2003.

[240] Douglas Watt. *Programming XC on XMOS Devices*. XMOS Ltd., September 2009.

[241] P. H. Welch and J. B. Pedersen. Santa Claus: Formal Analysis of a Process-Oriented Solution. *ACM Transactions on Programming Languages and Systems*, 32(4), April 2010. ISSN 0164-0925.

[242] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Alting Barriers: Synchronisation with Choice in Java using JCSP. To appear in CPE.

[243] Peter H. Welch. OEP/153: Dynamically-sized arrays. https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/153

[244] Peter H. Welch. Process Oriented Design for Java: Concurrency for All. In P.M.A. Sloot, C.J.K. Tan, J.J. Dongarra, and A.G. Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 687–687. Springer-Verlag, April 2002. ISBN 3-540-43593-X.

[245] Peter H. Welch. A Fast Resolution of Choice between Multiway Synchronisations. In Barnes et al. [23], pages 389–390.

[246] Peter H. Welch and Frederick R.M. Barnes. Mobile Data Types for Communicating Processes. In H.R.Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications 2001*, volume 1, pages 20–26. CSREA, CSREA Press, June 2001. ISBN 1-892512-66-1.

[247] Peter H. Welch, Neil C. C. Brown, James Moores, Kevin Chalmers, and Bernhard H. C. Sputh. Integrating and Extending JCSP. In McEwan et al. [136], pages 349–370.

[248] Peter H. Welch, Herman W. Roebbers, Jan F. Broenink, Frederick R.M. Barnes, Carl G. Ritson, Adam T. Sampson, Gardiner S. Stiles, and Brian Vinter, editors. *Communicating Process Architectures 2009*, volume 67 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, 2009. WoTUG, IOS Press. ISBN 978-1-60750-065-0.

[249] Peter H. Welch, Susan Stepney, Fiona A.C. Polack, Frederick R.M. Barnes, Alistair A. McEwan, Gardner S. Stiles, Jan F. Broenink, and Adam T. Sampson, editors. *Communicating Process Architectures 2008*, volume 66 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, 2008. WoTUG, IOS Press. ISBN 978-1-58603-907-3.

[250] P.H. Welch. *JCSP API Specification: class org.jcsp.lang.Bucket*. http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/

[251] P.H. Welch. *JCSP API Specification: class org.jcsp.lang.Crew*. http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-1.1-rc4/jcsp-doc/

[252] P.H. Welch. Emulating Digital Logic using Transputer Networks (Very High Parallelism = Simplicity = Performance). *Parallel Computing*, 9:257–272, January 1989. ISSN 0167-8191.

[253] P.H. Welch. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*, pages 310–317, Enschede, Netherlands, April 1989. Occam User Group, IOS Press, Netherlands. ISBN 90-5199-007-3.

[254] P.H. Welch and F.R.M. Barnes. Mobile Barriers for occam-pi: Semantics, Implementation and Application. In Broenink et al. [44], pages 289–316.

[255] P.H. Welch, F.R.M. Barnes, and F.A.C. Polack. Communicating Complex Systems. In Hinchey [100], pages 107–117.

[256] P.H. Welch and D.J. Beckett. Research into Deadlock, Livelock and Starvation Aware Design Tool (Final Report). Technical report, University of Kent at Canterbury, 1996.

[257] P.H. Welch, G.R.R. Justo, and C.J. Willcock. Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S.C. Hilton, M.R. Jane, and P.H. Welch, editors, *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, volume 2, pages 981–1004, Aachen, Germany, September 1993. IOS Press, Netherlands. ISBN 90-5199-140-1.

[258] P.H. Welch, H. Roebbers, and K. Wijbrans. A Generalized FFT Algorithm on Transputers. In *Transputer Research and Applications 4, Proceedings of NATUG 4*, pages 77–87, Ithaca, New York, October 1990. North America Transputer User Group, IOS Press, Netherlands. ISBN 90-5199-040-4.

[259] P.H. Welch, B. Vinter, and F.R.M. Barnes. Initial Experiences with occam-pi Simulations of Blood Clotting on the Minimum Intrusion Grid. In *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '05)*, volume 1, pages 20–26. CSREA Press, June 2005. ISBN 1-892512-66-1.

[260] Colin J. Willcock. occam-X11: EMR Project Report. Technical report, University of Kent at Canterbury, 1992.

[261] Colin J. Willcock. *A Parallel X Window Server*. PhD thesis, University of Kent at Canterbury, 1992.

[262] Phil Winterbottom. ALEF Language Reference Manual. In *Plan 9 Programmer's Manual, Volume 2*. 1993.

[263] Wiring. ISSN 2011-8376.
http://wiring.org.co/

[264] David C. Wood. KRoC – Calling C Functions from occam. Technical report, Computing Laboratory, University of Kent, Canterbury, August 1998.

[265] Matthew Wright. *Open Sound Control 1.0 Specification*, 2002.
http://opensoundcontrol.org/spec-1_0

[266] The X Window System.
http://www.x.org/

Index