COMMUNICATING HASKELL PROCESSES

a thesis submitted to
The University of Kent
in the subject of computer science
for the degree
of phd.

By
Neil Christopher Charles Brown
May 2011

Contents

List of Figures

Abstract

Concurrent programming supports multiple threads of concurrent execution. It is a suitable paradigm for interaction with the outside world, where many inputs (e.g. network packets, key-presses, mouse-clicks) may arrive at any time and need to be dealt with at the same time. Concurrent execution also potentially allows for parallel speed-up; multicore machines are now the standard for new PCs, but the quest to take full advantage of the available parallelism continues.

One approach to concurrent programming is process-oriented programming, which uses message-passing and is based on Hoare and Roscoe's Communicating Sequential Processes (CSP). CSP is a process calculus centred on concurrent processes that communicate with each other via synchronous channels. This is used as a formal underpinning for process-oriented programming, either directly (proving behaviours of programs with a CSP model checker) or indirectly (providing sound design principles and patterns).

Process-oriented programming has previously been provided in programming languages such as occam-π and libraries for other mainstream languages such as Java and C++. However, as yet process-oriented programming has failed to gain much traction via these implementations; occam-π is rooted in the much older language occam and thus lacks powerful data structures (among other features) which are standard in most modern languages – while languages such as Java and C++, where data is mutable and easily shared, can be an awkward fit to process-oriented programming.

Haskell is a functional programming language that is notable for its purity, type-classes, monads and lazy evaluation: modern features that provide interesting and powerful ways to program, including good support for imperative programming. Like other functional languages, it eliminates mutable data – which immediately removes a whole class of problems in concurrent programming.

This thesis contends that CSP and process-oriented programming fit well with Haskell. The thesis details the creation of a CSP library for Haskell (Communicating Haskell Processes: CHP) – which features powerful support for process composition – and its further augmentation with capabilities such as support for a new concurrency primitive (conjunction), tracing and the ability to generate formal models of CHP programs. This allows programmers to build concurrent message-passing systems with a strong formal underpinning in a modern programming language, with more features and less complications than process-oriented libraries for other languages.

Acknowledgements

My decision to begin a PhD was strongly influenced by my experience working in research (albeit in machine learning) at QinetiQ, after completing my undergraduate degree. I was fortunate to work with Zoë Webster (née Lock), Claire Thie (née Kennedy) and Emma Steele (née Peeling) from whom I learnt a great deal, and all of whom chose to pursue PhDs. Mike Kirton always encouraged me to do a PhD, while the late Chris Booth's staunch principles were an inspiration.

This PhD owes a great deal to Adam Sampson, a fellow student at Kent, who inadvertently set its course by writing an occam compiler in Haskell and thus tempting me to re-learn the language so that I could join in; once I was writing a compiler for concurrent languages in Haskell, it was a small step to exploring Haskell's own concurrency features. During the PhD I was pleased to be able to work with Marc Smith from Vassar College, who has been kind and supportive towards me ever since I met him at the very first conference I attended, back in 2003.

The concurrency group at Kent was a good environment in which to work, and my supervisor, Peter Welch, was always encouraging. Friends and family were understanding when I was always busy. Michael Kölling kindly volunteered some useful proof-reading of my thesis draft. The thesis was greatly improved by the input of my two examiners, Simon Thompson and Simon Marlow, for which I am very grateful.

And finally, thanks are due to Kristina, for everything.

Chapter 1

Introduction

Hoare and Roscoe's Communicating Sequential Processes (CSP) is a process calculus centred on concurrent processes that communicate with each other via synchronising events or channels [Hoare, 1985; Roscoe, 1997]. CSP provides a way to describe, with a formal notation and semantics, systems of interacting processes. A programming style that it underpins, process-oriented programming, supports encapsulation and polymorphism (initially popularised by object-oriented programming) with easily visualisable process networks and simple intuitive message-passing semantics.

Process-oriented programming began in a purpose-designed language, occam [INMOS, 1985] (later revised as occam-π [Welch and Barnes, 2005]), and has since been added to various imperative languages such as Java [Welch, 1997; Hilderink et al., 1997], C [Moores, 1999; Ritson et al., 2009], C++ [Brown and Welch, 2003; Hilderink and Broenink, 2003], and various others. These embeddings have generally suffered from being unable to enforce rules for safe concurrent behaviours (e.g. preventing shared mutable data between processes) as well as having awkward representations for processes (e.g. each process must be a new class) and other subtle issues [Brown, 2007].

Haskell is a functional programming language that is notable for its purity (i.e. careful control of side effects), type-classes, monads and lazy evaluation. Like other functional languages, it has type inference and it eliminates mutable data. Haskell's features, especially its careful containment of side effects, make it easy to write embedded domain specific languages [Stewart, 2009].

This thesis contends that CSP and process-oriented programming fit well with Haskell. The thesis details the creation of a CSP library for Haskell named Communicating Haskell Processes (CHP: chapter 3) – which features powerful support

for process composition (chapter 4) – and its further augmentation with capabilities such as support for a new conjunction primitive (chapter 5), tracing (chapter 6) and the ability to generate formal models of CHP programs (chapter 7).

The thesis makes the argument that Haskell is a suitable language in which to embed process-oriented programming, and explores several new features not previously available in any other process-oriented programming system. This is informative for process-oriented programmers, but is also relevant to those interested in concurrent programming in Haskell.

1.1 Contribution

This thesis makes the following contributions:

1. **Development of a library for Haskell that supports CSP-style concurrency.** This thesis details the design rationale of the CHP library that allows CSP-like concurrency to be used in Haskell (chapter 3).

2. **Development of new techniques for composing processes together.** This thesis explores new operators and functions to allow concise and powerful composition of processes (chapter 4).

3. **Invention of conjoined events.** This thesis introduces into process-oriented programming the idea of waiting for conjoined events, i.e. waiting for both a and b to occur, and only engaging in both or neither (chapter 5).

4. **Algorithms for channels, choice and conjunction using Software Transactional Memory.** This thesis explains how to use software transactional memory to implement the algorithms required to support the types of communication and choice that CSP provides (chapter 5).

5. **Invention of structural traces.** This thesis introduces a new style of trace that preserves more information about the history of a concurrent computation (chapter 6).

6. **Algorithms for recording traces.** This thesis presents algorithms for recording CSP, VCR and structural traces that can be used by any concurrent run-time (chapter 6).

7. **Technique for generating CSP models of CHP programs.** This thesis explains a lightweight approach to generating CSP models of CHP programs without the need for source-code analysis (chapter 7).

Chapter 2

Background

Communicating Sequential Processes (CSP) is a process calculus developed by Hoare and Roscoe [Hoare, 1985; Roscoe, 1997]. It provides a formal basis for describing concurrent message-passing systems. The FDR model-checker was later developed to allow automatic refinement checking of a CSP system against a specification [Roscoe, 1994; Formal Systems (Europe) Ltd, 1997].

The principles and concurrency model of CSP have been made available to programmers in several ways. The occam programming language was faithfully based on CSP [INMOS, 1985]. More recently, libraries which provide the CSP concurrency model have been developed for Java [Welch, 1997; Hilderink et al., 1997], C++ [Brown and Welch, 2003; Hilderink and Broenink, 2003] and many other similar imperative languages.

The Haskell programming language was created to unify research into functional programming [Peyton Jones, 2003; Hudak et al., 2007]. Its distinguishing features were its purity, lack of side effects, and lazy evaluation – the opposite of imperative languages such as C. However, the important concept of monads allowed imperative-style sequencing of operations to be added to Haskell [Jones and Wadler, 1993]. Haskell has support for basic concurrent primitives [Peyton Jones et al., 1996], and elegant support for Software Transactional Memory [Harris et al., 2005].

This thesis investigates embedding the CSP concurrency model into Haskell, in a library entitled Communicating Haskell Processes (CHP). Haskell provides a very different host language than previous work in the area which used languages such as Java. Haskell's functional nature may make it seem an odd fit for an imperative concurrency model, but this thesis argues that Haskell meshes well with CSP's declarative nature, and that Haskell's monads (amongst many other

features) mean that it is a good language to combine with CSP, allowing high-level approaches to process-oriented programming not seen in other frameworks or languages.

This chapter explores the background in the two areas: firstly, CSP and its implementation in various languages (sections 2.1–2.8), and secondly approaches to concurrency in functional languages such as Haskell (sections 2.9–2.15).

2.1 Communicating Sequential Processes

CSP is centred around the idea of events. Events may involve only a single process, but typically they are shared between several concurrent processes. These become synchronising events: for example, if two concurrent processes share the event a, then when either wants to complete event a, they must wait until the other process is also ready to do so.

Processes can be composed in various ways: sequentially, concurrently and using choice. The sequential composition of processes P and Q is written $P \fatsemi Q$ – when P completes successfully, Q will execute. The concurrent composition of processes P and Q is written $P \underset{\Sigma}{||} Q$ where Σ is the set of shared events.

The two primitive processes are SKIP and STOP; the former terminates successfully, the latter does not terminate. Processes may be prefixed by an event synchronisation, for example $a \rightarrow \text{SKIP}$ is the process that synchronises on event a and then terminates successfully. This operator is right-associative, which allows for processes such as $a \rightarrow b \rightarrow c \rightarrow \text{SKIP}$.

Processes can be composed using external choice: $P \Box Q$ is the external choice between P and Q. External choice between two processes means waiting for the first prefixed event in either process to become available. If $P = a \rightarrow P'$ and $Q = b \rightarrow Q'$, $P \Box Q$ will wait for either event a or event b to complete, and then continue behaving as P' or Q' respectively. With external choice, if only one event is available to complete, the process is not permitted to refuse, and must engage in that event. That is, if only a is available, $P \Box Q$ will engage in a and then behave as P'. If both are available, the choice between a and b is arbitrary.

The complement to external choice is internal, or non-deterministic choice, written $P \sqcap Q$. The choice is similar, but a process offered one option of an internal choice is not obliged to engage in it. Only if all options are offered is it obliged to take one of them. That is, if only a is available, $P \sqcap Q$ may do nothing (i.e. refuse a), but if both a and b are available, the process must engage in one

of them.

The final key part of CSP is channels. Theoretically, a channel can be conceived of as a set of events (that may be infinite). For example, a channel c carrying non-negative integers can be thought of as the set of events $\{c.0, c.1, c.2, \ldots\}$. In practical terms, a communication channel is a way of synchronously passing messages from one process to another.

2.2 occam 2.1

The occam 2.1 programming language [INMOS, 1985], which succeeded and superseded the previous revisions, was strongly influenced by Hoare's work on CSP. The language was designed with concurrency and synchronous communication channels at its heart – this came from a strong basis in Hoare's theory but was also a good match for embedded devices and the underlying chips connected by wires.

The occam language has various standard programming concepts: the while loop, if-then statements, case statements and statically scoped variables. There are no pointers or references, but instead scoped aliases can be declared that introduce a new name for an existing variable, or more usefully a reference to a particular part of an existing variable (e.g. a particular index or slice of an array).

The occam language's support for concurrency is obvious in its two main block types. Preceding an indented section[1] with `SEQ` caused the commands to be executed in sequence, whereas using `PAR` in its place would cause the commands to be executed concurrently. This placed support for imperative concurrency at the heart of the language. For example, this code runs process `P`, followed by `Q` and `R` in parallel – and once they both terminate, process `S` runs:

```
SEQ
  P()
  PAR
    Q()
    R()
  S()
```

This support for concurrency would only have (potentially) benefited performance, had it not also featured additional constructs to allow interaction between

[1] occam is an indentation-based language.

concurrently executing components. The designers of occam chose to ignore the general case of supporting CSP events, and instead specifically included communication channels.

Whereas Hoare's channels were simply sets of events, occam took the more practical route of having typed, synchronous communication channels that could carry any value of a declared type. This could be primitive types (integers, characters, etc), compound types (arrays, records, etc) but not channels, processes, functions or procedures. In C++ terminology this style of restriction is known as "plain old data". Protocols containing any specified sequence of these types were also permitted.

The commands for sending and receiving on these channels were given simple syntax. The command `c ? x` received a single data value from the channel `c` into the variable `x`. Correspondingly, `c ! x` sent a single data value `x` on the channel `c`.

The most powerful construct of occam was its support for choice via the `ALT` command. Without choice, communication channels are much less useful – a process in the system would always need to know on which channel it must communicate next, which typically restricts programs to simple pipelines or one-to-one client-server relationships. `ALT` allows a program to offer to read from one of a set of channels, choosing the next one with a process waiting to write to the other end of the channel. Choices between multiple such ready processes could either be resolved non-deterministically or with a priority ordering. For example, this code offers to read a value from channel `c` or `d`; in the former case it forwards the value on channel `e`, and in the latter it does the same but adds one to the value (each indented item in an `ALT` is a guard, followed by a further indented associated body to execute if the guard is chosen):

```
ALT
  c ? x
    e ! x
  d ? x
    e ! (x + 1)
```

This choice was actually only a subset of Hoare's choice in CSP; CSP allows choice over possible writes to channels or a mix of reads and writes, but occam only allowed choice between reads. This was done for implementation reasons, since the only way to allow choice at both ends of the same channel (a potential consequence of allowing choice on writes as well) in a distributed system was to

use an expensive two-phase commit protocol for all communications (even those not involving such choice).

The occam language's adherence to the theory was strong enough that Hoare and Roscoe were able to construct a paper filled with laws of occam, including transformations and equivalences [Roscoe and Hoare, 1988].

An additional interesting feature of occam 2.1 was its support for replication. Any `ALT`, `IF`, `SEQ` or `PAR` block could be replicated. For example:

```
SEQ i = 0 FOR 4
  P(i)
```

Is exactly equivalent to:

```
SEQ
  P(0)
  P(1)
  P(2)
  P(3)
```

Hence, a replicated `SEQ` block is the equivalent of a for loop from other languages. Since replication in occam 2.1 is primarily a compile-time substitution it is mainly useful to save repetition in the program code.

2.2.1 Critique

The occam 2.1 language was very static in several aspects. This was mainly due to practical considerations; it was necessary to know at compile-time how much memory each process would require, and how the external channels would be wired.

As a consequence, array sizes had to be static in occam and there was no support for recursion (both of which would have required dynamic memory). Dynamically-varying replication limits for sequential code were permitted as this would not require any additional memory, but the replication limits for parallel replicators had to be known at compile-time.

The process network in an occam program was also static. Processes could be created and finish during the course of a program, but channels could not dynamically be moved among processes, nor sent over channels. Thus, the externally-visible channels that a procedure was passed as its parameters were the only external channels it could ever use: any channels it created could only be used internally by subcomponents.

2.3 occam-π

The occam-π language was developed at the University of Kent [Welch and Barnes, 2005], taking inspiration from Milner's π-calculus [Milner, 1999] to introduce the concept of mobility and remove all these static constraints that were not necessary on modern computers which could easily support dynamically allocated memory.

Mobility is the idea of moving, or transferring, *mobile* resources between processes. Standard assignment and communication in occam had copy semantics, which worked simply for block-copying data, but could not be used to send references or channels without introducing the possibility of aliasing. If process C copies a resource reference to process D (be it data, or a channel) there is no easy way to prevent C using its reference in parallel with D using the copy of the reference.

Mobile resources are not copied, but moved. So when a reference to a mobile item is assigned from or sent down a channel, the source/sender loses the reference. This is enforced by the compiler: the reference is moved from one place to another each time, and thus only a single reference ever exists to the mobile resource. This prevents the possibility of concurrent aliasing, and is also efficient for sending large pieces of data (since they are sent by reference rather than block-copied).

The occam-π language supports mobile data, mobile channels and mobile processes: these are the primary changes to classical occam. Other changes include the introduction of channel bundles (tying several channels together for ease, typically a request and response channel) and other dynamism such as recursion and forking.

2.3.1 Critique

The occam-π language is at the forefront of message-passing concurrency research, but several other aspects remain dated in their occam 2.1 origins – such as the type system. Passive data structures such as doubly-linked lists and associative maps are difficult to define in the language with mobile data, and arrays remain the central data structure. Although tagged protocols bear a similarity to discriminated unions, the latter are also not present in occam-π. There is also no support for generics or templates in occam-π (discussed later in this chapter).

2.4 Process-Oriented Programming

The ideas behind CSP and occam have come to be known as process-oriented programming. No formal definition has been given for this term, but for the purposes of this thesis, the primary features of process-oriented programming are identified as follows:

- Several processes executing concurrently, without shared mutable data.
- Communication and synchronisation is made possible by typed anonymous communication channels (generally synchronised and unbuffered), and optionally via barriers.
- Choice is supported: at least the choice between reading from different channels, but also potentially full choice (which also includes writing to channels, and synchronising on barriers).

2.5 JCSP, C++CSP and more

The ideas of process-oriented programming evolved with the occam programming language, but there is no reason to *require* occam for process-oriented programming. The occam language does provide various safety guarantees for concurrent programs, but leaving those aside, the only requirement to implement process-oriented programming in other languages is a concurrent run-time supporting processes, channels and choice.

This allowed the development of JCSP, a Java library that provided support for processes (loosely layered on top of Java's threads), channels, barriers, choice and various other process-oriented features, built using Java monitors [Welch, 1997]. A similar library, CTJ, was developed around the same time [Hilderink et al., 1997].

After the Java libraries, two C++ libraries (C++CSP [Brown and Welch, 2003] and CTC++ [Hilderink and Broenink, 2003]) were developed. These followed roughly the same format as the Java libraries, but also took advantage of some of the appropriate features of C++ (such as templates). These have been followed by a slew of similar libraries for other languages, such as the .NET languages [Lehmberg and Olsen, 2006; Chalmers and Clayton, 2006], Python [Bjørndalen et al., 2007], Groovy [Kerridge, 2005] and Scala [Sufrin, 2008].

Each of these libraries has been unable to provide the safety guarantees that the occam compiler is able to (freedom from race hazards and concurrent aliasing). But they were each able (or forced) to blend the ideas of process-oriented programming with the particular host language. In the following subsections we describe the aspects of the different host languages that provided interesting features in a process-oriented setting.

2.5.1 Java

Java (prior to version 1.5) had no support for generic programming. Types were split into primitive types (integers, floating point numbers) and objects. All classes inherit from `Object`, so all the collection classes held collections of `Object`s which were down-cast where necessary in a user's program.

To provide channels, without needing to provide a separate class for every type in a user's program (many of which would not be known to the JCSP designers), JCSP provides a specialised integer channel and a general channel for `Object`s. The net effect was to allow dynamically-typed channels that could carry any object; usually these were used as channels that carried a single specific type, but a new feature had been added by the restrictions of the Java type system. (This feature has recently been corrected by the addition of generic channels to JCSP.)

2.5.2 C++

C++ is a superset of the C programming language that adds support for object-oriented programming and generic programming. It was developed in 1979 and the current latest standard was set in 1998.

The support for concurrency in the C++ language is equivalent to that of C; the only language feature at all related to concurrency is the keyword `volatile`, which is often used to mark an address in memory as potentially used in parallel. The utility (and the unclear concurrent semantics) of this keyword have been informally discussed by Robison [2007] and Boehm and Maclaren [2006]. This feature highlights problems with the underlying memory model [Boehm, 2005], which is important in languages with memory that can be shared between processes executing in parallel. Work continues to specify (at least partially) the memory model [Boehm and Adve, 2008].

Templates

The availability of templates in C++ (and lack of generics in Java) at the time of the development of the C++CSP library [Brown and Welch, 2003] made C++ a more suitable host language. The problem with providing a channel class for every type was moot; a single templated version of a channel class was provided, which could then be instantiated by users for whatever type they wished. Templated channels are effectively a C++ version of occam's channels, which could similarly carry any single type.

There are several basic processes that are used frequently in process-oriented programming. One of these is the identity process, that perpetually forwards values from one channel to another. In occam, this process must be written repeatedly, once for each type. So while the channels can carry any type (and are first-class types), a process must deal with a specific type.

In C++, just as the channels are templated, so the processes can be. The identity process can also be made a template, and can be instantiated by users for any type that they want. This reduces code duplication (and thus maintenance effort) and faciliates greater code re-use.

Poison

Sequential programs terminate when the top-level function exits. The same applies for concurrent programs, but the concurrent network of components that make up a process-oriented program can make termination difficult. For example, if two processes are connected by a channel and one process terminates, the other will deadlock if it attempts to communicate further on that channel (rather than also terminating).

Solving the termination problem using standard channel communications is difficult without unintentionally provoking deadlock [for a detailed explanation, see Welch, 1989b]. C++CSP introduced the idea of stateful poison: channels can be placed into a poisoned state by the reader or the writer. This change is permanent (channels can never be cured of poison) and repeating poisoning has no further effect.

Any attempt to read from or write to a poisoned channel will cause a poison exception to be thrown. In C++CSP, this was tied in with the exception mechanisms of the C++ language. Every process would execute its code inside a `try` block, and upon `catch`ing a poison exception, would poison all its channels and

terminate. Poison would thus spread about the network, jumping from channel to channel via processes which would then shut down. At the conclusion of this, every process should be shut down, and all channels poisoned.

2.5.3 Problems of Process-Oriented Programming in C++

There are several issues with programming in C++CSP, where mistakes can be made with respect to concurrency that cannot be detected by the compiler. These are representative of the problems of embedding a concurrent programming style into a language not designed for it. These problems were originally discussed in a publication at Communicating Process Architectures 2007 [Brown, 2007].

Channel Destruction

There are often situations in which the user of C++CSP will want to have a single server process serving many client processes. Assuming the communication between the two is a simple request-reply, the server needs some way of replying specifically to the client who made the request. One of the easiest ways of doing this is for the client to send a reply channel-end with its request (channel-ends being inherently mobile in C++CSP):

```
//requestOut is of type Chanout< pair< int,Chanout<int> > >
//reply is of type int
{
    One2OneChannel<int> replyChannel;
    requestOut << make_pair(7,replyChannel.writer());
    replyChannel.reader() >> reply;
}
```

The corresponding server code could be as follows:

```
//requestIn is of type Chanin< pair< int,Chanout<int> > >
pair< int, Chanout<int> > request;
requestIn >> request;
request.second << (request.first * 2);
```

For this trivial example, requests and replies are integers, and the server's answer is simply double the value of the request.

In the original C++CSP, this seemingly valid code was potentially unsafe. The following sequence of events was possible:

1. Client sends request, server receives it.
2. Server attempts to send reply, must block (waiting for the client).
3. Client reads reply, adds server back to the run-queue.
4. Client continues executing, destroying `replyChannel` (because it goes out of scope).
5. Server wakes up and needs to determine whether it woke because it was poisoned or because the communication completed successfully. To do this, the server must check the poison flag; a member variable of the now-destroyed `replyChannel`.

This situation thus leads to the server checking a flag in a destroyed channel. The algorithm was later corrected [Brown, 2007], but this demonstrates that destruction order in the prescence of concurrency can be difficult.

Scoped Forking

Scope is a useful part of structured programming. In most languages, variable storage is allocated when variables come into scope and de-allocated when variables go out of scope. In C++ classes this concept is built on with constructors (executed when an object variable comes into scope) and destructors (executed when an object variable goes out of scope). This feature, which is not present in Java in the same way, can be both useful and dangerous in the context of C++CSP. Both aspects are examined in this section.

C++CSP takes advantage of the scope of objects to offer a `ScopedForking` object that behaves in a similar manner to the `FORKING` mechanism present in occam-π [Barnes and Welch, 2002]. In occam-π, one might write:

```
FORKING
  FORK some.widget()
```

In C++CSP the equivalent is:

```
{
    ScopedForking forking;
    forking.fork(new SomeWidget);
}
```

The name of the `ScopedForking` object is arbitrary (`forking` is as good a name as any). At the end of the scope of the `ScopedForking` object (the end of the block in the above code), the destructor waits for the forked processes to terminate – the same behaviour as at the end of the `FORKING` block in occam-π.

The destructor of a stack object in C++ is called when the variable goes out of scope – this could be because the end of the block has been reached normally, or because the function was returned from, or an exception was thrown. In these latter two cases the destructor will still be executed.

For example:

```
{
    ScopedForking forking;
    forking.fork(new SomeWidget);

    if (something == 6)
        return 5;

    if (somethingElse == 7)
        throw AnException();
}
```

Regardless of whether the block is left because of the return, the throw, or normally, the code will only proceed once the `SomeWidget` process has terminated. Using such behaviour in the destructor allows us to emulate some language features of occam-π in C++, and even take account of C++ features (such as exceptions) that are not present in occam-π. However, there is one crucial difference – the occam-π compiler understands the deeper meaning behind the concepts, and can perform appropriate safety checks. In contrast, the C++ compiler knows nothing of what we are doing. There are two mistakes that can be made using the new `ScopedForking` concept, which are explained in the following two sub-sections.

Exception Deadlock: Consider the following code:

```
One2OneChannel<int> c,d;
try
{
    ScopedForking forking;
    forking.fork(new Widget(c.reader()));
    forking.fork(new Widget(d.reader()));
```

```
    c.writer() << 8;
    d.writer() << 9;
}
catch (PoisonException)
{
    c.writer().poison();
    d.writer().poison();
}
```

At first glance this code may seem sensible. The try/catch block deals with the poison properly, and the useful `ScopedForking` process makes sure that the sub-processes are waited for whether poison is encountered or not. Consider what will happen if the first `Widget` process poisons its channel before the example code tries to write to that channel. As part of the exception being thrown, the program will destroy the `ScopedForking` object *before* the catch block is executed. This means that the program will wait for both `Widget`s to terminate before poisoning the channels. If the second `Widget` is waiting to communicate on its channel, then deadlock will ensue.

This problem can be avoided by moving the declaration of the `ScopedForking` object to outside the try block. The general point, however, is that the C++ compiler can offer no protection against this mistake. In a purpose-designed language that offered both concurrency and poison exceptions, the compiler could avoid such problems by detecting them at compile-time, or by ensuring that all catch blocks for poison are executed before the wait for sub-processes.

Order of Destruction: Consider the following code:

```
{
    ScopedForking forking;
    One2OneChannel<int> c,d;
    forking.fork(new WidgetA(c.reader(),d.writer()));
    forking.fork(new WidgetB(d.reader(),c.writer()));
}
```

This code creates two processes, connected by channels, and then waits for them to complete. This code is unsafe. In C++, objects are constructed in order of their declaration. At the end of the block, the objects are destroyed in reverse order of their declaration. This means that at the end of the block in the above code, the channels will be destroyed, and then the `ScopedForking` object will be

destroyed. So the processes will be started, the channels they are using will be destroyed, and then the parent code will wait for the processes to finish, while they try to communiate using destroyed channels.

Again, this problem can be avoided by re-ordering the declarations. This code is dangerous (in the context of our example):

```
ScopedForking forking;
One2OneChannel<int> c,d;
```

This code is perfectly safe:

```
One2OneChannel<int> c,d;
ScopedForking forking;
```

The subtle difference between the two orderings, the non-obvious relation between the two lines, and the ramifications of the mistake (in all likelihood, a program crash) make for a subtle error that again cannot be detected by the C++ compiler. In languages such as occam-π, this mistake can be easily detected at compile-time (variables must remain in scope until the processes that use them have definitely terminated) and thus avoided.

The documentation for the C++CSP library explains these pitfalls, and offers design rules for avoiding the problems in the first place (for example, always declare all channels and barriers outside the block containing the `ScopedForking` object). The wider issue here is that adding concurrency to existing languages that have no real concept of it can be dangerous. Compile-time checks are the only real direct defence against such problems as those described here.

2.5.4 Mutability

The specific problems for Java and C++ have been discussed without mentioning the main problem with embedding process-oriented concurrency into such languages. Our definition of process-oriented programming in section 2.4 stated: "Several processes executing concurrently, without shared mutable data". Shared mutable data is the default in imperative languages such as Java and C++. Preventing it is impossible, and even if users do not intend to use shared mutable data, pointers and uncontrolled aliasing mean that processes can accidentally end up sharing mutable data. Problems with shallow cloning versus deep cloning abound. There are only three ways to outlaw shared mutable data between threads: to outlaw sharing of mutable data, to outlaw all shared data, or outlaw all mutable data.

The first two require special compiler knowledge of concurrency; the third leads to functional programming.

2.6 Honeysuckle

Honeysuckle [East, 2003] is a programming language intended to fix some of the shortcomings of occam, and is focused on the idea of using services (client/server relationships) to remove deadlock. Honeysuckle features a form of explicit mobility for transfer of ownership of objects. While mobile data is not a distinct type from other data (so one may say that all data is *implicitly* mobile), transfers are done using explicit syntax that is different from copying. Exceptions are featured in the language as asynchronous seemingly-external events delivered to the process which has encountered the exception. Honeysuckle currently exists only as a specification, without an implementation.

2.7 Polyphonic C#

Polyphonic C# [Benton et al., 2004] comprises two additions to the C# language, based on the join calculus [Fournet and Gonthier, 2000]. The first is asynchronous method calls. An asynchronous call returns no value and cannot propagate any exceptions. It is essentially a piece of code to be run on the object at some later time in another thread. It is not completely clear whether or not these asynchronous methods are prevented from running on the same object simultaneously with other synchronous and/or asynchronous methods – presumably this is prevented.

The more interesting feature is termed a 'chord'. A chord is a group of methods, with exactly one synchronous method, and one or more asynchronous methods. The synchronous method will not execute until all of the asynchronous methods have also been called. Multiple asynchronous calls may have been made to the asynchronous methods, and no guarantee is provided on ordering. For example, a buffer can be written by having a chord with a synchronous get method, and an asynchronous put method. The body of the chord merely returns to the get method the value provided by the put method. If multiple put calls are pending, there is no guarantee as to which is used for the next get call.

In terms of message-passing, a chord could be implemented by having the caller of the synchronous method read a message from each of the asynchronous callers, and then executing the synchronous method using that data.

2.8 Sing#

Sing# [Fähndrich et al., 2006] is the programming language invented for the Singularity Operating System [Hunt et al., 2005]. Like Polyphonic C#, Sing# is an extension of the C# language, but one that uses channels for message-passing.

A channel type in Sing# is represented by its contract: a list of messages that can pass across the channel, the constraints on the data and the order(s) in which the messages may be sent. The Sing# compiler performs static checks at compile-time that ensure type safety and protocol conformance.

Channel communication is not provided via explicit send and receive operations as in many other languages (such as occam). Instead, each message that could be sent is associated with a method call on the writing end, with the data to be sent as parameters. Similarly, each message that could be received is associated with a method call on the reading end that returns the data read.

These method calls on the channel ends are used directly by the writing end. However, the process using the reading end will rarely know exactly which message it might receive next (dependent upon the protocol of the channel) so a '`switch receive`' statement is provided that is very similar to the `ALT` statement in occam. It allows selection between receiving different types of messages from a single channel, or even from multiple channels.

2.9 Erlang

Erlang is a programming language built on the ideas of message passing [Armstrong et al., 1993]. Superficially similar to CSP, Erlang's communication has its roots in the Actor model [Hewitt et al., 1973]. Rather than synchronous communication over typed point-to-point channels, it uses asynchronous untyped commmunication with addressed mailboxes. Each process has a unique identifier that can be used to send messages to its mailbox.

This addressing approach alleviates the complexities of shared channels and reduces the need for typed channels. All a process needs to send a message is the address of the recipient, and the sender can include its own address in the message to facilitate replies. However, this explicit addressing of processes also loses some of the compositionality of process-oriented programming. When a process is given a channel in a language such as occam, it can either use the channel directly, or pass it to a sub-process that is part of a parallel composition. If the channel is

mobile, the reading end could be sent to another process without the sender being aware, and without requiring the original recipient to continue executing. With a direct addressing system, these indirections must be programmed explicitly (with code to forward messages).

There is little idea of a process network in Erlang. Process identifiers can be sent around unrestricted, so any process may send a message to any other at any time. This can make it hard to conceptually separate a system into subcomponents since it is hard to tell which parts of these subcomponents will be hidden from the outside world.

Erlang does not require a choice operator; when a process waits for the next message from its mailbox, it is implicitly waiting for a message from *any* process. Choice over outputs becomes inapplicable because outputs do not need to wait for inputs, so they can never not be ready. This simplifies the implementation of Erlang, by removing the need for complicated `ALT` algorithms.

2.10 Concurrent Haskell

Concurrent Haskell [Peyton Jones et al., 1996] is a library for the Haskell language that adds message-passing concurrency. The designers decided against an equivalent of occam's `PAR` construct as being "extremely inconvenient" (in the context of Haskell). Instead they use a forking mechanism that is effectively fork-and-forget.

The central communication mechanism in Concurrent Haskell involves using shared variables with atomic operations. Peyton Jones et al. [1996] demonstrate how asynchronous single-item and multi-item buffered channels can be built on top of these shared variables. They explicitly discuss the possibility of including a choice mechanism (even specifically mentioning occam's `ALT` statement). They decide against including one, due to implementation issues if both sides can have choice (the reason why occam only allows input guards) but also that such choice mechanisms are rarely used to their full. As further justification they cite Reppy's arguments on wrapping protocols, as described in section 2.12 [Reppy, 1995].

The authors of the paper show how shared channels can be built, and hence how shared channels can be used to process input from multiple sources. This is not the choice that `ALT` provides of choosing A or B or C. Instead, it is a matter of choosing A *and* B *and* C, but with the order undefined. The authors claim that the latter option is the most used form of choice, and that the behaviours that only the former can offer are rarely used.

2.11 Software Transactional Memory

More recent work in implementing concurrency in Haskell has focused on Software Transactional Memory (STM) [Harris et al., 2005]. STM is a vast improvement on the idea of locks for shared data. Locks have many well-recognised problems, particularly the deadlocks that can arise from claiming the locks, the race hazards that can arise from not claiming the locks, and the difficulties of dealing with multiple locks.

Software Transactional Memory focuses not on the individual locks, but on transactions: sequences of events on special transactional variables that must be performed atomically. The concept of the atomic transactions is to not allow any intermediate or partial states to be visible to other threads. Other threads can only observe the state before a transaction begins, or after it completes.

STM avoids many of the traditional problems of locks: in particular STM eliminates deadlocks (if the Haskell-only `retry` primitive is omitted). Although STM transactions may have to restart when conflicts occur, at least one thread must be making progress in order for the conflicts to occur. STM does retain the possibility of starvation, however. A transaction that is long-running may continuously conflict with other shorter transactions.

STM provides a different model of concurrency to message-passing, as it is focused on data manipulation. However, with Haskell's `retry` primitive it is possible to use STM to implement communication channels. Haskell also permits choice between transactions in STM. Several transactions can be joined with choice; if the first explicitly retries, the second is attempted, and so forth.

2.12 Concurrent ML

Concurrent ML was developed by Reppy [1995]. Reppy argues that mechanisms such as `ALT` actually combine four aspects: channel communications (the guards), the associated actions (bodies) for each, the choice between them, and the synchronisation (of waiting for the action to complete). He attempts to split these aspects up so that they can each be used independently or in combination.

The split that Reppy envisages is not possible in occam, nor in many other languages, but is well-suited to the more declarative style of programming that ML and Haskell provide; a Concurrent ML library has recently been developed for Haskell [Chaudhuri, 2009]. In Haskell terms, Reppy's work could be considered

to have types along these lines:

```
readChannel :: Chanin a -> Action a
choose :: [Action a] -> Action a
sync :: Action a -> IO a
```

Reppy also introduces a concept of negative acknowledgements. When a channel read is considered as part of a choose statement (the choice aspect) but is *not* chosen, a process waiting to write on that channel can be sent a negative acknowledgement – effectively a "you were not chosen" message. The idea is that if a process does not choose a particular communication, it is no longer interested in it; no discussion is given to the possibility that the choosing-process's next action will be to choose again, and this time may choose the communication.

2.13 Transactional Events

Reppy argues that selective communication is incompatible with abstraction. As an example, he explains how a request-reply protocol can be combined into a single compound function. Selective communication can only compose primitive communications (e.g. the request) – not a compound function (the request plus reply). This means, for example, that once the server is waiting to send the reply, it will not be able to choose to instead engage in other communications.

Long actions such as this would usually be composed in occam using the `PAR`allel constructor, not the `ALT`ernative constructor. An alternative is used when a program wants to choose only one of a number of options. But if a server is waiting to send a reply and wants to process other communications as well, then performing the separate operations in parallel would be more appropriate.

This criticism was re-iterated in work by Donnelly and Fluet [2006], who implemented a library for Haskell that allowed such choice between compound sequences of events. This work was in effect a hybrid of STM and process-oriented concurrency, supporting transactions made up of synchronisations (as opposed to reads and writes as in the case of STM).

One restriction of the transactional events approach is that transactions must align. For example, if one process P wishes to perform a transaction with event a followed by event b, and a second process Q tries to perform two corresponding transactions, the first with a and the second with b, deadlock will ensue: process Q will be unable to complete its first transaction until P's transaction is completed,

which in turn will wait for Q's *second* transaction. Thus, the events within a transaction must become part of a defined protocol, cf. work on session types.

2.14 Process-Oriented Concurrency in Haskell

One early work that presages this work is that of Hill [1995]. This work showed a basic monad-based API that is very similar to the CHP API that will be introduced in chapter 3. The original paper focused heavily on the channel usage checker, but the basic idea of using monads for imperative message-passing concurrency is one that is continued in this thesis (and greatly expanded).

2.15 Data Parallelism and Implicit Parallelism

This thesis is concerned with mechanisms for explicit *concurrency*. There is also other support for *parallel* execution in Haskell, particularly: explicit parallelism using the `par` combinator, implicit parallelism and nested data parallelism. These are briefly described here for completeness; Hammond [1994] provides a more complete overview of more approaches.

Explicit parallelism retains the purity of functional programming but adds hints to the compiler as to when it would be beneficial to evaluate two arguments in parallel. Being in a functional language, any evaluation order is permitted, so the run-time can perform a parallel evaluation without danger (contrast this to trying to parallelise two expressions in C). A combinator may be of the form: `par :: a -> b -> (a, b)` which appears to merely pair two arguments, but also indicates that the expressions should be evaluated in parallel. As Hammond [1994] points out, the level of granularity is crucial (this is applicable to all attempts to gain parallel speed-up). Too many annotations can cause the program to suffer from the overheads involved in the parallelism; too few will not take sufficient advantage of available multi-processors.

Implicit parallelism attempts to resolve the granularity problem automatically. Implicit parallelism refers to the process of adding in the `par` annotations automatically, either dynamically or statically. A recent example of such research is by Harris and Singh [2007], who attempted to use feedback from running the program to decide where to place the annotations. Their results showed a 10–70% speed-up at best, even on four cores, highlighting the difficulty of attempting to automatically choose the correct points for introducing parallelism.

Data Parallel Haskell is a small extension to the language that supports nested data parallelism [Chakravarty et al., 2001]. It adds a new list-like type: a parallel array. Many list functions (including list comprehensions) are offered for parallel arrays. A parallel array is *not* constructed inductively, and can be evaluated in parallel. This simple addition allows many uses of Haskell lists to be transformed into uses of parallel arrays, with the accompanying opportunity for parallelism when the program is run.

2.16 Summary

Process-oriented programming has previously been embedded in several different languages (e.g. occam), and libraries in host languages (e.g. C++CSP, JCSP, CML). The commonality between most of these languages is that they are imperative languages, with destructive updating. The idea of embedding process-oriented programming in a functional language has only been explored in CML. Erlang does not meet the definition of process-oriented programming given in section 2.4 due to its use of an addressed-mailbox model for message-passing.

One of the core traits of process-oriented programming is the absence of shared mutable data. Process-oriented libraries in imperative host languages cannot stop mutable data being shared between processes – and typically permit concurrent aliasing – which can lead to unintentional mutation of shared data structures by multiple processes. In non-garbage collected languages such as C++, subtle bugs can arise from the premature destruction of the library's concurrent primitives.

Such problems can be avoided in specialised languages – the occam-π compiler is able to prevent most of these problems. However, occam-π suffers from a lack of uptake. This is partly due to the chicken-and-egg problem of any small language: until it has a rich set of libraries (or interfaces to existing libraries in other languages), users will be hesitant to take it up, but to get such libraries on any scale, many users are required. The other reason is that while occam-π has very good concurrent features, it lacks good support for data structures such as linked lists or associative maps (partly due to the restricted aliasing in the language), and lacks support for generic programming and other such modern language features.

The work in Haskell and other functional languages (such as CML) shows that concurrency – including message-passing concurrency – can be added to functional languages successfully. This existing work provides the basics – running processes

in parallel and communicating over channels. Features such as barriers or common processes are not present. This thesis demonstrates what can be done by using concurrency in functional languages as a powerful basis.

This thesis proposes that embedding process-oriented programming in Haskell results in a useful and powerful library, that can both:

- provide interesting lessons for process-oriented programming, by showing how Haskell features and idioms can benefit concurrent programming, and
- provide a powerful alternative to other Haskell concurrency mechanisms, show how process-oriented programming compares to other concurrency approaches, and show how it can be built on to provide capabilities over and above all other message-passing concurrency libraries.

Chapter 3

Communicating Haskell Processes

This chapter details the Communicating Haskell Processes (CHP) software library (for Haskell) developed as part of this thesis. The chapter explores the embedding of CSP and process-oriented concepts into Haskell using a monadic approach. A brief introduction to Haskell is given in appendix B for those not familiar with the language.

The CHP library explored in this section forms the basis for most of the remainder of the thesis; the later chapters primarily explain additions to the CHP library that is presented in this chapter. An early version of the work in this chapter was published in the proceedings of Communicating Process Architectures 2008 [Brown, 2008].

3.1 Communicating Haskell Processes Concept and IO

Communicating Haskell Processes is a library that provides imperative process-oriented concurrency in Haskell. It supports parallel composition, synchronous message-passing on communication channels, multiway synchronisation on barriers, choice between these communications and synchronisations and other features besides.

At the core of the library is a CHP monad, which is a thin layer on top of Haskell's IO monad. Thus, it is possible to use the full range of IO interactions with the outside world during a CHP program, using a simple "lifting" function:

```
liftIO_CHP :: IO a -> CHP a
```

We return to its definition later. The Communicating Haskell Processes (CHP) library can thus be thought of either as a way to program process-oriented programs with access to IO, or as a way to augment Haskell IO programs with concurrency. At the top-level, CHP programs are run using the runCHP function:

```
runCHP :: CHP a -> IO a
```

To give a flavour of the library (ahead of introducing it properly), we present a short sample program that continually reads lines from stdin and prints out only the lines that feature the string "caterpillar":

```
import Control.Concurrent.CHP
import Control.Monad
import Data.List

filterLines :: Chanin String -> Chanout String -> CHP ()
filterLines input output = forever $ do
  x <- readChannel input
  when ("caterpillar" `isInfixOf` x)
       (writeChannel output x)

complete :: CHP ()
complete = do
  (rA, wA) <- newChannelRW
  (rB, wB) <- newChannelRW
  parallel_
    [ forever (liftIO_CHP getLine >>= writeChannel wA)
    , filterLines rA wB
    , forever (readChannel rB >>= liftIO_CHP . putStrLn) ]

main :: IO ()
main = runCHP complete
```

This program uses only the core primitive operations in CHP, which we will progressively introduce and explain in this chapter. The basic structure of a CHP program is shown; there is a top level process that takes several channels as arguments and has a behaviour in the CHP monad. This is then composed in a parallel composition with some other behaviours (which are written inline in the parallel composition) by connecting them with some allocated channels. Using the common processes and composition operators (which will be introduced in

	x, y	$\in$	*Variable*
Values	V	$::=$	$\backslash x \to M \mid l \mid con\ M_1 \ldots M_n$
			$\mid \text{return}\ M \mid M >>= N \mid \ldots$
Terms	M, N, O	$::=$	$x \mid V \mid M\ N \mid \ldots$
Evaluation Contexts	$\mathbb{E}$	$::=$	$[\cdot] \mid \mathbb{E} >>= M$

Figure 1: The syntax of values and terms.

chapter 4) available in CHP's support libraries it is possible to reduce the same program to:

```
import Control.Concurrent.CHP
import Control.Concurrent.CHP.Connect
import qualified Control.Concurrent.CHP.Common as Common
import Control.Monad
import Data.List

main :: IO ()
main = runCHP $ pipelineConnectComplete
  (\w -> forever (liftIO_CHP getLine >>= writeChannel w))
  [Common.filter ("caterpillar" `isInfixOf`)]
  (\r -> forever (readChannel r >>= liftIO_CHP . putStrLn))
```

3.2 Operational Semantics

In this chapter we will build up an operational semantics for the CHP library in the style of Peyton Jones [2001] and Harris et al. [2005]. We begin with the syntax, in figure 1. The syntax for values and terms is straightforward. Evaluation contexts formalise the rule that the left-most expression in a chained bind is the one to be executed: the $[\cdot]$ notation is an executable "hole" in the expression that can be filled.

Examples of using the evaluation context can be seen in our basic transition rules in figure 2. The rule $\{\mathbb{E}[\texttt{return N >>= M}]\} \Rightarrow \{\mathbb{E}[\texttt{M N}]\}$ states that given a thread of execution (the braces $\{\}$ indicate a thread) where the hole in the expression contains a return followed by a bind, the hole can be replaced by applying the right-hand function to the left-hand value.

We use the $\Rightarrow$ notation to indicate a "free" transition; this is a transition that

$$\{\mathbb{E}[\texttt{return N >>= M}]\} \Rightarrow \{\mathbb{E}[\texttt{M N}]\} \qquad \text{(LUNIT)}$$

$$\{\mathbb{E}[\texttt{M >>= return}]\} \Rightarrow \{\mathbb{E}[\texttt{M}]\} \qquad \text{(RUNIT)}$$

$$\frac{x \notin fn(N) \cup fn(O)}{\{\mathbb{E}[\texttt{(M >>= N) >>= O}]\} \Rightarrow \{\mathbb{E}[\texttt{M >>= (\x -> N x >>= O)}]\}} \qquad \text{(BIND)}$$

$$\frac{\varepsilon[M] = V \qquad M \not\equiv V}{\{\mathbb{E}[M]\} \Rightarrow \{\mathbb{E}[V]\}} \qquad \text{(FUN)}$$

Figure 2: The basic transition rules for monadic and functional code.

is usually a rearrangement, and that never affects other concurrent processes, and never resolves choices (the monad law LUNIT is a typical example). The general form of transitions in the system is written using the $\xrightarrow{\alpha}$ syntax, where α is a placeholder for a set of events that occur with that transition (such as might appear in a CSP trace). The semantics also use structural transitions, written using $\Rightarrow$, which merely facilitate rearrangement ready for the standard transitions. Some events may transition with an empty set of events. Figure 3 gives structural congruence and structural transitions.

3.3 Parallel Composition

The most basic component of a concurrent system is the primitive for supporting concurrency. In CHP this is the `parallel` operation:

```
parallel :: [CHP a] -> CHP [a]
```

The `parallel` operation can be implemented in terms of a binary operator `<||>`[1]:

```
parallel [] = return []
parallel (p:ps) = foldl parPair (p >>= \x -> return [x]) ps
  where
    parPair :: CHP [a] -> CHP a -> CHP [a]
    parPair ps p = do (x, xs) <- p <||> ps
                      return (x : xs)

(<||>) :: CHP a -> CHP b -> CHP (a, b)
```

[1] This is useful for explaining the semantics, but in reality the list form is implemented directly in a more efficient manner.

$$P \parallel Q \equiv Q \parallel P \qquad \text{(COMMUTE)}$$

$$P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R \qquad \text{(ASSOC)}$$

$$\nu x.\nu y.P \equiv \nu y.\nu x.P \qquad \text{(SWAP)}$$

$$(\nu x.P) \parallel Q \equiv \nu x.(P \parallel Q), \quad x \notin fn(Q) \qquad \text{(EXTRUDE)}$$

$$\nu x.P \equiv \nu y.P[y/x], \quad y \notin fn(P) \qquad \text{(ALPHA)}$$

$$\frac{P \xrightarrow{\alpha} Q}{P \parallel R \xrightarrow{\alpha} Q \parallel R} \qquad \text{(PAR)}$$

$$\frac{P \xrightarrow{\alpha} Q \qquad x \notin \alpha}{\nu x.P \xrightarrow{\alpha} \nu x.Q} \qquad \text{(NU)}$$

$$\frac{P \equiv P' \qquad P' \xrightarrow{\alpha} Q' \qquad Q \equiv Q'}{P \xrightarrow{\alpha} Q} \qquad \text{(EQUIV-L)}$$

$$\frac{P \equiv P' \qquad P' \Rightarrow Q' \qquad Q \equiv Q'}{P \Rightarrow Q} \qquad \text{(EQUIV-F)}$$

$$\frac{P \Rightarrow P' \qquad P' \xrightarrow{\alpha} Q}{P \xrightarrow{\alpha} Q} \qquad \text{(FREE)}$$

$$\frac{P \Rightarrow Q \qquad Q \Rightarrow R}{P \Rightarrow R} \qquad \text{(FTRANS)}$$

Figure 3: Structural congruences and structural transitions. We use ν for scoping names, and $\parallel$ for composing two threads of execution in parallel.

$$\frac{u \notin fn(\mathbb{E}) \qquad v \notin fn(\mathbb{E}) \qquad u \neq v}{\{\mathbb{E}[M \texttt{<||>} N]\}_t \xrightarrow{\{\}} \nu u.\nu v.(\{\mathbb{E}[u \sqcup v]\}_t \parallel \{M\}_u \parallel \{N\}_v)} \quad \text{(S-PAR)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\texttt{return x}\}_u \parallel \{\texttt{return y}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{return (x, y)}]\}_t \quad \text{(E-PAR-RR)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\texttt{throw e}\}_u \parallel \{\texttt{return y}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw e}]\}_t \quad \text{(E-PAR-TR)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\texttt{return x}\}_u \parallel \{\texttt{throw e}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw e}]\}_t \quad \text{(E-PAR-RT)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\texttt{throw e}\}_u \parallel \{\texttt{throw f}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw e}]\}_t \quad \text{(E-PAR-TT-L)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\texttt{throw e}\}_u \parallel \{\texttt{throw f}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw f}]\}_t \quad \text{(E-PAR-TT-R)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel [t \lightning e] \xrightarrow{\{\}} \{\mathbb{E}[u \sqcup v]\}_t \parallel [u \lightning e] \parallel [v \lightning e] \quad \text{(PAR-ASYNC)}$$

Figure 4: Operational semantics for beginning and ending parallel composition. The $\sqcup$ notation (which indicates that the parent thread is waiting for its two children to finish executing, often called a join instruction) is invented purely for describing the semantics, and is not visible to the programmer. Each execution thread is surrounded by thread brackets, with a subscript (e.g. t in $\{\ldots\}_t$) as a thread identifier (which is omitted when it is not relevant). Note that, unlike the Haskell function forkIO, the thread identifiers are not made visible to the programmer.

We also provide a form that discards the results of the composition, which can be defined simply as:

```
parallel_ :: [CHP a] -> CHP ()
parallel_ ps = parallel ps >> return ()
```

The operational semantics for the CHP <|||> operator (which should not be confused for the related || semantic parallel composition) is given in figure 4. Informally, the operator runs both actions concurrently, and when *both* have terminated, it returns both results. The only complication in these semantics comes from integrating Haskell's exceptions mechanism. If a branch of the parallel composition terminates with an uncaught exception, this will be re-thrown in the parent process once both branches have terminated. If both branches terminate with an uncaught exception, which one of them is chosen arbitrarily and propagated.

As well as it being possible that the child processes may terminate with an exception, it is possible that while the child processes are running, the parent process may receive an asynchronous exception [Marlow et al., 2001]. In this situation, the exception is duplicated to all child processes. This preserves certain intuitive laws in CHP (e.g. that `p <|||> return ()` behaves identically to `p`). Without this behaviour, throwing an asynchronous exception to a CHP process would have a different effect depending on whether the process is in the middle of a parallel composition or not.

As an example of parallel composition, given a function which reads in the entire contents of a file before returning it, e.g. `readFile :: FilePath -> IO Text` [2], this code:

```
liftIO_CHP (readFile fileA) <|||> liftIO_CHP (readFile fileB)
```

will read the contents of the two files concurrently, and once both files have been fully read, will return a pair of their contents.

3.4 Channel Communications

CHP provides channels for concurrently-executing processes (i.e. those composed together with the <|||> operator) to communicate with each other. This communication is synchronous and unbuffered.

[2] From the `Data.Text.IO` module of the Haskell text-0.11.0 library.

$$\frac{c \notin fn(\mathbb{E})}{\{\mathbb{E}[\texttt{newChannelRW :: CHP(Chanin a,Chanout a)}]\} \xrightarrow{\{\}} \nu c.\{\mathbb{E}[\texttt{return } (c_R, c_W)]\}} \text{(NEW-CHANNEL)}$$

$$\{\mathbb{E}_1[\texttt{readChannel } c_R]\}_t \mid\mid \{\mathbb{E}_2[\texttt{writeChannel } c_W \texttt{ x}]\}_u \xrightarrow{\{c\}} \{\mathbb{E}_1[\texttt{return x}]\}_t \mid\mid \{\mathbb{E}_2[\texttt{return ()}]\}_u \quad \text{(BASIC-COMM)}$$

Figure 5: The operational semantics for basic channel communication, without poison. We use the subscripts R and W to indicate the reading and writing ends of the channels. We use numbered subscripts to identify the different evaluation contexts for the writer and reader in the communication rule.

CHP uses the idea of channel-ends, as do most process-oriented frameworks. Channels are represented by an end from which input can be taken, Chanin, and an end to which output can be given, Chanout. This distinction at the type-level means that it is apparent from the type of a process whether it will use the channel for input or output, which aids readability and prevents confusion where two outputting processes could be wired together and cause deadlock. Channels are allocated using the newChannelRW function, which for now can be considered to have type:

```
newChannelRW :: CHP (Chanin a, Chanout a)
```

This type will be generalised in section 3.4.2. These channel ends can then be used for communication using the readChannel and writeChannel functions:

```
readChannel :: Chanin a -> CHP a
writeChannel :: Chanout a -> a -> CHP ()
```

The following code creates a channel, and sends a String across it from one concurrent process to another:

```
do (r, w) <- newChannelRW
   writeChannel w "Hello" <||> readChannel r
```

The operational semantics for channel creation and communication are given in figure 5. Informally, the channel communication is a synchronised passing of data from one concurrent process to another. There is no buffering in the channels, and each communication must have both a writer and a reader. The following code deadlocks because the write requires a concurrent read:

```
do (r, w) <- newChannelRW
```

```
writeChannel w "Hello"
readChannel r
```

Buffer processes can be constructed should buffering be needed; an example of such a process is given in section 3.9.2.

3.4.1 Channel Type Inference

It is instructive to note that this previous code sample is a complete fragment of Haskell code that needs no extra type annotations:

```
do (r, w) <- newChannelRW
   writeChannel w "Hello" <|*|> readChannel r
```

The Haskell type system is able to infer that the channel being constructed carries Strings from its use on the second line, meaning that the first line – where the channel is constructed – needs no annotations. This allows the channel creation code in CHP to be much less verbose than other systems such as occam-π where the type of each channel must be stated explicitly – but the program is still statically type-checked and is type-safe.

3.4.2 Shared Channels

The channels we have seen so far are one-to-one channels; they are intended for use by one writer and one reader. Many systems include support for shared channels, which allow multiple *sequential* writers or readers. With shared channels, each communication only has a single reader and a single writer, but it is possible for each communication to involve different readers and writers. For example, a channel might be used to receive results from many workers. This can be accomplished using an any-to-one channel: a channel with a single reader, but with multiple possible writers. Each writer can claim the shared writing end of the channel in order to communicate, and then release it again for use by other writers.

This sharing of the channel can be easily accomplished by providing a mutex for the ends of the channel that are shared. For this purpose we use a simple Shared wrapper (see section 3.12.1 for the implementation of the Mutex type) that is exported as abstract:

```
data Shared c a = Shared (Mutex, c a)
```

$$\mathbb{E} ::= [\cdot] \mid \mathbb{E} >>= M \mid \texttt{finallyCHP}\ \mathbb{E}\ M$$

$$\frac{c \notin fn(\mathbb{E})}{\{\mathbb{E}[\texttt{newChannelRW :: CHP (Shared Chanin a, Chanout a)}]\}_t \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return (}c_R\texttt{, }c_W\texttt{)}]\}_t \mid\mid \langle c_R \rangle)} \quad \text{(NEW-CHAN-SRW)}$$

$$\frac{c \notin fn(\mathbb{E})}{\{\mathbb{E}[\texttt{newChannelRW :: CHP (Chanin a, Shared Chanout a)}]\}_t \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return (}c_R\texttt{, }c_W\texttt{)}]\}_t \mid\mid \langle c_W \rangle)} \quad \text{(NEW-CHAN-RSW)}$$

$$\frac{c \notin fn(\mathbb{E})}{\{\mathbb{E}[\texttt{newChannelRW :: CHP (Shared Chanin a, Shared Chanout a)}]\}_t \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return (}c_R\texttt{, }c_W\texttt{)}]\}_t \mid\mid \langle c_R \rangle \mid\mid \langle c_W \rangle)} \quad \text{(NEW-CHAN-SRSW)}$$

$$(\{\mathbb{E}[\texttt{claim x m}]\}_t \mid\mid \langle x \rangle) \xrightarrow{\{\}} \{\mathbb{E}[\texttt{m x `finallyCHP` release x}]\}_t \quad \text{(CLAIM)}$$

$$\{\mathbb{E}[\texttt{release x}]\}_t \xrightarrow{\{\}} (\{\mathbb{E}[\texttt{return ()}]\}_t \mid\mid \langle x \rangle) \quad \text{(RELEASE)}$$

Figure 6: The operational semantics for shared channels. Communication is still carried out using the semantics for standard channels, but these semantics deal with sharing. The $\langle x \rangle$ notation is a dummy process indicating that the mutex associated with x is unclaimed – this is consumed/created by claiming/releasing, respectively. Note that $\langle c_R \rangle$ is a distinct mutex from $\langle c_W \rangle$. The release command is a semantic construct that is not accessible to the programmer, but is needed to release the channel after the claim block is complete.

This can be used as Shared Chanin Int or Shared Chanout String, for example. We provide a simple claim function that claims the mutex, executes some actions with the shared item, and then releases the mutex at the end of the execution (or in case of an exception):

```
claim :: Shared c a -> (c a -> CHP b) -> CHP b
```

This function is the only API that is necessary for the use of shared channels; all other functions such as reading and writing are done as before, but must be executed inside the block of code passed to claim as the second parameter. The only other consideration is channel creation.

We could provide functions such as newChannelAnyToOneRW for channel creation, but in fact we can re-use the same newChannelRW function as one-to-one channels by generalising the type. Recall that we previously defined the type of newChannelRW as:

```
newChannelRW :: CHP (Chanin a, Chanout a)
```

We can generalise this using a type-class:

```
class Channel r w where
  newChannelRW :: CHP (r a, w a)

instance Channel Chanin Chanout -- one-to-one
instance Channel (Shared Chanin) Chanout -- one-to-any
instance Channel Chanin (Shared Chanout) -- any-to-one
instance Channel (Shared Chanin) (Shared Chanout) -- any-to-any
```

This removes the need for a proliferation of channel creation functions, especially when we later consider broadcast and reduce channels in section 3.6. The semantics for creating and claiming shared channels are given in figure 6.

3.4.3 Extended Communication

CHP also offers the idea of extended communication, also known as extended rendezvous [Barnes and Welch, 2002]. The idea behind extended communication is that two processes synchronise on a channel and exchange data as with a normal channel communication, but either side (or both sides) may *extend* the communication, performing an additional action while the two are engaged together.

Informally, a communication can be generalised into its extended form as follows: two processes agree to communicate on a given channel. At this point, the

$$\{\mathbb{E}_1[\texttt{(extReadChannel } c_R \texttt{ M)}]\}_t \;||\; \{\mathbb{E}_2[\texttt{(extWriteChannel } c_W \texttt{ N)}]\}_u \xrightarrow{\{\tau\}} \{\mathbb{E}_1[\texttt{(readChannel } c_R \texttt{ >>= M) `finallyCHP` endComm } c]\}_t \;||\; \{\mathbb{E}_2[\texttt{(N >>= writeChannel } c_W\texttt{) `finallyCHP` endComm } c]\}_u \quad \text{(EXT-COMM-RW)}$$

$$\{\mathbb{E}_1[\texttt{(readChannel } c_R\texttt{)}]\}_t \;||\; \{\mathbb{E}_2[\texttt{(extWriteChannel } c_W \texttt{ N)}]\}_u \xrightarrow{\{\tau\}} \{\mathbb{E}_1[\texttt{readChannel } c_R]\}_t \;||\; \{\mathbb{E}_2[\texttt{N >>= writeChannel } c_W]\}_u \quad \text{(EXT-COMM-W)}$$

$$\{\mathbb{E}_1[\texttt{(extReadChannel } c_R \texttt{ M)}]\}_t \;||\; \{\mathbb{E}_2[\texttt{(writeChannel } c_W\texttt{)}]\}_u \xrightarrow{\{\tau\}} \{\mathbb{E}_1[\texttt{(readChannel } c_R \texttt{ >>= M) `finallyCHP` endComm } c]\}_t \;||\; \{\mathbb{E}_2[\texttt{writeChannel } c_W \texttt{ `finallyCHP` endComm } c]\}_u \quad \text{(EXT-COMM-R)}$$

$$\{\mathbb{E}_1[\texttt{endComm c}]\}_t \;||\; \{\mathbb{E}_2[\texttt{endComm c}]\}_u \xrightarrow{\{\}} \{\mathbb{E}_1[\texttt{return ()}]\}_t \;||\; \{\mathbb{E}_2[\texttt{return ()}]\}_u \quad \text{(END-COMM)}$$

Figure 7: The semantics for extended communication. The endComm call is a purely semantic construct that is not visible to the end user.

writer may perform an extended action (which may involve other synchronisations) which will give a result of the value to be communicated, while the reader waits. This value is then passed across, at which point the reader may perform an extended action with the value it has received, while the writer waits. After this action is done, the two processes can continue executing. This is reflected in the API:

```
extWriteChannel :: Chanout a -> CHP a -> CHP ()
extReadChannel :: Chanin a -> (a -> CHP b) -> CHP b
```

The second parameter to each function is the extended action. Note that normal (non-extended) communication can be seen as a special case of extended communication:

```
writeChannel c x = extWriteChannel c (return x)
readChannel c = extReadChannel c return
```

Note that either readChannel or extReadChannel can be used with writeChannel or extWriteChannel – that is, either side may choose to have an extended action, or not.

The formal semantics for extended communication is given in figure 7. The τ event is an internal hidden event, but it is required to get the correct behaviour with respect to the choice semantics later in the chapter. The finallyCHP function has similar semantics to the finally function; it ensures that the second argument is executed after the first has finished, even if the first finished by throwing an exception (or throws poison, which will be introduced later on).

3.5 Barriers

Barriers are primitives that offer multiway synchronisation between any number of processes, without the exchange of data that channels provide. They are useful in order to ensure that multiple concurrent processes have reached the same stage in their execution. The semantics of barriers is given in figure 8.

Informally, a barrier has an associated persistent count of the number of processes enrolled on a barrier, N. When one process wants to synchronise, it must wait for the other $N - 1$ processes to also synchronise, at which point they all proceed. The enrollment count can be dynamically increased or decreased at any time. Barriers can be re-used (several concurrent libraries feature a "barrier" primitive that is single-use only).

In this section we show an API for barriers that will be generalised in section 3.5.1 to include the idea of phases. A barrier is created with an Unenrolled wrapper. This is much like the Shared wrapper used to prevent use of shared channels without claiming; a barrier in an Unenrolled wrapper cannot be used for anything except enrolling. That is the rationale behind the three basic barrier functions:

```
newBarrier :: CHP (Unenrolled Barrier)
enroll :: Unenrolled Barrier -> (Barrier -> CHP a) -> CHP a
syncBarrier :: Barrier -> CHP ()
```

The newBarrier function creates a new barrier with no processes enrolled. The enroll function enrolls on a barrier for the duration of the given code block (second parameter), during which time the enrolled process can call syncBarrier to synchronise with all other processes enrolled on the barrier. The style of API for the enroll function avoids the need for an explicit resign call, by automatically resigning at the end of the given block.

$$\frac{b \notin (keys(B) \cup fn(\mathbb{E}))}{\{\mathbb{E}[\texttt{newBarrier}]\} \mid\mid \beta(B) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\texttt{return}\ \ b_U]\} \mid\mid \beta(B \triangleleft \{b \mapsto 0\}))} \quad \text{(NEW-B)}$$

$$\frac{(b \mapsto k) \in B}{\nu b.(\{\mathbb{E}[\texttt{enroll}\ \ b_U\ \ \texttt{M}]\} \mid\mid \beta(B)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\texttt{M}\ \ b_E\ \ \texttt{`finallyCHP`}\ \ \texttt{unenroll}\ \ b_E]\} \mid\mid \beta(B \triangleleft \{b \mapsto k+1\}))} \quad \text{(ENROLL-B)}$$

$$\frac{(b \mapsto k) \in B}{\nu b.(\{\mathbb{E}[\texttt{unenroll}\ \ b_E]\} \mid\mid \beta(B)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\texttt{return}\ \ ()]\} \mid\mid \beta(B \triangleleft \{b \mapsto k-1\}))} \quad \text{(UNENROLL-B)}$$

$$\frac{(b \mapsto k) \in B}{\nu b.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{syncBarrier}\ \ b_E]\}_{t_i} \mid i \in \{1..k\}\} \mid\mid \beta(B)) \xrightarrow{\{b\}} \nu b.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{return}\ \ ()]\}_{t_i} \mid i \in \{1..k\}\} \mid\mid \beta(B))} \quad \text{(SYNC-B)}$$

Figure 8: The operational semantics for barriers without a phase and without poison. We take B to be an associative map (with a single enrollment count value per key), indicated by a β wrapper notation, with the function *keys* that gets the set of keys, and the notation $B \triangleleft B'$ to mean the union with a bias to the right (i.e. the mappings from the right override those from the left). We use the subscripts U and E on barriers to indicate enrolled and unenrolled ends, but semantically they are both aliases for b. The `unenroll` function is a purely semantic construct that is not visible to the programmer, but is needed to reverse the effects of enroll after the enrolled block is complete.

To explore the uses of channels and barriers, consider the task of swapping the contents of two files. A sequential implementation would be as follows (using non-lazy readFile and writeFile functions):

```
do a <- liftIO_CHP ( readFile  fileA )
   b <- liftIO_CHP ( readFile  fileB )
   liftIO_CHP ( writeFile  fileA  b)
   liftIO_CHP ( writeFile  fileB  a)
```

A simple concurrent implementation could use two concurrent compositions:

```
do (a, b) <- liftIO_CHP ( readFile  fileA ) <||> liftIO_CHP ( readFile  fileB )
   liftIO_CHP ( writeFile  fileA  b) <||> liftIO_CHP ( writeFile  fileB  a)
```

For the purposes of illustration, we consider using a single concurrent composition. The version with channels would read the file, swap the contents with the other process using channels and then write back to the same file:

```
do (rA, wA) <- newChannelRW
   (rB, wB) <- newChannelRW
   (do a <- liftIO_CHP ( readFile  fileA )
       writeChannel wA a
       b <- readChannel rB
       liftIO_CHP ( writeFile  fileA  b)
    ) <||> (do b <- liftIO_CHP ( readFile  fileB )
               a <- readChannel rA
               writeChannel wB b
               liftIO_CHP ( writeFile  fileB  a) )
```

A barrier-based example could simply use the barrier to synchronise the two processes, thus indicating to each other that the files have been read, and it is now safe to write back to both of them:

```
do b <- newBarrier
   enroll  b (\bA -> enroll  b (\bB ->
    (do a <- liftIO_CHP ( readFile  fileA )
        syncBarrier  bA
        liftIO_CHP ( writeFile  fileB  a)
    ) <||> (do b <- liftIO_CHP ( readFile  fileB )
               syncBarrier  bB
               liftIO_CHP ( writeFile  fileA  b)
           )
   ) )
```

Note that the enroll calls take place outside the parallel composition. This is crucial for the correct use of barriers. This ensures that the enrollment count is two before the parallel composition begins. In general all uses of barriers should follow this pattern:

```
do b <- newBarrier
   enroll b (\bA -> enroll b (\bB -> p bA <||> q bB))
```

They should **not** follow this pattern:

```
do b <- newBarrier
   enroll b p <||> enroll b q
```

The problem with the latter is that it is possible for the following to happen: the left-hand side of the parallel composition may be scheduled first. This will enroll p on the barrier b, giving it an enrollment count of 1. All synchronisations on b during the execution of p will succeed immediately, as it is only synchronising with itself. p can then complete before the right-hand side of the parallel composition (with q) begins executing. The process p will then resign from the barrier, and q will enroll, leaving it with an enrollment count of one again, which means q can execute, always synchronising with itself. So while the processes will not deadlock, the intended behaviour (the two processes synchronising together) may not happen, or perhaps q will enroll after p has synchronised a few times. The way to guarantee that they always synchronise together is to enroll both before beginning the parallel composition. It is trivial to construct helper functions to this end, for example:

```
enrollAllPar :: Unenrolled Barrier -> [Barrier -> CHP a] -> CHP [a]
enrollAllPar ub = enrollAndRun []
  where
    enrollAndRun :: [CHP a] -> [Barrier -> CHP a] -> CHP [a]
    enrollAndRun ps [] = parallel ps
    enrollAndRun ps (q:qs) = enroll ub (\eb -> enrollAndRun (ps ++ [q eb]) qs)
```

This function enrolls all the given processes on a barrier before running them all in parallel. We will see more examples like this in chapter 4. This function allows us to rewrite our earlier example more simply:

```
do b <- newBarrier
   enrollAllPar b
    [\bar -> do a <- liftIO_CHP ( readFile  fileA )
                syncBarrier bar
                liftIO_CHP ( writeFile  fileB  a)
    ,\bar -> do b <- liftIO_CHP ( readFile  fileB )
                syncBarrier bar
                liftIO_CHP ( writeFile  fileA  b)
    ]
```

3.5.1 Phased Barriers

In the previous section we introduced barriers, which purely synchronised with no further information available. These simple barriers can be viewed as a particular subset of phased barriers. The idea behind phased barriers arises from a common mode of use in occam-π [Barnes et al., 2005; Ritson and Welch, 2010]: barriers are often used to separate the actions of concurrent agents into distinct phases, such as discovery and action phases in concurrent simulations. Typically these phases are documented in comments but are not used in the actual program code.

Phased barriers incorporate the idea of phases into barriers. A phased barrier always has a current phase value. Each synchronisation on the barrier moves its phase one position further through the cycle of phases. This allows two enhancements in the code: processes enrolled on the barrier can ensure that their belief about the current phase is correct, and processes newly enrolled on the barrier can discover the current phase and behave accordingly.

We start by showing the update to the barrier API to support phases, which can be of any type:

```
newPhasedBarrier' :: [phase] -> CHP (Unenrolled PhasedBarrier phase)
enroll  :: Unenrolled PhasedBarrier phase
        -> (PhasedBarrier phase -> CHP a) -> CHP a
syncBarrier  :: PhasedBarrier phase -> CHP phase
currentPhase :: PhasedBarrier phase -> CHP phase
```

(We will generalise the type of `enroll` yet further in section 3.6.) The argument to `newPhasedBarrier'` is an infinite list of phases. The starting phase will be the head of the list, and after the first synchronisation the phase will be the second item in the list and so on. The `syncBarrier` function now returns the new phase

$$\frac{b \notin (keys(B) \cup fn(\mathbb{E}))}{\{\mathbb{E}[\texttt{newPhasedBarrier' } h]\} \mid\mid \beta(B) \xrightarrow{\{\}} \nu b.\{\mathbb{E}[\texttt{return } b_U]\} \mid\mid \beta(B \triangleleft \{b \mapsto (0, h)\})} \quad \text{(NEW-BP)}$$

$$\frac{(b \mapsto (k, h)) \in B}{\nu b.(\{\mathbb{E}[\texttt{enroll } b_U \texttt{ M}]\} \mid\mid \beta(B)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\texttt{M } b_E \texttt{ `finallyCHP` unenroll } b_E]\} \mid\mid \beta(B \triangleleft \{b \mapsto (k+1, h)\}))} \quad \text{(ENROLL-BP)}$$

$$\frac{(b \mapsto (k, h)) \in B}{\nu b.(\{\mathbb{E}[\texttt{unenroll } b_E]\} \mid\mid \beta(B)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\texttt{return ()}]\} \mid\mid \beta(B \triangleleft \{b \mapsto (k-1, h)\}))} \quad \text{(UNENROLL-BP)}$$

$$\frac{(b \mapsto (k, o : h : h')) \in B}{\nu b.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{syncBarrier } b_E]\}_{t_i} \mid i \in \{1..k\}\} \mid\mid \beta(B)) \xrightarrow{\{b\}} \nu b.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{return } h]\}_{t_i} \mid i \in \{1..k\}\} \mid\mid \beta(B \triangleleft \{b \mapsto (k, h : h')\}))} \quad \text{(SYNC-BP)}$$

$$\frac{(b \mapsto (k, h : h')) \in B}{\nu b.(\{\mathbb{E}[\texttt{currentPhase } b_E]\} \mid\mid \beta(B)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\texttt{return } h]\} \mid\mid \beta(B))} \quad \text{(CUR-BP)}$$

Figure 9: The operational semantics for barriers with a phase and without poison. We now use a pair for the values in B, of an enrollment count and infinite list of phases (where the first item indicates the current phase). The synchronise function moves to the next phase, and returns this phase as the current phase.

after the synchronisation, and currentPhase can be used between synchronisations to ask for the current phase. Note that currentPhase has no danger of race hazards; only an enrolled process can use this function, and while the enrolled process is asking for the current phase it is not synchronising, and thus the current phase cannot change during the call.

It is trivial to add extra APIs for the creation of phased barriers and generation of the phase list. In most cases it is sufficient to use the Haskell type-classes Enum and Bounded:

```
newPhasedBarrier :: (Enum phase, Bounded phase) =>
                    phase -> CHP (Unenrolled PhasedBarrier phase)
newPhasedBarrier start
  = newPhasedBarrier' ([start..maxBound] ++ cycle [minBound..maxBound])
```

This makes using phased barriers trivial; a program can get a phased barrier with alternating discovery and action phases simply by writing:

```
data Phase = Discovery | Action deriving (Enum, Bounded)

do b <- newPhasedBarrier Discovery
```

It was stated earlier that the simple barriers of the previous section are just a special case of phased barriers. The obvious type for the phases is simply (), the unit type, so that barriers are simply PhasedBarrier (). The barrier API specialised for this phase type is very similar to our original API:

```
newBarrier :: CHP (Unenrolled PhasedBarrier ())
newBarrier = newPhasedBarrier' (repeat ())

enroll :: Unenrolled PhasedBarrier ()
       -> (PhasedBarrier () -> CHP a) -> CHP a
syncBarrier :: PhasedBarrier () -> CHP ()
```

The operational semantics for phased barriers is given in figure 9.

3.6 Broadcast and Reduce Channels

We have introduced standard one-to-one channels, shared channels with multiple readers or writers in sequence (but only one writer and one reader at a time), and barriers on which processes can enroll and resign. Broadcast channels allow multiple readers to enroll, and then all communicate to receive the same value

from a single writer in a single communication. Reduce channels allow multiple writers to enroll, and then all communicate values (which are reduced into a single value) to a single reader in a single communication.

We do not require any additional APIs for these channels, we merely use and generalise existing APIs. In particular, we combine the channels API with the enrolling API. Our first step is to generalise the enroll function seen earlier in the barriers section into a type-class:

```
class Enrollable b where
  enroll :: Unenrolled b p -> (b p -> CHP a) -> CHP a

instance Enrollable PhasedBarrier
```

This allows the same use as in the phased barriers, but we can now add instances for broadcast channels (where the reading end – the Chanin – is enrollable) and reduce channels (where the Chanout writing end is enrollable):

```
instance Enrollable Chanin
instance Enrollable Chanout
```

The next step is to add instances for creating these channels, using our earlier Channel type-class:

```
instance Channel (Unenrolled Chanin) Chanout
instance Channel (Unenrolled Chanin) (Shared Chanout)
instance Channel Chanin (Unenrolled Chanout)
instance Channel (Shared Chanin) (Unenrolled Chanout)
instance Channel (Unenrolled Chanin) (Unenrolled Chanout)
```

These channels are broadcast channels with unshared and shared ends, reduce channels with unshared and shared ends, and finally a broadcast-reduce channel. Thus we have implementations of all possible combinations of Chan*, Shared Chan* and Unenrolled Chan*.

The semantics of broadcast and reduce channels are given in figure 10. We note that broadcast and reduce channels are offered by frameworks such as MPI, but are not commonly offered in process-oriented frameworks. This is perhaps because choice over such channels is difficult to offer without choice over barriers – with the latter feature in place, choice over broadcast and reduce channels is trivial [Welch et al., 2010].

$$\frac{c \notin (keys(C) \cup fn(\mathbb{E}))}{\begin{array}{c}\{\mathbb{E}[\texttt{newChannelRW :: CHP(Unenrolled Chanin a, Chanout a)}]\} \mid\mid \gamma(C) \\ \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return } (c_{UR}, c_W)]\} \mid\mid \gamma(C \triangleleft \{c \mapsto 0\}))\end{array}} \quad \text{(NEW-BROAD)}$$

$$\frac{c \notin (keys(C) \cup fn(\mathbb{E}))}{\begin{array}{c}\{\mathbb{E}[\texttt{newChannelRW :: CHP (Chanin a, Unenrolled Chanout a)}]\} \mid\mid \gamma(C) \\ \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return } (c_R, c_{UW})]\} \mid\mid \gamma(C \triangleleft \{c \mapsto 0\}))\end{array}} \quad \text{(NEW-REDUCE)}$$

$$\frac{(c \mapsto k) \in C}{\begin{array}{c}\nu c.(\{\mathbb{E}[\texttt{enroll } c_U \texttt{ M}]\} \mid\mid \gamma(C)) \\ \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{M } c_E \texttt{ `finallyCHP` unenroll } c]\} \\ \mid\mid \gamma(C \triangleleft \{c \mapsto k+1\}))\end{array}} \quad \text{(ENROLL-BR)}$$

$$\frac{(c \mapsto k) \in C}{\begin{array}{c}\nu c.(\{\mathbb{E}[\texttt{unenroll } c]\} \mid\mid \gamma(C)) \\ \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return ()}]\} \mid\mid \gamma(C \triangleleft \{c \mapsto k-1\}))\end{array}} \quad \text{(UNENROLL-BR)}$$

$$\frac{(c \mapsto k) \in C \quad k \geq 1}{\begin{array}{c}\nu c.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{readChannel } c]\}_{t_i} \mid i \in \{1..k\}\} \\ \mid\mid \{\mathbb{E}_0[\texttt{writeChannel c x}]\}_u \mid\mid \gamma(C)) \\ \xrightarrow{\{c\}} \nu c.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{return x}]\}_{t_i} \mid i \in \{1..k\}\} \\ \mid\mid \{\mathbb{E}_0[\texttt{return ()}]\}_u \mid\mid \gamma(C))\end{array}} \quad \text{(COMM-BROAD)}$$

$$\frac{(c \mapsto k) \in C \quad k \geq 1}{\begin{array}{c}\nu c.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{writeChannel c } x_i]\}_{t_i} \mid i \in \{1..k\}\} \\ \mid\mid \{\mathbb{E}_0[\texttt{readChannel c}]\}_u \mid\mid \gamma(C)) \\ \xrightarrow{\{c\}} \nu c.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{return ()}]\}_{t_i} \mid i \in \{1..k\}\} \\ \mid\mid \{\mathbb{E}_0[\texttt{return (combine } x_1 \ldots x_k\texttt{)}]\}_u \mid\mid \gamma(C))\end{array}} \quad \text{(COMM-REDUCE)}$$

Figure 10: The operational semantics for broadcast and reduce channels. C is an associative map (akin to B) wrapped by γ, which maps channel identifiers to enrollment counts. Note that the behaviour of communicating on either channel when the count is zero is undefined.

3.7 Choice

Choice allows a CHP process to wait one of several different events to occur. For example, a process can choose between communicating on a channel or synchronising on a barrier. The process will engage in exactly one of the events that it is choosing between. The API is straightforward:

```
instance Alternative CHP where
  (<|>) :: CHP a -> CHP a -> CHP a
  empty :: CHP a

skip :: CHP ()
stop :: CHP a
stop = empty
alt :: [CHP a] -> CHP a
alt = foldr (<|>) empty
```

The <|> operator chooses between two actions; empty (with a synonym stop, to fit with CSP) is the never-ready choice, while skip is the always-ready choice. The alt function (named to match occam-π) is provided as a list form of choice.

The semantics of choice is given in figure 11. The intuition is as follows: consider a series of chained choices to be a flattened list of choices. Pull the choices from the list, until either the end of the list is reached, or an always-ready choice is found (but do not take this). If any of these pulled choices can complete, complete one of them (making a non-deterministic choice between them). If none can complete and there was an always-ready choice following them, execute the always-ready choice. If none can complete and there were no other choices, wait for one to complete. This is written in Haskell psuedo-code in figure 12. We will explain the semantic rules by considering various examples of choices.

Consider the code readChannel c <|> readChannel d. If readChannel c can currently perform a transition (using rule BASIC-COMM from figure 5) then according to rule CHLEFT it can complete (becoming return x) and the right-hand side of the choice will be discarded. If, on the other hand, readChannel c cannot complete but readChannel d can perform a transition (again using BASIC-COMM) then according to the rule CHRIGHT it can complete instead. Close examination of the pre-conditions shows that if both can perform a transition, either CHLEFT or CHRIGHT can fire; there is no precedence between events, and which occurs is a non-deterministic choice.

$$\{\mathbb{E}[\texttt{skip}]\} \xrightarrow{\{\}} \{\mathbb{E}[\texttt{return ()}]\} \qquad \text{(SKIP)}$$

$$\{\mathbb{E}[\texttt{return x <|> N}]\} \xrightarrow{\{\}} \{\mathbb{E}[\texttt{return x}]\} \qquad \text{(CHLRET)}$$

$$\frac{\{M\} \not\rightarrow \{M'\} \quad M \not\equiv (\texttt{A <|> B})}{\{\mathbb{E}[\texttt{(M <|> return x)}]\} \xrightarrow{\{\}} \{\mathbb{E}[\texttt{return x}]\}} \qquad \text{(CHRRET)}$$

$$\texttt{(M <|> N) <|> O} \Rightarrow \texttt{M <|> (N <|> O)} \qquad \text{(CHASSOC)}$$

$$\frac{\{M\} \xrightarrow{\alpha} \{M'\} \quad M \not\equiv (\texttt{A <|> B})}{\{\mathbb{E}[\texttt{M <|> N}]\} \xrightarrow{\alpha} \{\mathbb{E}[\texttt{M'}]\}} \qquad \text{(CHLEFT)}$$

$$\frac{\begin{array}{c}((\{M\} \not\rightarrow \{M'\} \wedge \{N\} \xrightarrow{\alpha} \{N'\}) \\ \vee(\{M\} \xrightarrow{\{\}} \{M'\} \wedge \{N\} \xrightarrow{\alpha} \{N'\} \wedge \alpha \neq \{\})) \\ M \not\equiv (\texttt{A <|> B})\end{array}}{\{\mathbb{E}[\texttt{M <|> N}]\} \xrightarrow{\alpha} \{\mathbb{E}[\texttt{N'}]\}} \qquad \text{(CHRIGHT)}$$

Figure 11: The semantics for the choice operator. The condition $M \not\equiv$ (A <|>B) on many of the rules forces the choice to be rearranged into a right-associative form (via CHASSOC) before proceeding. The condition $\{M\} \not\rightarrow \{M'\}$ is only satisfied when M cannot make any transition, while $\{M\} \xrightarrow{\{\}} \{M'\}$ is satisfied when M can make a transition with a non-empty set of events or M cannot make any transition.

```
data Choice = AlwaysReady | Other

choose :: [Choice] -> Choice
choose choices =
  let (pulled, rest) = span (/= AlwaysReady) choices
  in case filter canComplete pulled of
       [] -> case rest of
               [] -> waitForOne pulled
               (x@AlwaysReady:_) -> x
       poss -> complete (nonDeterministicPick poss)
```

Figure 12: Pseudo-Haskell code explaining the intuition behind choice resolution in CHP.

Consider the code (readChannel c <|> readChannel d) <|> readChannel e. In its current form, neither CHLEFT or CHRIGHT can fire because the left-hand side of the choice is itself a choice. We must use the rule CHASSOC (in addition to the rule FREE from figure 3) to rearrange the expression into a right-associative form (readChannel c <|> (readChannel d <|> readChannel e)). Given this form, either CHLEFT, CHRIGHT or a double application of CHRIGHT would allow c, d or e to be read from (assuming BASIC-COMM could fire with an appropriate writer).

Consider the code skip <|> readChannel c. Rules SKIP and CHLEFT state that this can transition to return (). Rule CHRIGHT cannot fire here, because its pre-conditions cannot be met; the left-hand side can make a {} transition so the right-hand side is prevented from firing.

Consider the code readChannel c <|> skip. Rules CHLEFT and BASIC-COMM would allow communicate on channel c if an appropriate writer is available. The rule CHRIGHT can fire only if readChannel c cannot currently make a transition (because the skip on the right-hand side can only make a {} transition).

Consider the code (readChannel c <|> skip) <|> (readChannel d<|>readChannel e). Again, it requires the CHASSOC and FREE rules to rearrange into a form that is right-associative: readChannel c <|> (skip <|> (readChannel d <|> readChannel e)). Given this form, CHLEFT and BASIC-COMM would allow communication on channel c if an appropriate writer is available. Rule CHRIGHT depends on what transitions the right-hand side of the outer choice can make: the only transition the right-hand side can make is through the SKIP rule, therefore this code is actually semantically equivalent to the previous readChannel c <|> skip example.

3.7.1 Choice and Monadic Bind

All the examples we have looked at so far involve single actions in a choice, e.g. readChannel c <|> readChannel d. In this section we will explain what happens when monadic bind is involved, e.g.:

```
(readChannel c >>= writeChannel e) <|> (readChannel d >>= \_ -> syncBarrier b)
```

The intuition is that only the first, or leading, action of each choice matters when considering which choice to take. So both the previous examples will make a choice between reading from channels c and d. Once the choice has been made, the result is fed into the appropriate bind operator (if any).

This behaviour requires no extra semantic rules; it is already encoded in the semantics that we have written. Consider the example p || q, where :

```
p = (readChannel c >>= writeChannel e)
    <|> (readChannel d >>= \_ -> syncBarrier b)
q = writeChannel c x
```

and assume that channel c can communicate due to a waiting writer. We can rewrite this to a form with evaluation contexts:

$\mathbb{E}_1 = [\cdot]$ >>= writeChannel e
$\mathbb{E}_2 = [\cdot]$ >>= _ -> syncBarrier b
$\mathbb{E}_3 = [\cdot]$
p = $\mathbb{E}_1$[readChannel c] <|> $\mathbb{E}_2$[readChannel d]
q = $\mathbb{E}_3$[writeChannel c x]

The rule CHLEFT states that the choice M <|>N can make a transition to M' if M can transition to M'. We can use this rule on p, because the left-hand side of p, together with q, satisfies the rule BASIC-COMM, meaning they can both transition (through CHLEFT and BASIC-COMM) to:

$\mathbb{E}_1 = [\cdot]$ >>= writeChannel e
$\mathbb{E}_3 = [\cdot]$
p = $\mathbb{E}_1$[return x]
q = $\mathbb{E}_3$[return ()]

Skip and return

The main subtlety involved in choice and binding is the behaviour of return. One of the monadic laws is that return x >>= f is equivalent to f x. Therefore, the expression (return c >>= readChannel) <|> readChannel d must have identical behaviour to the expression readChannel c <|> readChannel d (the former has no rules that can fire for the left-hand side as-is, but can transition into the latter using the structural transition LUNIT). We must maintain this equivalence while we implement another behaviour: a simple return statement (without an accompanying bind) in a choice should be an always-ready choice.

Our rule, CHLRET, states that when return is on the left-hand side of a choice *by itself*, it can resolve the choice with a {} transition, ignoring the right-hand side. Note that this rule only applies to single return statements; if the return statement is bound to another action inside the choice then the rule cannot fire. There is another rule, CHRRET, which allows a choice to resolve to a right-hand return (by itself) when the left-hand side is unable to fire.

The skip rule has a very similar semantic effect to return x (both are always-ready choices), but the SKIP rule is written differently to CHLRET and CHRRET for two reasons. Firstly, and most importantly, (skip >>= M) <|>N will be an always-ready choice that fires (as a {} transition) to return () >>= M, whereas (return x >>= M) <|>N will act as M x <|>N, i.e. the return statement will be lost in the rearrangement. Secondly, skip needs a rule that allows it to fire outside of a choice, so it makes sense to have the same single rule, rather than writing two separate rules.

3.7.2 Choice and Abstraction

CHP allows more rearrangement of program code, and pulling out code into functions, than other systems such as occam-π, and choice is one example of this. `ALT`s (choices) in occam-π are composable to a certain degree; directly nested `ALT`s are possible:

```
PRI ALT
  PRI ALT
    c ? x
      d ! x
    e ? x
      d ! x
  SKIP
    d ! 0
```

The above code chooses between inputs on `c` and `e`, but if neither are ready, it executes the `SKIP` guard. The body of each guard is an output on channel `d`.

However, in occam-π one *cannot* pull out guards into a separate procedure:

```
PROC alt.over.all (CHAN INT c?, CHAN INT e?, CHAN INT d!)
  PRI ALT
    c ? x
      d ! x
    e ? x
      d ! x
:
PRI ALT
  alt.over.all (c, e, d)
  SKIP
    d ! 0
```

This was a design decision, taken in classical occam, to treat the guards differently to normal code. Due to referential transparency and our decision not to treat choices differently to normal, this *is* valid in CHP:

```
altOverAll :: Chanin Int -> Chanin Int -> Chanout Int -> CHP ()
altOverAll c e d = alt [ do x <- readChannel c
                           writeChannel d x
                       , do x <- readChannel e
                           writeChannel d x ]

alt [ altOverAll c e d
    , do skip
         writeChannel d 0 ]
```

This new composability overcomes one of the shortcomings that Reppy pointed out in the `ALT` construct when he developed Concurrent ML [Reppy, 1995]. He noted that function composition was incompatible with choice. With implicit guards in CHP, this is not the case. This idea would also be possible to build into occam-π, where the presence of choice could be checked at compile-time.

3.7.3 Choice Between Transactions

We have already stated that CHP chooses between the leading actions of different choices. One alternative possibility to supporting choice would be to support choice between the entire block. That is, given the code below, the process could choose between reading and then writing on the "..A" channels (as one option) or reading and then writing on the "..B" channels (as another option):

```
(readChannel inputA >>= writeChannel outputA)
<|> (readChannel inputB >>= writeChannel outputB)
```

This would be a transactional semantics, where each transaction consisted of many event synchronisations. Donnelly and Fluet [2006] have previously explored this possibility in a Haskell setting. The greater power comes at the expense of a slower run-time resolution. The idea of using a transactional semantics is powerful, but in this thesis we will restrict ourselves to CSP's style of choice: that of choosing between individual leading events. We also note that the conjoined events feature in CHP introduced in the next section can emulate *some* of the capabilities of such transactions.

$$\frac{\{M\} \xrightarrow{\alpha} \{M'\} \quad \{N\} \xrightarrow{\beta} \{N'\}}{\{\mathbb{E}[M \texttt{ <\&> } N]\} \xrightarrow{\alpha \cup \beta} \{\mathbb{E}[M' \texttt{ <|> } N']\}} \text{(CONJ)}$$

Figure 13: The semantics of conjunction.

3.8 Conjunction

The final item in the CHP API is conjunction, written using the `<&>` operator. Conjunction can be thought of as the dual of choice. The expression `a <|> b` waits for either `a` or `b` to occur, but the conjunction `a <&> b` waits for both `a` and `b` to occur together. The API features that operator and a list-form:

```
(<&>) :: CHP a -> CHP b -> CHP (a, b)
every :: [CHP a] -> CHP [a]
```

The semantics of conjunction is given in figure 13. Conjunction illustrates why we use sets of events with our transition arrow: conjoining events merges their sets of events together. Intuitively, the conjunction operator conjoins the leading actions of the two sides (much as choice chooses between the leading actions), and any further actions are executed in parallel as per the normal parallel operator.

Conjunction is a novel feature, and is discussed in chapter 5, along with examples and details of its implementation.

3.9 CHP Examples

We have introduced the key primitives of CHP: parallel composition, channels, barriers and choice. We can now explore some simple example programs in CHP.

3.9.1 Simple Example

Here is a complete program, with a process that reads in a line of input, which sends it to a forwarding process which sends it back to the original process which prints it out:

```
import Control.Concurrent.CHP

forward :: Chanin a -> Chanout a -> CHP ()
forward input output = readChannel input >>= writeChannel output
```

```
getPut :: Chanout String -> Chanin String -> CHP ()
getPut output input
  = do liftIO_CHP getLine >>= writeChannel output
       readChannel input >>= liftIO_CHP . putStrLn

main :: IO ()
main = runCHP $ do
  (cr, cw) <- newChannelRW
  (dr, dw) <- newChannelRW
  forward cr dw <||> getPut cw dr
```

The forward process forwards a single item from one channel to another, and illustrates the common form of a CHP process: it takes its synchronisation/communication primitives (i.e. channels and barriers) as arguments and performs a CHP action. In chapter 4 we will look at ways of simplifying the code to create channels and pass around channel-ends.

3.9.2 Overwriting Buffer

We give here an example of an overwriting buffer (rather than a complete CHP program), which illustrates a use of choice:

```
overwritingBuffer :: Int -> Chanin a -> Chanout a -> CHP ()
overwritingBuffer n input output
  | n <= 0    = return ()
  | otherwise = go []
  where
    go s | null s = takeIn
         | length s == n = sendOut <|> takeInOverwrite
         | otherwise = takeIn <|> sendOut
      where
        takeIn = do x <- readChannel input
                    go (s ++ [x])
        takeInOverwrite = do x <- readChannel input
                             go (tail s ++ [x])
        sendOut = do writeChannel output (head s)
                     go (tail s)
```

Lists are not the most efficient data-type for this purpose, but we use them in our example for simplicity.

3.9.3 Dining Philosophers

A well-known concurrency example is the problem of the dining philosophers. We will discuss a novel solution to this problem in chapter 5, but here we present the classic deadlocking version to help introduce CHP:

```
import Control.Concurrent.CHP
import Control.Monad
import System.Random

data DownUp = Down | Up deriving (Eq, Enum, Bounded, Show)

type Bar = PhasedBarrier DownUp

philosopher :: Bar -> Bar -> CHP ()
philosopher leftBar rightBar
  = forever $ do randomDelay -- Thinking
                 (Up, Up) <- syncBarrier leftBar <||> syncBarrier rightBar
                 randomDelay -- Eating
                 (Down, Down) <- syncBarrier leftBar <||> syncBarrier rightBar
                 return ()
  where
    randomDelay = liftIO_CHP (getStdRandom (randomR (500000, 1000000)))
                    >>= waitFor

diningFork :: Bar -> Bar -> CHP ()
diningFork leftBar rightBar
  = forever (upDown leftBar <|> upDown rightBar)
  where
    upDown bar = do Up <- syncBarrier bar
                    Down <- syncBarrier bar
                    return ()

college :: Int -> CHP ()
college size
  = do b <- newPhasedBarrier Down
```

```
        enroll b (\bA -> enroll b (\bB ->
          foldr1 couple (concat $ replicate size [diningFork, philosopher])
            bA bB))

couple :: (Bar -> Bar -> CHP ())
       -> (Bar -> Bar -> CHP ())
       -> (Bar -> Bar -> CHP ())
couple p q left right
  = do b <- newPhasedBarrier Down
       enroll b (\bP -> enroll b (\bQ -> p left bP <||> q bQ right))
       return ()

main :: IO ()
main = runCHP (college 5)
```

The philosopher process takes as its arguments the barriers for synchronising with the forks to its left and right respectively. The philosopher waits, then picks up both forks in parallel, then waits again, then puts them both down in parallel. This code makes use of phased barriers to indicate that the two synchronisations on a particular barrier are first to pick up the forks, then to put them down.

The fork process continously chooses between being picked up and put down by the philosopher on its left or on its right. Note that the choice is between the first action; it chooses between the pick-up synchronisation on its left or the pick-up synchronisation on its right. Once one has happened, it is committed to execute the corresponding code, which is a put-down synchronisation on the same channel.

The forks and philosophers are wired together in a compositional manner, wiring pairs of processes together into a line using couple and finally sealing the line into a cycle in college. This technique is presented in much more detail (with combinators that can hide away the details) in chapter 4, while a novel solution to the dining philosophers deadlock using conjunction is presented in section 5.3.1.

3.9.4 Boids

As a further example we present an implementation of the classic boids simulation in CHP. We omit all of the logic for calculating new directions and new positions (which are uninteresting in terms of the concurrency) and instead focus on the architecture of the simulation.

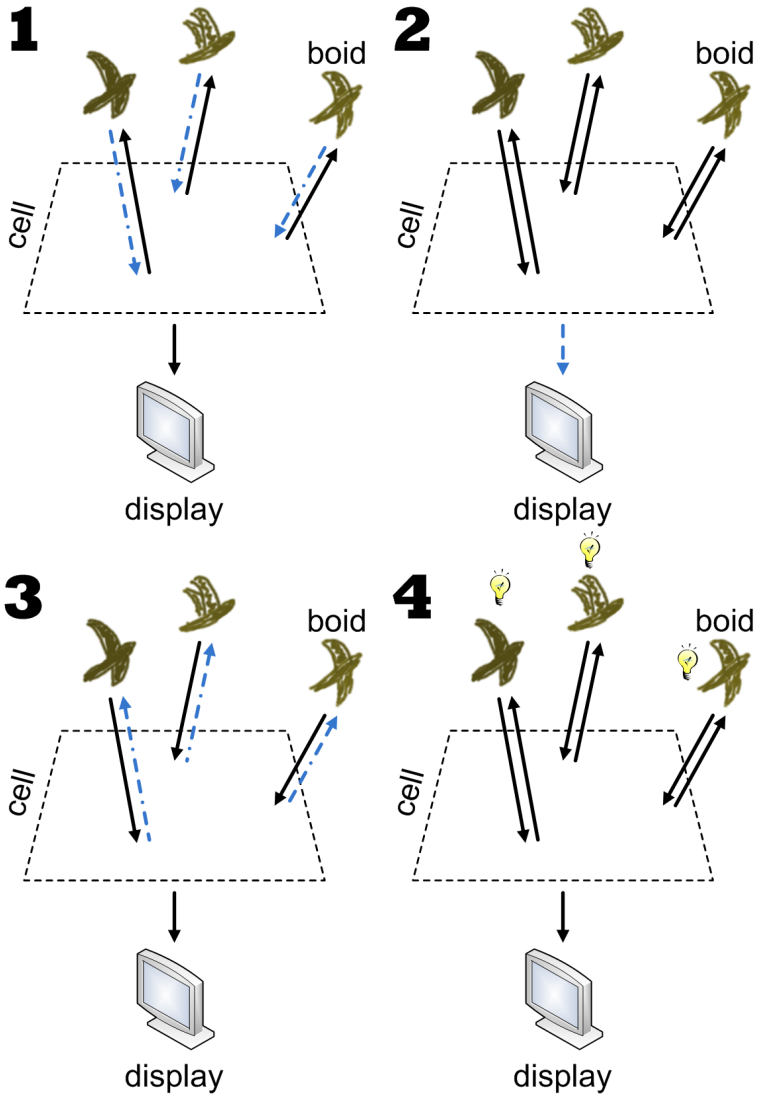


Figure 14: The architecture and communication pattern of the boids example. The arrows are channels, depicted in dashed form when they are being communicated on (and black solid otherwise). In the first phase, the many boid process communicate to the cell. In the second phase, the cell communicates to the display process. In the third phase, the cell communicates back to the boids. In the fourth phase the boids update their velocity, and then phase 1 begins again.

The simulation features many boids. Each boid knows some local state and its current velocity. It does not know its position, and it does not interact directly with any other boids. All the boids interact only with a "cell" process which communicates information to each boid about nearby neighbours. The cell is also responsible for sending information about boid positions to a drawing process that visualises the boids on-screen. The architecture and phases of communication are depicted in figure 14. We begin the code with relevant imports and all the boid types and useful non-monadic functions:

```
import Control.Applicative
import Control.Concurrent.CHP
import Control.Monad
import System.Random

data BoidVel = BoidVel { velX, velY :: Float }
data BoidNeighbour
  = BoidNeighbour { relX, relY :: Float, neighbourVel :: BoidVel }
data BoidInfo = BoidInfo { posX, posY :: Float, vel :: BoidVel }

updateInfo :: BoidInfo -> BoidVel -> BoidInfo
updateInfo = ...

-- For each boid, makes a list of neighbours in the given radius
nearby :: Float -> [BoidInfo] -> [[BoidNeighbour]]
nearby = ...
```

The boid process repeatedly: sends out its current velocity, reads in information about its neighbours (from the cell) and updates its velocity. Its definition is:

```
boid :: Chanout BoidVel -> Chanin [BoidNeighbour] -> BoidVel -> CHP ()
boid out input = boid' startState
  where
    boid' state cur
      = do writeChannel out cur
           neighbours <- readChannel input
           let (state', new) = update neighbours (state, cur)
           boid' state' new

    update = ...
    startState = ...
```

The cell process repeatedly: reads in the velocities of all the boids, updates each boid position accordingly, sends out the positions (to the drawing process) and then sends neighbour information to each boid. Its definition is:

```
cell :: [((Float, Float), Chanout [BoidNeighbour], Chanin BoidVel)]
     -> Chanout [BoidInfo] -> CHP ()
cell boidInfo outputPos = cell' (map makePos startPos)
  where
    (startPos, outputs, inputs) = unzip3 boidInfo
    makePos (x, y) = BoidInfo x y (BoidVel 0 0)

    cell' pos = do vels <- mapM readChannel inputs
                   let newPos = zipWith updateInfo pos vels
                   writeChannel outputPos newPos
                   zipWithM_ writeChannel outputs (nearby 0.05 newPos)
                   cell' newPos
```

Finally, the main process creates all the necessary channels, assigns the boids a starting position, and wires the whole process network together:

```
main :: IO ()
main = runCHP $ do
  let rand = replicateM 150 randomIO
  boidStartPos <- liftIO_CHP (zip <$> rand <*> rand)
  (cellFromBoid, boidToCell) <- unzip <$> replicateM 150 newChannelRW
  (boidFromCell, cellToBoid) <- unzip <$> replicateM 150 newChannelRW
  (drawRead, drawWrite) <- newChannelRW

  let (*$*) = zipWith ($) -- apply a list of functions to a list of arguments
      makeStartVel (x, y) = BoidVel (0.01 * (x - 0.5)) (0.01 * (y - 0.5))
      boids = repeat boid *$* boidToCell
                          *$* boidFromCell
                          *$* map makeStartVel boidStartPos

  parallel boids
    <||> cell (zip3 boidStartPos cellToBoid cellFromBoid) drawWrite
    <||> forever (do boidPos <- readChannel drawRead
                     liftIO_CHP (drawBoids boidPos)
                 )
  return ()
```

We could have used a reduce channel for the communication between the boids and the cell process, but we chose to use many one-to-one channels for the purposes of illustration. The boids example illustrates how to implement a concurrent simulation in CHP.

3.10 Laws of CHP

In this section we give various short laws of CHP. We use ≅ to indicate a form of equality; these laws indicate an intuitive semantic equality, and ignore small amounts of pure code that would be needed to rearrange the monadic results. For example, a rule may state simply that `p <||> q ≅ q <||> p` or `p <||> skip ≅ p`. Technically, for full equality, these rules should state `let swap (x, y) = (y, x) in p <||> q = swap <$> (q <||> p)` and `fst <$> (p <||> skip) = p`. Such additions obscure the intention of the rules and thus are omitted.

We note that the rules in this section only hold if the user processes do not trap (rethrowing is acceptable) or indefinitely mask asynchronous exceptions. As is noted several times in this chapter, CHP is not intended to be used with asynchronous exceptions.

For brevity, most of the rules are supplied without explicit proofs, but appendix E has a few example proofs. The remaining rules should be provable based on the semantics given in this chapter; wherever this is not obvious, a footnote is added.

3.10.1 Sequence and Parallel

`(p >> q) >> r ≅ p >> (q >> r)`	Sequence is associative
`p >> skip ≅ p`	SKIP is a right-unit of sequence
`stop >> p ≅ stop`	Nothing follows STOP[3]
`(p <||> q) <||> r` ≅ `p <||> (q <||> r)`	Parallel is associative
`p <||> q ≅ q <||> p`	Parallel is commutative
`p <||> skip ≅ skip >> p`	SKIP is a pseudo-unit (left- and right-, given the above law) of parallel
`(p <||> stop) >> q` ≅ `p >> stop`	A parallel with a STOP in it will never complete[4]
`parallel [p] ≅ p`	Parallel on a singleton doesn't matter
`parallel [] ≅ skip >> return []`	

Note that SKIP is *not* a left-unit of sequence, because it will act differently when placed in a choice. The code skip >> p is always ready, and will execute p if chosen. However, p is only ready if its first action is ready.

3.10.2 Choice

(p <|> q) <|> r ≅ p <|> (q <|> r)	Choice is associative
p <|> stop ≅ p	STOP is a right-unit of choice
stop <|> p ≅ p	STOP is a left-unit of choice
skip <|> p ≅ skip	SKIP is an always-ready left-zero of choice
(skip >> p) <|> q ≅ p	SKIP is always ready, and does nothing
alt [p] ≅ p	Choice on a singleton doesn't matter
alt [] ≅ stop	An empty list of choices is never resolved

3.10.3 Conjunction

(p <&> q) <&> r ≅ p <&> (q <&> r)	Every is associative
p <&> q ≅ q <&> p	Every is commutative
stop <&> p ≅ stop	STOP is a (left- and right-, given the above law) zero of every
skip <&> p ≅ p	SKIP is a (left- and right-, given the above law) unit of every
(skip >> p) <&> (skip >> q) ≅ p <||> q	Every is equivalent to parallel if all branches are immediately ready
every [p] ≅ p	Every on a singleton doesn't matter
every [] ≅ skip >> return []	

3.11 Assumptions and Undefined Behaviour

In this chapter we have presented various laws and semantics. In general, these rules avoid the discussion of bottom values (errors and non-terminating computations) with the monadic operations. For example, the results of all the below (where bottom is some Haskell bottom value) are undefined:

[3]There are (deliberately) no rules to transition stop, so neither side of the equivalence can ever progress, thus they are equivalent in their behaviour.

[4]Since stop will not complete, the right-hand side will execute p and then progress no further, thus it is equivalent to the right-hand side.

```
syncBarrier b >> bottom
readChannel c <|> bottom
p <||> bottom
```

The behaviour of primitives with infinite lists is also undefined, for example both of the below are undefined:

```
parallel (repeat (return ()))
alt (repeat (return ()))
```

In general, supplying any bottom of type CHP a to a CHP function will have an undefined behaviour. Bottoms with type Chanin a and similar will obviously also cause problems. The same is true already in the Haskell libraries with bottoms of type IO a; no specification is given for the behaviour, it is instead left to a common-sense assumption that this will cause undefined behaviour.

It is possible to have bottoms at the non-monadic level without error, however. For example, this code will return the two errors without problem:

```
return (error "a") <||> return (error "b")
```

Similarly, this code will communicate an error call over a channel successfully:

```
do (r, w) <- newChannelRW
   writeChannel w (error "") <||> readChannel r
```

The CHP library does make some assumptions about the use of the library by the programmer:

- There should be a single runCHP call that encompasses all uses of CHP. runCHP should therefore never be nested inside liftIO_CHP.
- One-to-one channels should only ever be used by one writer and one reader at a time.
- Barriers and channels should not be returned from inside an enroll call (which would allow them to be used while not enrolled) and should not be communicated to a process that could use it after the enroll call has completed. The same rule applies to channels and the claim call.
- A process cannot communicate with itself via conjunction. If r and w are the same channel, the behaviour of readChannel r <&> writeChannel w () is undefined. Similarly, a conjunction of synchronisations on the *same* barrier will have undefined behaviour.

- Conjunction should not contain choices or timeouts in either branch. That is, (a <|> b) <&> c is illegal.
- During an extended communication, processes should not use the channel that the communication is taking place on, other than to poison it.
- Writing to a broadcast channel with no enrolled readers (or reading from a reduce channel with no enrolled writers) has undefined behaviour.

Several of these assumptions could be statically checked by encoding extra information in the monad using parameterised monads [Atkey, 2009]. One approach is to effectively store the in-scope channels in the type of CHP monad, and use type-level indexes to access them. This would allow, for example, any parallel composition to be statically checked (as part of type-checking) to ensure that each one-to-one channel is only used by one writer and one reader. This is theoretically interesting, but practically it greatly complicates the library; to use **do**-notation we must replace the standard Haskell Monad type-class (using GHC's rebindable syntax), and the error messages from the type-checker when there is an unsafe usage are very hard to comprehend.

Similarly, a monadic region approach [Kiselyov and Shan, 2008] could be used to make the access for enroll and claim safer. This is less burdensome than the parameterised monad approach, but may still involve adding extra lifting calls inside the monadic region. This could be explored in future work.

Most of these rules are easy to follow and quite intuitive. The main rule that programmers may accidentally break is the correct use of one-to-one channels. This can be because of a misconceived process network or because they should have used a shared channel instead. The first problem can be alleviated by using of the wiring combinators that will be introduced in chapter 4, which enforced the one-to-one nature of the channels. The latter problem has no obvious solution, other than the the aforementioned parameterised monad solution.

3.12 Implementation

We can now explain the implementation of the core of CHP, without poison (which is introduced next, in section 3.13). The concurrency parts of CHP are built using forkIO and the Software Transactional Memory (STM) library, with occasional use of MVars (which are typically faster than STM) where possible. The simplest implementation of CHP uses a very basic reification with a GADT:

```
data CHP a where
  IO :: IO a -> CHP a
  Return :: a -> CHP a
  Bind :: CHP a -> (a -> CHP b) -> CHP b
  Choice :: [(Guard, CHP a)] -> CHP a
```

This states that the CHP monad can have four different values. It can be a lifted IO action, a return or bind (the standard monadic primitives), or it can be a choice between a list of guards with associated actions.

The monad instance is trivial with this representation:

```
instance Monad CHP where
  return = Return
  (>>=) = Bind
```

The implementation of a function like liftIO_CHP is also trivial:

```
liftIO_CHP :: IO a -> CHP a
liftIO_CHP = IO
```

The data structure is processed by a function like this:

```
goCHP :: CHP a -> IO a
goCHP (Return x) = return x
goCHP (Bind m k) = goCHP m >>= (goCHP . k)
goCHP (IO m) = m
goCHP (Choice gas) = selectFromGuards [(g, goCHP a) | (g, a) <- gas]
```

This function also forms the definition of the top-level runCHP function:

```
runCHP :: CHP a -> IO a
runCHP = goCHP
```

The selectFromGuards function is the one that resolves choices:

```
selectFromGuards :: [(Guard, IO a)] -> IO a
```

The details of the algorithm for resolving choice are in chapter 5. The lengthy but uninteresting matter of implementing selectFromGuards on top of these algorithms is explained in appendix C. Our Guard type is defined as follows:

```
data Guard = TimeoutGuard (IO (STM ()))
           | SkipGuard
           | EventGuard (STM ()) [Event]
```

The SkipGuard is the simple primitive guard that is always ready to complete. The TimeoutGuard represents a timeout, and is an IO action that sets up the timeout guard and returns an STM action that will succeed once the timeout has elapsed. The EventGuard has a list of events (in the absence of conjunction, this is a singleton) and an STM action to perform if the events are chosen.

3.12.1 Channels

On top of this Guard type we can build our channels as follows (using channels built with the STM library). Our readChannel and writeChannel functions delegate to suffixed versions that construct the guards and bodies:

```
data Chanin a = Chanin (STMChannel a)

data Chanout a = Chanout (STMChannel a)

readChannel :: Chanin a -> CHPBasic a
readChannel (Chanin c) = Choice [readChannelG c]

writeChannel :: Chanout a -> a -> CHPBasic a
writeChannel (Chanout c) x = Choice [writeChannelG c x]
```

Note that all communications on channels (and synchronisations on barriers) are implemented as choices. Even though this process may not compose its readChannel call in a choice (or a conjunction), the process at the other end may do so (and we cannot know *a priori*), and if either side is choosing we must use the choice algorithm that can handle this. Therefore we *always* use our choice algorithm for all communications and synchronisations, even though we pay a performance penalty if neither side actually needs choice or conjunction for a particular synchronisation.

The channel is implemented using the STMChannel type which combines an Event (see chapter 5) with a TVar for sending data and acknowledgements between the writer and reader:

```
newtype STMChannel a = STMChannel (Event, TVar (Maybe a, Bool))

newChannelRW :: CHP (Chanin a, Chanout a)
newChannelRW = liftIO_CHP $ do
  tv <- newTVarIO (Nothing, False)
  e <- newEvent
```

```
  let c = STMChannel (e, tv)
  return (Chanin c, Chanout c)
```

We define several helper functions for using the STMChannel. One pair of functions sends data by adding it to the STMChannel's TVar and consumes it (waiting, by retrying, if is not present):

```
sendData :: TVar (Maybe a, Bool) -> a -> STM ()
sendData tv x = do (_, b) <- readTVar tv
                   writeTVar tv (Just x, b)

consumeData :: TVar (Maybe a, Bool) -> STM a
consumeData tv = do (mx, b) <- readTVar tv
                    case mx of
                      Just x -> do writeTVar tv (Nothing, b)
                                   return x
                      Nothing -> retry
```

A similar pair of functions does the same with acknowledgements in the same TVar:

```
sendAck :: TVar (Maybe a, Bool) -> STM ()
sendAck tv = do (x, _) <- readTVar tv
                writeTVar tv (x, True)

consumeAck :: TVar (Maybe a, Bool) -> STM ()
consumeAck tv = do (mx, b) <- readTVar tv
                   if b
                     then writeTVar tv (mx, False)
                     else retry
```

We can then define the readChannelG and writeChannelG functions as follows:

```
readChannelG :: STMChannel a -> (Guard, CHP a)
readChannelG (STMChannel (e, tv))
  = (EventGuard (sendAck tv) [e], liftIO_CHP (atomically (consumeData tv)))

writeChannelG :: STMChannel a -> a -> (Guard, IO ())
writeChannelG (STMChannel (e, tv)) x
  = (EventGuard (sendData tv x) [e], liftIO_CHP (atomically (consumeAck tv)))
```

The readChannelG function sends the acknowledgement at the same time that the event completes, and the writeChannelG function similarly sends the data at

the same time that the event completes. This means that the data and acknowledgement will definitely be ready after the event completes, by the time that the two bodies occur afterwards, which consume the data and the acknowledgement.

With only these simple read and write functions, the acknowledgement is superfluous; it will always be present by the time that the reader consumes it, and no extra synchronisation is necessary after the event completes (which was already the necessary synchronisation). The acknowledgement is needed to support extended communication. We define our extended communications as follows:

```
extReadChannelG :: STMChannel a -> (a -> CHP b) -> [(Guard, CHP b)]
extReadChannelG (STMChannel (e, tv)) block
  = [(EventGuard (return ()) [e], body)]
  where
    body = do x <- liftIO_CHP (atomically (consumeData tv))
              r <- block x
              liftIO_CHP (atomically (sendAck tv))
              return r

extWriteChannelG :: STMChannel a -> CHP a -> [(Guard, CHP ())]
extWriteChannelG (STMChannel (e, tv)) block
  = [(EventGuard (return ()) [e], body)]
  where
    body = do x <- block
              liftIO_CHP (atomically (sendData tv x))
              liftIO_CHP (atomically (consumeAck tv))
```

The extended read consumes the data as part of the body in the same way as the normal read does. However, whereas the normal read "pre-loaded" the channel with an acknowledgement, the extended read only sends the acknowledgement after the extended action has completed. This means that the normal write will have to wait during its body for the acknowledgement, rather than completing immediately as it did with the normal write.

The extended write consumes the acknowledgement as part of the body in the same way as the normal write does. However, before that it executes its block and then sends the data on the channel. This means that the reader will not be immediately able to consume the data, and will have to wait until the writer has made it available. Note that it is important that the extended write sends the data and consumes the acknowledgement in two separate transactions. If they were one transaction, the data would only be available after the acknowledgement

could be consumed, which would cause a deadlock if an extended reader (which must consume the data before sending the acknowledgement) communicated with an extended writer.

Shared Channels

Shared channels are implemented using an MVar for the Mutex type:

```
type Mutex = MVar ()

claimMutex :: Mutex -> CHP ()
claimMutex = liftIO_CHP . takeMVar

releaseMutex :: Mutex -> CHP ()
releaseMutex = liftIO_CHP . flip putMVar ()

claim :: Shared c a -> (c a -> CHP b) -> CHP b
claim (Shared (m, c)) k
  = (claimMutex m >> k c) `finallyCHP` releaseMutex m
```

3.12.2 Choice

The implementation of the choice operator uses the Choice constructor, but flattens all the nested choices into a single list:

```
(<|>) :: CHP a -> CHP a -> CHP a
(<|>) a b = Choice (asGuards a ++ asGuards b)

asGuards :: CHP a -> [(Guard, CHP a)]
asGuards (Choice gs) = gs
asGuards (IO e) = [(SkipGuard, IO e)]
asGuards (Return x) = [(SkipGuard, Return x)]
asGuards (Bind m k) = [(g, Bind a k) | (g, a) <- asGuards m]

empty :: CHP a
empty = Choice []

skip :: CHP ()
skip = Choice [(SkipGuard, Return ())]
```

Note that the processing of Bind descends the left-hand side to find the left-most guard. The implementation of empty (i.e. stop) simply has no guards in the list.

3.12.3 Conjunction

The conjunction operator also uses the Choice constructor:

```
(<&>) :: CHP a -> CHP b -> CHP (a, b)
(<&>) a b = Choice (conjoin (asGuards a) (asGuards b))
  where
    conjoin [] _ = []
    conjoin _ [] = []
    conjoin [(SkipGuard, a)] [(SkipGuard, b)] = [(SkipGuard, a <||> b)]
    conjoin [(SkipGuard, a)] [(bg@(EventGuard _ _), b)] = [(bg, a <||> b)]
    conjoin [(ag@(EventGuard _ _), a)] [(SkipGuard, b)] = [(ag, a <||> b)]
    conjoin [(EventGuard aact aes, a)] [(EventGuard bact bes, b)]
      = [(EventGuard (aact >> bact) (aes ++ bes), a <||> b)]
    conjoin _ _ = error
      "Timeout guards and choices not allowed in conjunction"
```

3.12.4 Parallel

The main complication with parallel composition is the addition of code to handle the chosen exception semantics. We must make use of the existing Haskell primitive forkIO :: IO () -> IO ThreadId from Control.Concurrent and several functions from the Control.Exception module (in `base-4.3`):

```
catch :: Exception e => IO a -> (e -> IO a) -> IO a
throwTo :: Exception e => ThreadId -> e -> IO ()
mask :: ((forall b. IO b -> IO b) -> IO a) -> IO a
```

The catch function is a standard exception-handling function, while throwTo throws an asynchronous exception to another thread. An asynchronous exception may affect a thread while it is not masked, or while it is masked and in an interruptible operation (for the code below, that is the transaction featuring retry). The mask function masks supplies a "restore" function to its first parameter, and then masks any parts not surrounded by the restore function. That is, assuming that it is not inside an outer mask call, the code:

```
mask (\restore -> do a
                     restore b
                     c)
```

will perform a without registering asynchronous exceptions sent to it (unless a performs an interruptible operation), then it will perform b, during which it will notice asynchronous exceptions that it receives (or that were received during a), and finally it will perform c without registering asynchronous exceptions sent to it (unless c performs an interruptible operation). This is a newer version of the original block/unblock operations set out in the original paper [Marlow et al., 2001].

Here is the code for parallel composition:

```
(<||>) :: CHP a -> CHP b -> CHP (a, b)
(<||>) p q = IO $ mask $ \restore -> do
  ptv <- atomically (newTVar Nothing)
  qtv <- atomically (newTVar Nothing)
  pid <- forkIO (restore (Right <$> goCHP p)
                   `catch` (\(e::SomeException) -> return (Left e))
                 >>= atomically . writeTVar ptv . Just)
  qid <- forkIO (restore (Right <$> goCHP q)
                   `catch` (\(e::SomeException) -> return (Left e))
                 >>= atomically . writeTVar qtv . Just)
  let wait = restore (atomically $
                        (,) <$> (readTVar ptv >>= maybe retry return)
                            <*> (readTVar qtv >>= maybe retry return)
              ) `catch` (\(e::AsyncException) -> do throwTo pid e
                                                    throwTo qid e
                                                    wait)
  wait >>= merge

merge :: (Either SomeException a, Either SomeException b) -> IO (a, b)
merge (Left e, _) = throw e
merge (_, Left e) = throw e
merge (Right x, Right y) = return (x, y)
```

3.13 Poison

Poison is a technique for safely shutting down a process network, without inviting deadlock or forcefully aborting processes [Brown and Welch, 2003; Hilderink, 2005;

Welch, 1989b]. A channel can either be in a normal operating state, or it can be poisoned. Any attempt to read or write on a poisoned channel will result in a poison exception being thrown. A channel may never be cured of poison; once poisoned, it remains so forever.

Poison propagates throughout a network as follows. When a process catches a poison exception, it should poison all its channel-ends and barriers. Thus its neighbours (according to channel and barrier connections in a process graph) will also get poison thrown, and they will do the same, until all channels in a process network have been poisoned. Once processes have poisoned their channels, they shut down, and thus the process network terminates. This mechanism has previously been incorporated into C++CSP, JCSP and other libraries.

We define the following primitives:

```
onPoisonTrap :: CHP a -> CHP a -> CHP a
throwPoison :: CHP a

class Poisonable a where
  poison :: a -> CHP ()

instance Poisonable (Chanin a)
instance Poisonable (Chanout a)
instance Poisonable (PhasedBarrier phase)
```

The function onPoisonTrap runs its second argument when poison is thrown in its first argument. The throwPoison function throws poison directly, while poison can be used to introduce poison into channels and barriers. It is then trivial to define an additional helper function:

```
onPoisonRethrow :: CHP a -> CHP b -> CHP a
onPoisonRethrow a b = a `onPoisonTrap` (b >> throwPoison)
```

Processes should surround their outermost code with a call to onPoisonRethrow that poisons all their channels and barriers. For example, here is the identity process that forever forwards values from one channel to another:

```
id :: Chanin a -> Chanout a -> CHP ()
id input output = go `onPoisonRethrow` (poison input >> poison output)
  where
    go = forever (readChannel input >>= writeChannel output)
```

This process will forever forward values on, but when it encounters poison on either of its channels, the poison exception will "break out" of the forever

$$\mathbb{E} ::= [\cdot] \mid \mathbb{E} >>= M \mid$$
$$\texttt{finallyCHP}\ \mathbb{E}\ M \mid \texttt{onPoisonTrap}\ \mathbb{E}\ M$$

$$\{\mathbb{E}[\texttt{throwPoison} >>= M]\} \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throwPoison}]\} \quad \text{(BIND-POISON)}$$

$$\{\mathbb{E}[\texttt{throwPoison `onPoisonTrap` M}]\} \xrightarrow{\{\}} \{\mathbb{E}[M]\} \quad \text{(POISON-TRAP)}$$

$$\nu c.(\{\mathbb{E}[\texttt{poison c}]\} \mid\mid \Omega(P)) \quad \text{(POISON)}$$
$$\xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return ()}]\} \mid\mid \Omega(P \cup \{c\}))$$

Figure 15: The core operational semantics for poison.

call to the outer poison handler, which will poison both channels (poisoning an already poisoned channel does not cause any problems) to make sure they are both poisoned regardless of which one was already poisoned, and then the poison will be rethrown so that whatever parallel composition `id` is part of will also terminate with poison. In this way, all processes forward poison to all processes they are connected to, and eventually the entire system will shut down.

The core semantics of poison are shown in figure 15. The sequential rules are standard. The poison mechanism effectively has a global set of names (marked with Ω) that are currently poisoned, and poisoning something adds the name to the set.

All of the existing rules for communicating on channels are synchronising on barriers must now be formulaically altered to add a pre-condition that the channel is not poisoned, i.e. $c \notin P$ where P is the set inside the Ω wrapper. Additionally, rules are required that throw poison when an attempt is made to use a poisoned channel or barrier. All of these rules are given in full in appendix D.

3.13.1 Choice and Poison

The behaviour of choice in the presence of poison falls out of the semantics; a read or write on a poisoned channel is an always-ready event. This does not mean that a choice will always detect poison in its channels: `readChannel c <|> readChannel d` can successfully read from `c` if `c` is ready even if `d` is poisoned. However, poison will always be preferred on the left to choices on the right; if `c` is poisoned, the choice will always throw poison regardless of the readiness of `d`. This is explored via the formal semantics in appendix E (section E.6).

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\texttt{return x}\}_u \parallel \{\texttt{throwPoison}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throwPoison}]\}_t \quad \text{(E-PAR-RP)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\texttt{throwPoison}\}_u \parallel \{\texttt{return y}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throwPoison}]\}_t \quad \text{(E-PAR-PR)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\texttt{throwPoison}\}_u \parallel \{\texttt{throwPoison}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throwPoison}]\}_t \quad \text{(E-PAR-PP)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\texttt{throw e}\}_u \parallel \{\texttt{throwPoison}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw e}]\}_t \quad \text{(E-PAR-TP)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \parallel \{\texttt{throwPoison}\}_u \parallel \{\texttt{throw e}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw e}]\}_t \quad \text{(E-PAR-PT)}$$

Figure 16: The semantics for poison and parallel composition. Parallel composition still waits until both sides have terminated, and then when merging the results, exceptions take precedence over poison, which takes precedence over successful termination.

3.13.2 Parallelism and Poison

One previous problem with poison has been deciding on the semantics of poison and parallel composition. In short, the question arose: when p and q are composed in parallel and p exits due to poison, what should happen to q, and what should happen to their parent process?

Forcibly killing off q is an ugly solution that goes against the main principle of poison (allowing for controlled termination). Doing nothing at all is an odd solution, because the parent will not know whether its sub-processes terminated successfully or died because of poison. Consider the following process:

```
delta2 :: Chanin a -> Chanout a -> Chanout a -> CHP ()
delta2 input output0 output1
  = forever (do x <- readChannel input
                writeChannel output0 x <||> writeChannel output1 x)
    `onPoisonRethrow` (do
      poison input
      poison output0
      poison output1)
```

If the parent is never notified about its subprocesses dying of poison, the delta2 process would continue running if one, or even *both*, of its output channels was poisoned, because the poison exception would be masked by the parallel composition.

The semantics we have chosen is straightforward. The parent process spawns off all the sub-processes, and waits for them *all* to complete, either normally (no poison) or abnormally (with poison). Once they have all completed, if *any* of the sub-processes exited in a state of exception (with poison), the parallel function (or similar) throws a poison exception in the parent process.

This solution corresponds to the ideas in Hilderink's CSP exception operator [Hilderink, 2005] and Hoare's concurrent flowcharts [Hoare, 2007]. It maintains the associativity of PAR – the following two lines are semantically equivalent:

```
parallel [ parallel [p, q], r]
parallel [p, parallel [q, r]]
```

The other useful property that is preserved is that running one process in parallel is the same as running the process directly: parallel [p] is semantically identical to p. Commutativity of parallel composition is also maintained.

3.13.3 Augmented Implementation

To support poison, we modify the implementation of our earlier CHP monad to changed the type of the IO item and add a handler item, Trap:

```
data WithPoison a = Poison | NoPoison a

data CHP a where
  IO :: IO (WithPoison a) -> CHP a
  Return :: a -> CHP a
  Bind :: CHP a -> (a -> CHP b) -> CHP b
  Choice :: [(Guard, CHP a)] -> CHP a
  Trap :: CHP a -> CHP a -> CHP a
```

The modifications to liftIO_CHP and goCHP are straightforward:

```
liftIO_CHP :: IO a -> CHP a
liftIO_CHP = IO . liftM NoPoison

goCHP :: CHP a -> IO (WithPoison a)
goCHP (Return x) = return (NoPoison x)
```

```
goCHP (Bind m k) = do v <- goCHP m
                      case v of
                        NoPoison x -> goCHP (k x)
                        Poison -> return Poison
goCHP (IO m) = m
goCHP (Choice gas) = selectFromGuards [(g, goCHP a) | (g, a) <- gas]
goCHP (Trap m h) = do v <- goCHP m
                      case v of
                        NoPoison x -> return (NoPoison x)
                        Poison -> goCHP h
```

The selectFromGuards function has been appropriately re-typed:

```
selectFromGuards :: [(Guard, IO (WithPoison a))] -> IO (WithPoison a)
```

Finally, we must adjust the implementation of parallel composition, by adjusting the type and implementation of the merge function:

```
merge :: ( Either SomeException (WithPoison a)
         , Either SomeException (WithPoison b))
      -> IO (WithPoison (a, b))
merge (Left e, _) = throw e
merge (_, Left e) = throw e
merge (Right Poison, _) = return Poison
merge (_, Right Poison) = return Poison
merge (Right (NoPoison x), Right (NoPoison y)) = return (NoPoison (x, y))
```

3.13.4 Comparison to Other Termination Mechanisms

Poison allows a process network to be terminated gracefully, with each process handling poison (with a chance to close open files and so on) and spreading it to its neighbouring processes. It removes some of the reasoning about deadlock that follows if processes attempt to send poison messages around the channels in a process network (or the reasoning about who is responsible for killing who, with things like asynchronous exceptions).

One of poison's drawbacks is that it is only effective if processes frequently communicate on their channels. A process that computes for a long time will not terminate until it attempts to communicate the result on a channel. A process that is blocked waiting for a network message will not terminate until a message is received and it then communicates on a channel. This is a major limitation of

poison that is addressed by things like asynchronous exceptions: poison cannot terminate processes blocked on external I/O actions.

One interesting feature of Haskell is that if a thread is blocked on a communication primitive that is not referred to by another thread, it is identified as deadlocked (no other thread can ever touch the primitive to wake the thread) and is woken with an appropriate exception. This mechanism can be used to shut down concurrent process networks. However, it is too fragile to rely on completely: if another thread does retain a reference to the communication primitive, even if it will never use it, the original process will not be identified as deadlocked.

3.14 Indicative Benchmarks

The work in this thesis is primarily concerned with programming models, expressive power and concurrent capabilities rather than with speed. However, to give an idea of the cost in terms of speed of the expressive power of CHP as against other process-oriented systems, we provide the results of some simple benchmarks.

All the benchmarks have been tested on three systems – CHP, occam-π and JCSP – on the same Intel Core 2 Duo (i.e. dual-core) machine running Ubuntu Linux 10.10 with GHC 7.0.3. Results were recorded and summarised by the `criterion` benchmarking library for Haskell, and measured the complete execution time of the program (so that things such as garbage collection are included in the time). Each program was run 100 times. All confidence intervals were calculated using a non-parametric (i.e. without assumption of underlying distribution) bootstrap with bias-corrected acceleration [Efron and Tibshirani, 1993] using 100000 resamples. All results are given in seconds.

We give the program code in CHP for each benchmark; the code for occam-π and JCSP was a simple mechanical translation of this code.

3.14.1 Parallel Composition

The parallel composition benchmark measured the overhead of repeated parallel compositions:

```
import Control.Concurrent.CHP
import Control.Monad

main :: IO ()
```

```
main = runCHP $ replicateM_ iterations $ parallel_ $ replicate par (return ())
  where
    iterations = 10000
    par = 100
```

The results were as follows:

System	Mean	95% CI
occam-π	0.032	0.031 – 0.032
CHP	1.054	1.045 – 1.062
JCSP	2.462	2.446 – 2.487

It can be seen that occam-π, which uses its own fast purpose-designed scheduler [Ritson et al., 2009] is much faster than the other two systems. The Haskell runtime uses a threading system with multiple Haskell threads per single OS thread, while JCSP uses an OS thread for each process. The overhead of OS threads explains the low performance of the JCSP library.

3.14.2 CommsTime

CommsTime is a ubiquitous benchmark in the process-oriented programming literature. It features a ring of three processes: a prefix process that sends out a value then forever forwards values on, a successor process that forwards integers, incrementing them as they pass through, and a delta process that forwards values on while also sending them to a fourth consumer process. The benchmark primarily tests the costs of communication, in a small network with little or no computation. The CHP code (which uses the common processes that are provided in the `chp-plus` library) is as follows:

```
import Control.Monad

import Control.Concurrent.CHP
import Control.Concurrent.CHP.Connect
import qualified Control.Concurrent.CHP.Common as Common

consumer :: Chanin Int -> CHP ()
consumer c =
  do replicateM_ 10000 $ readChannel c
     poison c
```

```
main :: IO ()
main = runCHP $
  (do (rR, wR) <- newChannelRW
      consumer rR <|*|> cycle [Common.prefix 0, Common.succ,
                               \r w -> Common.parDelta r [w, wR]]
      return ()
  ) `onPoisonTrap` return ()
```

The cycle function is introduced in chapter 4. The results were as follows:

System	Mean	95% CI
occam-π	0.017	0.017 – 0.018
CHP	0.322	0.322 – 0.323
JCSP	2.354	2.329 – 2.387

There is an order of magnitude difference between occam-π and CHP, and between CHP and JCSP. Given that CHP supports three major extra capabilities on channels than occam-π (choice by writers, conjunction and poison) and one more than JCSP (conjunction), this is quite a good result. JCSP undoubtedly suffers again from using OS threads, which makes the frequent context-switching in the CommsTime benchmark carry a large penalty.

3.14.3 Choice

The final benchmark tests the speed of choice by having a single reader choosing between 20 channels, each of which has an attached writer that writes many values:

```
import Control.Applicative
import Control.Concurrent.CHP
import Control.Monad

main :: IO()
main = runCHP_ $
  do (rs, ws) <- unzip <$> replicateM 20 newChannelRW
     parallel_ $
       replicateM_ (20 * 5000) (alt (map readChannel rs))
       : [replicateM_ 5000 (writeChannel w (0::Int)) | w <- ws]
```

The results were as follows:

$$\mathsf{runCHP\ p} \equiv \{\mathsf{p}\} \parallel \beta(\{\}) \parallel \gamma(\{\}) \parallel \Omega(\{\})$$

Figure 17: The semantics for runCHP, which introduces the single β, γ and Ω sets.

System	Mean	95% CI
occam-π	0.008	0.008 – 0.008
CHP	1.357	1.356 – 1.358
JCSP	0.356	0.351 – 0.361

The results were not as strong for CHP here. It is during this larger choice that the library pays the penalty for the extra conjunction functionality it supports.

It is instructive to compare the results of this benchmark (which features a total of 10000 communications, but uses choice) with those for CommsTime (which features 40000 communications in all, but without choice). occam-π is only twice as fast for this benchmark, which suggests that the choice is a penalising factor. JCSP is around 8 times faster for this benchmark, which suggests that its choice implementation is fast. Further investigation revealed that the reader communicated with each writer repeatedly (rather than different writers in turn) which may have taken more advantage of caching (or of the dual-cores) than the commstime benchmark. CHP is around four times slower for this benchmark, which suggests that the large choice carries a comparatively large penalty.

3.15 Final Semantics and Complete API

The semantics for the top-level function runCHP is given in figure 17. This introduces the β, γ and Ω sets seen in the previous rule. There is only ever a single set for each throughout the entire program. Note that any renaming by the ALPHA rule must also perform renaming on the keys in the maps (and members in the sets).

The complete final semantics and API for CHP are given in appendix D.

3.16 Correspondence to CSP

Most process-oriented frameworks are directly or indirectly based on the CSP calculus, and this is true of CHP too. CSP is a declarative language; no assignment

features in CSP, only (immutable) name bindings. Similarly there are no loop structures in CSP, only recursion. These features are closer to those of functional programming than of imperative programming languages such as Java and C++. In this section we compare CSP and CHP in detail.

CSP separates the notions of events from processes. CHP has no such division; a single event (e.g. writing on a channel) has the same type as a sequence of events: `CHP a`. Doing otherwise would make using the monad interface in CHP very awkward.

CSP has event prefixing, e.g. $c!x \rightarrow P$. This is generally represented in CHP using the monadic sequence operator: `writeChannel c x >> p`. CSP's event prefixing also permits the binding of names in inputs, e.g. $c?x \rightarrow P(x)$. This is represented in CHP using monadic bind: `readChannel c >>= p`. The CSP prefix operator has a close correspondence to the monad operations.

CSP allows composition of processes using external choice: this chooses between the leading event synchronisations of each process. As seen in section 3.7.1, CHP copies this behaviour exactly. There is no direct correspondence to internal choice in the CHP library: as Roscoe [1997] notes, "[nondeterministic choices] are not constructs one would be likely to use in any program written for execution in the usual sense."

CSP supports composition of parallel processes – this composition is parameterised by the set of synchronising events. In CHP (and all other process-oriented frameworks), events are dealt with differently, by having a count of participants associated with each event (fixed at two for normal channels).

3.17 Discussion

In this chapter we have seen the design and implementation of the CHP library for Haskell. The CHP library is able to bear a closer resemblance to CSP than most other libraries. This is not purely syntactic; techniques such as choosing between leading actions would simply not be possible in a standard imperative language such as C++.

The functional, immutable nature of Haskell means that processes can be passed around without complications such as aliasing, which is a good fit with CSP's declarative style, and allows for higher-order processes (and processes that take functions as arguments). Most Haskell processes are tail-recursive, which is also true in CSP but not in most other frameworks such as occam or JCSP.

Chapter 4

Process Composition

The previous chapter introduced the basic elements and primitives of the CHP library. In this chapter we examine how these primitives can be hierarchically composed together into processes using high-level combinators that capture wiring patterns or repeated behaviours and reduce repeated code and boilerplate. Some of the work in this chapter has previously been published at Practical Aspects of Declarative Languages 2011 [Brown, 2011a].

We begin the chapter by examining the compositional properties of poison, and show how to reduce the boilerplate required to add poison handling to processes (section 4.1). We then go on to explore the uses of passing and sending functions and channels around between processes (section 4.2).

We explore ways to remove the boilerplate required to wire together a network of processes (sections 4.3 and 4.4) using combinators and monads. Finally, we look at ways to capture sequences of behaviours and choices in other combinators (section 4.5).

All of this chapter builds on top of the core primitives of CHP described in chapter 3, without adding any new primitives or semantics. The combinators described here are intended to replace boilerplate where possible, but not entirely replace the primitives seen in the last chapter. Many CHP programs will contain a mix of combinators for wiring together the uniform parts of the process network, along with some "manual" wiring for the non-uniform or intricate parts.

4.1 Composition and Poison

In this section we show how to compose several very simple processes that involve poison handling, and consider which processes are equivalent. Each process is

given both in Haskell code and using CSP notation (with parameters omitted). We borrow Hilderink's exception-handling operator [Hilderink, 2005]: $P \overrightarrow{\triangle} Q$ behaves as P, but if P throws a poison exception it behaves instead like Q. We also invent a process, $\Omega(..)$ that poisons all channels passed to it, and THROW that throws a poison exception.

The simplest common process in process-oriented programming is the identity process – but this already contains two compositions (sequence and iteration). We start here with a forward process that is one iteration of the identity process:

```
-- CSP: forward = input?x → output!x → SKIP
forward :: Chanin a -> Chanout a -> CHP ()
forward input output = do x <- readChannel input
                          writeChannel output x
```

This can then be composed into several other processes:

```
-- CSP: forwardForever = *forward
forwardForever :: Chanin a -> Chanout a -> CHP ()
forwardForever input output = forever (forward input output)

-- CSP: forwardSealed = forward ▷ (Ω(input, output))
forwardSealed :: Chanin a -> Chanout a -> CHP ()
forwardSealed input output
  = (forward input output)
    `onPoisonTrap` (do poison input
                       poison output)

-- CSP: forwardRethrow = forward ▷ (Ω(input, output) ; THROW)
forwardRethrow :: Chanin a -> Chanout a -> CHP ()
forwardRethrow input output
  = (forward input output)
    `onPoisonRethrow` (do poison input
                          poison output)
```

We include both the latter two processes so that we can demonstrate their relative composability below. Consider these further-composed processes:

```
-- CSP: id1 = forwardForever ▷ (Ω(input, output))
id1 :: Chanin a -> Chanout a -> CHP ()
id1 input output
  = (forwardForever input output)
```

```
    `onPoisonTrap` (do poison input
                       poison output)

-- CSP: id2 = forwardForever ⟿ (Ω(input, output) ⨟ THROW)
id2 :: Chanin a -> Chanout a -> CHP ()
id2 input output
  = (forwardForever input output)
    `onPoisonRethrow` (do poison input
                          poison output)

-- CSP: id3 = *forwardSealed
id3 :: Chanin a -> Chanout a -> CHP ()
id3 input output = forever (forwardSealed input output)

-- CSP: id4 = *forwardRethrow
id4 :: Chanin a -> Chanout a -> CHP ()
id4 input output = forever (forwardRethrow input output)

-- CSP: id5 = (*forwardRethrow) ⟿ SKIP
id5 :: Chanin a -> Chanout a -> CHP ()
id5 input output
  = (forever (forwardRethrow input output))
    `onPoisonTrap` skip
```

Intuitively, id2 is semantically identical to id4, and id1 is semantically identical to id5; proving this is left as an exercise for the reader. We prefer id4 and id5, which locate the poison-handling as close as possible in the composition to the channel-events. Processes id1 and id5 are *not* identical to id3, as the latter will never terminate, even if its channels are poisoned.

We can see that, pragmatically, the forwardRethrow function was much more composable than the forwardSealed function, because it revealed the poison rather than trapping it; processes using forwardRethrow will see the poison and terminate, rather than being oblivious as they would be when using forwardSealed. The implication in turn is that id2 and id4 will prove more composable than their "sealed" counterparts (id1 and id5); we believe that in practice, processes involving poison should always rethrow in order to make them more composable.

4.1.1 Auto-Poison

The recommended way of using poison in a process is to surround the process with an onPoisonRethrow block that poisons the channels and barriers connected to the process, for example:

```
delta :: Chanin a -> [Chanout a] -> CHP ()
delta input outputs
  = forever (do x <- readChannel input
                parallel [writeChannel c x | c <- outputs]
    ) `onPoisonRethrow` (poison input >> mapM_ poison outputs)
```

This can be tiresome, and adds clutter to the code; often in this thesis we will omit poison handling to focus on other aspects of the process. However, we can automate this poisoning for simple processes using a type-class trick. This will allow us to write the following, which is identical in behaviour to the above:

```
delta :: Chanin a -> [Chanout a] -> CHP ()
delta = autoPoison delta'
  where
    delta' input outputs = forever $ do
      x <- readChannel input
      parallel_ [writeChannel c x | c <- outputs]
```

This is done using an AutoPoison type-class which wraps a process with any number of arguments (this is a similar trick to implementing Printf in Haskell):

```
class AutoPoison p where
  autoPoison' :: [CHP ()] -> p -> p

instance AutoPoison (CHP a) where
  autoPoison' pois m = m `onPoisonRethrow` sequence_ pois

instance (AutoPoisonParam a, AutoPoison p) => AutoPoison (a -> p) where
  autoPoison' pois p x = autoPoison' (aPoison x : pois) (p x)
```

The autoPoison' function wraps something of type p, accumulating a list of CHP actions (the poison actions). The final case is the instance for CHP a, which wraps the process with an onPoisonRethrow block that carries out the poisoning. The other instance takes care of all the parameters to the process by poisoning them via the AutoPoisonParam type-class:

```
class AutoPoisonParam a where
  aPoison :: a -> CHP ()

instance AutoPoisonParam (Chanin a) where aPoison = poison
instance AutoPoisonParam (Chanout a) where aPoison = poison
instance AutoPoisonParam (PhasedBarrier phase) where aPoison = poison

instance AutoPoisonParam Int where aPoison _ = return ()
instance AutoPoisonParam Char where aPoison _ = return ()
```

There are instances for things which can be poisoned (e.g. Chanin a) and things which cannot be poisoned (e.g. Int). There are also instances for basic compound types, for example:

```
instance AutoPoisonParam a => AutoPoisonParam [a] where
  aPoison = mapM_ aPoison

instance (AutoPoisonParam a, AutoPoisonParam b) => AutoPoisonParam (a, b)
where
  aPoison (x, y) = aPoison x >> aPoison y
```

This takes care of most simple processes. The user can add instances for their own types if they wish. This mechanism is finally wrapped in the autoPoison function:

```
autoPoison :: AutoPoison p => p -> p
autoPoison = autoPoison' []
```

This mechanism works fine for processes that have all their channels and barriers as parameters. We will see in section 4.2.4 some examples of processes for which this is not true, and thus these processes need different/additional poisoning.

4.2 Processes and Functions

CHP processes can take any type as a parameter; common examples include integers, strings and channel-ends. Processes can also take functions as parameters, which allows powerful processes to be constructed. The most basic, but also one of the most useful, is the map process:

```
mapP :: (a -> b) -> Chanin a -> Chanout b -> CHP ()
```

```
mapP f input output
  = forever  (do x <- readChannel input
                 writeChannel output (f x))
```

This is very similar to the identity process but applies a modification function on the value as it is passed through (a strict version that forces evaluation can also be constructed).

It is also possible to construct a filter process[1]:

```
filterP  :: (a -> Bool) ->
            Chanin a -> Chanout a -> CHP ()
filterP  keep input output
  = forever  (do x <- readChannel input
                 when (keep x) (writeChannel output x))
```

This process repeatedly reads in values from an input channel, but only sends on those values which meet the given criteria.

The ability to pass functions into concurrent processes allows *communication* behaviour to be captured in a common process, parameterised by the "business logic" that operates on the values being passed around.

4.2.1 Capturing Common Communication Patterns

Given an arbitrary function that processes a list, we can deduce little of its behaviour. If we know that it is implemented as `map f`, we know much more. For example, we know that composing this with another `map` satisfies the law: `map f . map g = map (f . g)`. So having a common `map` function provides twin benefits: it can be re-used whenever a list needs to be processed one independent element at a time to making writing code shorter and less error-prone, and it provides a recognisable pattern for others who read the code later.

We have already seen the definition of the corresponding `mapP` process. It always produces one output for each input, and it processes each input independently. We can form a generalisation of `mapP` that allows it to also implement `filterP` , by relaxing the constraint that it must always output:

```
mapMaybeP :: (a -> Maybe b) ->
             Chanin a -> Chanout b -> CHP ()
mapMaybeP f input output
  = forever  (do x <- readChannel input
```

[1] In fact, it is possible to construct analogues of most of the standard list processing functions.

```
    case f x of
      Nothing -> return ()
      Just y -> writeChannel output y)
```

The mapMaybeP process takes an input and then, depending on the output of the function, either outputs nothing or a single transformed result. We can generalise this process yet further to allow multiple outputs (zero, one or more) from one input:

```
mapManyP :: (a -> [b]) ->
            Chanin a -> Chanout b -> CHP ()
mapManyP f input output
  = forever (do xs <- readChannel input
                mapM_ (writeChannel output) xs)
```

Neither mapP, nor mapMaybeP, nor mapManyP have any persistent state – all of the processes produce their output solely based on the latest value. Our final generalisation removes this restriction by making the outputs dependent on all previous inputs:

```
mapHistoryP :: ([a] -> [b]) ->
               Chanin a -> Chanout b -> CHP ()
mapHistoryP f input output = mapHistoryP' []
  where
    mapHistoryP' hist
      = do mapM_ (writeChannel output) (f hist)
           new <- readChannel input
           mapHistoryP' (hist ++ [new])
```

The function parameter takes a list of all inputs to the process thus far, and returns a list of values to send out. This is more general than the previous processes because it supports a historical view of all the inputs, and also allows the process to output before input arrives (by allowing the function to return values in response to the empty list).

All the processes in the previous section are focused on sending output in response to input, and do so by wrapping pure functions into processes. This is very common and provides a good fit to stream-processing and data-flow programming. These processes are not, however, the full extent of CHP's capabilities. Processes can also be implemented with many more channels, and complex choice between the channels – section 4.2.3 and section 4.5 have examples of such processes.

4.2.2 Deducing Process Behaviour from Types

The free theorems [Wadler, 1989] and associated ideas of parametricity allow properties of functions to be deduced solely from their type. For example, given the type `a -> a`, it is apparent that since this function must work for any type `a` and values cannot be formed for arbitrary types, the only possible implementation is to return the argument given (the only value available of the correct type). (The free theorems typically ignore the possibility of bottoms.)

A more complex example is the type `(a -> b) -> [a] -> [b]`. From this type it can be reasoned that any values of type `b` that occur in the result list must be the result of applying the given function to an item in the input list. The resulting list must be some collection of transformed original values, potentially filtered, duplicated or permuted. These operations must be done arbitrarily, without access to any information from the values themselves, since this must work for any types `a` and `b` and no functions are given to inspect them.

Similar reasoning can also be used with processes. Given the process with type `Chanin a -> Chanout a ->CHP ()`, we can reason about it similarly to the function with type `[a] -> [a]`. Any items that are sent out of the process must have previously been sent into it – this also means that the process must perform at least one input before it will produce any output. Items could be dropped, duplicated or permuted, but only arbitrarily. We can similarly recognise the types of some non-productive processes. A process with type `Chanout b ->CHP ()` cannot produce any output (in a similar way to a value with type `a` being unimplementable).

The type of a process can inform us about the *values* that the process may send out, based on the values sent in. The type cannot, however, reveal information about the *communication behaviour* of an arbitrary process. A process with type `Chanin a -> Chanout a -> CHP ()` may refuse to read any input, or it may drop all its input, or it may accept one value and then repeatedly output it forever. Communication behaviour can instead be recognised using common processes such as those shown in the previous section.

4.2.3 Dynamic Processes

As well as being passed in as parameters, functions can be sent over channels. So we can construct "dynamic" versions of the processes shown in section 4.2.1, wherein the functions can be changed while the process is running. For example, here is a dynamic version of the `mapP` process:

```
mapDynP :: (a -> b) -> Chanin (a -> b) ->
           Chanin a -> Chanout b -> CHP ()
mapDynP origF finput input output = mapDynP' origF
  where
    mapDynP' f
      = (do x <- readChannel input
            writeChannel output (f x))
            mapDynP' f)
        <|> (readChannel finput >>= mapDynP')
```

This process chooses between receiving a value (and passing it on with the current function applied) or receiving a new function to replace the old one. In this process the recursion is explicit rather than using the forever function, because the behaviour changes according to some persistent state; most of the other processes seen so far have had repeating behaviour without any changing state.

A filterDynP process can be similarly constructed:

```
filterDynP :: (a -> a) -> Chanin (a -> Bool) ->
              Chanin a -> Chanout a -> CHP ()
filterDynP origF finput input output = filterDynP' origF
  where
    filterDynP' f
      =     (do x <- readChannel input
                when (f x) (writeChannel output x)
                filterDynP' f)
        <|> (readChannel finput >>= filterDynP')
```

This process chooses between: reading a value from its input channel, and passing it on if it meets the current filtering criteria (embodied in the function f), and: reading in a new filtering function to replace the old one. Such a process could be useful, for example, in a firewall system. The process would filter out blocked requests, but could be updated because of changes in the configuration while it is running.

Another use of passing functions over channels, and of choice, is to implement a process representing a shared variable. The process has an outgoing channel on which it is willing to send out the current value, and an incoming channel on which it will accept a function to be applied to the current value. The value can be replaced by sending a function of the form const x. The definition of this process is as follows:

```
sharedVar :: a -> Chanin (a -> a) -> Chanout a -> CHP ()
sharedVar start input output = sharedVar' start
  where
    sharedVar' x = (do f <- readChannel input
                       sharedVar' (f x))
                   <|> (do writeChannel output x
                           sharedVar' x)
```

Without the choice operator supporting choice between an input and an output, this process would be impossible to implement with this interface. Without the ability to send functions over channels, it would not be possible to perform an atomic update of the value held.

4.2.4 Sending Channels Over Channels

It is also possible to send processes and channels over channels. This allows programming patterns such as those supported by the π-calculus [Milner, 1999] to be implemented in CHP.

The following process is an identity process that also sends off copies of the messages to registered listeners. Listeners register by sending in the channel-end that they wish to receive messages on:

```
tap :: Chanin a -> Chanout a -> Chanin (Chanout a) -> CHP ()
tap input output register = tap' []
  where
    tap' listeners = do
      listeners' <- (do x <- readChannel input
                        parallel_ [writeChannel c x | c <- output : listeners]
                        return listeners)
                    <|> (do listener <- readChannel register
                            return (listener : listeners))
      tap' listeners'
```

Dynamic processes that receive channels must take account of this in their handling of poison to make sure they poison all channels on which they are currently (potentially) communicating. In the case of the above process, we must poison all the listeners as well as the primary channels given as parameters:

```
tap :: Chanin a -> Chanout a -> Chanin (Chanout a) -> CHP ()
tap input output register = tap' []
  where
```

```
tap' listeners = do
  listeners' <- ((do x <- readChannel input
                     parallel_ [writeChannel c x | c <- output : listeners]
                     return listeners)
                 <|> (do listener <- readChannel register
                         return (listener : listeners))
  ) `onPoisonRethrow` (do
      poison input >> poison register
      mapM_ poison (output : listeners))
  tap' listeners'
```

4.3 Wiring

In message-passing systems with typed channels, a substantial part of the programming model is the composition of processes using channels. For example, we may want to compose together the mapP and filterP processes described in section 4.2 into a process that filters out negative numbers and then turns the remaining positive numbers into strings:

```
showPosP :: Chanin Int -> Chanout String -> CHP ()
showPosP input output
  = do (w, r) <- newChannelWR
       filterP (> 0) input w <||> mapP show r output
       return ()
```

This is shown diagrammatically in figure 18. It is instructive to note that the composition of two such processes with a single input channel and single output channel is itself a process with a single input channel and a single output channel. This component can then be re-used without any knowledge required of its internally concurrent implementation.

4.3.1 Simple Operator

This composition of two single-input, single-output processes is so common that it is worth capturing in an associative operator:

```
(==>) :: (Chanin a -> Chanout b -> CHP ())
      -> (Chanin b -> Chanout c -> CHP ())
      -> (Chanin a -> Chanout c -> CHP ())
```

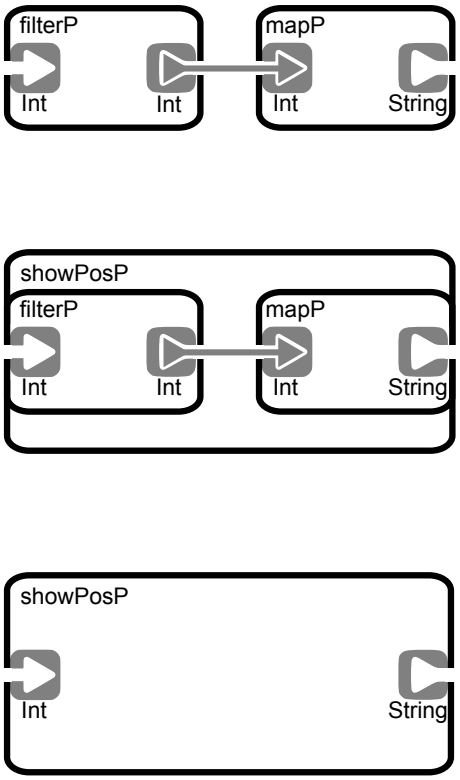


Figure 18: The composition of filterP and mapP, as shown in the top diagram. This composition becomes an opaque box to other components, as shown in the lower diagrams. This component can then be further composed in a similar manner. The programming model used in CHP is thus compositional, allowing complex networks to be built from joining together different components without regard to their internal implementation.

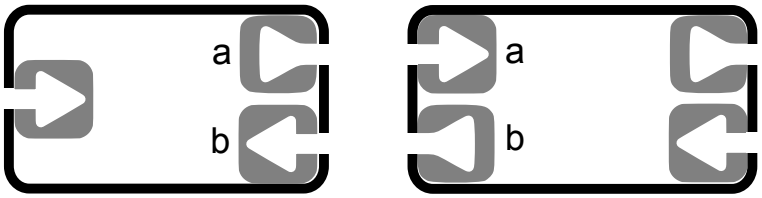


Figure 19: An example of slightly different process composition than figure 18. The letters indicate the types of the channel-ends that each process takes. It is readily apparent, both that these processes *can* be composed, and *how* they should be composed (with a pair of channels).

```
(==>) p q r w = do (mw, mr) <- newChannelWR
                   p r mw <| |> q mr w
                   return ()
```

The previous composition of mapP and filterP can be written using this operator as follows:

```
showPosP :: Chanin Int -> Chanout String -> CHP ()
showPosP = filterP (> 0) ==> mapP show
```

This point-free style is clearer and more elegant. By not introducing extra variable names we eliminate potential mistakes (mis-wiring). It can be seen that this process composition operator is an analogue of function composition.

We do not, however, always want to connect processes merely with a single unidirectional channel. We may want to connect processes with a pair of channels (one in each direction) or three channels, or a channel and a barrier, etc – which means that we need a more general operator than the one above.

4.3.2 Richer Operator

Figure 19 shows another example of process composition, requiring different connections than figure 18. The types and directions of the channels needed to compose the processes are readily apparent – it should be just as easy to join these processes with two channels as it was to join filterP and mapP with one.

To generalise the variety of composition possible, we use Haskell's type-class mechanism. We define a two parameter type-class, Connectable, an instance of which indicates that the two parameters can be wired together in some fashion, and provide a function that must be implemented to do so:

```
class Connectable l r where
  connect :: ((l, r) -> CHP a) -> CHP a
```

Instances for channels (in both directions) are trivial[2]:

```
instance Connectable (Chanout a) (Chanin a) where
  connect p = newChannelWR >>= p

instance Connectable (Chanin a) (Chanout a) where
  connect p = (swap <$> newChannelWR) >>= p
    where
      swap (x, y) = (y, x)
```

We choose this style of function to compose the processes, rather than say `connect :: CHP (l, r)`, because we may need to enroll the processes on the synchronisation object for the duration of their execution. Our chosen style of function allows us to do just that for an instance involving barriers:

```
instance Bounded phase =>
    Connectable (PhasedBarrier phase) (PhasedBarrier phase) where
  connect p
    = do b <- newPhasedBarrier minBound
         enroll b (\b0 -> enroll b (\b1 -> p (b0, b1)))
```

The instance that grants much greater power to the Connectable interface is the one that works for any pair of Connectable items:

```
instance (Connectable lA rA, Connectable lB rB) =>
         Connectable (lA, lB) (rA, rB) where
  connect p
    = connect (\(ax, ay) ->
        connect (\(bx, by) ->
          p ((ax, bx), (ay, by))))
```

This instance means that two processes p and q can easily be wired together if their types were:

```
p :: (Chanin Int, PhasedBarrier ()) -> CHP ()
q :: (Chanout Int, PhasedBarrier ()) -> CHP ()
```

Similar instances can also be constructed for triples and so on. Programmers may also create their own instances (as with any Haskell type-class) for synchronisation primitives not known to the library, or for compound data structures

[2]The <$> operator is a synonym for fmap and can be thought of for the purposes of this chapter as having the type (a -> b) -> CHP a ->CHP b – it applies a pure function on the left-hand side to the return value of a monadic action on the right-hand side.

that feature several synchronisation primitives that need to be wired together differently.

The Connectable interface is a suitable basic API, but it is too unwieldy to compose processes together. We can use it to define a more general version of the composition operator seen earlier:

```
(<=>) :: Connectable l r =>
         (a -> l -> CHP ()) ->
         (r -> b -> CHP ()) ->
          a -> b -> CHP ()
(<=>) p q x y = connect (\(l, r) -> (p x l <||> q r y) >> return ())
```

The type of this operator is very general. No restrictions are placed on the "outer" types a and b (which may be channels, but are not required to be so). This operator composes together any pair of two-argument processes where the second argument of the first process can be connected to the first argument of the second process. We can also trivially define other operators that are useful at the start and end of a process pipeline, respectively:

```
(|<=>) :: Connectable l r =>
          (l -> CHP ())
       -> (r -> b -> CHP ())
       ->  b -> CHP ()

(<=>|) :: Connectable l r =>
          (a -> l -> CHP ())
       -> (r -> CHP ())
       ->  a -> CHP ()
```

4.3.3 Capture Common Topologies

We do not always want to simply compose two adjacent processes. Another common requirement is to wire together a pipeline of processes. We can do this by building on top of our connectable operator, meaning that the helper function is parameterised by the type of connection between processes, but fixes the topology:

```
pipeline :: Connectable r l =>
            [l -> r -> CHP ()] -> l -> r -> CHP ()
pipeline = foldr1 (<=>)
```

We can easily extend this to a cycle:

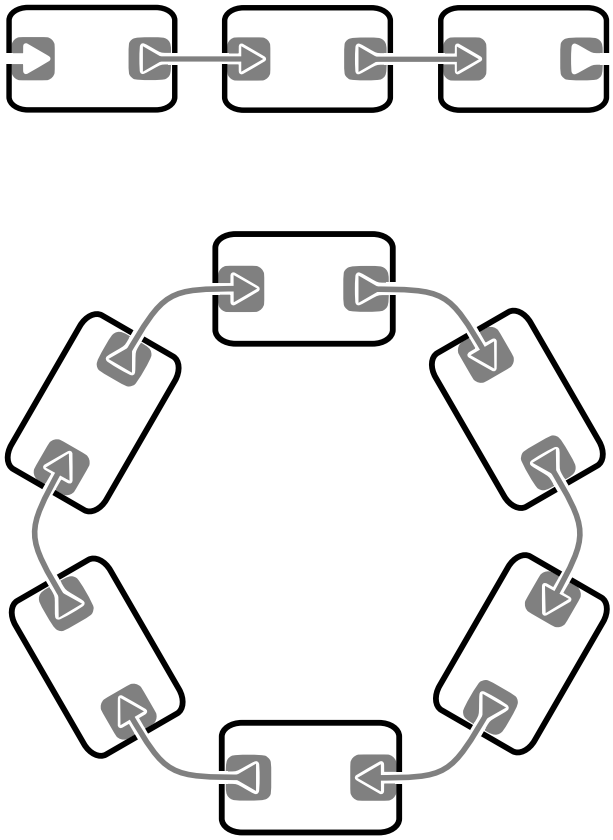


Figure 20: The pipeline topology (top) and cycle topology (bottom). It can be seen that a cycle can be formed simply by connecting the two end points of a pipeline together. The processes are illustrated here by connecting them with a single channel, but any regular interface could be connected together using the Connectable class described in this chapter.

```
cycle :: Connectable r l =>
         [l -> r -> CHP ()] -> CHP ()
cycle ps = connect (\(l, r) -> pipeline ps l r)
```

Both topologies are depicted in figure 20.

This idea of capturing topology extends beyond such one-dimensional structures. A common requirement when building concurrent simulations with the CHP library is to form a regular two-dimensional (or three-dimensional) grid, either with or without diagonal connections. Producing such wiring, especially with diagonal connections, is verbose and potentially error-prone. Without the Connectable interface, it would have to be replicated for each type of channel used, increasing the possibility for error. But we can now write the function once, test it to show its correctness once, and re-use it repeatedly in different programs. We

show an example type here but omit the lengthy definition[3]:

```
grid4way :: (Connectable right  left ,
             Connectable bottom top) =>
  [[above -> below -> left  -> right  -> CHP r]] ->
  CHP [[r]]
```

The parameter is a list of rows of processes (which must be rectangular); the result is a corresponding list of rows of results. The processes are wired together into a regular grid where the far right edge also connects to the far left edge, and the bottom edge to the top: this forms a torus shape.

Any topology (especially regular topologies) can be captured in helper functions like those given above, and re-used regardless of the channel types required to connect the processes.

4.3.4 API

We gather together the API for the connectable combinators here:

```
class Connectable l r where
  connect :: ((l, r) -> CHP a) -> CHP a

instance Connectable (Chanout a) (Chanin a)
instance Connectable (Chanin a) (Chanout a)
instance Bounded phase =>Connectable (PhasedBarrier phase) (PhasedBarrier phase)
instance (Connectable lA rA, Connectable lB rB) =>
         Connectable (lA, lB) (rA, rB)

(<=>) :: Connectable l r =>
         (a -> l -> CHP ())
      -> (r -> b -> CHP ())
      ->  a -> b -> CHP ()

(|<=>) :: Connectable l r =>
          (l -> CHP ())
       -> (r -> b -> CHP ())
       ->  b -> CHP ()
```

[3]It can be found in the chp-plus library at `http://hackage.haskell.org/package/chp-plus`; an alternate short implementation is given later in section 4.4.2 of this chapter.

```
(<=>|) :: Connectable l r =>
          (a -> l -> CHP ())
       -> (r -> CHP ())
       ->  a -> CHP ()

(|<=>|) :: Connectable l r =>
           (l -> CHP ())
        -> (r -> CHP ())
        -> CHP ()

pipeline :: Connectable r l =>
            [l -> r -> CHP ()] -> l -> r -> CHP ()

cycle :: Connectable r l =>
         [l -> r -> CHP ()] -> CHP ()
```

4.3.5 Dining Philosophers Example

As an example of using the connectable combinators, we return to the dining philosophers example from chapter 3. Here are the wiring parts of the original code:

```
data DownUp = Down | Up
  deriving (Eq, Enum, Bounded, Show)

type Bar = PhasedBarrier DownUp

philosopher :: Bar -> Bar -> CHP ()

diningFork :: Bar -> Bar -> CHP ()

couple :: (Bar -> Bar -> CHP ())
       -> (Bar -> Bar -> CHP ())
       -> (Bar -> Bar -> CHP ())
couple p q left right
  = do b <- newPhasedBarrier Down
       enroll b (\bP -> enroll b (\bQ ->
         p left bP <||> q bQ right))
       return ()
```

```
college :: Int -> CHP ()
college size
  = do b <- newPhasedBarrier Down
       enroll b (\bA -> enroll b (\bB ->
         foldr1 couple (concat $ replicate size [diningFork, philosopher])
           bA bB))

main :: IO ()
main = runCHP (college 5)
```

It can now be seen that the couple operator is a form of the connectable operators we have introduced in this chapter. The wiring code can be replaced with our combinators, and thus simplified to:

```
data DownUp = Down | Up
  deriving (Eq, Enum, Bounded, Show)

type Bar = PhasedBarrier DownUp

philosopher :: Bar -> Bar -> CHP ()

diningFork :: Bar -> Bar -> CHP ()

college :: Int -> CHP ()
college size = cycle (concat $ replicate size [diningFork, philosopher])

main :: IO ()
main = runCHP (college 5)
```

4.4 Compositional Wiring

The previous section outlined ways to compose processes into a complete whole. We often have situations where a process needs not just one set of connections, but also some other cross-cutting connection. For example, a cycle of processes may all be connected to their neighbours with a channel – but they may also all be enrolled together on a barrier (as illustrated in figure 21).

Consider how to implement such an arrangement with the combinators that

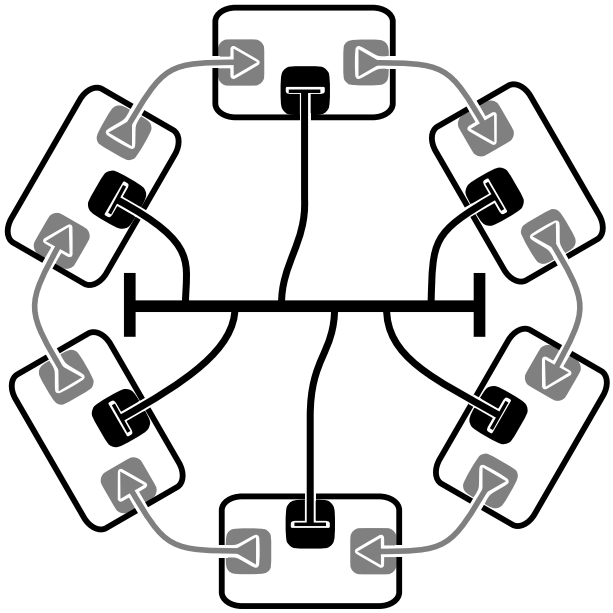


Figure 21: A ring of processes connected to their neighbours with a single channel, and also all enrolled together on the same central barrier.

we have introduced thus far; we have (with the types slightly specialised for illustration):

```
enrollAll :: Unenrolled PhasedBarrier phase
          -> [PhasedBarrier phase -> CHP a] -> CHP [a]

pipeline :: [Chanin a -> Chanout a -> CHP ()] ->
            Chanin a -> Chanout a -> CHP ()
```

Both processes expect a list of processes that take exactly the required arguments (a barrier or a channel pair, respectively) and return a CHP process. Neither supports partial application that would return a process ready to be wired up by the other function: in short, these combinators do not compose.

We cannot simply create a function without the CHP monad, such as:

```
pipeline' :: [Chanin a -> Chanout a -> b] ->
             Chanin a -> Chanout a -> [b]
```

We require access to the CHP monad in order to run the processes in parallel, and to create the channels used to connect them together. This means that we need a different strategy in order to support composing these combinators in a useful way. To that end, we introduce a Composed monad.

4.4.1 The Composed Monad

We need to abstract over the return types of the processes being composed together while still allowing access to the functionality in the CHP monad. We therefore create functions as follows (again with types specialised for illustration):

```
enrollAllR :: Unenrolled PhasedBarrier phase
           -> [PhasedBarrier phase -> a] -> Composed [a]

pipelineR :: [Chanin a -> Chanout a -> b] ->
             Chanin a -> Chanout a -> Composed [b]

cycleR :: [Chanin a -> Chanout a -> b] -> Composed [b]
```

Given a list, `processes :: [PhasedBarrier () -> Chanin a -> Chanout a ->CHP ()`, we can compose them as depicted in figure 21, simply using:

```
enrollAllR b processes >>= cycleR
```

The meaning of composition in this monad is not intuitively the sequencing of actions as is often the case for monads (in fact, the monad is conceptually commutative in many cases). It is instead a form of nesting – the code above enrolls the processes on the barrier, and inside the scope of that enrollment it wires them together in a cycle.

The type of the Composed monad is:

```
newtype Composed a = Composed
  { runWith :: forall b. (a -> CHP b) -> CHP b }
```

This type is not without precedence as a monad; it is equivalent to `forall b. ContT b CHP a`, the continuation-passing monad transformer on top of CHP, and is technically the codensity monad of CHP. The monad is not used to pass continuations, however. The intuition is that any type wrapped in Composed needs to be told how it can be turned into a CHP action, and then it becomes that CHP action. At the outer-level this is accomplished with `parallel` :

```
run :: Composed [CHP a] -> CHP [a]
run ps = ps `runWith` parallel
```

The output of any Composed block is almost always such a list of complete CHP processes ready to be run in parallel.

The monad instance for Composed is as follows:

```
instance Monad Composed where
```

```
return x = Composed (\r -> r x)
(>>=) m f = Composed (\r -> m `runWith` ((`runWith` r) . f))
```

4.4.2 Composed Wiring Functions

We can re-define all the wiring functions seen earlier in the new Composed monad. The most basic are the connectR and enrollR functions:

```
connectR :: Connectable l r => ((l, r) -> a) -> Composed a
connectR p = Composed (\r -> connect (r . p))

enrollR :: Enrollable b p => Unenrolled b p
        -> (b p -> a) -> Composed a
enrollR b p = Composed (\r -> enroll b (r . p))
```

The latter can easily be expanded into an enrollAllR function:

```
enrollAllR :: Enrollable b p => Unenrolled b p
           -> [b p -> a] -> Composed [a]
enrollAllR b ps = mapM (enrollR b) ps
```

The enrollAllR function enrolls a list of processes on the given barrier. Without the Composed monad it is an intricate recursive function, but with the Composed monad it is a non-recursive and straightforward mapM call.

We can define the pipelineR function as follows:

```
pipelineR :: Connectable l r => [r -> l -> a]
                  -> Composed (r -> l -> [a])
pipelineR [] = return (\_ _ -> [])
pipelineR (firstP :restP) = foldM adj (\x y -> [firstP x y]) restP
  where
    adj p q = connectR (\(l, r) x y -> (p x l) ++ [q r y])
```

As before, the cycleR function is a small addition to the pipelineR function:

```
cycleR :: Connectable l r =>
          [r -> l -> a] -> Composed [a]
cycleR [] = return []
cycleR ps = pipelineR ps >>= connectR . uncurry . flip
```

With these composition operators we can now easily define the 4-way grid composition discussed earlier in section 4.3.3:

```
grid4wayR :: (Connectable below above,
              Connectable right  left ) =>
  [[above -> below -> left  -> right  -> a]] ->
  Composed [[a]]
grid4wayR =     (mapM cycleR . transpose)
            <=< (mapM cycleR . transpose)
```

The inherent symmetry, and regularity, of the combinator is exposed, and its definition trivial based on the cycleR function – with the help of the standard list function transpose that swaps rows for columns in a list of lists. (The <=< function composes two monadic functions; its type is Monad m =>(b ->m c) -> (a -> m b) -> a -> m c.)

It is possible for users to define their own wiring functions using this monad. For example, a user may find that frequently in their program they have a list of processes where they wish to enroll all the processes at odd positions in the list on one barrier, and all the processes at even positions on another barrier. They could write a function to do this, and use it different situations in combination with other functions – for example, one such list may further be wired into a pipeline, while another may be wired into a star topology.

The use of wiring combinators avoids explicitly declaring and naming the channels, barriers, etc required to construct the process network. This makes the code shorter, and prevents errors (such as passing the wrong channel-end to the wrong process, which can occur if they have the same type). It also means that common topologies (such as pipelineR) can be recognised by name when reading code – it is not straightforward to recognise wiring patterns when they are written out "long-hand" with all the individual components named and passed around to the various processes.

4.5 Behaviours

A common pattern in simulations built with CHP is the tick pattern. Each process enrolls on a barrier named tick that has a low priority. The tick barrier divides time into timesteps (tick occurs on the boundary between two timesteps). Each process offers a choice between performing various actions with other processes, and performing the tick event. Since the tick event has the lowest priority and is enrolled on by all the processes, it will only performed when no other choices by any process can be performed.

Consider a process representing a physical site in a simulation. It may or may not be currently holding an agent. It will allow another agent to enter the space (a maximum of one per time-step) and it will independently allow its current agent to move on. So it has three actions; moveIn (which may happen once or not at all), moveOut (which may happen once or not at all) and tick (which will happen once, and will end this current behaviour). Afterwards we want a list of the agents now in the site.

We must implement this functionality as follows:

```
start cur = (moveIn >>= movedIn cur)
          <|> (moveOut >> movedOut)
          <|> (tick >> return [cur])

movedIn old new = (moveOut >> tick >> return [new])
                <|> (tick >> return [old, new])

movedOut = (moveIn >>= \new -> tick >> return [new])
         <|> (tick >> return [])
```

There are 5 different paths through this functionality; each move may or may not occur (and if they both occur, they may occur in either order), followed by tick. If we added a third movement option, there would be 16 different paths and the number would grow exponentially.

The above mechanism would be a poor abstraction with which to program such simulations. We instead want an API that allows us to clearly and concisely express our intention: we wish to perform two actions once or not at all, ended at any time by a third action, and we want to know afterwards which actions occurred.

4.5.1 Behaviour API

Our solution to this is called behaviours. A behaviour is an item that has type `CHPBehaviour a`[4], and represents some action (potentially repeated) that will result in a value of type `a`. Such a behaviour can be executed using the `offer` function:

```
offer :: CHPBehaviour a -> CHP a
```

[4] `CHPBehaviour` has a `Functor` instance for modifying the return value of the item, but is neither a monad nor an applicative functor.

We allow combination of behaviours (iterated choice) using the alongside combinator:

```
alongside :: CHPBehaviour a ->
             CHPBehaviour b ->
             CHPBehaviour (a, b)
```

Its type aside, alongside is semantically associative and commutative. We define two fundamental behaviour types: those which end the current offer (using endWhen), and those which do not (all other functions):

```
endWhen :: CHP a -> CHPBehaviour (Maybe a)

once :: CHP a -> CHPBehaviour (Maybe a)

upTo :: Int -> CHP a -> CHPBehaviour [a]

repeatedly :: CHP a -> CHPBehaviour [a]

repeatedlyRecurse :: (a -> CHP (b, a)) -> a -> CHPBehaviour [b]
```

The Maybe return of endWhen indicates that it may or may not have occurred. This may seem odd, but it is possible to have two or more endWhen functions combined with alongside, and then precisely one will have occurred (with a Just return) and the others will not have (a Nothing return).

The latter four functions can all be thought of as special cases of one another; once is equivalent to listToMaybe . upTo 1, while repeatedly is upTo without an upper bound, and the function repeatedlyRecurse offers persistence of state through repeated executions of the action.

Some laws for trivial behaviours naturally follow from the definitions of these behaviours:

```
offer (repeatedly p) = forever p

offer (once p) = p >> stop -- i.e. it does not finish

offer (endWhen q) = Just <$> q

offer (endWhen p `alongside` endWhen q)
  = ((\x -> (Just x, Nothing)) <$> p)
    <|> ((\x -> (Nothing, Just x)) <$> q)
```

```
offer (once p `alongside` endWhen q)
  = (p >>= (\x -> q >>= \y -> return (Just x, Just y)))
    <|> (q >>= \y -> return (Nothing, Just y))
```

Most further examples have no such simple rearrangement, which is of course the incentive to express them as behaviours.

Our earlier example can now be expressed with behaviours:

```
do ((old, new),_) <-
       offer ((once moveIn
                `alongside` once moveOut)
                `alongside` endWhen tick)
```

4.5.2 Implementation

The CHPBehaviour type is made up of a value that would be returned if the offer were terminated now, and a possible choice to offer as part of the behaviour; if the latter part is Nothing, this indicates that the offer should terminate now:

```
data CHPBehaviour a
  = CHPBehaviour a (Maybe (CHP (CHPBehaviour a)))

instance Functor CHPBehaviour where
  fmap f (CHPBehaviour x m)
    = CHPBehaviour (f x) (fmap (fmap (fmap f))) m)
```

The alongside combinator is implemented using CHP's choice operator, and whichever option occurs, it is joined together with the other half of the alongside call again for the next choice:

```
alongside oa@(CHPBehaviour a (Just fa))
          ob@(CHPBehaviour b (Just fb))
  = CHPBehaviour (a, b)
      (Just ((flip alongside ob <$> fa)
               <|> (alongside oa <$> fb)))

alongside (CHPBehaviour a _) (CHPBehaviour b _)
  = CHPBehaviour (a, b) Nothing
```

In the latter case, at least one of the two behaviours has a Nothing value that indicates termination, and this is perpetuated up the tree of the alongside calls.

The repeatedly combinator accumulates a list (adding at the head for efficiency):

```
repeatedly m = reverse <$> repeatedly' []
  where
    repeatedly' xs
      = CHPBehaviour xs (Just ((repeatedly' . (: xs)) <$> m))
```

The once combinator will execute its action once, and afterwards will only offer stop, the choice that can never be taken, thus preventing it occurring a second time, without terminating the call to offer:

```
once = CHPBehaviour Nothing . Just .
         fmap (flip CHPBehaviour (Just stop) . Just)
```

The endWhen combinator will become Nothing after its execution, thus terminating the call to offer:

```
endWhen = CHPBehaviour Nothing . Just .
            fmap (flip CHPBehaviour Nothing . Just)
```

Finally, the offer call itself simply repeats the behaviour until a Nothing value is found for the action:

```
offer (CHPBehaviour _ (Just m)) = m >>= offer
offer (CHPBehaviour x Nothing) = return x
```

4.5.3 Generalisation Beyond CHP

We have shown the idea of behaviours here, specialised to CHP. In fact, the concept of behaviours is not specific to CHP, and should work for any type that it is an instance of Monad and Alternative. A suitably generalised version of behaviours has been written up in The Monad Reader electronic magazine [Brown, 2011b].

4.5.4 Relation to Grammars

The way that we originally explored programming the behaviour of our system at the start of section 4.5 is a context-free grammar (CFG), and the program code is similar to how such a system might be encoded in a form such as Backus-Naur Form (BNF). This grammar is a grammar over the sequence of actions that the particular process might take – but it is more like a generative grammar than a parser.

CFGs are generally used for certain sequences of actions, e.g. one a followed by many bs followed by a c. What we require here, and what CFGs and notations such as BNF are ill-suited to capture, is patterns such as: many bs with at most one a amongst them. This must be captured using a BNF rule with two states, one where a has not happened yet and one where a has happened. Trying to make this more complex, e.g. many bs with at most two a and three c anywhere among them, does not scale when using this finite-state automata approach.

Our Haskell solution to the problem does not correspond to a CFG – its use of higher-level combinators takes it further than a CFG. There do exist grammar systems that can capture the same as our Haskell solution – particularly two-level grammars [Cleaveland and Uzgalis, 1977]. Intuitively, a two-level grammar is a grammar that generates a grammar; a higher-order grammar.

4.5.5 Two-Level Grammar for Behaviours

The appeal of a two-level grammar for representing behaviours is that the majority of the two-level grammar need only be defined once, for all possible uses of the behaviour system. We begin with the only part of the grammar that need be altered to adapt the two-level grammar system to the particular behaviour at hand: the three classes of events:

- `END`, the events that end the behaviour,
- `OPTIONAL`, the events that can happen at most once, and
- `REPEATABLE`, those events which may occur any number of times.

We leave including the events that can occur a finite number of times as an exercise for the reader (such systems are explored by Cleaveland and Uzgalis [1977]), as the use of `upTo` is very rare. For our earlier example, the three event meta-productions are:

```
END :: tick.
REPEATABLE :: .
OPTIONAL :: moveIn; moveOut.
```

In this notation, colon introduces a definition, comma is sequence and semi-colon is choice or branching. These meta-productions are further referenced by other constant meta-productions:

```
EVENT :: END; OPTIONAL; REPEATABLE.
EMPTY :: .
SEQ :: EVENT ; SEQ EVENT.
```

EMPTY is the empty sequence. EVENT is any event of the three sets already introduced. SEQ is any sequence of one or more EVENT.

We name the top-level hyper-rule in our system that represents the complete grammar, "g":

```
g: g', END.
g': EMPTY; SEQ check.
```

This grammar, g, is the grammar g' followed by the END terminal. The grammar g' can either be empty or the meta-production SEQ followed by check – the latter item is a post-fix tag used to match with the latter three hyper-rules (in effect, it invokes these hyper-rules by needing to be parsed, akin to a function call). The check item can be parsed in three different ways (think of them as different pattern-matching cases in a Haskell function):

```
OPTIONAL SEQ check: OPTIONAL symbol, SEQ check,
                    OPTIONAL notin SEQ.
REPEATABLE SEQ check: REPEATABLE symbol, SEQ check.
EVENT check: EVENT symbol.
```

The first check hyper-rule matches an OPTIONAL event at the front of a SEQquence of events – this is like a head-tail pattern match. This hyper-rule translates into a hyper-rule that matches an OPTIONAL meta-production followed by a further SEQuence (also with a post-fix check tag to recursively invoke these rules on the remainder of the sequence), where the optional item is not in that sequence. That is, this rule matches an optional item before a sequence and ensures that the optional item does not occur again in the following sequence. The notin notation is itself a rule that can be constructed that only parses if the left-hand side does not appear in the right-hand side – see Cleaveland and Uzgalis [1977] for the length definition.

The second check hyper-rule matches a REPEATABLE meta-production at the front of a SEQuence. It is simply a REPEATABLE symbol followed by a further SEQuence with a check tag.

The third check hyper-rule is for a sequence with a single event (the base case, in effect) that matches just that event.

4.6 Conclusions

The Communicating Haskell Processes library is an imperative message-passing library built in a functional programming language with a clean, simple API. This chapter has shown how the ideas of higher-order functions, type-class-based abstractions and re-usable combinators can be taken from functional programming and applied to message-passing programming, with all of the same benefits.

The mapP and filterP processes presented in section 4.2 transfer long-standing functional programming ideas directly into message-passing programming. The versions of these processes presented in section 4.2.3 demonstrate how such processes can be dynamic, allowing the parameterisation of their behaviour to be altered while they are running in a safe and controlled manner.

CHP programs are made up of many components composed together concurrently, and connected by channels, barriers or other synchronisation primitives. The "long-hand" way of composing these processes – manually declaring channels and passing the ends to the right processes – is tedious, verbose and error-prone. The combinators discussed in this chapter allow for an elegant and concise point-free style, composing processes together without ever naming the channels that connect the processes.

The Connectable type-class allows the wiring functions to abstract away from the mechanisms used to compose adjacent processes and to instead focus on capturing topology. This allows complicated functions (such as two-dimensional grids with diagonal connections) to be written once and re-used. The Composed monad takes this further and allows complicated composition with several cross-cutting concerns to be done easily and compositionally, which makes for completely flexible wiring of processes.

The work on behaviours shows that the most straightforward way to program some processes using just the imperative monad can end up intricate and unclear. The combinator-based higher-order approach can again be used to provide a simpler API to capture these repeated behaviours more clearly.

All of this work is only possible because functions and processes are first-class in CHP, and can thus be passed as arguments. Implementing these combinators in message-passing frameworks in other languages would either be overly verbose and awkward (e.g. using interfaces and classes in Java) or simply not possible (e.g. in the message-passing language occam, where higher-order programming is not possible). Furthermore, the use of type-classes supports easy parameterisation of

the combinators according to the types involved, supporting abstractions where functionality can be determined (e.g. in the wiring together of channel ends).

Chapter 5

Choice and Conjunction

This chapter begins by explaining a Software Transactional Memory (STM) algorithm for implementing choice (at both ends of a channel, and by all parties in a barrier synchronisation). The chapter then focuses on adding a new feature not seen in any other framework, conjunction, and extends the algorithms to support conjunction. Much of the work in this chapter was previously published in Advances in Message Passing 2010 [Brown, 2010].

5.1 Choice Algorithm

In this section we explain how to implement an algorithm to resolve choice, without considering conjunction. The setting is as follows: there are many processes, each of which is willing (or offering) to synchronise on one of a set of events. Each event has an enrollment count. If an event has as many processes offering to synchronise as its enrollment count, the synchronisation takes place and all the synchronising processes' other offers are discarded. Until this is the case, the processes must all wait.

All of the algorithms in this chapter for implementing choice between events do not involve any polling or similar. A process wanting to synchronise on some events performs a single check of all the events to see if any can complete. If one can complete, then it is completed, all other processes that engaged in the event are woken up and the process can then proceed. If none can complete, the process records its offers to synchronise, and sleeps until it is woken by a later process that completes this event. Thus there is no busy-waiting or polling or active search; each process performs one check and is then either finished, or it sleeps until another process wakes it up (at which point it is finished).

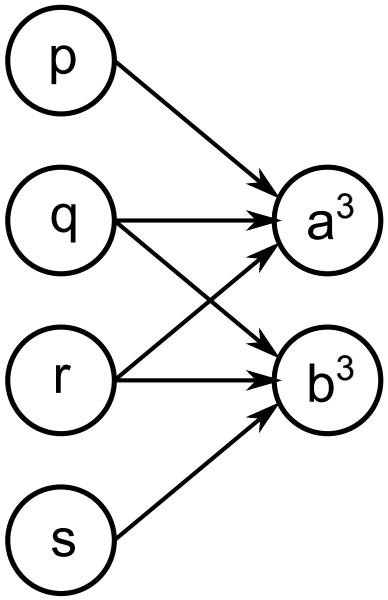


Figure 22: A diagram of four processes (p, q, r, s) on the left making various offers (indicated by arrows) to engage in the events a and b on the right, which both have an enrollment count of 3 (indicated by superscript).

Note that in these algorithms, it is the responsibility of the process completing the event to withdraw the offers of all the other processes involved in the completed event. Consider the scenario in figure 22, and consider if p and s arrive at roughly the same time, and p will complete event a. It is p's responsibility to remove the offers by q and r on event b. If it was left to q and r once they wake up and are rescheduled, there is a possible problem with s seeing their old (now invalid) offers on b.

This highlights the main challenge in implementing choice by all parties in a synchronisation: that of atomicity. The offer by p will allow a to complete, and the offer by s will allow b to complete – but only one of these can happen, as a can only be completed if q and r commit to a (foregoing b), and b can only be completed if q and r commit to b (foregoing a). For correct behaviour, q and r in this scenario must both make the same choice. The problem becomes one of atomicity; there must be a way to ensure that between the two concurrent processes, p and s, a single decision is made as to whether a completes or b does. If there is only a per-event lock or a per-process lock, it may be possible for p to believe a should complete and s to believe b should complete, resulting either in an unnecessary deadlock or in q and r believing that different events were chosen.

Welch et al. [2006a] have previously described how to implement choice by all parties in a synchronisation using a single central lock-protected data structure.

Here, we give a de-centralised implementation based on STM that does not require any such central structure, but does retain correct atomicity.

For the purposes of this algorithm, an "offer" made by a process is a single event on which it wishes to synchronise; a process makes a set of offers from which it wants to choose and engage in exactly one. The implementation of an event has an enrollment count and a list of current offer-sets in which it features. The basic pattern of the choice algorithm is as follows: a new process wishing to make a set of offers checks to see if (with itself included) any of the events can now complete. If an event can complete, it is completed by revoking all of its associated offer-sets from all events, and waking up all the processes involved. If an event cannot be completed, the process adds its offer to all the involved events and waits to be woken up by a later process completing one of its events.

The code is shown in figure 23. Making and revoking offers (lines 11–19) is a straightforward manipulation of the list of offers held by an event. The primary function is `offerAll`, which takes a list of triples (an STM action to perform when the event is chosen, the event itself, and an associated return value), waits until one of the given events can complete, and then returns the associated value.

The `offerAll` function first checks if any of the events can complete immediately, using the `checkComplete` function. This function reads all the events in turn. If it finds any event with an enroll count equal to the current offers plus one (since we are also making an offer on that event, but have not yet recorded this), it performs all the actions associated with the events, removes all the other offers from the now-committed other processes and then returns. If no events are completed, the process executing `offerAll` records its offers in all the events and returns a transactional variable which is currently `Nothing` but that will become a `Just` value once the process has been committed to an event later on.

The process then performs a second transaction to wait for the transacational variable (named `rtv`) to become a `Just` value. It is crucial that this is a separate transaction; the first transaction must complete and have its effects (i.e. the process's offers) visible to other processes, and then it must wait for another process's transaction to alter the transaction variable.

This algorithm replaces the single central lock structure of Welch et al. [2006a] by using STM's guarantee of atomic transactions across multiple transaction variables, rather than pooling all the variables into a single lock-protected structure. This allows multiple events with totally disjoint participants to proceed entirely in parallel.

```
1  data Offer = Offer { offerThreadId :: ThreadId, offerEvents :: [EventImpl] }
2
3  instance Eq Offer where (==) = (==) `on` offerThreadId
4
5  data EventImpl = EventImpl { enrollCount :: Int,
6                               offersTV :: TVar [(STM (), Offer)] }
7
8  newEventImpl :: Int -> IO EventImpl
9  newEventImpl n = EventImpl n <$> atomically (newTVar [])
10
11 makeOffer :: Offer -> (STM (), EventImpl) -> STM ()
12 makeOffer o (act, e) = do offers <- readTVar (offersTV e)
13                           writeTVar (offersTV e) ((act, o):offers)
14
15 revokeOffer :: Offer -> STM ()
16 revokeOffer off = mapM_ remove (offerEvents off)
17   where
18     remove e = do offers <- readTVar (offersTV e)
19                   writeTVar (offersTV e) (filter ((/= off) . snd) offers)
20
21 offerAll :: [(STM (), EventImpl, a)] -> IO a
22 offerAll off
23   = do tid <- myThreadId
24        rtv <- atomically $ checkAll tid
25        atomically $ readTVar rtv >>= maybe retry return
26   where
27     checkAll tid = do
28       rtv <- newTVar Nothing
29       let offer = [(act >> writeTVar rtv (Just x), e) | (act, e, x) <- off]
30       complete <- checkComplete offer
31       unless complete $
32         mapM_ (makeOffer (Offer tid [e | (_, e, _) <- off])) offer
33       return rtv
34
35 checkComplete :: [(STM (), IntEvent)] -> STM Bool
36 checkComplete [] = return False
37 checkComplete ((act, e) : rest)
38   = do offers <- readTVar (offersTV e)
39        if enrollCount e /= length offers + 1
40          then checkComplete rest
41          else do sequence_ (act : map fst offers)
42                  mapM_ (revokeOffer . snd) offers
43                  return True
```

Figure 23: The STM-based algorithm for choice between events (without conjunction).

5.2 Conjunction

This chapter introduces an "AND" operator for message-passing systems, that allows a process to engage in one event AND another (we call this conjoining the events). This is different from performing the two events in sequence or parallel: both events must happen, or neither. They are dynamically (and temporarily) fused into behaving like a single event. This enables the use of new design patterns.

In section 5.3 we explore uses of this new operator, and in section 5.4 we explore some of its properties. In section 5.5 we explain an algorithm for its implementation, built on top of Software Transactional Memory (STM), and in section 5.6 we benchmark this implementation against comparable message-passing systems to examine the cost of supporting conjunction.

5.3 Examples

To motivate the implementation of this new conjunction feature, we give several examples here of how it can be used.

5.3.1 Dining Philosophers

The dining philosophers is a classic problem in concurrency literature, invented by Dijkstra [Dijkstra, 1971] and popularised by Hoare in his book on Communicating Sequential Processes (CSP) [Hoare, 1985]. Philosophers spend time thinking or eating. When they sit down to eat, they must pick up two forks; one to the left, and one to the right. With the same number of philosophers as forks, this can lead to deadlock if all of the philosophers pick up the fork on their left, and all wait for the fork on their right to be free (or vice versa).

There is an obvious dining philosophers solution with conjoined events. Each philosopher waits to pick up their left and right forks as a conjunction of two events: one event being the picking up of its left fork and the other being the picking up of its right fork. Conjunction means that neither event will occur without the other, and thus the classic deadlock with all philosophers holding one fork is eradicated.

We represent conjunction in CSP using the operator "$\wedge$": "$a \wedge b$" is an event representing the conjunction of events a and b. Thus we can describe our dining philosophers in CSP with conjunction (CSPc) as follows:

$$
\begin{aligned}
PHIL(i) =& thinking.i \rightarrow (left.i \wedge right.i) \\
& \rightarrow eating.i \rightarrow (left.i \wedge right.i) \rightarrow PHIL(i) \\
FORK(i) =& left.i \rightarrow left.i \rightarrow FORK(i) \\
& \Box right.(i+1)\%5 \rightarrow right.(i+1)\%5 \rightarrow FORK(i) \\
PHILS =& PHIL(0) \,|||\, PHIL(1) \,|||\, PHIL(2) \,|||\, PHIL(3) \,|||\, PHIL(4) \\
FORKS =& FORK(0) \,|||\, FORK(1) \,|||\, FORK(2) \,|||\, FORK(3) \,|||\, FORK(4) \\
COLLEGE =& PHILS \,[\![\, \{left, right\} \,]\!]\, FORKS
\end{aligned}
$$

We use % to signify the modulo operation.

New Solution in Standard CSP

We can use this conjoined solution to create the new solution to the dining philosophers in standard CSP illustrated in figure 24. Philosophers engage in the same external thinking and eating events as before (not shown in the figure). An eating event is controlled by an internal event, $forks.i$, that represents the picking up and putting down of two forks (shown as a filled circle in the figure). There is one such control event for each philosopher. Forks on either side of a philosopher must also engage in this controlling event.

A philosopher engages with its control event before and after eating. A fork initially offers to engage with the control event of the philosophers on either side – it is available to be *picked up*. Having been picked up by a philosopher, it offers only the control event of that philosopher – it waits to be *put down*. Since three parties (one philosopher and two forks) are required for *pickup* to take place, there is no time when a philosopher has only one fork. Hence, the classical deadlock (where each philospher gets one fork) cannot happen.

This can be written in standard CSP (without conjunction) as follows:

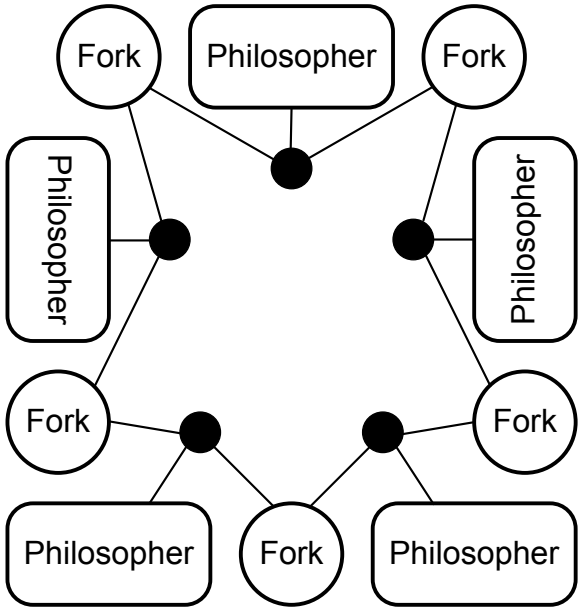


Figure 24: New solution to the dining philosophers. Each filled circle represents an event, $forks.i$, that represents alternatively the picking up and putting down of two forks by a philosopher. The diagram indicates that three processes (a philospher and two forks) must synchronise for any $forks.i$ event to happen.

$$
\begin{aligned}
PHIL(i) =& thinking.i \rightarrow forks.i \rightarrow \\
& eating.i \rightarrow forks.i \rightarrow PHIL(i) \\
FORK(i) =& forks.i \rightarrow forks.i \rightarrow FORK(i) \\
& \Box forks.(i+1)\%5 \rightarrow forks.(i+1)\%5 \rightarrow FORK(i) \\
PHILS =& PHIL(0) \;|||\; PHIL(1) \;|||\; PHIL(2) \;|||\; PHIL(3) \;|||\; PHIL(4) \\
FORKS =& (((FORK(0) \;[\![\, forks.1 \,]\!]\; FORK(1)) \;[\![\, forks.2 \,]\!]\; FORK(2)) \\
& [\![\, forks.3 \,]\!]\; FORK(3)) \;[\![\, forks.0, forks.4 \,]\!]\; FORK(4) \\
COLLEGE =& PHILS \;[\![\, \{forks\} \,]\!]\; FORKS
\end{aligned}
$$

We believe that this solution to the problem is new.

5.3.2 Moving Agents

Consider a topology of sites – say, a one-dimensional line of sites – with a pair of communication channels (one in each direction) connecting adjacent sites. Each site may either be empty or contain a single agent. Each agent may move to an adjacent site within a time-step.

One way to implement this is to make the site behave as follows in each time-step:

- if it is empty, it offers an exclusive choice between receiving an agent from each adjacent site, whereas
- if it is full, it offers an exclusive choice between sending its agent on an outward channel (to any of the sites to which the agent is willing to move).

This simple implementation has two problems in the case where a site is full. Firstly, it prohibits an agent moving into a site on the same time-step as an agent moves out (the "slow" problem); this may not be a property we want. Secondly, if two sites adjacent to each other both try to send an agent to each other, neither will progress (the "blocking" problem). In a one-dimensional line of sites, the blocking problem is particularly problematic and can easily lead to deadlock.

An erroneous correction to this implementation would be to offer (when the site is full) to send the agent out in parallel with offering to receieve a new agent. This could result in receiving a new agent without sending on the old agent; this would leave two agents in a site, breaking our design.

A correct solution involves the use of conjunction. When a site is full and has an agent wanting to move left, it offers to either: (send the agent left) or (send the agent left *and* recieve a new agent from the left). This way, if the site to the left is empty, the agent will be sent. If the site to the left is full and its agent wants to move right into our site, the agents are swapped. This solves our blocking problem. If we also want to solve the slow problem, we offer to either: (send the agent left) or (send the agent left *and* recieve a new agent from the left) or (send the agent left *and* recieve a new agent from the right). This solution generalises to any number of adjacency connections (and thus to any topology, such as two- and three-dimensional regular grids). It allows the problem to be solved using a different local design rather than a system-wide solution involving extra events.

5.3.3 Platelet Pipeline

We consider a problem similar to that posed by Schneider et al. [2006]. Platelets move (in a consistent direction) along a one-dimensional pipeline. On each time-step a platelet may move or not move, with the following rule: if there are platelets immediately before or immediately after it in the pipeline, it will only move forwards if they do so too. We also add the feature that each platelet will refuse to move on a time-step with probability p (typically 5%).

The CSP algorithm [Schneider et al., 2006] requires knowledge of the platelets from more than the adjacent sites, and multiway synchronisations; a later solution in the occam-π language used a higher-level clot process to group together the movement decisions of adjacent platelets [Welch et al., 2006b]. We can use conjunction to implement a solution wherein each platelet only requires (two-party) synchronisations with its immediate neighbours, and an optional global synchronisation.

Ticking

One methodology for designing concurrent simulations is the tick pattern; in each time-step, each process offers the choice between some actions, and a lower-priority tick event. If the actions are chosen, a further choice is offered between any remaining actions and the tick event (and so on). All the actions that can occur will do, followed by the tick event that signifies the end of this time-step (and thus the beginning of the next).

In our platelet simulation, each site may communicate two different things to each neighbour: a platelet (signalling a move) or an empty signal. An empty site begins by offering:

- to read a signal from the site behind it in the pipeline, *or*
- to send an empty signal to the site ahead of it, *or*
- to synchronise on the tick event.

If either of the former two happens, the remaining two of three are offered again, and so on, so that the process may do the first none or once, the second none or once, until the third happens – at which point it becomes full if it read a platelet from the site behind it (i.e. if it did perform the first event), and remains empty otherwise.

This can be written in CSP as follows:

$$
\begin{aligned}
&EMPTY(prev, next)\\
&\quad = prev.move \rightarrow ((next.empty \rightarrow tick \rightarrow FULL(prev, next))\\
&\qquad\qquad \Box\ (tick \rightarrow FULL(prev, next)))\\
&\quad \Box\ next.empty \rightarrow ((prev.move \rightarrow tick \rightarrow FULL(prev, next))\\
&\qquad\qquad \Box\ (tick \rightarrow EMPTY(prev, next)))\\
&\qquad \Box\ tick \rightarrow EMPTY(prev, next)
\end{aligned}
$$

A full site generates a random probability for the time-step; with a 5% chance it simply waits for the tick event. In the other 95% of cases it offers a choice:

- to read a signal from the site behind it in the pipeline *and* send the current platelet forward in the pipeline, *or*
- to synchronise on the tick event.

If the former happens, the process subsequently commits to the latter. Once the tick event has happened, the process is empty if it read an empty signal from the site behind it, and remains full otherwise.

This can be written in CSP with conjunction (CSPc) as follows[1]:

$$
\begin{aligned}
FULL(prev, next) &= FULL_M(prev, next) \sqcap FULL_S(prev, next)\\
FULL_S(prev, next) &= tick \rightarrow FULL(prev, next)
\end{aligned}
$$

$$
\begin{aligned}
&FULL_M(prev, next)\\
&\quad = prev.move \wedge next.move \rightarrow tick \rightarrow FULL(prev, next)\\
&\quad \Box\ prev.empty \wedge next.move \rightarrow tick \rightarrow EMPTY(prev, next)\\
&\quad \Box\ tick \rightarrow FULL(prev, next)
\end{aligned}
$$

[1] We use $\wedge$ for conjunction.

The pipeline can be terminated with a site that repeatedly reads in any signal, and started with a site that offers signals according to some pattern of platelet generation.

From these simple rules for behaviour we get emergent clotting behaviour; no site that is full can move unless its neighbours are empty (and thus willing to engage in events with it) or are full and willing to move – but not if they are unwilling to move. This becomes transitive; no clot (contigious group of full sites) can move unless all are willing to move. This would not be possible so simply (and with entirely local rules such as these) without conjunction.

Implementing this system with a concurrent process per site may be inefficient due to the large number of synchronisations that would be required. It is possible to instead construct a hybrid approach that simulates a group of many contiguous sites sequentially, and uses the above design/protocol at the edges of each group to communicate between the concurrent processes that simulate entire groups of sites.

Tickless

It is also possible to construct a solution that uses neither a tick event nor priority. Without the tick event it is necessary to have every site communicate with both of its neighbours on every time-step, to keep time-step synchronisation across the pipeline. This leads to a problem if we only use the empty and move events: there is no possible synchronisation for a stationary platelet to engage in with its neighbours.

Introducing a stay event does not solve the problem: a process willing to move must also offer the stay event to its neighbours (in case they wish to remain stationary), but this can result in a clot full of processes willing to move making the arbitrary choice to synchronise on the stay events rather than the move events. Our solution introduces two more events between processes beyond empty and move: a can-stay event and a must-stay event.

Our new $FULL_M$ process (that is willing to move) is as follows:

$$
\begin{aligned}
&FULL_M(prev, next)\\
&\quad = prev.empty \wedge next.move \rightarrow EMPTY(prev, next)\\
&\quad \Box prev.move \wedge next.move \rightarrow FULL(prev, next)\\
&\quad \Box prev.empty \wedge next.canstay \rightarrow FULL(prev, next)\\
&\quad \Box prev.canstay \wedge next.canstay \rightarrow FULL(prev, next)\\
&\quad \Box prev.muststay \wedge next.muststay \rightarrow FULL(prev, next)
\end{aligned}
$$

The first two cases are just as before, in the original version. The middle case is for when the process is at the beginning of the clot; it synchronises on empty with the process behind it, and can-stay with the process ahead of it in the clot. The last two cases can be thought of as perpetuating the can-stay and must-stay events throughout the clot.

The new $FULL_S$ process (that will remain stationary) is as follows:

$$
\begin{aligned}
&FULL_S(prev, next)\\
&\quad = prev.empty \wedge next.muststay \rightarrow FULL(prev, next)\\
&\quad \Box prev.muststay \wedge next.muststay \rightarrow FULL(prev, next)\\
&\quad \Box prev.canstay \wedge next.muststay \rightarrow FULL(prev, next)
\end{aligned}
$$

The first case is for if this process is at the beginning of the clot: it synchronises on empty with the process behind it, and must-stay with the process ahead of it. Looking up at the $FULL_M$ process, we can see that any $FULL_M$ process (from the last case) and any $FULL_S$ process (from the middle case immediately above) that synchronises on must-stay with the process behind it will also synchronise on must-stay with the process ahead of it. So if the process at the start of the clot synchronises on must-stay, all processes ahead of it in the clot will also synchronise on must-stay (by induction).

The third case of the $FULL_S$ process indicates that the processes behind the stationary one may offer can-stay, and it will then offer must-stay to all the processes ahead of it. The can-stay event will only be offered from the previous

process if it is in the $FULL_M$ state ($FULL_S$ only offers must-stay to the process ahead of it, and we will see shortly that $EMPTY$ only offers empty to the process ahead of it), which must then synchronise either on can-stay with the process behind that (which, again, must be a $FULL_M$ process) or empty (which means it's the start of the clot). All the full processes after $FULL_S$, following the logic in the previous paragraph, will synchronise on must-stay regardless of their state.

The new $EMPTY$ process is as follows:

$$
\begin{aligned}
&EMPTY(prev, next) \\
&\quad = prev.empty \wedge next.empty \rightarrow EMPTY(prev, next) \\
&\quad \Box prev.muststay \wedge next.empty \rightarrow EMPTY(prev, next) \\
&\quad \Box prev.move \wedge next.empty \rightarrow FULL(prev, next)
\end{aligned}
$$

All cases offer the empty event to the process ahead of it. It will accept from behind: the empty case (when the previous site is empty), the move case (when the previous site is full and able to move forward) and the must-stay event (when the previous site is part of a clot that cannot move). It does not accept the can-stay event, which is crucial, for reasons explained in the next section.

Tickless Proof

Since conjunction is an extra feature in CSP, there is no direct model-checking support for it. We will instead offer inductive proofs about clots. By proving a statement for the beginning site, and optional middle/end sites based on the neighbour behind them, this should inductively cover all non-empty clots. This can be done by considering the pairings of prev and next events, to see when offering a set of events from the previous site, what might be offered to its next neighbour.

Consider a clot of willing platelets. The site at the beginning of the clot can only synchronise on prev.empty (as that is all the empty site before it will offer). Therefore the site at the beginning of a clot will only offer move or can-stay to the next site. Any middle site that synchronises on move or can-stay with the previous site will offer the same thing to the next site. So inductively the last site of the clot will only offer move or can-stay. We can see that the empty site

following the clot will only accept move, not can-stay, so a clot of willing processes may only move and may not stay put. This solves the problem that we had with the ticking version, and is why the *EMPTY* process does not offer to synchronise on can-stay. (This result also shows that any line of sites at the beginning of a clot will only offer move or can-stay to the sites ahead of it.)

Now let us consider a clot with one or more stationary platelets somewhere along its length (but not the beginning). We have seen in the previous paragraph that the willing sites at the beginning of the pipeline will offer move or can-stay to the first stationary site in the clot. This stationary site appearing after these willing sites will only accept can-stay, and will then offer must-stay ahead of it. We can see that all full sites, stationary and willing, will only synchronise on prev.muststay with next.muststay, so regardless of the stationary/willing state of sites ahead of the first stationary site, must-stay will be the event offered at the end of the clot. The empty site will accept this, and so a clot with one or more stationary sites after the beginning will all agree to stay. If a stationary site is at the beginning, it will synchronise on prev.empty and next.muststay, and then the rest of the clot will also synchronise on must-stay, so again the clot will stay put. Thus any clot with one or more stationary sites will stay put.

5.3.4 Hidden Process

When two processes communicate with each other, they also synchronise. Placing a process in the middle of the two that forwards the messages introduces buffering, which may be unwanted. An added problem is that if the buffer process offers to read in a value, it is continually available – whereas the original receiver may not be, thus altering the synchronisation behaviour; if the writer only intends to write when the reader is available (and choose something else otherwise), its choice will be affected.

The solution to this problem of "hiding" a process is surprisingly complicated, and involves the use of conjunction and another feature: extended communications. An extended communication involves making the second party in a communication wait while the first performs an action. An extended write means that once both parties have agreed to synchronise, the writer performs an action before sending the value (typically, something that involves calculating the value). An extended read means that once the data has been passed, the reader can perform an additional action (typically a synchronisation) before the writer is freed. The

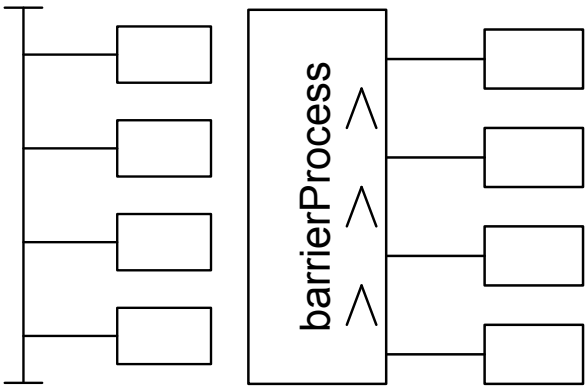


Figure 25: The equivalence between N processes enrolled on an N-party event (left), and N processes synchronising on 2-party events with a single "barrier process" that synchronises on the conjunction of all of them (right).

other party in an extended transaction does not need to know (at the program level) that the other party has performed an extended communication, and both sides may extend the communication.

A process may be hidden as follows. It creates an internal communication channel. It then performs the conjunction of an extended input on its external input channel, and an extended output on its external output channel. This means that it maintains the synchronisation behaviour, because it fuses together its input and output; they behave as if they were the same event (i.e. as if the process was not present at all).

During the extended input phase of the input (i.e. while the outside writer is waiting), the value read from the external input is sent on the internal communication channel. During the extended output phase of the output (i.e. while the outside reader is waiting for a value), the value is read from the internal communication channel, and is used as the value for the external outward communication. Thus, the value received is passed across and communicated on the next channel. This implements a hidden identity process that does not modify the value; any processing on the value should take place during one of these two extended phases, or by having a more complex internal structure than a simple forwarding channel.

5.3.5 Conjunction of 2-Party Events

In addition to two-party synchronisations (as channel communications often are), many frameworks (including CHP) also support N-party synchronisations. N processes synchronising on one item together is equivalent to N processes each

synchronising on a 2-party event with a further process that conjoins them all (this is illustrated in figure 25). This means that any system with only two-party synchronisations and conjunction can emulate N-party synchronisations.

Furthermore, it is possible to form a similar correspondence for partial events. Partial events are events where any X processes of N may synchronise together (where $X < N$; if $X = N$ this is a full event). To implement a partial event, the extra process must offer a choice between all subsets of size X of the N two-party events that it has been given. It is possible to further constrain the subsets to treat certain parties differently (e.g. the process could offer to synchronise with one of two special master processes and any three of ten worker processes).

These correspondences are useful in theory, and for thinking about expressivity; however, it is likely that the systems involving conjunctions of two-party events will be slower than implementing the full or partial events directly.

A hybrid approach is interesting – consider 201 processes wanting to synchronise on a barrier together. They could all enroll on one barrier and synchronise together – all 201 would contend for the same barrier. Consider instead two barriers – 100 processes would solely enroll on barrier and 100 solely on the other. One process would enroll on both and would conjoin the two. This has the same behaviour as the original barrier that was twice as large as these two new barriers, but (using the implementation described in this chapter) the participants would contend on only their one barrier until it was ready, and only then would they examine the other barrier. Thus conjunction could be used to reduce the contention on large barriers.

5.4 Properties, Terminology and Notation

We use a notation borrowed from boolean logic, with $a \oplus b$ meaning that a process waits for either a or b but will only perform one of them, and $a \wedge b$ means that a process waits for both a and b. The operators are associative and commutative. The unit of $\oplus$ is the never-ready event STOP, and the unit of $\wedge$ is the always-ready event SKIP. Conversely, STOP is a zero of $\wedge$.

Conjunction does distribute over choice: $a \wedge (b \oplus c) \equiv (a \wedge b) \oplus (a \wedge c)$, as demonstrated in the event table in figure 26. However, choice does not distribute over conjunction: $a \oplus (b \wedge c)$ will not behave the same as $(a \oplus b) \wedge (a \oplus c)$. The difference is revealed in the event table in figure 27; alternatively one could say that the given semantics of the latter are indeterminate.

a	b	c	$a \wedge (b \oplus c)$	$(a \wedge b) \oplus (a \wedge c)$
✗	✗	✗	None	None
✗	✗	✓	None	None
✗	✓	✗	None	None
✗	✓	✓	None	None
✓	✗	✗	None	None
✓	✗	✓	$\{a, c\}$	$\{a, c\}$
✓	✓	✗	$\{a, b\}$	$\{a, b\}$
✓	✓	✓	$\{a, b\}$ or $\{a, c\}$	$\{a, b\}$ or $\{a, c\}$

Figure 26: The event table showing the equivalence $a \wedge (b \oplus c) \equiv (a \wedge b) \oplus (a \wedge c)$. The event table is akin to a boolean logic truth table; each event is either ready (indicated by ✓) or not ready (indicated by ✗), and the possible resolutions (sets of events that could complete together) are indicated for each compound choice.

a	b	c	$a \oplus (b \wedge c)$	$(a \oplus b) \wedge (a \oplus c)$
✗	✗	✗	None	None
✗	✗	✓	None	None
✗	✓	✗	None	None
✗	✓	✓	$\{b, c\}$	$\{b, c\}$
✓	✗	✗	$\{a\}$	$\{a\}$
✓	✗	✓	$\{a\}$	$\{a\}$ or $\{a, c\}$
✓	✓	✗	$\{a\}$	$\{a\}$ or $\{a, b\}$
✓	✓	✓	$\{a\}$ or $\{b, c\}$	$\{a\}$ or $\{a, b\}$ or $\{a, c\}$ or $\{b, c\}$

Figure 27: The event table showing that $a \oplus (b \wedge c)$ will not behave the same as $(a \oplus b) \wedge (a \oplus c)$. It can be seen that the right-hand formula admits more possible resolutions than the left-hand formula.

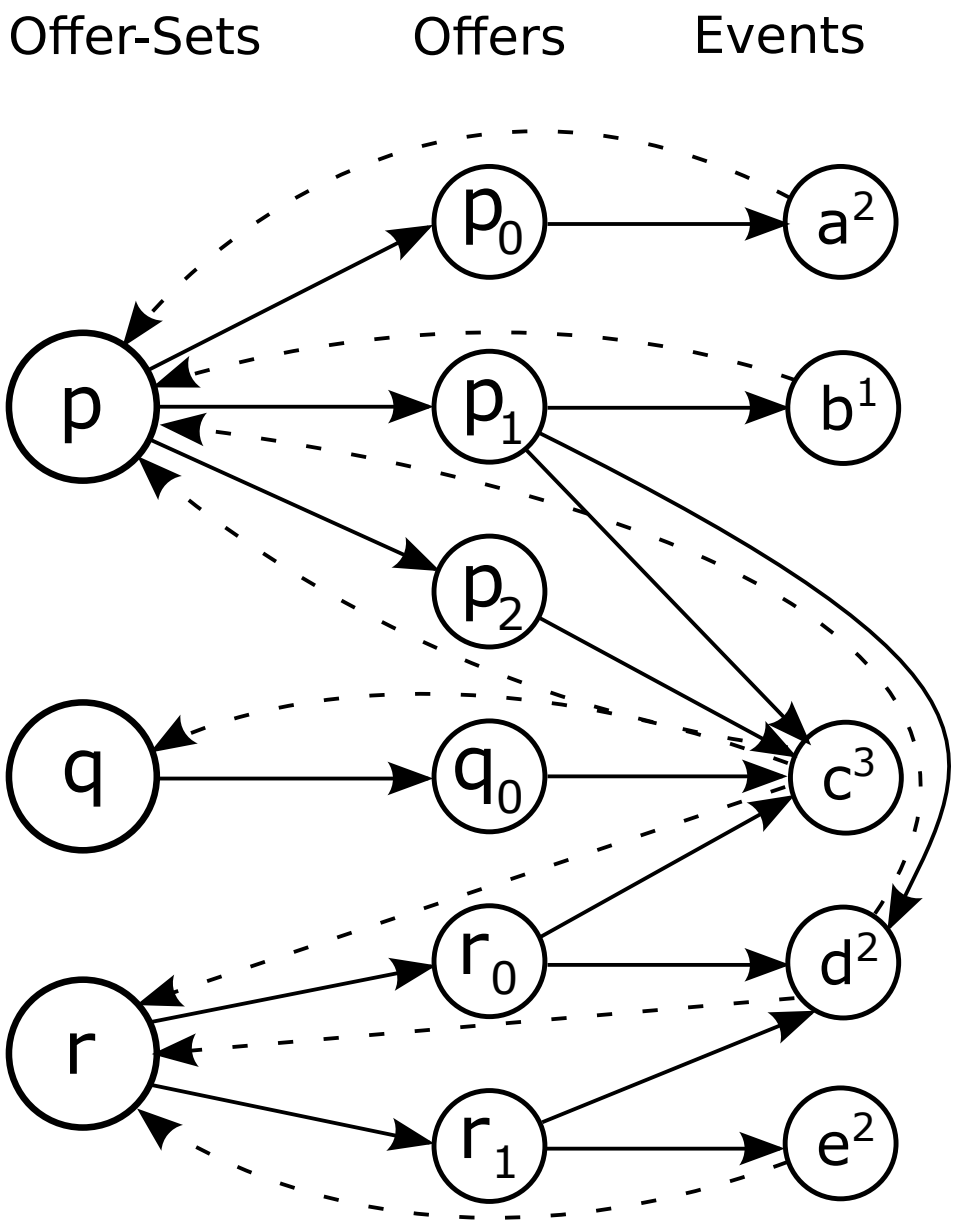


Figure 28: Tripartite graph of offer-sets (left), offers (middle) and events (right). The direction of the arrows indicates the direction of references in the system, and also the direction that the search follows. The line styles are for visual clarity and have no special meaning. Each offer is always referenced by exactly one offer-set. Events reference an offer-set if and only if the offer-set contains an offer featuring that event. Multiple edges between the same nodes are not permitted anywhere in the graph. The superscripts on the events are the current enrollment counts of those events.

Our algorithms deal with a choice normal form (a choice of conjunctions). Due to the distributivity properties, it is easier to transform complex offers into this choice normal form than it would be into, say, conjunctive normal form (a conjunction of choices).

We define a synchronisation event as being a concurrency primitive that has a persistent membership count of processes that can dynamically be increased (by enrolling) or decreased (by resigning). An event with membership count N can only complete when all N members agree to synchronise on the event. Events with a static count are a special case of those with a dynamic count.

Each *offer* can be a single event or a conjunction of several events; we choose to represent an offer as a set of events, where multiple events indicate a conjunction, a singleton set is the contained single event, and the empty set is not permitted.

A process can make multiple offers, forming an *offer-set*, but will only complete exactly one of those offers. If one or more offers can complete immediately, an arbitrary choice will be made between them (see section 5.5.8 for discussion of priority). If no offers can currently complete, the new process must wait until another process later resolves the offers. We can evisage offer-sets, offers and events as a tripartite graph (see figure 28). A resolution is a collection of offers (each from a distinct offer-set) such that each event that appears in any offer appears in exactly as many offers as it has enrollees.

5.5 Conjunction Algorithm

We will build up to an optimised version of our conjunction algorithm by first starting with a search based on simple lists (section 5.5.3), then moving onto a backtracking-based search (section 5.5.4) before finishing with our actual search algorithm, the forward-filtering search algorithm (section 5.5.5). Building the algorithm slowly allows the basic idea to be understood before the optimisations are progressively integrated.

5.5.1 Basic definitions

We begin with basic definitions that we will use in all versions of our algorithm. We assume the existence of a Map and Set type like those in the `containers` library – although *ad hoc* benchmarks have shown that list-based implementations are faster. The first definition is the event type:

```
newtype EventId = EventId Unique deriving (Eq, Ord)

data Event = Event {
  getEventId :: EventId, -- Event identifier
  getEventTVar :: TVar (Int, Set OfferSet) } -- (Enrolled count, current offers)

instance Eq Event where (==) = (==) `on` getEventId
instance Ord Event where compare = compare `on` getEventId

newEvent :: Int -> IO Event
newEvent n = Event <$> (EventId <$> newUnique)
                   <*> atomically (newTVar (n, Set.empty))
```

The next definition is that of an offer:

```
newtype SignalValue = Signal Int

type SignalVar = TVar (Maybe SignalValue)

data Offer = Offer {
  signalValue :: SignalValue,
  offerAction :: STM (),
  eventsSet :: Set Event }
```

We can now define a type for offer-sets:

```
newtype OfferSetId = OfferSetId ThreadId deriving (Eq, Ord)

data OfferSet = OfferSet {
  signalVar :: SignalVar, -- Variable to use to signal when committed
  offerSetId :: OfferSetId -- Id of the process making the offer
  offersInSet :: [Offer] } -- A list of all offers

instance Eq OfferSet where (==) = (==) `on` offerSetId
instance Ord OfferSet where compare = compare `on` offerSetId

makeOfferSet :: [(SignalValue, STM (), Set Event)] -> IO OfferSet
makeOfferSet offs = do
  tv <- newTVarIO Nothing
  tid <- myThreadId
  return $ OfferSet tv (OfferSetId tid) [Offer v a s | (v, a, s) <- offs]
```

Finally, a return type related to enabling:

```
data EnabledActions = EnabledActions
  {waitForEvent :: STM SignalValue, retractOffers :: STM (Maybe SignalValue)}
```

5.5.2 Helper functions

We also define various helper functions:

```
allEventsInOfferSet :: OfferSet -> Set.Set Event
allEventsInOfferSet = foldr1 Set.union . map eventsSet . offersInSet

modifyTVar :: (a -> a) -> TVar a -> STM ()
modifyTVar f tv = readTVar tv >>= writeTVar tv . f

fireOfferSets :: [(OfferSet, Int)] -> STM ()
fireOfferSets offerSets =
  do mapM_ (offerAction . snd) offers
     sequence_ [ writeTVar (signalVar os) (Just $ signalValue offer)
               | (os, offer) <- offers ]
  where
    offers = [(os, offersInSet os !! i) | (os, i) <- offerSets]

modifyAllEvents :: (OfferSet -> Set.Set OfferSet -> Set.Set OfferSet)
                -> OfferSet -> STM ()
modifyAllEvents f offerSet
  = mapM_ (modifyTVar m . getEventTVar)
          (Set.toList $ allEventsInOfferSet offerSet)
  where
    m (enrolled, offerSets) = (enrolled, f offerSet offerSets)

retractOfferSet :: OfferSet -> STM ()
retractOfferSet = modifyAllEvents Set.delete

offerOfferSet :: OfferSet -> STM ()
offerOfferSet = modifyAllEvents Set.insert

readAndRetractOfferSet :: OfferSet -> STM (Maybe SignalValue)
readAndRetractOfferSet offer
  = do x <- readTVar $ signalVar offer
```

```
-- Since the transaction will be atomic, we know
-- now that we can disable the barriers and nothing fired :
when (isNothing x) ( retractOfferSet offer )
return x
```

5.5.3 Pure search

We begin with a pure (i.e. non-monadic) search algorithm that performs the search in non-monadic code. In order to be able to do this we need a first phase that reads in the information from the full connected network of events:

```
spiderAllEvents :: Map EventId (Int, Set OfferSet) -> OfferSet
                -> STM (Map EventId (Int, Set OfferSet))
spiderAllEvents m os = foldM spiderOffer m ( offersInSet os)

spiderOffer :: Map EventId (Int, Set OfferSet) -> Offer
            -> STM (Map EventId (Int, Set OfferSet))
spiderOffer m o = foldM spiderEvent m (Set.toList $ eventsSet o)

spiderEvent :: Map EventId (Int, Set OfferSet) -> Event
            -> STM (Map EventId (Int, Set OfferSet))
spiderEvent m e
  | getEventId e `Map.member` m = return m
  | otherwise = do x <- readTVar (getEventTVar e)
                   let m' = Map.insert (getEventId e) x m
                   foldM spiderAllEvents m' (Set.toList $ snd x)
```

The only thing to note about this code is that it must keep track of which events it has visited, and avoid visiting the same event twice. This is not simply because it is redundant to read it twice – there are always cycles in the offer-set/offer/event graph, so we must make sure not to follow this cycle in an infinite loop.

The above spidering code that reads in all the data is used as the first phase, followed by a second search phase. We will define the search function shortly, which has type:

```
search :: Map EventId (Int, Set OfferSet) -> [Map OfferSet Int]
```

We first define the top-level function of the whole search, which takes an offer-set and returns the result (either the value associated with a completed offer,

or some actions related to waiting and withdrawing – see appendix C for more details):

```
enableEvents' :: OfferSet -> STM (Either SignalValue EnabledActions)
enableEvents' offerSet = do
  offerOfferSet offerSet
  eventInfo <- spiderAllEvents Map.empty offerSet
  case search eventInfo of
    (oss:_) -> do fireOfferSets (Map.toList oss)
                  mapM_ retractOfferSet (Map.keys oss)
                  Just val <- readTVar (signalVar offerSet)
                  return (Left val)
    [] -> return $ Right $ EnabledActions
            (readTVar (signalVar offerSet) >>= maybe retry return)
            (readAndRetractOfferSet offerSet)
```

The search function returns a list of possible completions. If there are any, one is picked arbitrarily. The offer-sets involved are notified, and then the offer-sets are retracted from all relevant events. If there are no completions, we return an EnabledActions item with the code that will wait for a completion and the code that will retract the offer sets.

The search function is defined as follows:

```
1  search :: Map EventId (Int, Set OfferSet) -> [Map OfferSet Int]
2  search m
3    = map (Map.fromList . map fst) .
4      filter (checkEvents . concatMap snd) .
5      filter (not . null) . map catMaybes $
6      sequence [ Nothing : map Just (allChoices os) | os <- allOfferSets ]
7    where
8      checkEvents :: [Event] -> Bool
9      checkEvents es = and . Map.elems . Map.intersectionWith ((==) . fst) m $
10                      Map.fromListWith (+) [(getEventId e, 1) | e <- es]
11
12     allChoices os = [ ((os, i), Set.toList (eventsSet o))
13                     | (i, o) <- zip [0..] (offersInSet os)]
14
15     allOfferSets = Set.toList . Set.unions $ map snd (Map.elems m)
```

The search function is given a map from event identifier to enrollment count and current offers. The function first goes through all the offer-sets, and produces

a list of each choice (i.e. one for each offer) that the offer-set could make. All the choices are labelled with a Just item, and the possibility that the offer-set does not complete here (the Nothing item) is also included (line 6). This gives a list; for example, based on figure 28, this would be:

```
[[Nothing, Just ((p, 0), [a]), Just ((p, 1), [b, c, d]), Just ((p, 2), [c])]
,[Nothing, Just ((q, 0), [c])]
,[Nothing, Just ((r, 0), [c, d]), Just ((r, 1), [d, e])]
]
```

The sequence function is applied to this list. In the list monad, as here, this produces a list of lists, where each list pulls one from an original inner list. Continuing our example, applying sequence gives the following – each item is a set of completions:

```
[[Nothing, Nothing, Nothing]
,[Nothing, Nothing, Just ((r, 0), [c, d])]
,[Nothing, Nothing, Just ((r, 1), [d, e])]
,[Nothing, Just ((q, 0), [c]), Nothing]
,[Nothing, Just ((q, 0), [c]), Just ((r, 0), [c, d])]
,[Nothing, Just ((q, 0), [c]), Just ((r, 1), [d, e])]
,[Just ((p, 0), [a]), Nothing, Nothing]
,[Just ((p, 0), [a]), Nothing, Just ((r, 0), [c, d])]
,[Just ((p, 0), [a]), Nothing, Just ((r, 1), [d, e])]
,[Just ((p, 0), [a]), Just ((q, 0), [c]), Nothing]
,[Just ((p, 0), [a]), Just ((q, 0), [c]), Just ((r, 0), [c, d])]
,[Just ((p, 0), [a]), Just ((q, 0), [c]), Just ((r, 1), [d, e])]
,[Just ((p, 1), [b, c, d]), Nothing, Nothing]
,[Just ((p, 1), [b, c, d]), Nothing, Just ((r, 0), [c, d])]
,[Just ((p, 1), [b, c, d]), Nothing, Just ((r, 1), [d, e])]
,[Just ((p, 1), [b, c, d]), Just ((q, 0), [c]), Nothing]
,[Just ((p, 1), [b, c, d]), Just ((q, 0), [c]), Just ((r, 0), [c, d])]
,[Just ((p, 1), [b, c, d]), Just ((q, 0), [c]), Just ((r, 1), [d, e])]
,[Just ((p, 2), [c]), Nothing, Nothing]
,[Just ((p, 2), [c]), Nothing, Just ((r, 0), [c, d])]
,[Just ((p, 2), [c]), Nothing, Just ((r, 1), [d, e])]
,[Just ((p, 2), [c]), Just ((q, 0), [c]), Nothing]
,[Just ((p, 2), [c]), Just ((q, 0), [c]), Just ((r, 0), [c, d])]
,[Just ((p, 2), [c]), Just ((q, 0), [c]), Just ((r, 1), [d, e])]]
]
```

Since the offer-sets have 3, 1 and 2 offers in them, and each offer-set also may not complete, there are $(3+1) \times (1+1) \times (2+1) = 24$ possible completions. Not all of these are valid because they do not match the enrollment counts, but these all the possible cases ignoring enrollment counts.

We then filter out all the Nothing items, which are no longer needed, and also filter out any sets of completions that are empty, i.e. that have no offer-sets completing (both line 5).

The next step is to filter out all sets of completions that have too few processes synchronising on an event. This is checked by the checkEvents function. Line 10 builds up a map from event identifier to the count of processes wishing to synchronise on that event. This is then checked against the enrollment count for the event (held in the map m), to see if the counts match. If all of the counts match, this is a valid completion. If not, it will be removed from the list.

For example, the completion:

```
[Just ((p, 2), [c]), Just ((q, 0), [c]), Just ((r, 0), [c, d])]
```

is not valid because although the event c has the correct number of associated offer-sets/processes (recall from figure 28 that its enrollment count is 3), d only has one offer-set associated with it, which is not equal to its enrollment count of 2. The only completion that is valid in this instance is:

```
[Just ((p, 1), [b, c, d]), Just ((q, 0), [c]), Just ((r, 0), [c, d])]
```

5.5.4 Backtracking Monadic Search

The search algorithm shown in the last section is very inefficient. One major problem is that it performs a full traversal of the event graph before searching. It may be able to complete the first event it searches but it will still read all the rest. Should any of those (irrelevant) events change during the transaction, this creates an STM conflict and the transaction will be needlessly re-run, because our search algorithm has touched more events than it needed to.

Similarly, the search algorithm makes offers on all its events before searching. This again needlessly "touches" the TVars involved in the events – if any event can complete, the offers will be immediately revoked anyway. A more logical thing to do is search first, and only record the offers if no events can complete.

Our improvement on the pure search integrates the search with the reading of events, to make sure we only read from as many TVars as the search actually

requires. In a sense, this is a lazier search even though it involves effects, as it only reads from as many events as it needs to. To still get the proper search behaviour, we use a simple backtracking monad. The backtracking transformer is shown here:

```
data BacktrackT m a = BacktrackT { run :: m (Maybe (a, BacktrackT m a)) }

runBacktrackT :: Monad m => BacktrackT m a -> m (Maybe a)
runBacktrackT m = do x <- run m
                     return (fmap fst x)

instance Monad m => Monad (BacktrackT m) where
  return x = BacktrackT $ return $ Just (x, empty)
  m >>= f = BacktrackT $ do x <- run m
                            case x of
                              Just (r, n) -> run $ f r <|> (n >>= f)
                              Nothing -> return Nothing

lift :: Monad m => m a -> BacktrackT m a
lift m = BacktrackT $ do x <- m
                         return $ Just (x, empty)

instance Monad m => Functor (BacktrackT m) where
  fmap = liftM

instance (Monad m) => Applicative (BacktrackT m) where
  pure = return
  (<*>) = ap

instance (Monad m) => Alternative (BacktrackT m) where
  empty = BacktrackT $ return Nothing
  (<|>) a b = BacktrackT $ do
                x <- run a
                case x of
                  Nothing -> run b
                  Just (r, m) -> return $ Just (r, m <|> b)

backtrack :: Alternative f => f a
backtrack = empty
```

Many of these definitions are straightforward. The monad transformer itself either returns Nothing (if there are no more paths to try), or a value combined with an alternate search path. The Alternative instance will run the right-hand side of a choice if the left-hand side fails. We define backtrack as a synonym for empty; whenever this is encountered, an alternate search path will be tried.

Having defined the backtracking monad, the code for the search is quite short:

```
type SearchState = (Map OfferSet Int, Map Event Int)
 -- Pair:
 -- map from offer-set to chosen offer, and
 -- map from event to number of participants still needed.

type SearchM = BacktrackT STM

searchOfferSet :: SearchState -> OfferSet -> SearchM SearchState
searchOfferSet ss@(visited, eventCounts) os
  = if Map.member os visited
      then return ss -- Already visited
      else foldr (<|>) backtrack
             [searchOffer (Map.insert os i visited, eventCounts) o
             | (i, o) <- zip [0..] (offersInSet os)]


searchOffer :: SearchState -> Offer -> SearchM SearchState
searchOffer ss offer = foldM searchEvent ss (Set.toList $ eventsSet offer)

searchEvent :: SearchState -> Event -> SearchM SearchState
searchEvent (visited, eventCounts) e
  = do (enrollCount, offerSets) <- lift $ readTVar $ getEventTVar e
       let ss = (visited,
                 Map.alter (Just . maybe (enrollCount - 1) pred) e eventCounts)
       foldM searchOfferSet ss (Set.toList offerSets)
```

This code is just a rewritten form of the non-monadic search that incorporates the reading of events.

The key element is after all the offer-sets have been searched, when the events are checked for the correct number of synchronisers:

```
1 enableEvents' :: OfferSet -> STM (Either SignalValue EnabledActions)
2 enableEvents' offerSet = do
```

```
3         result <- runBacktrackT $ do
4           (os, es) <- searchOfferSet (Map.empty, Map.empty) offerSet
5           when (any (/= 0) $ Map.elems es) backtrack
6           return os
7         case result of
8           Just oss -> do fireOfferSets (Map.toList oss)
9                          mapM_ retractOfferSet (Map.keys oss)
10                         Just val <- readTVar (signalVar offerSet)
11                         return (Left val)
12          Nothing -> do
13            offerOfferSet offerSet
14            return $ Right $ EnabledActions
15              (readTVar (signalVar offerSet) >>= maybe retry return)
16              (readAndRetractOfferSet offerSet)
```

On line 5, still inside the backtracking monad, the events are checked to ensure that no events in the set of events being completed require any more participants. If this check fails, backtracking ensues to try another possible completion.

Otherwise this function is very similar to the previous pure-search, except that the offer-set is only recorded when no completions were found.

5.5.5 Forward-filtering search

The search algorithm in the previous section avoids reading events when it does not need to, but it is still inefficient. The search algorithm can spend a lot of time checking paths that obviously cannot complete. To use an example, consider figure 28 again. Consider beginning the search with offer-set q. The search will choose offer q_0 (the only choice), which features event c. The search will then continue by searching either p or r. If if chooses p, it may then choose offer p_0. At this point it has chosen a dead-end. The reason is that we have already chosen an offer featuring event c. For the event to complete, all enrolled members must synchronise together. Since p is making an offer on c and is therefore an enrolled member of c, for c to complete we require p. So any choice of p's offers that does not feature c (i.e. choosing p_0) is bound to fail. Similarly, if we had explored r, choosing r_1 would be bound to fail for exactly the same reason.

In this section we introduce a forward-filtering search that utilises the above observation. During this search we maintain two maps, one with all the offer-sets that we have already visited (from which we have picked an offer to search)

and one with all the offer-sets that we know we still need to visit (a work list, essentially):

```
data SearchState = SS {
  visited  :: Map OfferSet Int,
  toVisit  :: Map OfferSet [TrimmedOffer] }

addPick :: OfferSet -> Int -> SearchState -> SearchState
addPick os r (SS v nv) = SS (Map.insert os r v) nv
```

The TrimmedOffer data type is a processed form of an Offer that holds its index in the offer-set (for notifying the offer-set later on) and the current trimmed set of events in the offer:

```
data TrimmedOffer = TrimmedOffer {
  index :: Int,
  trimmedEvents :: Set Event }
```

We trim the events in our toVisit work-list as we process them to ensure that we do not visit any event twice (which is needless).

The checkEvent function is the one that visits the events:

```
checkEvent :: (Bool, OfferSet) -> SearchState -> Event -> STM SearchState
checkEvent (isNew, cos) ss e = do
  (enrollCount, offers) <- readTVar $ getEventTVar e
  let numOffers = Set.size offers
  if numOffers == enrollCount || (numOffers == enrollCount - 1 && isNew)
    then maybe retry return $
      filterWork (if isNew then offers else Set.delete cos offers) e ss
      -- No need to delete new offerset as it won't be there
    else retry
```

The first parameter is the offer-set that led to this event being processed and a Bool to indicate if it's a new offer-set; this search does not record the offers from the latest offer-set before searching. The given event is read from. It can only complete if either: the number of existing offers matches the enroll count, or: the number of existing offers is one less than the enroll count and the offer-set containing the event is new. If neither of those conditions match, we abort this current search path using STM's retry function. If the conditions are met, we filter out the current offer-set from the offerers on the event (if needed) and filter our work-list.

The work-list filtering is performed by this function:

```
1  filterWork :: Set OfferSet -> Event -> SearchState -> Maybe SearchState
2  filterWork offerers e ss
3    | Set.null offerers = Just ss
4    | not . Set.null $ Set.intersection offerers (Map.keysSet $ visited ss)
5        = Nothing
6    | otherwise =
7        let updatedToVisit
8              = unionMap id process (\x _ -> process x)
9                  (toVisit ss)
10                 (Map.fromAscList $ map (\x -> (x, getOffers x))
11                                        (Set.toAscList offerers))
12       in if any null (Map.elems updatedToVisit)
13            then Nothing
14            else Just (ss { toVisit = updatedToVisit })
15   where
16     process :: [TrimmedOffer] -> [TrimmedOffer]
17     process os = [ TrimmedOffer i (Set.delete e es)
18                  | TrimmedOffer i es <- os, Set.member e es]
19
20 getOffers :: OfferSet -> [TrimmedOffer]
21 getOffers os = zipWith TrimmedOffer [0..] (map eventsSet (offersInSet os))
22
23 unionMap :: Ord k => (a -> v) -> (b -> v) -> (a -> b -> v)
24           -> Map.Map k a -> Map.Map k b -> Map.Map k v
25 unionMap f g h a b = ...
```

If there are no offerers to process (this can happen if the event has enrollment count one and is offered by the new offer-set), there is nothing to be done (line 3).

Line 4 is an optimisation. When we process an offer-set (in the searchOfferSet function, below), we pick an offer and process all its events before processing another offer-set, and we ensure (by filtering the work list) that the event will not be processed a second time by picking any of those offer-sets. Therefore, if we encounter an event with an offer-set we have already seen, we must have picked from that offer-set an offer which does not contain this event. Therefore, this event will not be completeable and we abandon this search path.

The main body of the filterWork function uses the unionMap function. Its implementation is omitted for brevity, but its type illustrates its purpose. It joins two maps together. If a key only occurs in the first map, a, it is inserted into the

final map with f applied to its value. Similarly, if a key only occurs in the second map, b, it is inserted into the final map with g applied to its value. If a key occurs in both maps, it is inserted into the final map with h applied to the two values.

The unionMap function is used by filterWork to merge the old work-list with the new work-items from the event. If an offer-set is only in the old work-list, it is retained as-is. If an offer-set is only in the new work-items, it is processed. If an offer-set is in the old work-list and the new work-items, the version from old work-list is processed. The processing keeps only those offers that contain the event being processed (this forces that event to be chosen by all the offers), and then remove the event as it will not need to be processed again during the search (it has been chosen and all the offerers committed to it). We later inspect these trimmed offers to ensure that they are all non-empty. If any are empty then not all the offerers can commit to the event, and our search path must be abandoned.

The different search paths are investigated from a searchOfferSet function:

```
searchOfferSet :: (Bool, OfferSet) -> [TrimmedOffer] -> SearchState
               -> STM (Map OfferSet Int)
searchOfferSet nos@(_, os) offers ss
  = foldr orElse retry (map searchOffer offers)
  where
    searchOffer offer
      = do ss' <- addPick os (index offer) <$>
                    foldM (checkEvent nos) ss (Set.toList $ trimmedEvents offer)
           case Map.minViewWithKey (toVisit ss') of
             -- All visited:
             Nothing -> return (visited ss')
             -- At least one left:
             Just ((os, next), rest) ->
               searchOfferSet (False, os) next (ss' { toVisit = rest })
```

The parameters are an offer-set to search (and Bool indicating if it is new), the trimmed offers associated with that offer-set, and the current search state. The returned value is a valid search result; if no such result can be found, the function will abandon with the use of STM's retry function.

The searchOfferSet function tries searching for a completion with each different offer in the current trimmed offers. It checks all the events in a given offer, which updates the toVisit work-list, which is then processed using a recursive call. If no offer-sets remain to be visited, a valid result has been found.

Finally, this function is wrapped up in an enableEvents' function that is effectively identical to the one used in the backtracking search:

```
1  enableEvents' :: OfferSet -> STM (Either SignalValue EnabledActions)
2  enableEvents' new = do
3    r <- (Just <$> searchOfferSet (True, new) (getOffers new)
4                                  (SS Map.empty Map.empty))
5          `orElse` return Nothing
6    case r of
7      Nothing ->
8        -- Must record our offers
9        do offerOfferSet new
10          return $ Right $ EnabledActions
11            (readTVar (signalVar new) >>= maybe retry return)
12            (readAndRetractOfferSet new)
13     Just act ->
14       do fireOfferSets (Map.toList act)
15          let oldOffers = Map.delete new act
16          mapM_ retractOfferSet (Map.keys oldOffers)
17          Just chosenItem <- readTVar (signalVar new)
18          return $ Left chosenItem
```

To illustrate how this works, we consider the scenario depicted in figure 28, where p and r have already made their offers, and q is enabling its events.

The function searchOfferSet is called with q as the first parameter, the second parameter being the trimmed offers [TrimmedOffer 0 (Set.fromList [c])] and the third being a blank search state. checkEvent is then called on c. The enrollCount of c is 3, and numOffers is 2, but isNew is True so it passes the **if** condition. filterWork is then called with the two offer-sets p and r, the event c and a blank search state. Both of the two guards in filterWork fail so the third guard is used. Since toVisit ss is blank, the unionMap call effectively just calls process on all the items in the new work-list, which is:

```
Map.fromList [(p, [TrimmedOffer 0 (Set.fromList [a]),
                   TrimmedOffer 1 (Set.fromList [b, c, d]),
                   TrimmedOffer 2 (Set.fromList [c])])
             ,(r, [TrimmedOffer 0 (Set.fromList [c, d]),
                   TrimmedOffer 1 (Set.fromList [d, e])])]
```

The process function keeps all the offers containing c, and deletes c from the event sets, leaving the new updatedToVisit item as:

```
Map.fromList [(p, [TrimmedOffer 1 (Set.fromList [b, d]),
                   TrimmedOffer 2 (Set.fromList [])])
             ,(r, [TrimmedOffer 0 (Set.fromList [d])])]
```

This map is returned to searchOfferSet which picks the minimum item (an arbitrary pick) to process next. We will assume for the purposes of illustration that p < r, and we will also re-order the trimmed offers (which have an arbitrary offer) to explore what happens if the offer p_2 is searched next by searchOfferSet. In this case, checkEvent is not called because the set of events in the trimmed offer is empty, and searchOfferSet will be called with toVisit as simply:

```
Map.fromList [(r, [TrimmedOffer 0 (Set.fromList [d])])]
```

Now checkEvent will be called on d. The enrollment count is 2, and numOffers is also 2, so it passes the **if** condition, and filterWork will be called. The second guard performs the intersection of the offerers (p and r) and the visited offer-sets (p and q). This is non-empty, and thus the algorithm backtracks. To complete d, we need p. But we already visited p and did not choose an offer that featured d, hence we are only processing d when we come to process the offer-set r.

The backtrack will go back and try offer p_1 instead. This will call checkEvent on b. This will pass the enrollment checks, and filterWork will be called with an empty set of offerers (p having been deleted before the call), which succeeds on the first guard. checkEvent is then also called on d, which passes the enrollment checks and calls:

```
filterWork (Set.fromList [r]) d
  (SS (Map.fromList [(p, 1), (q, 0)])
      (Map.fromList [(r, [TrimmedOffer 0 (Set.fromList [d])])]))
```

This results in updatedToVisit being the union of two maps both featuring only r, which means that r's previous entry in the toVisit map gets processed. The trimmed offer there does feature d, so it is retained with d deleted, meaning that updatedToVisit is:

```
Map.fromList [(r, [TrimmedOffer 0 (Set.fromList [])])]
```

This is returned, and once searchOfferSet processes this work item (which does not require any checking of events) the work queue is empty, and the accumulated result is returned:

```
Map.fromList [(p, 1), (q, 0), (r, 0)]
```

Note that the algorithm performed very little backtracking. The use of the forward-filtering prevents the exploration of search paths that will definitely come to nothing, which allowed the search to quickly progress to the single valid solution.

5.5.6 Scalability and Complexity

One way to implement this complicated choice algorithm would be with a single central data structure with a lock. Each offering process would acquire the lock, then search for completions and either resolve successfully or record their choice and wait. The problem with this approach would be that it would cause independent processes in the system (one trying to complete event a, another trying to complete event z) to contend for the lock.

The advantage of the algorithm described above, using Software Transactional Memory, is that processes will only contend if they share an event in their offers. Processes trying to complete event a and event z will do so separately without causing any contention. Thus the algorithm can be said to scale well as more cores are added to the system; only connected processes will interact with each other, and processes that are separate in this regard can all be executed simultaneously on many cores without issue.

The worst-case algorithmic complexity for the conjunction algorithm is very high. The worst case is that each participant will search all possible completions (which in the worse case is every possible subset of the connected events, i.e. 2^e where e is the number of events). Searching a possible completion involves, in the worst case, searching all the events and participants in the completion. This gives an upper limit of $O((e + p)2^e)$ where e is the number of events and p is the number of processes. In practice, this upper limit is rarely approached; the cell-pipeline example explained earlier in the chapter can be run and displayed in real-time (i.e. at 20 frames per second).

5.5.7 Partial Events

A partial event is one where only X out of N enrolled processes are required to complete the event ($X < N$). When $X \neq N$, a central assumption of our algorithm is broken. It is no longer the case that when an offer-set features an event but does not choose the offer containing that event, that event can no longer complete. So the way the offers in the work queue are pruned cannot work the same way. Additionally, our strategy that once an event is featured in one offer

it must feature in all other offers that might be chosen is no longer valid.

Our existing pruning strategy can be formulated as the calculation that X participants are required for the event, and since there are N enrolled processes and for full events $X = N$, when at least one participant refuses to participate, the entire event cannot complete. For partial events, this means that when at least $N - X$ participants refuse to participate, the event can no longer occur – and in fact, if there are N' processes offering on the event ($N' \leq N$), when at least $N' - X$ participants refuse to participate, the event can no longer occur. So we would need to keep a count associated with each event of participants who had so far (on the search path) refused to engage in the event; when this count exceeds $N' - X$, we would backtrack.

5.5.8 Priority

One important feature not yet discussed is priority. Priority is useful when programming simulations: in particular to allow low-priority events. Our discrete-time simulations invariably have the following pattern: perform actions X and/or Y if possible, and then synchronise on a global tick event with all other agents in the simulation. It is important that the global tick has a lower priority than the other actions. If it does not, non-determinism can result because a process may tick rather than performing a ready alternative action. The intention in the design is that tick should happen only when nothing else can.

It is clear that local priorities are in general a poor solution: if one process offers a or b preferring a, and a second offers a or b preferring b, an event cannot be chosen that will satisfy both. We instead consider global priorities, where each event possesses an intrinsic constant priority. Without conjunction, this would be a simple matter, semantically: when offering a or b and both are ready, choose the event with the highest priority. With conjunction it is less clear; given the choice of completing (a and b) or (c and d), how should it be decided, given the priorities of a, b, c and d?

Note also that featuring priority makes the choice algorithm less optimal; if the priority of a resolution is decided based on all the events being completed, we must find all resolutions (which involves examining potentially many more events than finding the first resolution) and then calculate their priority before deciding on one. This is easily implementable with a small change to our current algorithm, but could have devastating consequences for the performance.

An alternative, which would require an even smaller change to the algorithm, is to guarantee that resolution A will be chosen over resolution B if all events in A have a higher priority than all of those in B: $(\forall\, a \in A. \forall\, b \in B. a > b) \Rightarrow A > B$. In any other case, an arbitrary choice will be made. This can then be implemented quite simply: by sorting the new offers by the priority of an arbitrary event (the first in the set). This will compare arbitary events from the resolutions that would result from successfully searching this offer; this gives consistent priority if the above condition is satisfied. Note also that if no processes are using conjunction, each resolution will contain a single event, and thus this system is equivalent to global priorities on events without conjunction. Only a small modification is required to our forward-filtering search. On line 3, the getOffers new call simply has to be wrapped with a sortBy call, for example:

```
(sortBy
  ( flip $ comparing ( getEventPriority . Set.findMin . trimmedEvents))
  ( getOffers new))
```

The findMin is essentially an arbitrary pick. The flip applied to comparing ensures the list is sorted in descending order of priority, leaving the highest priority at the head of the list. The searchOfferSet function will then search this list in order (descending depth-first), with the highest priority items earlier in the list.

5.6 Benchmarks

Given that conjunction is a new feature, it is not possible to benchmark our implementation of conjunction against another. Instead, we give indications of the cost of supporting conjunction for standard channel communications, as this is the primary negative effect of adding conjunction to existing message-passing frameworks.

Currently, we have only implemented the conjunction algorithm in our Haskell library CHP (Communicating Haskell Processes) [Brown, 2008], due to Haskell's good support for Software Transactional Memory [Harris et al., 2005]. Therefore it seems appropriate to benchmark against other Haskell-based message-passing libraries. These include synchronous channel implementations based on Haskell's MVars and STM itself, Concurrent ML (a Haskell implementation of the ML library) and an STM implementation of synchronisations with choice (a de-centralised version of the Oracle algorithm [Welch et al., 2006b]).

5.6.1 Methodology

All the benchmarks were carried out on the same 8-core Intel Xeon E5310 (1.6GHz) machine with 4GB RAM, running Debian GNU/Linux, kernel version "2.6.26-1-686-bigmem".

The benchmarks were timed using the Criterion benchmarking library by performing 100 iterations of each benchmark. Each benchmark recording included all startup and shutdown costs (particularly crucial in a garbage-collected language), with iterations set sufficiently high to make the fixed costs for an empty program ($\mu = 5.227$ms; 95% CI: 5.190ms, 5.287ms) immaterial.

All confidence intervals were calculated using a non-parametric (i.e. without assumption of underlying distribution) bootstrap with bias-corrected acceleration [Efron and Tibshirani, 1993] using 100000 resamples.

5.6.2 Pairs of Communicators

To test the speed of channel communications in the various systems, we create a program with N pairs of communicators, where each pair consists of a writer and a reader, with the former repeatedly communicating to the latter (100000 times). The run-time system underlying all our systems use light-weight threads scheduled flexibly (i.e. with migration) across the different processor cores.

It can be the case that having the writer and reader on different cores, running without contention, is actually slower than having them on the same core, continually being switched out for each other. Due to the effects of scheduling and migration with small numbers of pairs (low values of N), we set N to be suitably high. Our benchmarks were carried out with $N = 100$ to avoid such problems.

The results are presented in figure 29. It can be seen that the results for the implementation of conjunction in CHP are approximately a factor of four to six slower than those implementations without choice (MVar, STM) and a factor of two slower than the fastest implementation with choice (Sync)[2].

This result shows that adding conjunction to existing systems does not introduce a major *factor* difference for basic channel communications; given that adding choice added a factor of two or three, a further factor of two for conjunction seems an acceptable parallel in exchange for the extra expressive power.

[2]The CML implementation is particularly slow due to triggering a pathological case in the run-time system; this problem is currently being fixed but we believe CML will still be slower than CHP.

System	Mean	95% CI
MVar	0.7188	0.7069 – 0.7319
STM	1.0392	1.0215 – 1.0610
Sync	1.9387	1.9166 – 1.9635
CML	34.2610	34.1744 – 34.3670
CHP	4.1625	4.1345 – 4.1927

Figure 29: Times (means and 95% confidence intervals) for 100 pairs of writer and reader processes, communicating 100000 times (seconds, to 4 d.p.)

Type	Mean	95% CI
Single	4.1625	4.1345 – 4.1927
Conjunction	16.6746	16.5818 – 16.9561

Figure 30: Times (means and 95% confidence intervals) for 100 pairs of writer and read processes communicating 10000 times on a pair of channels in conjunction versus a single pair in CHP (seconds, to 4 d.p.)

5.6.3 Conjunctive Pairs

To give an idea of the speed of conjunction itself, we compare it to that of standard channel communications. A benchmark similar to the one in the previous section was constructed, but each pair of communicators were communicating on a conjunction of two channels, instead of solely on one.

The results are presented in figure 30. It can be seen that for CHP, communicating on the conjunction of two channels is a factor of three slower than communicating on a single channel; ideally, we might hope that this factor was less than two (the cost for the two channel communications separately). The time is around one order of magnitude worse than communicating on a single channel in the fastest systems (see figure 29).

5.7 API and Wider Applicability

The work on conjunction is not specifically dependent on CHP. The same feature could be implemented in other libraries (e.g. JCSP) and other languages (e.g. occam-π). The implementation in these other languages could use STM – which can be implemented using atomic compare-and-swap operations, see Fraser and Harris [2007].

Producing a language binding or library interface for conjunction is trivial: an operator is required that joins two event synchronisations into one, and it should

be allowed in guards in choices. The interface in CHP is as follows:

```
(<&>) :: CHP a -> CHP b -> CHP (a, b)
every :: [CHP a] -> CHP [a]
```

The latter function is a list-form of the former operator. Thus, the every function joins all of the leading actions in the synchronisation into one conjoined action (and returns a list of the results). If the actions have bodies beyond a leading action, those bodies are executed in parallel after the initial conjoined synchronisation has completed.

In CHP, channel communications (both inputs and outputs) and barrier synchronisations can be used in a conjunction. Attempting to conjoin any other action treats the action as a skip guard – since this is a unit of conjunction, other actions are effectively ignored.

5.8 Conjunction and Related Work

One of the key features of the work on conjunction in this chapter is that it is possible to make a choice between conjunctions, e.g. (a AND b) OR (c AND d). This choice can be made by participants at both ends of the synchronous events. Most work that appears to be similar to conjunction does not feature some of these capabilities.

Sulzmann et al. [2008] describe an Erlang-like library for message-passing where the receiver can specify a pattern which matches multiple input messages. For example, a receiver may specify they want a message satisfying pattern A, and a message satisfying pattern B. These patterns are similar to conjunction – but they can only be specified by the receiver, and they are matched against (asynchronously-sent) messages in a mailbox. This is therefore less powerful than the conjunction presented here, and the complexity is in the rule-matching rather than in any added concurrent behaviour.

Various other systems, for example by Vasudevan et al. [2008], feature communications that can occur among multiple readers and writers (in our terminology, these are many-to-many channels). The main differences between these and the conjunction presented here is that conjunction can join events together *ad hoc* rather than requiring all parties to synchronise on a pre-ordained channel, and we allow choices between conjunctions, which libraries with multi-way communications often do not.

5.9 Formal Embedding

We have begun some preliminary work to modify the traces model of the CSP process calculus [Hoare, 1985; Roscoe, 1997] to provide formal reasoning support for conjunction (which we term CSPc – CSP with conjunction). A mapping to plain CSP is difficult as it requires analysis of the whole system at once, and conjoined events interact awkwardly with CSP's hiding operator. However, modifying the traces model itself is a more promising avenue of investigation. The work is most simply explained by describing the changes necessary to the original CSP traces model described by Roscoe [1997] – all page numbers refer to that text.

5.9.1 Traces

Roscoe [1997] gives some parts of the traces model on page 37, for example: $traces(a \rightarrow P) = \{\langle\rangle\} \cup \{\langle a\rangle \hat{} s \mid s \in traces(P)\}$. All these rules remain much the same, but whenever there is a single event in CSP, e.g. $a \rightarrow P$, then in the conjoined model CSPc a is a non-empty set of conjoined events rather than a single event. Thus the trace $\langle a\rangle$ is a conjoined trace with a sequence (here, just one) of sets, and more generally, CSPc traces are a sequence of non-empty sets of conjoined events, rather than CSP's sequence of events.

5.9.2 Parallel

Roscoe [1997] gives the rules for the parallel operator on page 70. The traces of CSP's generalised parallel composition are defined using a parallel operator on traces:

$$traces(P \underset{X}{||} Q) = \bigcup \{s \underset{X}{||} t \mid s \in traces(P) \wedge t \in traces(Q)\}$$

This parallel operator on traces is defined as follows, where x and x' are members of X, while y and y' are any events not in X:

$$s \underset{X}{||} t = t \underset{X}{||} s$$

$$\langle\rangle \underset{X}{||} \langle\rangle = \{\langle\rangle\}$$

$$\langle\rangle \underset{X}{||} \langle x\rangle = \{\}$$

$$\langle\rangle \underset{X}{||} \langle y\rangle = \{\langle y\rangle\}^3$$

$$\langle x\rangle \hat{} s \underset{X}{||} \langle y\rangle \hat{} t = \{\langle y\rangle \hat{} u \mid u \in (\langle x\rangle \hat{} s) \underset{X}{||} t\}$$

$$\langle x\rangle \hat{} s \underset{X}{||} \langle x\rangle \hat{} t = \{\langle x\rangle \hat{} u \mid u \in s \underset{X}{||} t\}$$

$$\langle x\rangle \hat{} s \underset{X}{||} \langle x'\rangle \hat{} t = \{\} \text{ if } x \neq x'$$

$$\langle y\rangle \hat{} s \underset{X}{||} \langle y'\rangle \hat{} t = \{\langle y\rangle \hat{} u \mid u \in s \underset{X}{||} (\langle y'\rangle \hat{} t)\} \cup \{\langle y'\rangle \hat{} u \mid u \in (\langle y\rangle \hat{} s) \underset{X}{||} t\}$$

The latter four cases deal respectively with the following situations:

- One side offers an event in the synchronisation set and must wait; the other side offers an event outside the synchronisation set and may proceed.
- Both sides offer the same event from the synchronisation set, and thus synchronise on it.
- The two sides offer different events from the synchronisation set, and thus deadlock (no further traces possible).
- The two sides offer different events from outside the synchronisation set, and thus either event may occur first, followed by the corresponding further traces.

We must adjust the definition of the operator to the following, where a, b etc are conjoined sets of events (single non-conjoined events are simply singleton sets):

[3] We believe there is a small mistake here. We think it should read:
$\langle\rangle \underset{X}{||} \langle y\rangle \hat{} t = \{\langle y\rangle \hat{} u \mid u \in (\langle\rangle \underset{X}{||} t)\}$, and similarly the line above should read: $\langle\rangle \underset{X}{||} \langle x\rangle \hat{} t = \{\}$.

$$
\begin{aligned}
a \underset{X}{||} b &= b \underset{X}{||} a \\
\langle\rangle \underset{X}{||} \langle\rangle &= \{\langle\rangle\} \\
\langle\rangle \underset{X}{||} \langle a\rangle^\frown t &= \{\} \text{ if } a \cap X \neq \emptyset \\
\langle\rangle \underset{X}{||} \langle a\rangle^\frown t &= \{\langle a\rangle^\frown u \mid u \in (\langle\rangle \underset{X}{||} t)\} \text{ if } a \cap X = \emptyset \\
\langle a\rangle^\frown s \underset{X}{||} \langle b\rangle^\frown t &= \{\langle b\rangle^\frown u \mid u \in (\langle a\rangle^\frown s) \underset{X}{||} t\} \text{ if } a \cap X \neq \emptyset \wedge b \cap X = \emptyset \\
\langle a\rangle^\frown s \underset{X}{||} \langle b\rangle^\frown t &= \{\} \text{ if } a \cap X \neq \emptyset \wedge b \cap X \neq \emptyset \wedge (a \cup b) \cap X \neq a \cap b \\
\langle a\rangle^\frown s \underset{X}{||} \langle b\rangle^\frown t &= \{\langle a \cup b\rangle^\frown u \mid u \in s \underset{X}{||} t\} \text{ if } (a \cup b) \cap X = a \cap b \wedge a \cap b \neq \emptyset \\
\langle a\rangle^\frown s \underset{X}{||} \langle b\rangle^\frown t &= \{\langle a\rangle^\frown u \mid u \in s \underset{X}{||} (\langle b\rangle^\frown t)\} \cup \{\langle b\rangle^\frown u \mid u \in (\langle a\rangle^\frown s) \underset{X}{||} t\} \\
&\qquad \text{if } a \cap X = \emptyset \wedge b \cap X = \emptyset
\end{aligned}
$$

The latter four cases deal respectively with the following situations:

- One side offers a conjunction that overlaps with the synchronisation and must wait; the other side offers a conjunction that is entirely outside the synchronisation set and may proceed.
- The two sides offer conjunctions that both overlap with the synchronisation set, and their intersection is anything other than the parts of the two that are in the synchronisation (i.e. the intersection is either less, in which case one side is not offering a synchronisation event necessary to the other side, or it is more and both sides are offering an event outside the synchronisation set – they are interleaving on it, so only one may complete at any one time, not both). Neither side may proceed and there is deadlock.
- The two sides offer a conjunction where they overlap, but the common events are exactly the part of either that is in the synchronisation set. The two sides thus synchronise on the union of their offerings.
- The two sides offer non-overlapping conjunctions that are both entirely outside the synchronisation set. They may proceed in either order.

5.9.3 Hiding

Roscoe [1997] provides a simple definition of the effect of hiding on traces:

$$traces(P \setminus X) = \{s \upharpoonright (\Sigma \setminus X) \mid s \in traces(P)\}$$

We can re-use the same definition, providing we re-define the trace-restriction operator, $\upharpoonright$, as follows:

$$\begin{aligned} \langle\rangle \upharpoonright A &= \langle\rangle \\ \langle s\rangle^\frown t \upharpoonright A &= t \text{ if } s \subseteq A \\ \langle s\rangle^\frown t \upharpoonright A &= (s \setminus A)^\frown t \text{ if } s \not\subseteq A \end{aligned}$$

That is, hiding an event removes it from all the conjoined events in the trace as would be expected. The event only disappears from the trace if all events in the conjunction are hidden.

5.9.4 Example

We give a short example of our trace rules. Consider the system:

$$\begin{aligned} P &= a \wedge h \rightarrow \text{SKIP} \\ Q &= h \wedge b \rightarrow \text{SKIP} \\ R &= (P \underset{\{h\}}{||} Q) \setminus \{h\} \end{aligned}$$

The process P conjoins a with a hidden event h. The process Q conjoins b with the same hidden event h and they synchronise on it. The effect should be to synchronise a and b. The traces of P and Q are:

$$traces(P) = \{\langle\{a, h\}\rangle\}$$
$$traces(Q) = \{\langle\{h, b\}\rangle\}$$

The traces of R can be evaluated using our rules:

$$traces(R) = traces((P \underset{\{h\}}{||} Q) \setminus \{h\})$$
$$traces(R) = traces(P \underset{\{h\}}{||} Q) \upharpoonright (\Sigma \setminus \{h\})$$
$$traces(R) = traces(P \underset{\{h\}}{||} Q) \upharpoonright \{a, b\}$$
$$traces(R) = \{(\langle\{a, h\}\rangle \underset{\{h\}}{||} \langle\{h, b\}\rangle) \upharpoonright \{a, b\}\}$$
$$traces(R) = \{\langle\{a, h, b\}\rangle \upharpoonright \{a, b\}\}$$
$$traces(R) = \{\langle\{a, b\}\rangle\}$$

We can see that the effect is as we expected.

5.10 Conclusions

This chapter has introduced a conjunction operator that allows two events to be synchronised on together, so that each event can only occur together with the other. These conjunctions can be combined with a standard choice to permit choice between conjunctions. The algorithm has been described, including extensions to support partial events and/or priority between events. Furthermore, preliminary work on augmenting the CSP traces model to support conjunction has been described.

Conjunction allows new design patterns to be expressed, and for existing systems (such as the dining philosophers [Hoare, 1985] and blood platelet model [Schneider et al., 2006; Welch et al., 2006b]) to be expressed more simply. Conjunction is not specific to CHP (although Haskell's support for STM makes it easy to implement) and may be implemented in any other process-oriented library or language.

One benefit of the CSP programming model is that it can easily be transferred to distributed systems. Channels can easily be "stretched" over the network to join two separate machines without any modification to the original program. There is no obvious way to implement conjunction in a distributed system – this is reserved for future work.

Chapter 6

Tracing

A trace is a record of a single execution of a program. For example, a CSP trace consists of a single ordered sequence of events. A trace is useful for examining the behaviour of a process for the purposes of debugging, understanding or detecting aberrant behaviour.

Concurrent programs can be hard to debug because the timing can introduce non-determinism. For example, a program may have a bug where it deadlocks if two processes being run in parallel, $a \rightarrow b \rightarrow$ SKIP and $c \rightarrow d \rightarrow$ SKIP, execute in the order $\langle a, c, d, ... \rangle$. The concurrent aspect means that the relative scheduling may be non-deterministic (leading to a bug that manifests sometimes, but not always) and also that it can be hard to determine which order actually occurred. Adding logging (through the use of primed events) could give $a \rightarrow a' \rightarrow b \rightarrow b' \rightarrow$ SKIP and $c \rightarrow c' \rightarrow d \rightarrow d' \rightarrow$ SKIP, but a valid execution ordering would then be $\langle a, c, c', d, d', ... \rangle$ which would produce the deadlock bug, but before the a' logging event had fired.

In this chapter we detail the addition of built-in tracing to the CHP library. The event synchronisations and their recording in the trace are atomic, so that the trace is the true record of the program's behaviour. The tracing is easy to enable or disable, and requires no extra annotations in the user's program (although they can supply annotations to name events). The work in this chapter was previously published in Communicating Process Architectures 2008 and 2009 [Brown and Smith, 2008, 2009].

6.1 Background

Tracing is the recording of a program's behaviour. Specifically, a trace is a record of all the events that a process has engaged in. A trace of an entire program is typically a sequential interleaving of all the events that occurred during the run-time of the program. In this section we briefly recap CSP and VCR tracing, and introduce a new form of trace that will be used in this chapter: structural tracing.

For our trace examples in this section, we will use the following CSP system:

$$(a \rightarrow b \rightarrow c \rightarrow \text{STOP}) \underset{\{a\}}{\parallel} (a \rightarrow d \rightarrow e \rightarrow \text{STOP})$$

6.1.1 CSP Tracing

A CSP trace is a sequential record of all the events that occurred during the course of the program. Hoare originally described the trace as being recorded by a perfect observer who saw all events [Hoare, 1985]. If the observer saw two (or more) events happening simultaneously, they were permitted to write down the events in an arbitrary order.

For example, these are the possible maximal traces of our system:

$$\langle a, b, c, d, e\rangle \quad \langle a, b, d, c, e\rangle \quad \langle a, b, d, e, c\rangle$$
$$\langle a, d, b, c, e\rangle \quad \langle a, d, b, e, c\rangle \quad \langle a, d, e, b, c\rangle$$

6.1.2 VCR Tracing

A VCR trace is similar to a CSP trace, but instead of recording a sequence of single events in a trace, the observer records a sequence of multisets [Smith, 2000, 2004]. In the original model of VCR traces, each multiset represented a collection of simultaneous events, preserving information that the CSP observer had lost. In practice, simultaneity is a difficult concept to define, reason about or observe.

We have therefore adapted the meaning of event multiset traces in VCR. In our implementation in CHP each multiset is a collection of *independent* events. We term a and b to be independent events if a did not observably require b to occur first, and vice versa. The implication is that it was possible for events a and b to occur in either order without altering the meaning of the program.

By convention in this chapter, we write multisets of size one without set notation. These are the possible maximal traces of our system:

$$\langle a, b, c, d, e\rangle \quad \langle a, b, d, c, e\rangle \quad \langle a, b, d, e, c\rangle$$
$$\langle a, d, b, c, e\rangle \quad \langle a, d, b, e, c\rangle \quad \langle a, d, e, b, c\rangle$$
$$\langle a, b, \{c, d\}, e\rangle \quad \langle a, b, d, \{c, e\}\rangle$$
$$\langle a, d, b, \{c, e\}\rangle \quad \langle a, d, \{b, e\}, c\rangle$$
$$\langle a, \{b, d\}, \{c, e\}\rangle$$

VCR also permits multiple observers and imperfect observation [Smith, 2004] – the idea that some events may not be recorded – but unless specifically stated, in this chapter we will be using a single observer that records all events.

6.1.3 Structural Tracing

To explore other forms of trace recording, we have created structural tracing. CSP and VCR tracing are both based on recording the events in a single, fairly flat trace. Structural tracing is a constrasting approach where a structure is built up that reflects the parallel and sequential composition of the processes being traced. Traces can be composed in parallel with the || operator, and these parallel-composed traces may appear inside normal sequential traces.

There is only one possible maximal trace of our system:

$$\langle a, b, c\rangle \,||\, \langle a, d, e\rangle$$

Note that a single occurrence of an event is recorded multiple times (once by each process engaging it), which is distinct from both CSP and VCR tracing. This makes structural traces quite different from the classic notion of tracing.

6.2 Implementation

The CHP library has two primary types of synchronisations: channel communications (that transmit data) and barrier synchronisations (that have no data). This is also true for most CSP-derived languages and libraries. We have altered CHP so that after the completion of every successful event, the event is recorded. For

CSP and VCR this is done by the process that completes the synchronisation. For structural traces, every process that engages in the event records it. This highlights the fundamental difference between structural tracing and the other two forms: under structural tracing, events are recorded multiple times, but under CSP and VCR tracing events are only recorded once.

Tracing can be enabled or disabled at the start of the program's execution. Switching on tracing is as simple as changing a single line in the user's program – this makes the tracing facility very easy to use.

6.2.1 CSP Traces

CSP tracing is the most straightforward to implement. Hoare's perfect observer can be implemented by modifying a single sequence shared between all processes – with appropriate mechanisms, such as a mutex, to handle the concurrent updates.

Recording Bottleneck

The main problem with CSP tracing is that it serialises the whole program. All processes end up contending for access to the central trace, and thus with multiple threads this can slow down the program. An alternative approach would be to have each process record its own trace with events timestamped, and merge these sorted traces by timestamp from sub-processes into their parent process when the sub-processes complete.

The problem with such timestamps is that modern computers (from the point of view of a portable high-level language) often have timers with relatively coarse frequency, such as once per millisecond. CHP is able to complete around a hundred events in that time, and thus events that occur within a millisecond of each other would be recorded in arbitrary order. Therefore CSP traces are still recorded using a single shared sequence.

6.2.2 VCR Traces

As described in section 6.1.2, we group independent events into multisets for VCR traces. If we can deduce a definite ordering of two events, we take the latter event to be dependent on the former. Otherwise, we take the events to be independent. For example, we know that two events must be sequential if they are both performed by the same process-thread. To be able to reason about this and other details, we record events with associated process identifiers.

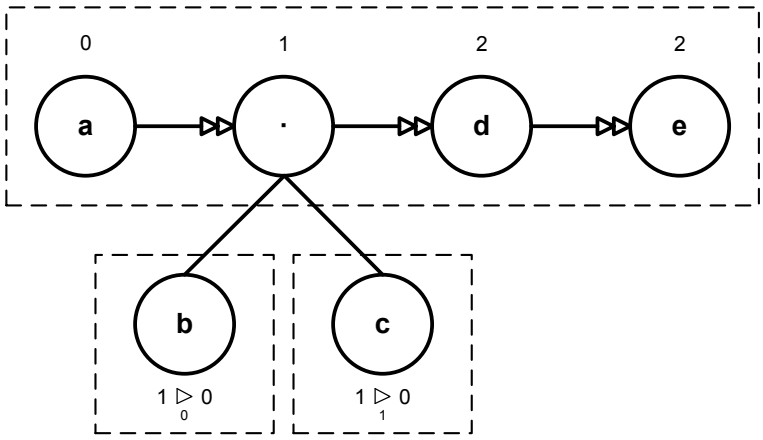


Figure 31: The program $(a \rightarrow (b \rightarrow \text{SKIP} \mid\mid\mid c \rightarrow \text{SKIP})) \underset{\S}{;} (d \rightarrow e \rightarrow \text{STOP})$, with the identifiers for each event.

Process Identifiers

We use process identifiers to deduce information about sequencing. Consider these two CSP systems:

$$(a \rightarrow (b \rightarrow \text{SKIP} \mid\mid\mid c \rightarrow \text{SKIP})) \underset{\S}{;} (d \rightarrow e \rightarrow \text{STOP})$$
$$(f \rightarrow \text{SKIP} \mid\mid\mid g \rightarrow \text{SKIP}) \underset{\S}{;} (h \rightarrow \text{STOP} \mid\mid\mid i \rightarrow \text{STOP})$$

We wish to be able to deduce the following facts, where $<$ is the strict partial ordering of the events in time:

$$a < b, a < c$$
$$b < d, c < d$$
$$d < e$$
$$f < h, g < h$$
$$f < i, g < i$$

Process identifiers are one or more integers (sequence numbers) joined together by a sub-process operator, $\underset{x}{\triangleright}$ where x is itself an integer (a parallel branch identifier). The top-level process of the program starts with the identifier 0. Process identifiers only change when the process runs a parallel composition. Before and after the composition, the last (right-most) sequence number is incremented. The identifiers for the sub-processes are formed by appending $\underset{x}{\triangleright}0$ to the current identifier, where x is distinct for each sub-process.

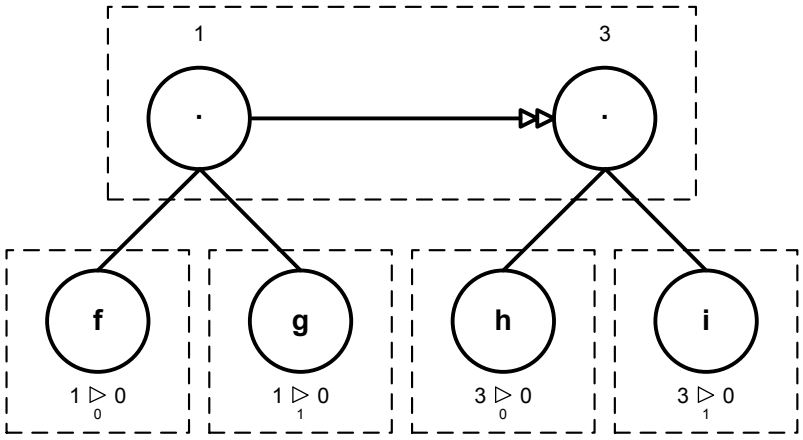


Figure 32: The program $(f \rightarrow \text{SKIP} \;|||\; g \rightarrow \text{SKIP}) \fatsemi (h \rightarrow \text{STOP} \;|||\; i \rightarrow \text{STOP})$, with the identifiers for each event.

Our example programs, with process identifiers, are shown in figures 31 and 32. Each event is given in a circle. The open double-headed arrows indicate sequencing, and sub-processes are shown in a vertical relation, with an empty (dot) event as a parent. The events are grouped into dashed boxes, which represent the actual processes that will be created when the program is run. Next to each event is the associated process identifier that will be recorded with it.

The sequence numbers in the top row of each figure are not in error. Events d and e are associated with the same sequence number. Event e will definitely be recorded after event d (since the same process is doing them in sequence) so we do not need to use the process identifier to tell them apart. It is only in the case of parallel composition that we must change the sequence numbers, in order to deduce an ordering between the parent's events (such as a and d) and the sub-processes' events (b and c). The incrementing before and after both parallel compositions is what leads to the sequence numbers being 1 and 3 for the second figure.

The following Haskell code explains the algorithm for comparing process identifiers:

```
data ProcessId = PID Integer (Maybe (Integer, ProcessId))

lessThan :: ProcessId -> ProcessId -> Bool
lessThan (PID x Nothing) (PID y Nothing) = x < y
lessThan (PID x (Just (xpar, xpid))) (PID y (Just (ypar, ypid)))
  = if x == y
      then
```

```
    (if xpar == ypar
       then lessThan xpid ypid
       else False)
else x < y
```

A process identifier has sequence number, and an optional (indicated by the Maybe type) parallel branch identifier with the further part of the process identifier. So the identifier $2 \underset{1}{\triangleright} 4$ would be represented as PID 2 (Just (1, (PID 4 Nothing))).

If the process identifiers are both single sequence numbers, we simply compare these sequence numbers. If they are a compound identifier, we again start by comparing the sequence numbers. If they are not equal, we return the value of x < y. If however they are equal, we compare their parallel branch identifiers. If these identifiers are not equal, we know the identifiers come from parallel siblings, and we return false. If they are equal, we proceed with comparing the remainder of the process identifiers.

Due to the incrementing of the sequence number before and after running sub-processes, no recorded identifier will ever be an exact prefix of another. In our diagrams (figures 31 and 32), the parent with the same prefix is represented as a dot, and engages in no events itself. Thus, pairs of recorded identifiers will always differ within the length of the shorter identifier, or both identifiers will be equal – hence there is no need for a case in the function above where one process identifier has a Just sub-process identifier and the other has a Nothing value.

Recording Rules

In the previous section we explained our process identifiers and a partial order over them. We will now show how we use this to record VCR traces, for the moment considering events that only involve one process.

When recording an event in a VCR trace, a process must look at the most recent set of parallel events, here termed Θ. Θ is a set[1] of pairs of an event (for which we use α, β, etc) and a process identifier (for which we use p, q, etc). We must determine, for a given (β, q) whether to add the new event to Θ or whether to start a new set of parallel events. Our rule is straightforward: if $\exists(\alpha, p) \in \Theta : p < q \vee p = q$, we must start a new set of parallel events because an event in the most recent set provably occurred before the event we are adding. Otherwise, add the new event to Θ (the events are independent).

[1] Technically in VCR, it is a multiset, but due to our recording strategy, no pair of event and process identifier will occur multiple times in the same set.

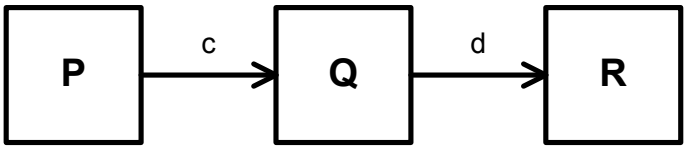


Figure 33: A standard pipeline of three processes.

Consider the following CSP traces of our example processes from figures 31 and 32, annotated with process identifiers:

$$\langle a[0], c[1 \underset{1}{\triangleright} 0], b[1 \underset{0}{\triangleright} 0], d[2], e[2] \rangle$$
$$\langle f[1 \underset{0}{\triangleright} 0], g[1 \underset{1}{\triangleright} 0], i[3 \underset{1}{\triangleright} 0], h[3 \underset{0}{\triangleright} 0] \rangle$$

Following our rules, the same traces recorded in VCR form would be:

$$\langle a, \{b, c\}, d, e \rangle$$
$$\langle \{f, g\}, \{h, i\} \rangle$$

Multiple Processes

We have so far considered events with just a single process – however, events usually involve multiple processes synchronising. Consider a process pipeline, where three processes are connected with channels c and d, as shown in figure 33. We will assume here that the reader happens to record the events (recall that events are recorded by the last process to engage in the synchronisation). Process Q will read from channel c, and record this event. Then it will write to channel d and the process R will read and record the event. Because the process identifiers for Q and R have no ordering, the events will be recorded as being independent. However, the communication on channel d was clearly dependent on the communication on from channel c, since process Q communicated on channel c before communicating on channel d.

In order to solve this problem, we modify our synchronisation events so that the party that records the event can know the process identifier of all the other processes that engaged in the synchronisation. The event is recorded with a set (in barriers, there may be more than two participants) of process identifiers rather than a single identifier.

Thus we can adjust our previous rule, now that Θ is a set of pairs of single event identifier and a *set* of process identifiers (for which we will use capital P,

Q, etc). For adding a new pair of event and identifier set (β, Q): if $\exists(\alpha, P) \in \Theta, \exists\, p \in P, \exists\, q \in Q : p < q \vee p = q$, start a new set of parallel events. Otherwise, add the new event to Θ.

In our previous example, the communication on channel c will have been recorded with the process identifiers for processes P and Q. When process R then records the communication on channel d with the identifiers for Q and R, the events will be seen to be dependent.

The reader may wonder what happens should it also be the case that $\exists\, p \in P, \exists\, q \in Q : q < p$ in the above rule. This is not possible in the CHP library because the recording of an event is bound into the synchronisation, so events will always be recorded in order. It cannot be the case that an event a that provably occurred *before* b will be recorded *after* b.

Relevance of Event Identifiers

The event identifier currently plays no role in our rules – it is usually subsumed by considering the identifiers of processes engaging in the events. Therefore the only cases where it would be of use are where an entirely different (unrelated) set of processes may engage on two successive occurrences of the same event.

Communication on unshared channels, and channels with only one shared end, are taken care of by considering the processes involved. In these cases, one process will always be involved[2] in the subsequent communications, and we can use this to deduce sequence information. Therefore we need only consider here channels that are shared at both ends.

With channels that are shared at both ends, it is possible for P and Q to communicate once on the channel, then release the ends, after which R and S claim them and perform another communication. In such a situation, we do not consider the second communication to be dependent on the first, and so we do not use the event identifier to deduce any sequence information.

Independence and Inference

The VCR theory describes events in the same multiset as being observed simultaneously [Smith, 2004], but we have altered this and implemented our traces to record independent events in the same multiset. This means that some of the

[2]Or a process and its sub-processes, that we can deduce sequence information about.

theoretically possible VCR traces can never actually be recorded in our implementation, and also that some sequence information can be retroactively inferred.

Recall our CSP system from the example in section 6.1:

$$(a \to b \to c \to \text{STOP}) \underset{\{a\}}{\parallel} (a \to d \to e \to \text{STOP})$$

Imagine that the first two events to occur are a and b, giving us a partial trace: $\langle a, b \rangle$. According to the VCR theory, the possible maximal traces that may follow from this are:

$$\langle a, b, c, d, e \rangle \quad \langle a, b, d, c, e \rangle \quad \langle a, b, d, e, c \rangle$$
$$\langle a, b, \{c, d\}, e \rangle \quad \langle a, b, d, \{c, e\} \rangle \quad \langle a, \{b, d\}, \{c, e\} \rangle$$

Which trace is recorded will depend on which event occurs next. If the next event is c, the theoretical traces are:

$$\langle a, b, c, d, e \rangle \quad \langle a, b, \{c, d\}, e \rangle$$

In fact, the trace recorded in our implementation will always be the second trace. Because c and d are independent events, they will always be recorded in the same multiset. If c happens first, the partial trace will be $\langle a, b, c \rangle$. But then when d is recorded, because it is independent of all events in the most recent multiset (c, in a singleton multiset), it will be added to that multiset, forming $\langle a, b, \{c, d\} \rangle$, then become the second trace shown above. The trace $\langle a, b, c, d, e \rangle$ can never actually be recorded using our implementation.

If instead the next (third) event is d, the possible theoretical traces are:

$$\langle a, b, d, c, e \rangle \quad \langle a, b, d, e, c \rangle \quad \langle a, b, d, \{c, e\} \rangle$$
$$\langle a, b, \{c, d\}, e \rangle \quad \langle a, \{b, d\}, \{c, e\} \rangle$$

For similar reasons to the previous example, it is not possible to record any of the top three traces. The pairs of events $(b,\ d)$, $(c,\ d)$ and $(c,\ e)$ are each independent, so the traces would always be recorded using the latter two traces (with appropriate multisets) rather than the earlier three traces with only singleton

multisets.

It is even possible to infer sequence from a trace with multisets. Consider the trace $\langle a, b, \{c, d\}, e\rangle$. If event d had occurred before c, then b and d would have been grouped into the same multiset (since they are independent). Therefore to produce this trace, c must have happened before d.

These issues reflect the difference between the theory of VCR and our actual implementation of its tracing. Our trace may seem to add obfuscation over and above the CSP trace. But it also abstracts away from some of the unnecessary detail. Grouping two events into a multiset implies that they *could* have happened in either order. For trace compression (see section 6.2.5), comprehension and visualisation (see section 6.5), a more regular trace that abstracts away from some of the scheduling-dependent behaviour of the program may well be appealing in some circumstances.

6.2.3 Trace Recording in Software Transactional Memory

Both CSP and VCR traces must be recorded centrally, that is with one trace for the whole system. In a concurrent program, many processes will add events to the trace. Thus we require a mechanism to ensure that all concurrent updates will be safe and not result in an incorrect trace (for example, by events being overwritten by another process). An obvious mechanism to protect the central traces would be to use a lock or mutex.

The event synchronisations take place as part of a Software Transactional Memory (STM) transaction (see chapter 5 for details). We can make the recording indivisible from the event synchronisation, while also protecting the trace, by making the trace a transactional variable, and updating it in the same transaction as the completion of the event synchronisation.

6.2.4 Structural Traces

A structural trace is the most natural – and fastest – to record. Each running process records events it has engaged in using a local trace. Since the trace is unshared, there is no contention or need for locks, and hence it is faster than any other method of recording. Sequential events are recorded by adding them to a list.

When a process runs several sub-processes in parallel, they all also separately record their own trace. Upon completion they all send back their traces to the

parent process. These traces are received by the parent and form a hierarchy in the parent's trace. This process of joining traces means that the end result is one single trace that represents the behaviour of the entire program.

The drawback with structural traces is that information about the ordering of events between non-descendant processes is lost. Consider the trace:

$$\langle a, a, a, a, a, a \rangle \;||\; \langle b, b, b, b, b, b \rangle$$

We cannot tell from this trace whether all the events a happened before all the events b, vice versa, a strict interleaving or any other possible pattern.

6.2.5 Compression

One problem of recording the trace of a system is that a large amount of data is generated. If the system may generate an event every microsecond[3], that is a million events a second. Assuming on a 32-bit machine that an event could be reduced to eight bytes per event (four bytes for an identifier, four for a pointer in a linked list or similar), in the worst case nearly a gigabyte of data would be generated approximately every two minutes. Generally programs are likely to fall far below this upper bound, but ideally we would like to reduce the space required.

Processes usually iterate (by looping or recursion), and thus display repetitive behaviour. This repetition is often diluted by observing the behaviour of several processes, but even large process networks can display regular behaviour.

Repeated patterns appearing in the trace means that it should be possible to compress the trace by removing this redundancy. To gain much benefit from this, we need to compress the trace on-line, while the program is still running, rather than off-line after the program has finished.

The obvious compression approaches are forms of run-length encoding that can reduce consecutive repeated behaviour, and dictionary-based compression methods that can spot common patterns and reduce them to an index in a lookup table of frequently occurring patterns.

We also note that viewing the compressed version is often more comprehensible to the programmer than its raw, uncompressed form. Most of the structural traces later in this chapter have been left in their compressed form for that reason.

[3]This figure is a very rough average of channel communication times on modern machines from various CSP implementations.

6.2.6 Rolling Trace

A primary use of traces is for post-mortem debugging, especially in the case of deadlock or livelock. In this case the programmer is interested to see what the program was doing leading up to the failure.

This failure could occur after the program has been running for hours, or days. The behaviour of the program much earlier will probably not be of interest. Therefore a good policy in these cases would be to keep a rolling trace which records, say, the most recent thousand events. This would require a constant amount of memory rather than accumulating the trace as normal (recording using an array, rather than a linked list).

This would allow tracing to place some time overhead on the program, but only a small amount of memory overhead, and would still be useful in the case of program failure to see the events immediately before the failure, but not those that occurred long before.

6.3 Example Traces

In this section we present examples of traces of several different small programs. Each trace is recorded from a different run of the program, so the traces will not (necessarily) be direct transformations of each other.

6.3.1 CommsTime

Commstime is a classic benchmark used in process-oriented systems. Its configuration is shown in figure 34. For this benchmark, we show approximately six iterations of the commstime loop.

CSP

⟨ prefix-delta, delta-recorder, delta-succ, succ-prefix,
prefix-delta, delta-succ, delta-recorder, succ-prefix,
prefix-delta, delta-succ, delta-recorder, succ-prefix,
prefix-delta, delta-succ, delta-recorder, succ-prefix,
prefix-delta, delta-succ, delta-recorder, succ-prefix,
prefix-delta, delta-succ, succ-prefix, delta-recorder,
delta-succ ⟩

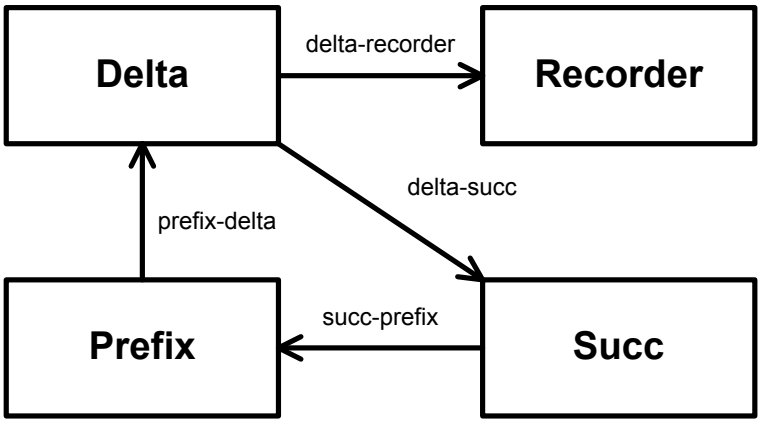


Figure 34: The CommsTime Network.

VCR

⟨ {prefix-delta}, {delta-succ}, {delta-recorder, succ-prefix},
{prefix-delta}, {delta-succ}, {delta-recorder, succ-prefix},
{prefix-delta}, {delta-recorder, delta-succ}, {succ-prefix},
{prefix-delta}, {delta-recorder, delta-succ}, {succ-prefix},
{prefix-delta}, {delta-recorder, delta-succ}, {succ-prefix},
{prefix-delta}, {delta-recorder, delta-succ}, {succ-prefix}, {delta-succ} ⟩

Structural

⟨⟨⟨7*⟨delta-succ? , succ-prefix!⟩, delta-succ?⟩
|| ⟨7*⟨prefix-delta! , succ-prefix?⟩ , prefix-delta!⟩
|| ⟨6*⟨delta-recorder?⟩⟩
|| ⟨7*⟨prefix-delta? , ⟨delta-recorder! || delta-succ!⟩⟩⟩⟩⟩

Summary

The CSP trace reflects the interleaving that occurred between all the parallel processes, and shows how the order of events changes slightly at the beginning and end of the trace.

The lines in the middle of the VCR trace are probably what we would expect; singleton sets of prefix-delta and succ-prefix, and a set of the two parallel events from the delta process: delta-recorder and delta-succ. However, the early traces show that there is a different recording that can occur, with the delta-succ event happening first, and the delta-recorder happening in parallel with succ-prefix. This is a valid behaviour of our process network.

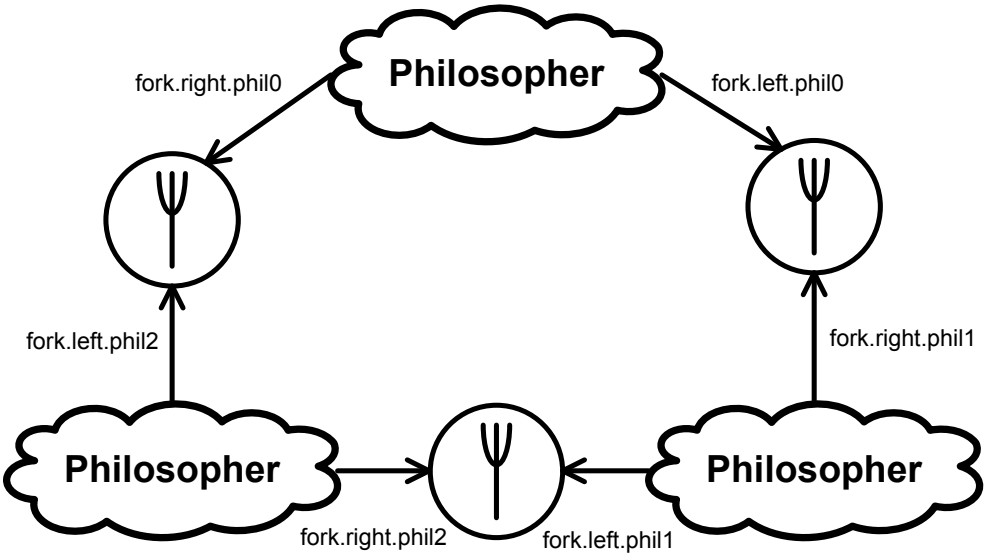


Figure 35: Dining Philosophers with Three Philosophers.

It is easy to see the four different processes in the structural trace, each with distinct behaviour. This would be visible even if the channels were not labelled. The regular repetition of all of the processes is also clear.

6.3.2 Dining Philosophers

The dining philosophers is a classic concurrency problem described by Hoare in his original book on CSP [Hoare, 1985]. We use the deadlocking version for our benchmark. To keep the traces simpler and to provoke deadlock more easily, we use only three philosophers. The fork-claiming channels are named according to the philosopher they are connected to – the names of all of the channels are shown in figure 35. We show only the tail-end of the traces (leading up to the deadlock) as the full traces are too long to display here.

CSP

⟨ . . . , fork.right.phil1, fork.left.phil0, fork.left.phil2, fork.right.phil2, fork.right.phil0, fork.left.phil1, fork.right.phil0, fork.left.phil0, fork.right.phil1, fork.left.phil2, fork.left.phil1, fork.right.phil2, fork.right.phil1, fork.left.phil0, fork.left.phil2, fork.right.phil0, fork.right.phil2, fork.left.phil1, fork.left.phil0, fork.right.phil0, fork.right.phil1, fork.left.phil2, fork.left.phil1, fork.right.phil1, fork.right.phil2, fork.left.phil0, fork.left.phil2, fork.right.phil2, fork.right.phil0, fork.left.phil1, fork.right.phil0, fork.left.phil0, fork.left.phil2, fork.left.phil0 ⟩

VCR

⟨ . . . , {fork.left.phil2}, {fork.right.phil0, fork.right.phil2}, {fork.left.phil0, fork.right.phil0, fork.left.phil1}, {fork.right.phil1, fork.left.phil2}, {fork.left.phil1, fork.right.phil1}, {fork.left.phil0, fork.right.phil2}, {fork.left.phil2, fork.right.phil2}, {fork.right.phil0, fork.left.phil1}, {fork.left.phil0, fork.right.phil0}, {fork.right.phil1, fork.left.phil2}, {fork.left.phil1, fork.right.phil1}, {fork.left.phil0, fork.right.phil2}, {fork.left.phil2, fork.right.phil2}, {fork.right.phil0, fork.left.phil1}, {fork.left.phil0}, {fork.right.phil0, fork.right.phil1}, {fork.left.phil1, fork.right.phil1}, {fork.left.phil0, fork.left.phil1, fork.left.phil2} ⟩

Structural

⟨ ⟨ 18 * ⟨ fork.left.phil0 || fork.right.phil0 ⟩, fork.left.phil0 ⟩
|| ⟨ 20 * ⟨ fork.left.phil1 || fork.right.phil1 ⟩, fork.left.phil1 ⟩
|| ⟨ 14 * ⟨ fork.left.phil2 || fork.right.phil2 ⟩, fork.left.phil2 ⟩ ⟩

Summary

The dining philosophers problem is an example of a larger process network with less regular behaviour. Another problem is that it is hard to identify which messages are "pick up" messages and which are "put down" messages. An automated tool for dealing with traces could easily fix this by labelling alternating messages on the same channel differently. At the end of each trace it can be seen (most visibly in the VCR and structural traces) that each philosopher communicated to its left-hand fork but not its right-hand fork, immediately before the deadlock.

6.3.3 Token Cell Ring

This example is a token-passing ring using barriers. The process network is a ring of cells, where each cell is enrolled on two barriers (here termed "before" and "after") and has a channel-end from a "tick" process. A cell can have two states: full or empty.

If the cell is full, it offers to either engage on its "after" barrier (to pass the token on) or read from its "tick" channel. If the barrier is the event chosen, the cell then commits to read from its "tick" channel. If the cell is empty, it offers to either engage on its "before" barrier (to receive a token) or read from its "tick" channel. Again, if the barrier is chosen, it then commits to reading from the "tick" channel.

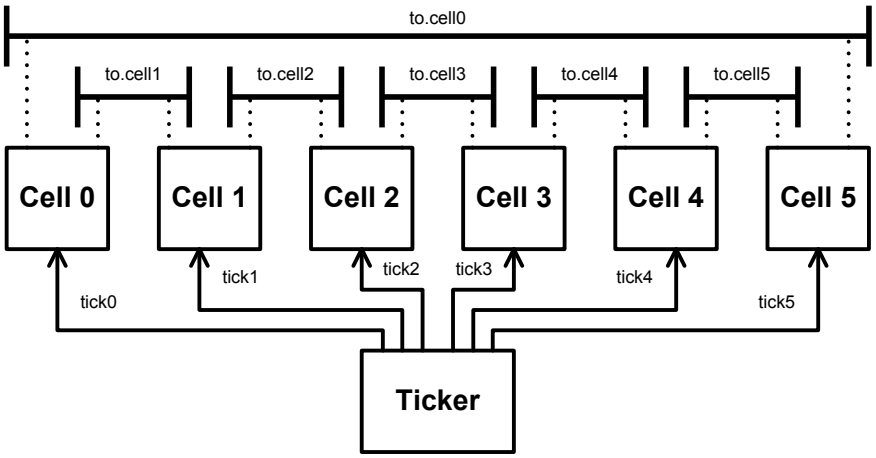


Figure 36: Token Cell Ring Network.

The writing ends of all the tick channels are connected to a "ticker" process that writes in parallel to all its channels, then repeats. Six cells are wired in a ring, and initially the first (index: 0) is full, with the rest empty. All the barriers are named according to the cell following it – that is, the barrier that is "after" for cell 2 and "before" for cell 3 is named "to.cell3".

This network is portrayed in figure 36. The idea is a simpler version of the model used for blood clotting in the TUNA project [Welch et al., 2006a], and featured in chapter 5 of this thesis. Eight iterations (i.e. eight ticks) of the network were run.

CSP

⟨ to.cell1, tick0, tick1, tick3, tick4, to.cell2, tick2, tick5,
tick2, tick0, tick1, tick3, tick5, to.cell3, to.cell4, tick4,
to.cell5, tick0, tick2, tick5, tick1, tick3, to.cell0, tick4,
tick0, tick2, tick3, tick5, to.cell1, tick1, to.cell2, tick4,
tick1, tick2, to.cell3, tick4, tick0, tick3, to.cell4, tick5,
tick2, tick3, tick4, tick5, tick0, to.cell5, tick1,
tick0, tick2, tick3, tick5, to.cell0, tick1, tick4,
tick1, tick2, tick4, tick0, tick3, to.cell1, tick5,
tick0, tick1, tick2, tick3, tick4, tick5 ⟩

VCR

⟨ {tick1, tick2, tick4, tick5, to.cell1}, {tick0, tick3, to.cell2},
{tick0, tick1, tick2, tick3, tick4, tick5},

{tick1, tick2, to.cell3}, {tick0, tick3, tick4},
{tick2, tick3, tick5, to.cell4}, {tick0, tick1, tick4, tick5},
{tick2, tick3, tick4, to.cell5}, {tick0, tick1, tick5},
{tick0, tick1, tick2, to.cell0}, {tick3, tick4, tick5, to.cell1},
{tick0, tick1}, {tick2, tick3, tick4, tick5, to.cell2},
{tick0, tick1, tick3, to.cell3}, {tick4, to.cell4}, {tick2, tick5, to.cell5},
{to.cell0}, {tick0}, {tick1}, {tick2}, {tick3}, {tick5}, {tick4} ⟩

Structural

⟨⟨⟨to.cell1* , 4*⟨tick0?⟩ , to.cell0* , tick0? , to.cell1* , 2*⟨tick0?⟩ , to.cell0* , tick0? , to.cell1* , tick0?⟩
|| ⟨to.cell1* , tick1? , to.cell2* , 4*⟨tick1?⟩ , to.cell1* , tick1? , to.cell2* , tick1? , to.cell1* , tick1? , to.cell2* , tick1?⟩
|| ⟨to.cell2* , tick2? , to.cell3* , 4*⟨tick2?⟩ , to.cell2* , tick2? , to.cell3* , 2*⟨tick2?⟩ , to.cell2* , tick2?⟩
|| ⟨8*⟨tick0! || tick1! || tick2! || tick3! || tick4! || tick5!⟩⟩
|| ⟨tick3? , to.cell3* , tick3? , to.cell4* , 4*⟨tick3?⟩ , to.cell3* , tick3? , to.cell4* , 2*⟨tick3?⟩⟩
|| ⟨2*⟨tick4?⟩ , to.cell4* , tick4? , to.cell5* , 3*⟨tick4?⟩ , to.cell4* , tick4? , to.cell5* , 2*⟨tick4?⟩⟩

|| ⟨3*⟨tick5?⟩ , to.cell5* , tick5? , to.cell0* , 2*⟨tick5?⟩ , to.cell5* , tick5? , to.cell0* , 2*⟨tick5?⟩⟩⟩⟩

Summary

We have separated the CSP and VCR traces approximately into the eight iterations of the process network – one per line. It can be seen that each iteration interleaves its events slightly differently as the token passes along the process pipeline. It is possible to see the movement of the token in each of the different styles of tracing. In this instance, the structural traces are no clearer than the other styles.

6.3.4 I/O-PAR Example

An I/O-PAR process is one that behaves deterministically and cyclically, by first engaging in all the events of its alphabet in parallel, then repeating this behavior. I/O-PAR (and I/O-SEQ) processes have been studied extensively by Welch, Martin, Roscoe and others [Welch, 1989a; Welch et al., 1993; Martin et al., 1994; Martin and Welch, 1996; Roscoe and Dathi, 1987]. Roscoe and Welch separately proved I/O-PAR processes to be deadlock-free, and further that I/O-PAR processes are closed under composition. This proof is not simple, and reasoning about I/O-PAR processes from their traces is not straight-forward.

For an example of a simple IO-PAR network, we use the example originally presented by Burgin and Smith [2006]. One process repeats a in parallel with b ten times. The other process repeats b in parallel with c ten times. The two processes are composed in parallel, synchronising on b together.

CSP

⟨ a, b, c, c, a, b, a, b, c, a, b, c, a, b, c, a, b, c, a, b, c, a, b, c, a, b, c, a, b, c ⟩

VCR

⟨ {a, b, c}, {a, b, c}, {a, b, c}, {a, b, c}, {a, b, c}, {a, b, c}, {a, b, c}, {a, b, c}, {a, b, c}, {a, b, c} ⟩

Structural

⟨⟨10*⟨a || b⟩⟩ || ⟨10*⟨b || c⟩⟩

Summary

Given a simple process network, the structural trace again has a direct mapping to the original program. The VCR trace shows the regular parallelism in the system, whereas the CSP trace reveals a slight mis-ordering in the second triple of events. This was predicted in the original chapter; two c events happen in-between two b events in the CSP trace, but this slush is ironed out in the VCR trace.

6.4 Conversion

Each trace type is a different representation of an execution of a system. It is possible to convert between some of the representations. One VCR trace can be converted to many CSP traces (see section 6.4.1). A VCR trace can thus be considered to be a set of CSP traces. One structural trace can be interleaved in different ways to form many VCR traces and (either directly or transitively) many CSP traces (see section 6.4.2). A structural trace can thus be considered to be a set of VCR (or CSP) traces. This gives us an ordering on our trace representations: one structural trace can form many VCR traces, which in turn can form many CSP traces. These conversions are deterministic and have a finite domain. In general, conversions in the opposite direction (e.g. CSP to Structural) can be non-deterministic and infinite – for example, the CSP trace $\langle a \rangle$ could have been generated by an infinite set of structural traces: a, or $(a \parallel a)$, or $(a \parallel a \parallel a)$, and so on.

```
type CSPTrace = [Event]
type VCRTrace = [Set Event] -- All sets must be non-empty.

cartesianProduct :: [a] -> [b] -> [(a, b)]
cartesianProduct xs ys = [ (x, y) | x <- xs, y <- ys ]

vcrToCSP :: VCRTrace -> Set CSPTrace
vcrToCSP [] = singleton []
vcrToCSP (s : ss) =
  fromList [ a ++ b | a <- permutations (toList s), b <- toList (vcrToCSP ss)]
```

Figure 37: An algorithm for converting a VCR trace into a set of CSP traces.

6.4.1 Converting VCR Traces to CSP Traces

One VCR trace can be transformed to *many* CSP traces. A VCR trace is a sequence of multisets; by forming all permutations of the different multisets and concatenating them, it is possible to generate the corresponding set of all possible CSP traces. A Haskell function for performing the conversion is given in figure 37 – an example of this conversion is:

$$\langle\{a, b\}, \{c\}, \{d, e\}\rangle \mapsto \{\langle a, b, c, d, e\rangle, \langle a, b, c, e, d\rangle, \langle b, a, c, d, e\rangle, \langle b, a, c, e, d\rangle\}$$

Note that the number of resulting CSP traces is easy to calculate: for all non-empty VCR traces `tr`, the identity `length (vcrToCSP tr) == product (map Set.size tr)` holds. That is, the number of generated CSP traces is equivalent to the product of the sizes of each set in the VCR trace (permitting duplicates).

6.4.2 Converting Structural Traces to VCR or CSP Traces

The scheme for recording structural traces explained in section 6.2.4 does not permit their conversion into CSP or VCR traces *post hoc* because it is not clear how events "line up". Consider the structural trace:

$$\langle a, a\rangle \parallel a$$

It is not clear which (if either!) of the events from the LHS of the parallel composition are the same synchronisation as the RHS. Therefore it is not clear whether this should become the CSP trace $\langle a, a, a\rangle$ or $\langle a, a\rangle$.

A structural trace can be converted into CSP or VCR traces if we record a

little extra information. Specifically, we must record a sequence number with each communication event in the trace. In CSP terms, we effectively replace all synchronisations on each event a with a corresponding external choice:

$$(a \rightarrow P) \mapsto (\Box\, A \rightarrow P) \qquad \text{where} \quad A = \{a_i \mid i \in \mathbb{N}_0\}$$

All uses of the event a in sets for hiding and parallel composition must also be replaced by the events in A. We must then compose our existing system P in parallel with a new process for that event:

$$P \mapsto (P \underset{A}{||} SEQ_{a,0}) \qquad \text{where} \quad SEQ_{a,i} = a_i \rightarrow SEQ_{a,i+1}$$

In the implementation, the sequence number becomes part of the data structure underlying channels and barriers, and adds no significant overhead to the cost of communications.

With this sequence number, it is possible to match up the communications from different parts of the structural trace. Our earlier example would become $(a_0 \rightarrow a_1) \,||\, a_0$, if the first synchronisation on the LHS was the one featured on the RHS. Furthermore, the format of the structural trace means that it was already possible to derive the process identifiers needed to form the VCR trace. These two aspects combined will allow us to generate CSP and VCR traces from structural traces.

Algorithm Description

We define the algorithm for converting structural traces to CSP traces in Haskell, in figure 38. First, we define our data structures, based around an `Event` type with hidden definition (see figure 38, lines 1–7). A CSP trace is a list of events; a structural trace is either empty or a sequential trace. A sequential trace is an event synchronisation (an event identifier paired with a sequence identifier) or a parallel trace, followed by an optional further sequential trace in a *cons*-fashion. A parallel trace is a collection of two or more sequential traces.

To convert a structural trace, we must first know how many processes were involved in each synchronisation. We thus define a `prSeq` function that builds up a map from an event and sequence identifier pair to the number of processes that synchronised on that event (see figure 38, lines 9–16).

The remainder of our algorithm can be implemented in a continuation-like

style. The structural trace is explored, and a map is built up with all the events initially available (and the number of processes available to engage in them), as well as a function that, given an occurred event, will return the next map and function. This is done by the convSeq function (see figure 38, lines 18–36).

The final piece is a function that iteratively picks the next available event from the merged maps and records it in a new CSP trace, until the trace is complete (see figure 38, lines 38–49). This picking of available events and then continuing the trace is strongly reminiscent of the execution of the CSP program. Effectively, our algorithm is the environment, picking arbitrarily from the available set of events and then continuing the computation. Of course, our structural trace differs from a typical CSP system in that it is finite, lacks any choice, and by definition our replaying of a real structural trace is deadlock-free.

Example

We give here an example of converting a structural trace to its CSP and VCR forms:

$$
\begin{aligned}
&(a_0 \to b_0 \to b_1 \to a_1) \,||\, (c_0 \to b_0 \to (c_1 \,||\, d_0) \to b_1) \\
&\qquad \mapsto \{\langle a, c, b, c, d, b, a\rangle \\
&\qquad\quad , \langle a, c, b, d, c, b, a\rangle \\
&\qquad\quad , \langle c, a, b, c, d, b, a\rangle \\
&\qquad\quad , \langle c, a, b, d, c, b, a\rangle\} \\
&(a_0 \to b_0 \to b_1 \to a_1) \,||\, (c_0 \to b_0 \to (c_1 \,||\, d_0) \to b_1) \\
&\qquad \mapsto \{\langle \{a, c\}, \{b\}, \{c, d\}, \{b\}, \{a\}\rangle\}
\end{aligned}
$$

It can be seen that converting the set of CSP traces is the same as would be produced by converting the VCR trace into CSP traces.

Trace Generation and VCR Algorithm

Although the algorithm in figure 38 creates one trace through picking events arbitrarily, it would be trivial to modify it to generate all possible traces through exploring all options. From a more abstract perspective, the set of all traces is of interest: generating an arbitrary trace is a needless specialisation. From a

```haskell
type CSPTrace = [Event]

type SeqId = Integer
data StructuralSeq = Seq (Either (Event, SeqId) StructuralPar) (Maybe StructuralSeq)
data StructuralPar = Par StructuralSeq StructuralSeq [StructuralSeq]
data StructuralTrace = TEmpty
                     | T StructuralSeq

combine :: [Map (Event, SeqId) Integer] -> Map (Event, SeqId) Integer
combine = foldl (Map.unionWith (+)) Map.empty

prSeq :: StructuralSeq -> Map (Event, SeqId) Integer
prSeq (Seq (Left e) s)
  = combine [Map.singleton e 1, maybe Map.empty prSeq s]
prSeq (Seq (Right (Par sA sB ss)) s)
  = combine (maybe Map.empty prSeq s : prSeq sA : prSeq sB : map prSeq ss)

data Cont = Cont (Map (Event, SeqId) Integer) ((Event, SeqId) -> Cont)
          | ContDone

convSeq :: StructuralSeq -> Cont
convSeq (Seq (Left e) s) = c
  where
    c = Cont (Map.singleton e 1)
             (\e' -> if e /= e' then c else maybe ContDone convSeq s)
convSeq (Seq (Right (Par sA sB ss)) s)
  = merge (convSeq sA : convSeq sB : map convSeq ss) `andThen` maybe ContDone convSeq s

andThen :: Cont -> Cont -> Cont
ContDone `andThen` r = r
Cont m f `andThen` r = Cont m (\e -> f e `andThen` r)

merge :: [Cont] -> Cont
merge cs = case [ m | Cont m f <- cs] of
             [] -> ContDone
             ms -> Cont (combine ms) (\e -> merge [f e | Cont m f <- cs])

structuralToCSP :: StructuralTrace -> CSPTrace
structuralToCSP TEmpty = []
structuralToCSP (T s) = iterate (convSeq s)
  where
    participants = prSeq s

    iterate :: Cont -> CSPTrace
    iterate ContDone = []
    iterate (Cont m f) = fst e : iterate (f e)
      where
        es = Map.filter (\(n, n') -> n == n') (Map.intersectionWith (,) m participants)
        e = Map.findMin es -- Arbitrary pick
```

Figure 38: The algorithm for converting structural traces to CSP traces.

practical perspective, the set of all traces may be overwhelming and thus it may be easier to explore an arbitrary single converted trace.

The algorithm for converting structural traces to VCR traces is a combination of the algorithm given here for converting structural to CSP traces, and the pre-existing algorithm for recording VCR traces based on process identifiers described in section 6.2.2. One interesting aspect is the choice of available events. In our conversion to CSP, we pick arbitrarily (figure 38, line 49). In VCR, the choice (if we only wish to generate one trace) makes a noticeable difference to the resulting trace. Consider the structural trace:

$$(a \rightarrow b) \mid\mid c$$

Our first choice may be a, giving us the partial VCR trace $\langle\{a\}\rangle$. If our next choice is b, the VCR trace becomes $\langle\{a\}, \{b\}\rangle$, but if our next choice is c, the VCR trace becomes $\langle\{a, c\}\rangle$. We could favour the second choice if we wanted more independence visible in the trace – which may aid understanding. We emphasise again that this is a moot point for generating the set of *all* traces, but we believe that for single traces, a trace with more independence (i.e. fewer but larger sets in the VCR trace) will be easier to follow.

6.4.3 Practical Implications of Trace Styles

As discussed earlier in the chapter, a CSP trace is most straightforward to record, by using a mutex-protected sequence of events as a trace for the whole program. A VCR trace uses a mutex-protected data structure with additional process identifiers. Both of these recording mechanisms are inefficient and do not scale well, due to the single mutex-protected trace. With four, eight or more cores, the contention for the mutex may cause the program to slow down or alter its execution behaviour. This deficiency is not present with structural traces, which are recorded locally without a lock, and thus scale perfectly to more cores.

We could also consider recording the traces of a distributed system. Implementing CSP's Olympian observer on a distributed system (a parallel system with delayed messaging between machines) is a very challenging task. VCR's concepts of imperfect observation and multiple observers would allow for a different style of observation. Structural traces could be recorded on different machines without modification to the recording strategy, and merged afterwards just as they normally are for parallel composition.

The structural trace should also be easier to compress, due to regularity that is present in branches of the trace, but that is not present in the arbitrary interleavings of a CSP trace. Given that the structural trace is more efficient to record in terms of time and space (memory requirements), supports distributed systems well and can be converted to the other trace types *post hoc*, this makes a strong case for recording a structural trace rather than CSP or VCR.

6.5 Visual Traces

It is possible to visualise the different traces in an attempt to gain more understanding of the execution that generated the trace. For an example, we will use the following CSP system, P:

$$P = (Q \fatsemi Q) \underset{\{b\}}{||} ((b \rightarrow b \rightarrow \text{SKIP}) \,|||\, (c \rightarrow c \rightarrow \text{SKIP}))$$
$$\text{where } Q = (a \rightarrow \text{SKIP}) \,|||\, (b \rightarrow \text{SKIP})$$

One possible CSP trace of this system is:

$$\langle a, b, a, c, c, b \rangle$$

This trace is depicted in figure 39a. It can be seen that for CSP traces, this direct visualisation offers no benefits over the original trace. As an alternative, the trace is also depicted in figure 39b. This diagram is visually similar to a finite state automata, with each event occurrence being a state, and sequentially numbered edges showing the paths from event to event – but these edges must be followed in ascending order.

A possible VCR trace of this system is:

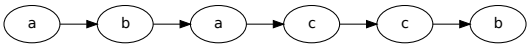
$$\langle \{a, b\}, \{a, c\}, \{c, b\} \rangle$$

This trace is depicted straightforwardly in figure 39c.

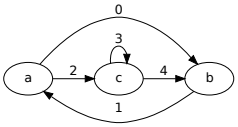
The structural trace of this system, with event sequence identifiers, is:

$$((a_0 \,||\, b_0) \rightarrow (a_1 \,||\, b_1)) \,||\, ((b_0 \rightarrow b_1) \,||\, (c_0 \rightarrow c_1))$$

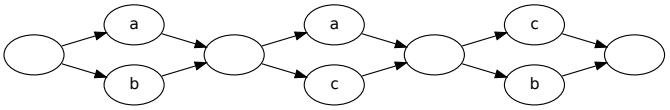
A straightforward depiction is given in figure 39d. An alternative is to merge together the nodes that represent the same event and sequence number, and use



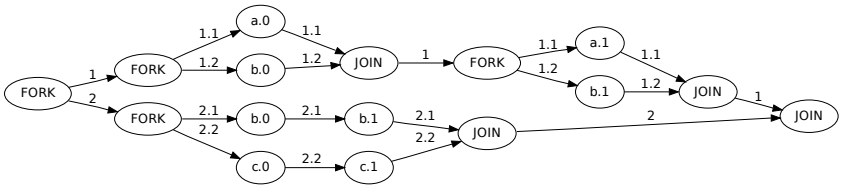
(a) A simple rendering of the CSP trace: $\langle a, b, a, c, c, b \rangle$.



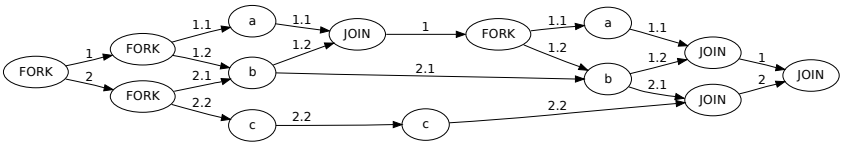
(b) A more complicated rendering of the CSP trace from (a): event synchronisations are rendered as state nodes, and ascending transition identifiers label the edges.



(c) A simple rendering of the VCR trace: $\langle \{a, b\}, \{a, c\}, \{c, b\} \rangle$



(d) A rendering of the structural trace: $((a_0 \parallel b_0) \rightarrow (a_1 \parallel b_1)) \parallel ((b_0 \rightarrow b_1) \parallel (c_0 \rightarrow c_1))$. Event synchronisations are represented by nodes with the same name, and the edges are labelled threads of execution joining together the event synchronisations.



(e) A rendering of the structural trace: $((a_0 \parallel b_0) \rightarrow (a_1 \parallel b_1)) \parallel ((b_0 \rightarrow b_1) \parallel (c_0 \rightarrow c_1))$. Event synchronisations are represented by a single node, and the edges are labelled threads of execution joining together the event synchronisations.

Figure 39: Various graphical representations of traces that could be produced by the CSP system $P = (Q \underset{\{b\}}{\parallel} Q) \parallel ((b \rightarrow b \rightarrow \text{SKIP}) \mid\mid\mid (c \rightarrow c \rightarrow \text{SKIP}))$ where $Q = (a \rightarrow \text{SKIP}) \mid\mid\mid (b \rightarrow \text{SKIP})$.

edges with thread-identifiers to join them together: this is depicted in figure 39e.

6.5.1 Discussion of Visual Traces

Merely graphing the traces directly adds little power to the traces (figures 39a, 39c and 39d). The contrast between the CSP, VCR and structural trace diagrams is interesting. The elegance and simplicity of the CSP trace (figure 39a) can be contrasted with the more information provided by the VCR trace of independence (figure 39c) and the more complex structure of the structural trace (figure 39d).

The merged version of the structural trace (figure 39e) is a simple change from the original (figure 39d), but one that enables the reader to see where the different threads of execution "meet" as part of a synchronisation.

Figure 39e strongly resembles the *dual* of a graphical representation of a Petri net [Reisig, 1985], a model of true concurrency. The nodes in our figure (FORK, JOIN and events) are effectively transitions in a standard Petri net, and the edges are places in a standard Petri net. Thus we can mechanically form a Petri net equivalent, as shown in figure 40.

Figure 39e is also interesting for its relation to VCR traces. Our definition of dependent events is visually evident in this style of graph. An event b is dependent on a iff a path can be formed in the directed graph from a to b. Thus, in figure 39e, it can be seen that the c events are mutually independent of the a and b events, and that the right-most a event is dependent on the left-most b event, and so on.

6.5.2 The Starving Philosophers

CSP models, coupled with tools such as FDR [Formal Systems (Europe) Ltd, 1997], allow formally-specified properties to be proved about programs: for example, freedom from deadlock, or satisfaction of a specification. Some properties of a system can be difficult to capture in this way – we present here an example involving starvation, based on Peter Welch's "Wot, No Chickens?" example [Welch, 1998].

Our example involves three philosophers. All the philosophers repeatedly attempt to eat chicken (no time for thinking in this example!). They are coupled with a chef, who can serve two portions of chicken at a time. In CSP with conjunction (see chapter 5), our example is:

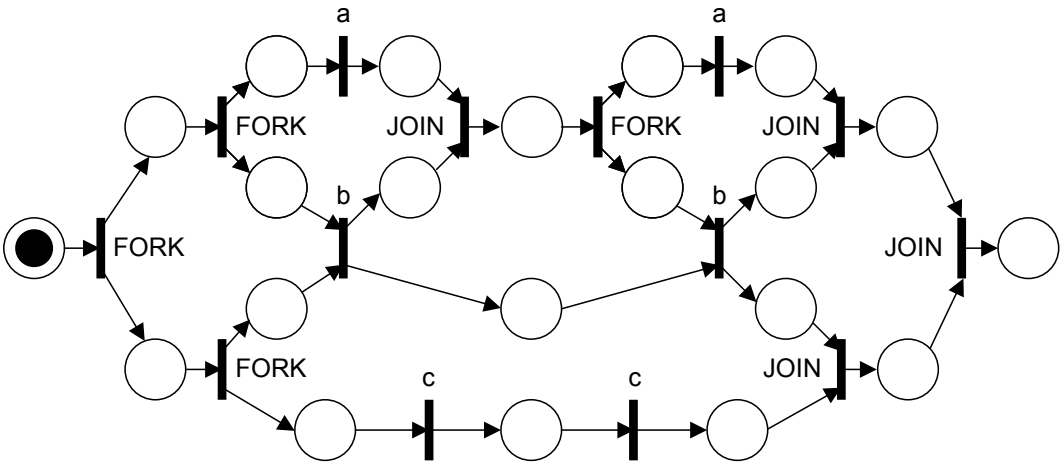


Figure 40: A petri net formed from figure 39e by using the nodes as transitions and the edges as places (unlabelled in this graph). Being from a trace, the petri net is finite and non-looping; it terminates succesfully when a token reaches the right-most place.

$$PHILOSOPHER(n) = chicken.n \rightarrow PHILOSOPHER(n)$$

$$CHEF = ((chicken.0 \wedge chicken.1) \rightarrow \text{SKIP}$$

$$\Box\ (chicken.0 \wedge chicken.2) \rightarrow \text{SKIP}$$

$$\Box\ (chicken.1 \wedge chicken.2) \rightarrow \text{SKIP}) \fatsemi CHEF$$

$$SYSTEM = (PHILOSOPHER(0) \ |||\ PHILOSOPHER(1) \ |||\ PHILOSOPHER(2)) \underset{\{chicken.0, chicken.1, chicken.2\}}{||} CHEF$$

Our intention is that the three philosophers should eat with roughly the same frequency. However, the direct implementation of this system using the CHP library leads to starvation of the third philosopher. Adding additional philosophers does not alleviate the problem: the new philosophers would also starve. In the CHP implementation, the chef always chooses the first option (to feed the first two philosophers) when multiple options are ready. This is a behaviour that is allowed by the CSP specification but that nevertheless we would like to avoid. Attempting to ensure that the three philosophers will eat with approximately the same frequency is difficult formally, but by performing diagnostics on the system we can see if it is happening in a given implementation. By recording the trace and visualising it (as shown in the structural trace in figure 41) we are able to readily spot this behaviour and attempt to remedy it.

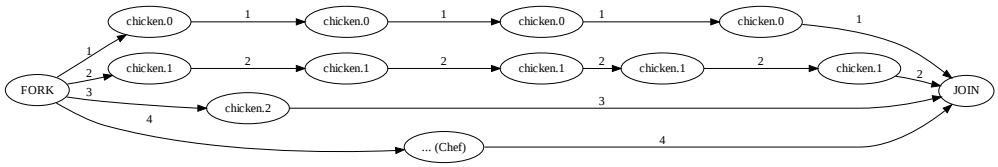


Figure 41: An example structural trace of five iterations of the starving philosophers example. The chef is collapsed, as it could be in an interactive visualisation of the trace. It is clear from this zoomed out view that one philosopher (shown in the third row) has far fewer synchronisations than the other two, revealing that it is being starved.

6.5.3 Interaction and Future Work

The diagrams already presented are static graphs of the traces. One advantage of the structural traces are that they preserve, to some degree, the organisation of the original program. The amenability of process-oriented programs to visualisation and visual interaction has long been noted – Simpson and Jacobsen provide a comprehensive review of previous work in this area [Simpson and Jacobsen, 2008]. The tools reviewed all focused on the design of programs, but we could use similar ideas to provide tools for interpreting traces of programs. We wish to support examination of *one* trace of a program, in contrast to the exploration of all traces of a program that tools such as PRoBE provide [Roscoe, 1997].

One example of an interactive user interface is shown in figure 43. This borrows heavily from existing designs for process-oriented design tools – and indeed, it would be possible to display the code for the program alongside the trace in the lower panel. This could provide a powerful integrated development environment by merging the code (what can happen) with the trace (what did happen).

An alternative example is given in figure 44. This shows the process hierarchy vertically, rather than the previous nesting strategy. Synchronising events are shown horizontally, connecting the different parts of parallel compositions. What this view makes explicit is that for any given event, there is an expansion level in the tree such that all uses of that event are solely contained with one process. For a, b and c, this process is numbers; for d it is the root of the tree.

The implementation of such tools is left for future work.

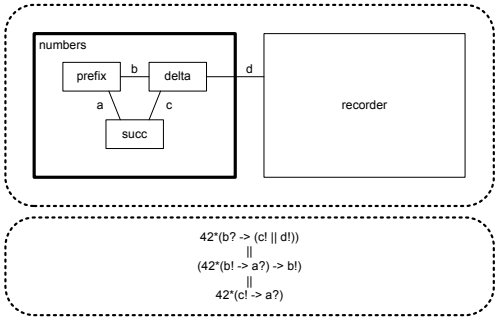
$$
\begin{aligned}
\text{CommsTime} &= (\text{numbers}(d) \underset{\{d\}}{||} \text{recorder}(d)) \setminus \{d\} \\
\text{numbers}(out) &= (\text{delta}(b, c, out) \underset{\{b\}}{||} \text{prefix}(a, b)) \underset{\{a,c\}}{||} \text{succ}(c, a)) \setminus \{a, b, c\} \\
\text{delta}(in, outA, outB) &= in?x \rightarrow (outA!x \rightarrow \text{SKIP} \;|||\; outB!x \rightarrow \text{SKIP}) \fatsemi \\
&\quad \text{delta}(in, outA, outB) \\
\text{prefix}(in, out) &= out!0 \rightarrow \text{id}(in, out) \\
\text{id}(in, out) &= in?x \rightarrow out!x \rightarrow \text{id}(in, out) \\
\text{succ}(in, out) &= in?x \rightarrow out!(x + 1) \rightarrow \text{succ}(in, out) \\
\text{recorder}(in) &= in?x \rightarrow \text{recorder}(in)
\end{aligned}
$$

(a) The CSP specification for the CommsTime network, *CommsTime*.

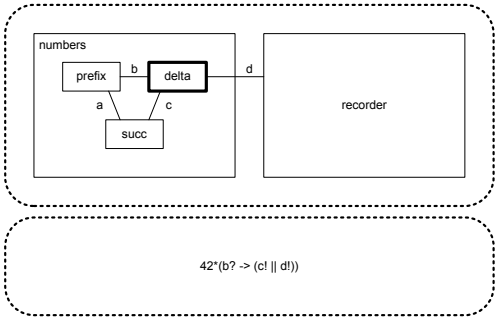
$$
\begin{aligned}
&(\text{numbers}: \\
&\qquad (\text{delta}{:}42 * (b? \rightarrow (c! \;||\; d!))) \\
&\;||\; (\text{prefix}{:}(42 * (b! \rightarrow a?) \rightarrow b!)) \\
&\;||\; (\text{succ}{:}42 * (c! \rightarrow a?))) \\
&\;||\; (\text{recorder}{:}42 * d?))
\end{aligned}
$$

(b) The structural trace for the CommsTime network. Note that the labels come from code annotations.

Figure 42: The specification and trace of the CommsTime example network. Note that the hiding of events – important for the formal specification – is disregarded for recording the trace in our implementation in order to provide maximum information about the system's behaviour.

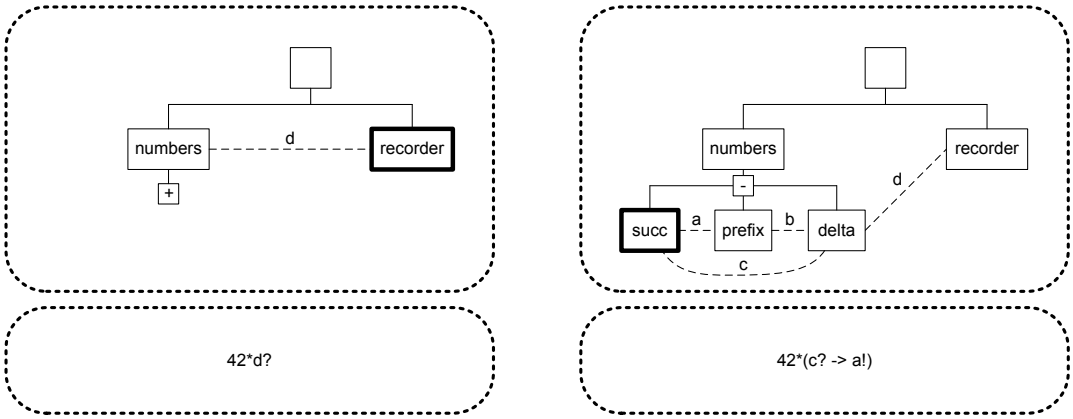


(a) The selected process is numbers.



(b) The selected process is delta.

Figure 43: An example user interface for exploring the trace (from figure 42) of a process network. The top panel displays the process network, with its compositionality reflected in the nesting. Processes can be selected, and their trace displayed in the bottom panel. Inner processes could be hidden when a process is not selected (for example, recorder could have internal concurrency that is not displayed while it is not being examined).



(a) The recorder process is selected. (b) The succ process is selected.

Figure 44: An example user interface for exploring the trace (from figure 42) of a process network. The top panel shows a tree-like view for the process network. Processes can be selected and their trace displayed in the bottom panel. Notably, the internal concurrency of processes (and the associated events) can be hidden, as in the left-hand example for numbers. On the right-hand side, the numbers process is expanded to reveal the inner concurrency.

6.6 Conclusions

Tracing is relevant to program diagnostics; investigating bugs or strange behaviours (such as unexpected inefficiency) typically involves comparing the actual behaviour of a program to the developer's intentions. Tracing allows the actual behaviour of a program to be recorded and examined. Adding print statements to a program is a well-known and much used methodology for sequential programs but this does not directly translate for concurrent programs, because timing and scheduling issues can separate the print statement from the event that it relates to.

Tracing, where the events are recorded directly as they occur, can provide an accurate record of a program's behaviour. There are different approaches to be taken, three of which are CSP traces, VCR traces and structural traces. A CSP trace is an elegant, flat sequence of events. VCR traces retain more structure concerning the concurrency of events. Structural traces retain the most structure, and deviate from the other traces in that each event synchronisation is recorded multiple times, once by each participant.

There is an interesting comparison to be made here between the tracing support available for concurrent CHP programs, and tracing functional Haskell programs [Chitil et al., 2000]. Tracing lazy functional programs is notoriously hard

due to the difficulties with evaluation orders and the possibility of repeated evaluation of the same value – or the single evaluation of shared values. In contrast, tracing the events that occur in CHP programs is relatively straightforward. This is an advantage of constructing programs in a process-oriented style in Haskell using CHP.

Chapter 7

Modelling

This chapter details an interesting technique for generating formal models of CHP programs without using source code analysis. This work was previously published in Automated Verification of Critical Systems (AVoCS) 2009 [Brown, 2009].

7.1 Motivation and Approach

The CHP library is closely related to the CSP process calculus, as explained in chapter 3. CSP provides a good concurrency model and design principles that can be carried into the CHP library. It should also be possible to take programs written in CHP, produce a CSP model of them, and use formal techniques and tools, such as the FDR proof-checker [Formal Systems (Europe) Ltd, 1997], to verify the behaviour of the model and thus the program.

There are several ways to produce the CSP model of a CHP program. One is to produce the model by hand: this is time-consuming and error-prone, although it does allow expert knowledge to be used in the conversion (which can sometimes allow simpler models to be produced than with tools). A second way is to use a tool to produce the model by analysing the program's source code. Such tools are a much easier way of generating a model, but they can be difficult to originally develop. The entirety of a program, including all the functional aspects of a CHP program unrelated to the concurrency, would need to be understood and processed in order to produce the model of a program by analysing its source code.

In this chapter we present a third approach. The central idea of our approach is to take an existing program, written to use the CHP monad, and to switch the imported definition of the CHP monad for one that will not run the full program, but instead discover its structure and produce a CSP model of the communication

behaviour of the program. This approach is limited in several respects (which will be discussed throughout the chapter) but provides an alternative tool-based option that requires less time to develop, and may be useful during prototype development where the power of full source-code analysis is not required.

7.2 CSP Models

Our approach centres around a type for specifying models:

```
type Spec' proc comm = [SpecItem' proc comm]

data SpecItem' proc comm
  = Par [Spec' proc comm]
  | Alt [Spec' proc comm]
  | Call proc
  | Sync comm
  | Stop
  | Repeat (Spec' proc comm)

type SpecItem = SpecItem' ProcessId CommId

type Spec = Spec' ProcessId CommId

newtype SpecMod = SpecMod (Spec -> Spec)

instance Monoid SpecMod where
  mempty = SpecMod id
  mappend (SpecMod f) (SpecMod g) = SpecMod (f . g)

finalise :: SpecMod -> Spec
finalise (SpecMod f) = f []
```

The Spec' type is a chronological sequence of individual specification items (SpecItem'). We will return to the ProcessId and CommId types later in the chapter. The SpecMod item is a function that modifies a specification; these can be turned into specifications by applying them to an empty specification (the empty list).

7.3 Model Generation Monad

To generate the model, we will use the following monad:

```
type CHP a = CHPSpecT (StateT CHPState IO) a

newtype CHPSpecT m a = CHPSpecT (forall r. ContT r (WriterT SpecMod m) a)

runSpec :: CHPSpecT m a -> (a -> m (b, SpecMod)) -> m (b, SpecMod)
runSpec (CHPSpecT m) f = runWriterT (runContT m (WriterT . f))

instance Monad m => Monad (CHPSpecT m) where
  return x = CHPSpecT (return x)
  CHPSpecT m >>= k = CHPSpecT (m >>= (\x -> let CHPSpecT m' = k x in m'))
```

The CHPSpecT monad is a **newtype**-wrapper around ContT and WriterT. A pictorial aid to the monad is given in figure 45.

7.4 Barriers and Sequential Composition

Barriers are straightforward to redefine. A barrier becomes a simple wrapper around an event identifier:

```
type EventId = Integer
type CommId = Either EventId (EventId, Dir, Integer)
data PhasedBarrier phased = Barrier EventId

syncBarrier :: PhasedBarrier () -> CHP ()
syncBarrier (Barrier x)
  = addSpec $ return ((), Sync $ Left x)
```

The communication identifier is either an event identifier (as here, for barriers) or a triple (for channel communications, explained later in this chapter). The addSpec function is used for adding an event to the sequence of events in the model:

```
addSpec :: forall m a. Monad m => m (a, SpecItem) -> CHPSpecT m a
addSpec m = CHPSpecT $ do
  (x, s) <- lift (lift m)
  lift (tell $ SpecMod (s :))
  return x
```

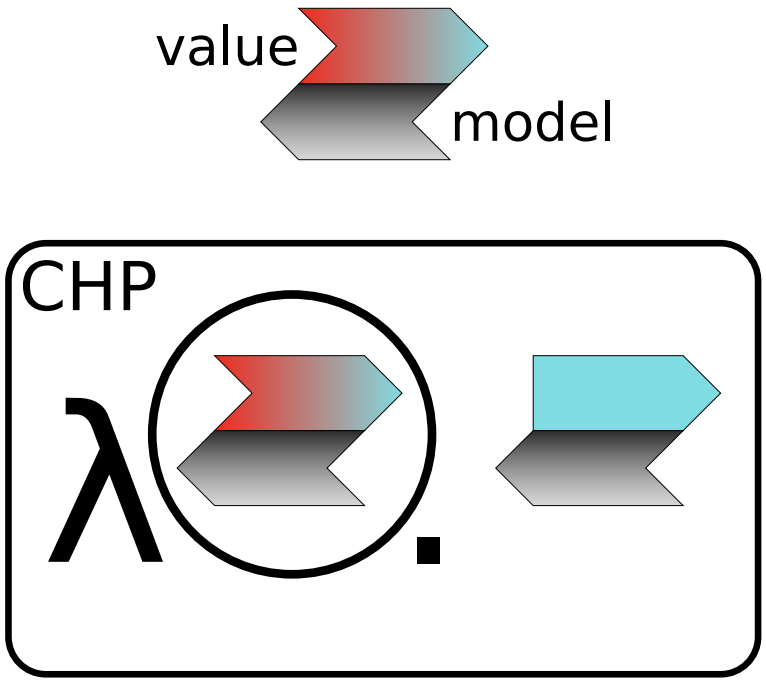


Figure 45: A depiction of the redefined CHP monad for modelling. The top diagram illustrates an item of type `b -> m (r, SpecMod)`: values are depicted on the top, and flow "forwards" through the monadic binds. In contrast, specifications are depicted on the bottom, and flow "backwards" through the model: the functions are composed by being applied to all future models in order to form the current model. The lower diagram shows the full CHP monad: given a function of the aforementioned type `b -> m (r, SpecMod)`, it becomes a complete item of type `m (r, SpecMod)`.

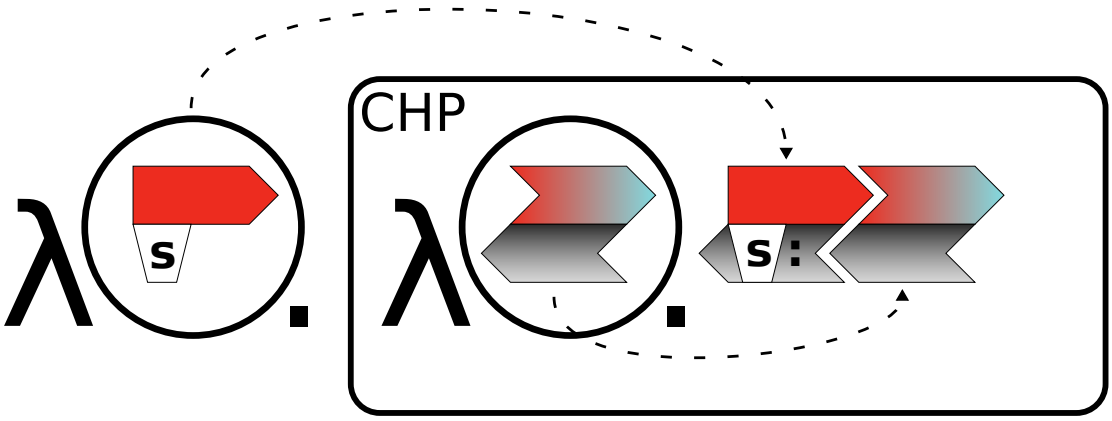


Figure 46: An illustration of the addSpec function. Given an item that produces a value and a single specification item, it gives back an action in the CHP monad that puts the item on the front of future specifications once the value is fed in.

It adds the given specification item to the front of the specification before returning the given value. The specifications are built up backwards, by applying each modification to the models formed by future operations; hence, we must add to the front of the specifications of the future behaviour. This function is depicted in figure 46.

7.5 Parallel Composition

Parallel composition is the most straightforward type of composition. It takes a list of parallel branches, and generates their models, then adds a Par constructor and passes them to the addSpec function to be joined into the model in sequence with the subsequent process:

```
parallel :: [CHP a] -> CHP [a]
parallel = addSpec . liftM (second Par . unzip) . mapM finSpec

finSpec :: Monad m => CHPSpecT m a -> m (a, Spec)
finSpec m = second finalise <$> runSpec m (\x -> return (x, mempty))
```

The finSpec function can be used to generate a model, by binding the model generation to the return function, and then finalising the specification-modifying function. The parallel function is illustrated in figure 47.

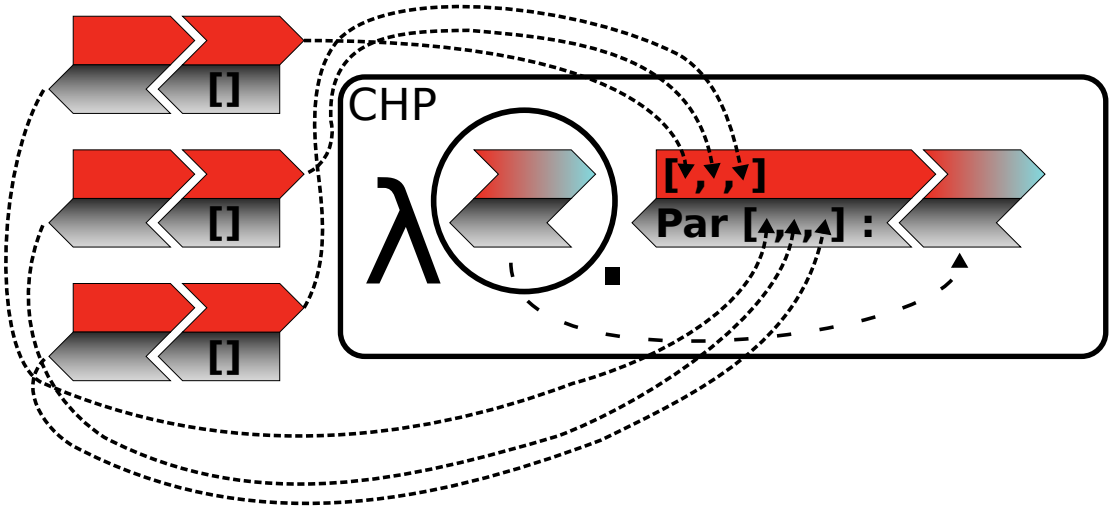


Figure 47: The behaviour of the redefinition of the parallel function. The specification functions of the processes are finalised by feeding them the empty list: this is shown on the left. The return values of the processes are used to form the list of return values (the top arrows). The specifications are put into a list with the Par constructor and joined on to the front of future specifications (the bottom arrows).

7.6 Communication

We have already seen how barriers can be modelled in section 7.4. Channels can be modelled similarly – but there is one important difference with channels. Reading from a channel returns a value. This value can be used for decisions about what to do next, and/or communicated on other channels. In both cases, we would like to capture this in our model. We return to the use of values for decisions in section 7.7; in this section we consider tracking which values read from one channel are passed to another.

Haskell contains a function error :: String -> a, which causes an error if it is evaluated, and gives the given String as an error message. These errors can be trapped in the IO monad; in combination these two features allow us to create identifiable values of any type (that give an error if evaluated). This technique is used to good effect elsewhere – for example, in the Lazy SmallCheck library [Runciman et al., 2008].

When we read a value from an input channel, we can create such an identifiable bottom value (chpNextBottom is a strict Integer member of CHPState):

```
bottomPrefix :: String
bottomPrefix = "__CHP.bottom__"
```

```
fakeCommIn :: Integer -> CHP a
fakeCommIn chanId = addSpec $ do
  st <- get
  put $ st { chpNextBottom = succ $ chpNextBottom st }
  return (error $ bottomPrefix ++ show (chpNextBottom st)
         ,Sync $ Right (chanId, DirInput, chpNextBottom st))
```

We then create some corresponding code for performing outputs that detects these identifiable bottom values:

```
fakeCommOut :: Integer -> a -> CHP ()
fakeCommOut chanId val = addSpec $ do
  possErr <- lift $ try $ evaluate val
  case possErr of
    Left (C.ErrorCall s) | bottomPrefix `isPrefixOf` s
      -> return ((), (Sync $ Right
           (chanId, DirOutput, read $ drop (length bottomPrefix) s)))
    -- Might be a bottom, but not ours:
    _ -> return ((), Sync $ Left chanId)
```

We can later use these identifiable values to name the inputs from channels, and produce an identical name where these inputs are directly passed on to other channels. There is a limitation with these technique, however: if the value being sent is not identical to that received from the channel, but uses it in a way that will trigger the original error when the new value is evaluated, this will not be detected. So for example, if the process increments a number it receives by one before sending it on, this modification of the number will be ignored by the model generation, which will generate a model as if the value was passed on unmodified.

7.7 IO and Nondeterminism

It is possible to execute IO actions in the CHP monad in the normal CHP library, using the `liftIO_CHP :: IO a -> CHP a` function. If we did not supply this function in the mirror library, this would cause problems for most CHP programs. The values that the function returns – and values returned from channels, as discussed in the previous section – can be used to make decisions about what to do next. This is actually the key capability that a monad supplies that, say, functors and applicative functors do not [McBride and Paterson, 2008].

We are thus faced with the problem that we need to know all the behaviours a process may adopt based on the value it is given. This is akin to testing, and we can borrow a technique from testing for it. The Lazy SmallCheck library [Runciman et al., 2008] provides techniques to generate partially-defined values (specialising them as required) in order to test a functions behaviour when given these values as arguments.

Our continuation-passing approach means that we have the continuation – the rest of the process to be executed given the return of the current item. We combine this with our testing techniques to examine all the behaviours a process may take, based on the value. We first define a fuzz function that will find possible behaviours of a given monadic action based on the value given:

```
fuzz :: (Serial a) => (a -> StateT s IO b) -> StateT s IO ([b], Bool)
```

The Serial type-class is used by Lazy SmallCheck to generate values. The returned item is the list of possible values (specifically: the list of possible models), paired with a Bool value that indicates whether the search was complete or partial.

This function is used in our redefinition of liftIO_CHP:

```
liftIO_CHP' :: Serial a => IO a -> CHP a
liftIO_CHP' _ = usingCont $ k -> WriterT $
    do (vals, complete) <- fuzz (runWriterT . k)
       unless complete $
         liftIO $ putStrLn "Incomplete fuzzing of IO computation"
       nonces <- replicateM (length vals) newEvent
       let firstLabel = "IO_" ++ show (head nonces)
       zipWithM_ labelEvent nonces (map (firstLabel ++) suffixes)
       modify $ \st -> st { chpIOEvents = Set.union
                              (Set.fromList nonces) (chpIOEvents st) }
       return (error "liftIO return", SpecMod $ \s ->
         [Alt $ zipWith (\n (_, SpecMod f) -> Sync (Left n) : f s) nonces vals])
  where
    suffixes = map (:[]) ['A'..'Z'] ++ map show [(0::Integer)..]

usingCont :: (forall b. (a -> WriterT SpecMod m b) -> WriterT SpecMod m b)
          -> CHPSpecT m a
usingCont m = CHPSpecT (ContT m)
```

The function completely ignores the actual IO action that it is given. It takes all the different models found in the search, and puts an artifically created event

before each, then joins all of these in an external choice. The external choice is a correct way to model such a change in behaviour based on unknown external input in CSP. Effectively, each dummy event (which will not be synchronising with any other process) represents a possible class of input values from the external computation, and the external choice between non-synchronising events reflects that it is unknown which event may occur.

7.8 Recursion

Recursion allows the behaviour of a process to be longer than its definition. In particular, many processes run forever, but their definition is finite. It is important when creating models that we are able to prevent the model becoming infinite (which would also mean that our model-finding would not terminate). In short, we want our model generation to terminate, even when the program may run forever.

There has been work into detecting such cycles in program code, and reifying them into a representation where the cycle is expressly represented [Gill, 2009]. This work relies on the particular implementation of "stable names", but according to the semantics this technique is not guaranteed to work at all. In this chapter we adopt a different approach by making the programmer use helper functions that deal with the infinite behaviour. There are two methods by which processes typically recurse; through the `forever` helper function, and by calling themselves. We will deal with each case in turn.

7.8.1 Recursion Forever

The `forever` monadic function allows a process to be made to run continually:

```
syncForever :: PhasedBarrier () -> CHP ()
syncForever a = forever (syncBarrier a)
```

This process is obviously repeating its behaviour, and expands out to:

```
syncsForever a
  = syncBarrier a >> syncBarrier a >> syncBarrier a ...
```

From observing this behaviour, we have no way of telling that the behaviour definitely repeats forever (compare the behaviour observed with the above process to: `replicateM 1000 (syncBarrier a) >> syncBarrier b`). To solve this problem, we

introduce our own foreverP function that programmers must use in place of forever, with the same semantics. In the standard CHP library, foreverP is also defined such that programs can work as normal (foreverP = forever). The definition of foreverP for model generation is:

```
foreverP :: CHP a -> CHP b
foreverP = stopSpec . liftM (Repeat . snd) . finSpec

stopSpec :: Monad m => m SpecItem -> CHPSpecT m a
stopSpec m = CHPSpecT $ ContT $ const $
  do sp <- lift m
     tell $ SpecMod (const [sp])
     return $ error "stopSpec"
```

The foreverP function generates the specification for its inner body, and then wraps it in a Repeat constructor. The stopSpec function stops modelling of any further processes in this monadic sequence. The const call ignores the continuation bound to this one, and instead it uses the specification sp as its model, with a dummy return value (that should never be examined).

7.8.2 Recursion by Calling

Processes do not just recurse by using the forever function, but also by direct recursion: calling themselves, potentially with parameter(s) to allow persistent state. As discussed in the previous section, we have no way observationally of detecting repeated behaviour and recursion. We therefore require the programmer to add a process annotation. This annotation must be placed such that it gives the following guarantees:

1. The types of the arguments of the process, and its return type, must support the Typeable type-classes, which supports dynamic typing (in order to store them in a central data structure). The arguments of the process must also support the Eq type-class in order to compare them for equality on subsequent analyses of the same process.

2. The behaviour of the program will not be infinite between encountering process annotations (or foreverP, discussed in the previous section).

3. The behaviour of an annotated process will be deterministic given the same

list of arguments (apart from results returned in the monad by communications and IO actions – i.e. it will have the same model), where "same" is determined by the Eq instance on the arguments.

4. The combination of the process name and argument-types must uniquely determine each process in the program, and must determine the return type.

5. The result of a potentially infinitely-recursive process must be the unit type.

The annotation is placed as follows:

```
syncRepeated :: PhasedBarrier () -> CHP ()
syncRepeated = process "syncRepeated" $ \b -> syncBarrier b >> syncRepeated b

syncLeftRight :: PhasedBarrier () -> PhasedBarrier () -> Bool -> CHP ()
syncLeftRight = process "syncLeftRight" $
  \a b lr -> syncBarrier (if lr then a else b) >> syncLeftRight a b (not lr)
```

It can be seen that this is a relatively small addition to the code. We recognise repeatedly calls to the same process with same arguments (which, recall, we take to always have the same model) by storing them in a data structure. The arguments are heterogeneously-typed (for multiple processes, and even within one process) so we store them using dynamic typing. In fact, we do not need to store the arguments themselves: we instead store a function that will check the arguments for being the same:

```
type CheckArg = Dynamic -> Bool

checkArgs :: [Dynamic] -> [CheckArg] -> Bool
checkArgs ds fs
  | length ds /= length fs = False
  | otherwise = and $ zipWith ($) fs ds
```

To implement our process annotation to work with any appropriate type of process, we use a Haskell type-class:

```
process :: Process p => String -> p -> p
process s = process' True s []

class Process p where
  process' :: Bool -> String -> [(Dynamic, CheckArg)] -> p -> p
```

We will return to the Bool parameter later; it is merely passed on by the instance below, as is the String parameter that names the process. The process' function takes as its third parameter the list of arguments: we pass both the values themselves in a Dynamic wrapper to check them against previous arguments, and a function that will similarly do the checking should we record this process for the future.

One instance of the type-class is used to capture the arguments to the process and add them to the list:

```
instance (Eq a, Typeable a, Process b) => Process (a -> b) where
  process' topLevel name args f x
    = process' topLevel name
        (args ++ [(toDyn x, (== Just x) . fromDynamic)]) (f x)
```

These parameters are built up until they reach the last instance for CHP a items which contains all the logic:

```
data CHPState = CHPState { ... ,
  , chpProcessMap :: Map.Map String (Map.Map Integer ([CheckArg], (Spec,
  Dynamic)))
  , chpFreeNames :: [(Dynamic, CheckArg)]
  , chpNextProcess :: !Integer }

instance (Typeable a, Eq a) => Process (CHP a) where
  process' topLevel name immArgs p = addSpec $ do
    st <- get
    let possibles = Map.toList <$> (Map.lookup name $ chpProcessMap st)
        args = if topLevel then immArgs
                  else chpFreeNames st ++ immArgs
    case possibles >>= find (checkArgs (map fst args) . fst . snd) of
      Just (n, (_, (_, r)))
        -> return (flip fromDyn (error "process-lookup") r, Call n)
      Nothing ->
        do let n = chpNextProcess st
           put $ st { chpProcessMap = insertMapMap name n
                        (map snd args, (error ``process'', toDyn ())) $
                        chpProcessMap st
                    , chpFreeNames = args
                    , chpNextProcess = succ n
                    }
```

```
(r, f) <- finSpec p
modify $ \st' -> st' { chpProcessMap = insertMapMap name n
                         (map snd args, (f, toDyn r)) $
                         chpProcessMap st'
                     -- Restore original free names:
                     , chpFreeNames = chpFreeNames st }
return (r, Call n)
```

The case statement searches for any existing models of the process. If it finds one (the top case), the return value is pulled out of the data structure and returned along with the model that simply calls the stored process. Thus the model can short-circuit at this point when it finds a previously-recorded process.

If no matching process is found (the lower case), we record one and put it in the state afterwards. It is crucial that the state is first updated with an entry for the process – that way, when the process recurses, it can find itself in the collection of recorded processes (if the recursion uses the same argument values). The dummy entry has the right parameters but an invalid model (that will never be accessed before it is later updated) and an incorrect return type; processes that recurse and then examine the value of the recursive call are not supported. However, almost every recursive CHP process is tail-recursive, which can be modelled just fine (if they werent tail recursive, they would probably feature a space-leak). After the process has been modelled, its entry is updated with the real return value and real specification.

It should be noted that this approach is only valid for recursive processes if eventually the process recurses with the same arguments (i.e. if the process has a finite number of states). Processes that, for example, output continually-ascending integers could not be modelled by this approach. However, this is not as major a limitation as it first appears – the FDR model-checker also requires that the process must have a finite number of states in order to be checked, so the model generation is only as limited as the model-checker into which it feeds.

The `process` annotation is also defined in the standard CHP library to be `process = const id`, allowing programs with this annotation to work as normal with the standard CHP library.

7.9 Choice

Implementing choice in our framework can be done very simply and avoiding many potential problems by using an observation about CHP[1]:

```
alt [p >>= k, q >>= k] = alt [p, q] >>= k
```

When modelling alts, we can simply join all subsequent continuations onto each branch of the alt in order to capture the model.

Our definition of alt is as follows:

```
alt :: [CHP a] -> CHP a
alt [] = stopSpec $ return Stop
alt ps = altSpec ps

altSpec :: Monad m => [CHPSpecT m a] -> CHPSpecT m a
altSpec ms = usingCont $ \k -> WriterT $
  do xfs <- mapM (flip runSpec k) ms
     return ( error "alt return"
            , SpecMod $ \s -> [Alt [f s | (_, SpecMod f) <- xfs]])
```

The return value of the alt can only be examined if the alt is used inside a process annotation: we currently do not support this. The continuation after this alt call (which is supplied to us in our continuation-passing style) is run with each branch of the alt, and these specification functions are applied to all future specifications (which will only be the empty lists from a finalise call) before being put together in an Alt item. This is depicted in figure 48.

7.10 Post-Processing

After the model for a program has been determined, it is a relatively simple matter to print it out in the machine-readable CSP form that the FDR model checker expects.

One small change needed is that while Hoare [1985] had an infinite repetition operator in his CSP (a prefix *), FDR does not include this. Therefore the body of a Repeat item must be pulled up into a new process (that always recurses), and the item itself turned into a call to this process.

[1]Note: this rule only holds if p and q are *not* return calls. We elide the special handling of return in this case for simplicity.

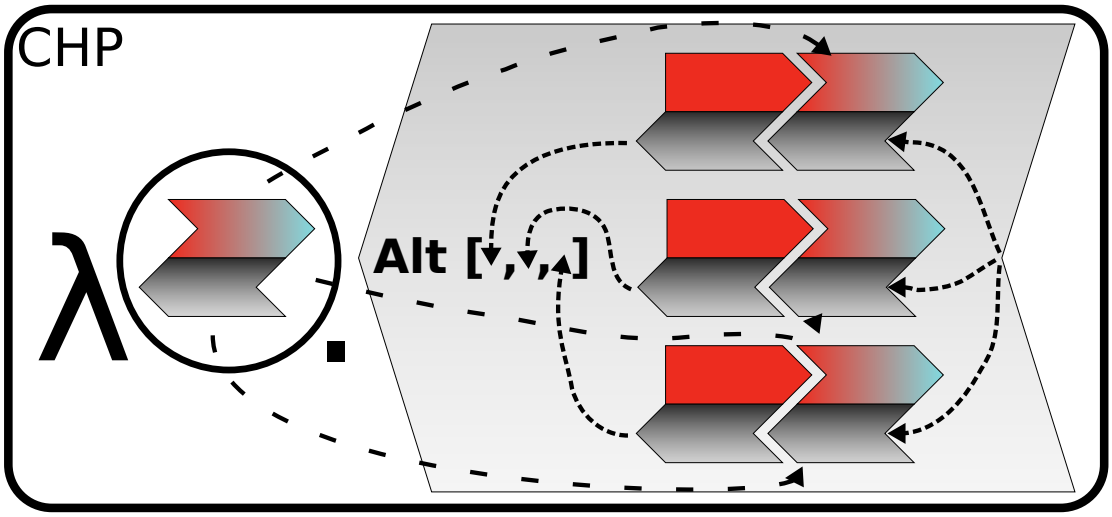


Figure 48: The redefined alt method. Each of the given values are joined with the given continuation; the specification-funcions from each of these items are applied to the specification following the alt call, and these are all put into a list with an Alt item. The return values of the alt branches are discarded.

Each different combination of arguments to an annotated process will correspond to a different process in the generated output. We could have more concise models if, in future, we did some post-processing to merge back any processes that differ only in the events used. Consider the simple system:

```
blackHole :: Chanin a -> CHP ()
blackHole = foreverP . readChannel

p = blackHole c <||> blackHole d
```

This would generate a model such as:

```
channel c
channel d
blackHole_1= (c?x_1 -> blackHole_1)
blackHole_2= (d?x_2 -> blackHole_2)
p = (((blackHole_1) ||| (blackHole_2)))
```

Even though the behaviour of the two black hole processes is identical except for the channel involved, we currently generate two instantiations of the process rather than one parameterised process. In future we would like to maintain the correctness of the specification, but also reduce its verbosity by merging such processes together.

7.11 Alphabets

In CHP, the synchronisation rules are as follows. A channel requires exactly two processes to synchronise on it. A barrier has an enrollment count, and a number of processes equal to that count must synchronise on the barrier for it to complete.

In Roscoe's CSP [Roscoe, 1997], an event can have any number of parties (one upwards). Which processes must synchronise with each other on an event is determined by the shared alphabet when the two processes are composed in parallel. Given this CSP:

$$P = a \rightarrow b \rightarrow c \rightarrow \text{SKIP}$$
$$Q = b \rightarrow \text{SKIP}$$
$$R = c \rightarrow \text{SKIP}$$
$$ALL = (P \underset{b}{||} Q) \underset{c}{||} R$$

The event a will involve just P, whereas b will involve P and Q, and c will involve P and R. Should the event a have been included in the alphabet of either parallel composition, P would cause deadlock as the other process in the composition would not be offering the event a.

In translating CHP programs into CSP models, we must infer the alphabets for parallel compositions. Since all of our events have unique identifiers we are able to follow a simple rule: the events in the alphabet of parallel composition are the intersection of the sets of events engaged in by the two processes being composed.

This rule works correctly for most programs (including the dining philosophers example). However, consider the following program:

```
p = do a <- newBarrierWithLabel "a"
       b <- newBarrierWithLabel "b"
       enroll a (\a0 -> enroll a (\a1 -> enroll b (\b0 -> enroll b (\b1 ->
            parallel [do syncBarrier a0
                         syncBarrier b0
                     , syncBarrier a1])))
```

This program will deadlock when run, as only one process (of two enrolled) will synchronise on the barrier b. If the specification is generated with our simple rule, we get this specification:

```
p = (a -> b -> SKIP) [|{| a |}|] (a -> SKIP)
```

This specification will not deadlock, as the event b is not in the shared alphabet with any other process. More generally, if fewer processes are using a barrier than are enrolled, or if only one process is using a particular channel, the model generated with our simple alphabet rule will be incorrect.

The simplest solution to this is as follows. We augment our framework to track how many processes should be using a particular event (two for channels, the enrollment count for barriers). If the actual number of processes turns out to be lower than this, we compose the relevant processes in parallel with a dummy process (`SKIP`) with the events shared, for example:

```
p = ((a -> b -> SKIP) [|{| a |}|] (a -> SKIP))
      [|{| b |}|] SKIP
```

This new model will reveal the deadlock in the original program.

7.11.1 Dining Philosophers Example

As an example of generating a CSP model from a CHP program and checking for deadlock, we use the dining philosophers problem. The code for the deadlocking dining philosophers, that can be executed normally, or used to produce traces Brown and Smith [2008], is as follows:

```
fork :: PhasedBarrier () -> PhasedBarrier () -> CHP ()
fork = process "fork" (\left right ->
    foreverP ((do syncBarrier left
                  syncBarrier left )
              <|> (do syncBarrier right
                      syncBarrier right )))

philosopher :: PhasedBarrier () -> PhasedBarrier () -> CHP ()
philosopher = process "philosopher" (\left right ->
  foreverP (
   do randomDelay
      syncBarrier left <||> syncBarrier right
      randomDelay
      syncBarrier left <||> syncBarrier right ))
  where
```

```
    randomDelay :: CHP ()
    randomDelay = liftIO_CHP (getStdRandom (randomR (500000, 1000000)))
                  >>= waitFor

college :: Int -> CHP ()
college = process "college" (\nPhil ->
  withBarrierPairListWithStem nPhil "fork_left_phil" (\forkLeftChans ->
  withBarrierPairListWithStem nPhil "fork_right_phil" (\forkRightChans ->
   parallel_ (
    [fork (fst (forkRightChans !! n))
          (fst (forkLeftChans !! ((n + 1) `mod` nPhil)))
    | n <- [0.. nPhil - 1]]
    ++
    [philosopher (snd (forkLeftChans !! n)) (snd (forkRightChans !! n))
    | n <- [0.. nPhil - 1]]))))
```

The process annotations on the fork and philosopher are not strictly necessary (the use of foreverP catches the infinite behaviour) but help to label the processes in the generated model.

This real running code can then be changed to use the CHP-model library by changing a single import statement from importing the main CHP library to importing the modelling library. The generated model for three philosophers is shown in figure 49. If we append the line `assert main :[ deadlock free ]` to that script, FDR produces a trace of one of the deadlocks in the system:

```
BEGIN TRACE example=0 process=0
fork_right_phil2
fork_right_phil1
fork_right_phil0
END TRACE example=0 process=0
```

7.12 Shallow and Deep Embedding

The techniques detailed in this chapter substitute the definition of the library primitives that execute the CHP code for definitions with the same type (at least, with types that have the same name) that instead generates a model of the program. This is effectively swapping one shallow embedding of CHP for another.

An alternative approach would have been to have a deep embedding of CHP, that holds a single reified representation of the CHP program, which could then be

executed by an evaluation function, or modelled by a modelling function. A deep embedding would also facilitate analysis that ensured that the program followed the guidelines for using the library safely (e.g. not returning a barrier from an enroll block).

The intent of CHP is to be a Haskell library that can be easily programmed and easily combined with other Haskell code. This is more difficult in a deep embedding. In a sense, this thesis can be seen as answering the question: how much can be done with a shallow embedding of process-oriented concurrency in Haskell? Future work could involve instead investigating a deep embedding of process-oriented concurrency.

7.13 Conclusions

An obstacle to the wider use of formal methods by programmers has been the relative complexity of formal methods. Hence it is important to provide tools to allow as much automatic use of formal methods as possible. We have demonstrated a technique for generating formal models that can be used automatically for a certain class of programs.

The changes required to output a CSP model from an existing CHP program are minimal:

- Import a different module (Control.Concurrent.CHPSpec rather than Control.Concurrent.CHP), which could be toggled using a preprocessor.
- Change the top-level of the program from runCHP to specifyCHP. Again, this could be done with a preprocessor.
- Add process annotations on recursive processes, and switch the use of forever to foreverP. The changed code will still work with the original library exactly as before, i.e. the annotations can be permanent.

This can easily be combined with an external script to automatically feed the output straight into FDR (no manual editing of the model is needed). This process could therefore be integrated into the software development cycle, similar to the automatic running of a test suite. Normally, the program will run – but with the switch of a preprocessor flag, it will generate its formal model. Thus, the model will always remain up-to-date as the program changes.

The output of FDR, when checking freedom from deadlock, is a trace of events that correspond to the names given to the events in library calls. Therefore it is possible, even without any formal training, to read the trace and relate it back to the original program.

The machine-readable CSP format that our framework generates is standardised, and thus it is possible to use other model-checkers (such as ProB [Leuschel and Fontaine, 2008]) or tools (such as FDR explorer [Freitas and Woodcock, 2009]) on the specification. An alternative mode of use is to use a tool such as ProBE [Roscoe, 1997] to explore the possible traces of a program by deciding which of all the available events should happen next.

Limitations of this work primarily stem from its approach of avoiding source code analysis. Communication channels and IO actions that have a large codomain cannot feasibly be checked using our approach, as we must blindly check all possible values in order to fully specify a process. For types with an infinite range an exhaustive check is not even possible. However, we have shown that we are able to check functions to a given depth, which may well suffice for many programs.

This work does not model conjunction, as there is no CSP representation (or simple transformation into CSP) for conjunction. Details on extending CSP to support conjunction are given in chapter 5, but this work targets standard CSP (and tools that work on standard CSP).

```
channel fork_left_phil0
channel fork_left_phil1
channel fork_left_phil2
channel fork_right_phil0
channel fork_right_phil1
channel fork_right_phil2
college_1=
  (((fork_2)
    [|{| fork_left_phil1 , fork_right_phil0 |}|]
    ((fork_3)
     [|{| fork_left_phil2 , fork_right_phil1 |}|]
     ((fork_4)
      [|{| fork_left_phil0 , fork_right_phil2 |}|]
      ((philosopher_5) ||| ((philosopher_6) ||| (philosopher_7)))))))
fork_2= (repeated_8)
fork_3= (repeated_9)
fork_4= (repeated_10)
main_0= (college_1)
philosopher_5= (repeated_11)
philosopher_6= (repeated_12)
philosopher_7= (repeated_13)
repeated_8=
  (((fork_right_phil0 -> fork_right_phil0 -> SKIP)
    []
    (fork_left_phil1 -> fork_left_phil1 -> SKIP))
   ; repeated_8)
repeated_9=
  (((fork_right_phil1 -> fork_right_phil1 -> SKIP)
    []
    (fork_left_phil2 -> fork_left_phil2 -> SKIP))
   ; repeated_9)
repeated_10=
  (((fork_right_phil2 -> fork_right_phil2 -> SKIP)
    []
    (fork_left_phil0 -> fork_left_phil0 -> SKIP))
   ; repeated_10)
repeated_11=
  (((fork_left_phil0 -> SKIP) ||| (fork_right_phil0 -> SKIP))
   ; ((fork_left_phil0 -> SKIP) ||| (fork_right_phil0 -> SKIP))
   ; repeated_11)
repeated_12=
  (((fork_left_phil1 -> SKIP) ||| (fork_right_phil1 -> SKIP))
   ; ((fork_left_phil1 -> SKIP) ||| (fork_right_phil1 -> SKIP))
   ; repeated_12)
repeated_13=
  (((fork_left_phil2 -> SKIP) ||| (fork_right_phil2 -> SKIP))
   ; ((fork_left_phil2 -> SKIP) ||| (fork_right_phil2 -> SKIP))
   ; repeated_13)
main = main_0
```

Figure 49: The generated model for the deadlocking version of the dining philosophers with three philosophers.

Chapter 8

Conclusions

This thesis has detailed the design, implementation and augmentation of Communicating Haskell Processes (CHP), a process-oriented Haskell library that bears a close resemblance to the CSP process calculus. In this chapter we explore this library's advantages and disadvantages when compared both to other process-oriented libraries and languages (section 8.1) and to other Haskell concurrency mechanisms (section 8.2). We then evaluate the new features of CHP – conjoined events, tracing and model generation (section 8.3).

8.1 Comparison to Other Process-Oriented Libraries and Languages

CHP is the latest in a long history of process-oriented languages and libraries, beginning with occam 2.1 [INMOS, 1985] – which was later augmented to form occam-π [Welch and Barnes, 2005] – and continued with libraries such as JCSP [Welch, 1997] and CTJ [Hilderink et al., 1997], C++CSP [Brown and Welch, 2003] and CTC++ [Hilderink and Broenink, 2003], various .NET CSP libraries [Chalmers and Clayton, 2006; Lehmberg and Olsen, 2006], PyCSP [Bjørndalen et al., 2007] and CSO [Sufrin, 2008]. Here, we compare CHP against these predecessors and contemporaries.

8.1.1 Strengths

The user-defined operators in CHP allow for concise expression of behaviour without introducing confusion (as operator overloading in C++ can). To read a value from the first available channel of `c` and `d` and send it on channel `e` requires simply:

```
(readChannel c <|> readChannel d) >>= writeChannel e
```

Contrast this with the occam equivalent:

```
INT x:
ALT
  c ? x
    e ! x
  d ? x
    e ! x
```

The operators also allow close correspondence to the CSP process calculus on which the CHP library is based. We give here a quick illustrative list of CSP expressions (left) and their CHP equivalent (right)[1]:

$a \rightarrow P$	`syncBarrier a >> p`
$c!x \rightarrow P$	`writeChannel c x >> p`
$c?x \rightarrow P(x)$	`readChannel c x >>= p`
$P \parallel Q$	`p <||> q`
$(a \rightarrow P) \Box (b \rightarrow Q)$	`(syncBarrier a >> p) <|> (syncBarrier b >> q)`

Haskell's idioms and common functions can be directly applied when writing CHP code. For example, the `mapM` function allows a function with monadic side-effects to be applied to every element of a list; thus, `mapM (writeChannel c) xs` sends the values in the list `xs` in order on the channel `c`. This is very concise and less error-prone than indexed loops in other languages, such as the occam equivalent:

```
SEQ i = 0 FOR SIZE xs
  c ! xs[i]
```

Recursion is another Haskell feature that is natural to use in CHP. Due to the problems of pre-allocated stacks in occam, C++CSP and most other implementations, recursion has generally been viewed as problematic: if a process has any recursion with a dynamic, potentially-large recursion count, this could overflow a fixed-size stack. The Haskell run-time is already suited to implement such recursive functions that are a mainstay of functional programming, so recursive processes are also easily supported. Recursion provides an elegant and concise way of expressing processes, as demonstrated in the next example.

[1] Given that the CSP external choice operator "[]" could not be used – nor anything similar – a decision was made to use the existing `<|>` operator from the `Alternative` type-class in Haskell instead.

The CHP library provides support for choice on writing (also known as output guards) which until now has only been offered in JCSP [Welch et al., 2010]. This feature turns out to simplify some programming logic (at a slight run-time cost). For example, here is a process that maintains a piece of data, continuously offering to either input a new value or output the current value:

```
value :: Chanin a -> Chanout a -> a -> CHP ()
value input output x
  = (readChannel input >>= value input output)
    <|> (writeChannel output x >> value input output x)
```

This process is a good example of a recursive process, and demonstrates a simple way of emulating shared-variables in process-oriented programming. In occam and other systems, this process would need to have a channel for inputting new values, and a request channel asking for the current value with an accompanying reply channel for sending the current value after a request. Allowing choice with outputs eliminates this request channel – instead the other process may just attempt to read from the reply channel. The process above is slightly awkward to use, however: any update that requires the previous value must read the current value (typically from a shared channel) then send a new one, which can introduce a race hazard, just as it can when using unprotected shared variables. We can avoid this problem by sending an update *function* instead of a new value:

```
value' :: Chanin (a -> a) -> Chanout a -> a -> CHP ()
value' input output x
  = (readChannel input >>= value input output . ($ x))
    <|> (writeChannel output x >> value input output x)
```

This technique of sending functions down channels is not one that can be easily accomplished in most other process-oriented libraries, and not as naturally as in Haskell, where first-class functions are often used.

Using Haskell as the language for implementing a process-oriented library immediately eliminates several problems of using other languages (e.g. C++) for concurrency. We do not have to deal with the accidental or deliberate misuses of shared mutable data between threads because Haskell is functional and all values are immutable (and thus safely shareable) by default. We do not need to worry about the order of destruction of channels (see section 2.5.3 for an example of how channels may be destroyed in C++CSP while they are still in use) because the channels are garbage-collected by the Haskell run-time when, and only when, they are no longer in use.

Haskell's referential transparency and first-class functions mean that it is easy to move around and refactor blocks of code. Any portion of a Haskell **do** block may be pulled out into a new function, or any function may be inlined into a **do** block. For example, writeChannel c x <||>writeChannel d y is the composition of two channel-writes in parallel; the individual channel-writes may be written as here or pulled out into their own function. This is not the case in C++, where parallel sub-processes *must* be pulled out into their own class. CHP is more flexible than occam, which does not allow guards to be pulled out to separate procedures (CHP does allow this).

The first-class functions also allow for processes that take functions as arguments, or processes. The pipeline-wiring functions (that take a list of arbitrary processes as an argument and wire them up with channels) and processes such as CHP's map process (that takes a function as an argument to apply to data passing through the process) cannot be implemented in occam, nor easily in C++ or Java or most other languages that have process-oriented libraries. The map process is again concise:

```
map :: (a -> b) -> Chanin a -> Chanout b -> CHP ()
map f input output
  = forever (readChannel input >>= writeChannel output . f)
```

The process-wiring functions supplied in CHP allow processes to be easily composed. For example, the above map process may be combined with a filter process to produce a single process that outputs the string representation of non-negative numbers:

```
filter (> 0) <=> map show
```

The composition operators allow topology to be captured in a helper function, parameterised by the types of connections between the processes. For example, a two dimensional grid of sites in a boids simulation can then be wired together just using a call to a helper function:

```
grid4way ( replicate 100 ( replicate 100 site ))
```

This helper function could not be implemented in occam without fixing the channel types and directions used to connect the processes, and would also be very awkward to implement in frameworks such as C++CSP and JCSP. Making this composition of processes simple and short encourages re-use of code, and programming large systems as a composition of small processes: that is, CHP embodies the process-oriented programming model.

8.1.2 Weaknesses

The implementation of CHP supports several concurrency features that are supported by few (or no) other process-oriented languages and libraries. CHP supports choice at both the reading and writing end of a channel (and by all participants in a barrier synchronisation), which until now has only been supported by the JCSP library. In contrast to JCSP's centralised oracle algorithm (using monitors), CHP implements a decentralised algorithm (i.e. without a single central data structure) using Software Transactional Memory (STM). Benchmarks indicate that this incurs a factor of two slowdown over standard synchronous channel communications, at least in Haskell (see section 5.6).

This extra feature, along with conjoined events, comes with a (performance) cost. CHP allows conjunction and choice on all channels and barriers, and this support means that channel communications will be slower, even when not using these features. One way to alleviate this problem is to offer fast channels that support less features (but that can use a quicker algorithm), but we chose simplicity in the API over raw performance.

Thus the main weakness of CHP is its speed (see section 3.14 for the results of some simple benchmarks). This is not a direct result of its expressiveness, nor particularly of its functional nature. Instead, it is a result of the extra capabilities of the library: choice at both ends of a channel, and conjunction. It would be possible to produce a version of CHP with less capabilities that should be closer in speed to the other frameworks.

8.2 Comparison to Other Haskell Concurrency Mechanisms

The original major concurrency extension for Haskell was Concurrent Haskell [Peyton Jones et al., 1996], which used MVars as a communication mechanism between different threads. An MVar is effectively a one-place buffered communication channel, but one that omits the possibility of choice between MVars; it is not possible to wait for the first available value from two different MVars. In process-oriented terminology, any-to-any shared channels are provided but not choice between one-to-one channels. This precludes certain styles of programming, eliminating the useful synchronisation aspect of process-oriented communications. For example, an overwriting buffer process [Brown, 2008] cannot be implemented

with MVars. Overall, the capabilities of CHP subsume those of MVars. The key advantage of MVars is their greater efficiency (see section 5.6).

The subsequent major concurrency development for Haskell was the implementation of Software Transactional Memory (STM) [Harris et al., 2005]. STM does support the choice that MVars lacked, and in general supersedes MVars (which can be easily emulated using STM, just as they can with CHP). STM provides the base layer for the implementation of CHP, and therefore can provide all of CHP's capabilities.

STM can be considered as a safer and more elegant version of shared variables with locks. The approach is centred around shared mutable variables – but the transactional nature removes several common problems with locks, such as determining lock acquisition order when multiple variables must be accessed. Shared variable concurrency suffers from numerous problems such as contention for centralised resources and designs that hard to compartmentalise and compose. STM has also been accused of being of limited utility in terms of parallel performance [Cascaval et al., 2008].

Since the development of the CHP library, the Concurrent ML library has been implemented in Haskell [Chaudhuri, 2009]. This library supports choice at both ends of channels and could potentially be used as a core and built on to provide a process-oriented concurrency library with most or all of the capabilities of CHP; as it currently stands, CML features only the basic concurrency primitives of forking, channel communication and choice – lacking barriers, poison or any of the new features of CHP such as conjunction and in-built support for tracing.

Alternatively, there exist concurrency mechanisms for Haskell that have capabilities beyond those of CHP. The transactional events work of Donnelly and Fluet [2006] supports a transactional approach to communication, effectively fusing the communications of CHP with the transactional approach of STM. The transactional events suffers, as CHP does, from inefficiency, but is in some respects even more powerful than CHP: it allows choice between a series of communications, rather than only a choice between a single communication (or a conjunction of communications) as CHP does. An interesting avenue of future work would be to look into combining transactional events with session types in CHP.

8.2.1 Strengths and Weaknesses

CHP features more capabilities and a richer API than all the other Haskell concurrency mechanisms. Where MVars support only reading and writing, CHP also supports: barriers and broadcast/reduce channels (including dynamic enrollment), clocks, poisoning, conjunction, choice, tracing, composition operators, testing aids, behaviours, common processes and an arrow interface. In a way they are perhaps in different classes: MVars are a small concurrency feature designed to support simple message-passing and further abstractions on top of them, whereas CHP is a concurrent framework designed to support many different aspects of concurrency, including pragmatic tools and high-level APIs. This is both a strength and, to a certain extent, a weakness: CHP is very well-featured and powerful, but this can be slightly intimidating to new programmers.

CHP is designed to support process-oriented concurrency. It can be used simply for passing a few messages between a handful of threads, or for passing work and results between workers and farmers in parallel tasks – but MVars would be a more efficient alternative for these purposes. The strength of CHP is that it supports a richer programming model: hierarchical compositions of process networks and CSP-based design principles. Using the library without adopting some of this programming model incurs CHP's weaknesses without taking advantage of its strengths, and thus programmers must whole-heartedly adopt it for it to be worthwhile, rather than using small elements. This point also shows up with regard to an element of CHP's design: the CHP monad.

One disadvantage of CHP is its use of its own monad. At the top-most level, a Haskell program is an action in the IO monad. Haskell programs tend to be a mix of imperative actions in the IO monad, and pure (side-effect free) code in the classic functional programming style. Explicitly concurrent programs tend to have IO actions as a significant part of the program.

The Haskell concurrency extensions mentioned above fit well with the use of the IO monad. The MVars mentioned above are accessed from the IO monad. STM has its own (eponymous) monad, in which transactions are written – but the atomically call then executes this transaction from the IO monad. In contrast, CHP's eponymous monad is a layer on top of the IO monad: CHP programs must be written in the CHP monad, with all IO actions inside them wrapped with a liftIO_CHP call[2]. This makes CHP programs slightly more verbose, and also means

[2]This problem can be reduced with the use of the library `http://hackage.haskell.org/package/monadIO` that automatically lifts common IO actions.

that introducing it into an existing project has the overhead of moving much of the existing code into the CHP monad.

The CHP monad is needed instead of the IO monad in the library for three reasons. The first reason is to support poison: the poison handlers are more elegant to write with the CHP monad, but the exception mechanism in the IO monad could be used instead. The second reason is that the tracing mechanism described in chapter 6 requires access to some local values (the read-only addresses of mutable variables, and the current process-identifier). This reason could be removed if traces were removed from the library, or alternatively a different solution could perhaps be found. The third reason is to support choice between the leading actions of CHP processes; an alternative would be to use a CML-like design where the choice is explicitly made between special action types rather than monadic blocks. Thus, one could imagine a slightly cut-down, or less elegant CHP library where all the actions took place in the IO monad, which might make the library more accessible for Haskell programmers.

8.3 New Features

CHP adds two further features from the formal CSP process calculus that are not available in any other process-oriented libraries (or Haskell concurrency mechanisms): the ability to record traces (not found elsewhere) and the ability to generate CSP models of the program (something previously explored in occam-π). The library also adds conjoined events, a novel feature not found in CSP (or any other process-oriented concurrency library).

8.3.1 Traces

Traces were described by Hoare [1985] as observations of a process's behaviour, and later used in proofs about processes' behaviour. The work presented in chapter 6 of this thesis takes traces as purely an observation of behaviour – analogous to a log-file generated by a classic sequential imperative program – and provides run-time support for generating them, to support diagnostics and debugging of process-oriented programs. The technique is not restricted to CHP, and should be possible to implement in any process-oriented program or library.

Traces can be used for testing: when a CHP test-case fails, the trace of the failing program is automatically printed out for the user to see. This is a very

useful to any developer using CHP. Traces can also be enabled for full programs, which can be especially useful in debugging. Monitoring which communications occurred between processes, including the values that were communicated, helps greatly when tracking down errors in a concurrent program. The trace is guaranteed to be a true reflection of what occurred, unlike print statements in a program following the communication, which can suffer from latency, re-ordering and interleaving and thus not reflect the true order of events.

8.3.2 Model Generation

Some work has already been undertaken to augment an occam compiler to generate a CSP model alongside the compiled object code [Barnes and Ritson, 2009]. The approach described in chapter 7 of this thesis is for a concurrency library without any such modification to the compiler. Our approach even avoids the need for source-code analysis by using the organisation of Haskell's monads (plus a few annotations) to produce a model of the program using a mirror implementation of the CHP library. This approach is experimental and ultimately limited in its power, but has provided an interesting exploration into methods for generating formal models of process-oriented programs in other languages.

Generating an accurate model of a concurrent program allows the program to be model-checked (using FDR or similar). Model-checking allows us to check that the program will never deadlock (a common error in concurrent programs) or that it meets a particular formal specification. Automatically generating the model would remove the burden from the programmer, enabled the easier use of formal methods during concurrent program development.

8.3.3 Conjunction

Conjoined events are a new feature that allow for dynamic "fusion" of events into a single event at run-time. This enables patterns such as invisible buffer processes that do not disrupt the synchronisation behaviour of the processes around them (see section 5.3.4). It also allows old problems to be solved in new ways, which led to the development of a new dining philosophers solution in standard CSP (see section 5.3.1). We believe that conjoined events are a useful design tool, and this thesis has introduced and motivated them – and given an algorithm for implementing them (see chapter 5).

8.4 Conclusions

Communicating Haskell Processes is the intersection of process-oriented programming and Haskell concurrency – and it is informative for both. The repeated use of higher-order functions and recursion show how powerful they are for process-oriented programming, and Haskell's immutable data avoids many pitfalls of shared mutable data in other imperative host languages such as C++.

CHP does not just provide the basic concurrency primitives such as choice, concurrency and channels: it goes further and provides higher-level constructs on top of these. Features such as process-wiring and behaviours show that traditional process-oriented features are good building blocks – and that there are exciting possibilities to build powerful high-level frameworks on top of them. CHP also includes features such as tracing – which was inspired by the theoretical notion of tracing in the CSP calculus, but it can be used to provide practical benefits during software testing by automatically tracing failing programs. CHP is a very powerful and very richly-featured framework, which means it can provide great benefit to programmers – but also that it can be unaccessible to novices who only want to use a small part of the framework.

CHP includes novel capabilities such as conjoined events and tracing. These do not feature in any other process-oriented languages or libraries – but there is nothing that ties these features to CHP. Details have been provided on how to implement conjunction and tracing, and we hope that this will allow these features to be provided in other process-oriented frameworks in the near future, allowing for more investigation into their usefulness (and into further improvements).

The work done in this thesis also provides evidence that Haskell is a powerful host language in which to embed programming models beyond functional programming. The imperative, declarative concurrency model of CSP easily fitted into the functional model of Haskell by using features such as type-classes and monads – which in turn allowed for many new features beyond the process calculus of CSP.

The work described in this thesis supports the development of well-designed concise concurrent programs with less errors. The increased interest in concurrency brought about by multicore processors means that this work can provide relevant approaches and tools for developers who wish to write their programs concurrently, without difficult-to-understand and error-prone threads and locks, without the problems of shared data, and with support to help avoid long wiring

code, with APIs that prevent the introduction of errors, and with support for testing and debugging their programs.

8.5 Future Work

This thesis has primarily focused on new features and programming patterns – not on efficiency. The recent advent of multicore processors means that many people are turning to concurrent programming to achieve speed-up on multicore machines. The performance of CHP is dependent both on its algorithms, and the speed of the underlying STM system. Optimising these further would help attract those who want to use CHP for parallel speed-up.

Possibilities for visualisation techniques for traces were suggested in chapter 6, but these have yet to be implemented. In future, automated interactive visualisation could be implemented for traces, and improved through use on real problems. This would provide a useful diagnostic tool for deployed CHP programs, and allow for easy inspection of unit test failures during development.

The approach taken to modelling CHP programs in chapter 7 was quite simple and lightweight, eschewing source code analysis. In future, it would be beneficial to attempt to use full source code analysis and associated tools in order to provide a more accurate and complete model of a CHP program that did not require adding any annotations.

8.6 Contribution

This thesis makes the following contributions:

1. **Development of a library for Haskell that supports CSP-style concurrency.** This thesis details the design rationale of the CHP library that allows CSP-like concurrency to be used in Haskell (chapter 3).

2. **Development of new techniques for composing processes together.** This thesis explores new operators and functions to allow concise and powerful composition of processes (chapter 4).

3. **Invention of conjoined events.** This thesis introduces into process-oriented programming the idea of waiting for conjoined events, i.e. waiting for both a and b to occur, and only engaging in both or neither (chapter 5).

4. **Algorithms for channels, choice and conjunction using Software Transactional Memory.** This thesis explains how to use software transactional memory to implement the algorithms required to support the types of communication and choice that CSP provides (chapter 5).

5. **Invention of structural traces.** This thesis introduces a new style of trace that preserves more information about the history of a concurrent computation (chapter 6).

6. **Algorithms for recording traces.** This thesis presents algorithms for recording CSP, VCR and structural traces that can be used by any concurrent run-time (chapter 6).

7. **Technique for generating CSP models of CHP programs.** This thesis explains a lightweight approach to generating CSP models of CHP programs without the need for source-code analysis (chapter 7).

Appendix A

Glossary

A

Alt: The name for CSP's external choice that is used in occam and subsequent process-oriented programming libraries (short for alternative).

C

C++CSP: Communicating Sequential Processes for C++, a C++ library supporting process-oriented programming [Brown and Welch, 2003; Brown, 2004, 2007].
CHP: Communicating Haskell Processes, a Haskell process-oriented programming library that was developed as part of this thesis [Brown, 2008].
CommsTime: A standard process-oriented benchmark that involves a prefix process (sends the value 0 then behaves as the identity process), a successor process (like the identity process, but adds one to the data as it passes through) and the delta process (duplicates data on two outputs) in a ring, with the second delta output connected to a recorder that times how long it takes for N numbers to be generated.
Conjoined Events: See conjunction.
Conjunction: The ability to wait for multiple events in a single guard. The guard will only be chosen if all the events in the guard can be completed together, atomically.
CSP: Communicating Sequential Processes, a theory of concurrency originally

created by Hoare [1985] and developed further by Roscoe [1997].

D

Dining Philosophers: A classic concurrency example from the book by Hoare [1985].
Disjunction: An alternate name for CSP's external choice and alts (in contrast with conjunction).

G

Guard: One option in an alt (external choice).

J

JCSP: Communicating Sequential Processes for Java, a Java library supporting process-oriented programming [Welch, 1997; Welch et al., 2007].

M

Mobility: The notion from occam-π of mobile data, processes and communications primitives. Mobile items can only have one reference to them in a program at any time, and to preserve this attribute they use a movement (transferral) semantics that destroys the source reference.
Monad: A Haskell construct that can be used to implement sequencing.
Monad Transformer: A way to combine the effects of several monads into a single monad.
MVar: An MVar is a shared mutable variable in Concurrent Haskell [Peyton Jones et al., 1996] that can also be used as a one-place buffered channel (but without support for choice).

O

occam 2.1: Also referred to as classical occam. The last occam language update released by INMOS, described in the occam reference manual [INMOS, 1985].
occam-π: The updated occam language, a superset of occam 2.1 that adds mobility concepts from the pi-calculus [Welch and Barnes, 2005].
Output Guard: The use of a channel-write in an alt (external choice) – historically, only input guards have been allowed in most process-oriented systems.

P

π-calculus: A process calculus developed by Milner [1999].
Poison: A technique for terminating process networks by setting channels into a persistent unusable state [Brown and Welch, 2003; Sputh and Allen, 2005].
Process-Oriented Programming: See section 2.4 for a definition.

S

STM: Software Transactional Memory, an approach to concurrency that has been implemented in Haskell [Harris et al., 2005].
Structural Traces: A different style of trace that is hierarchical and records each event multiple times – once per participant.

T

Trace: A record of the behaviour a process (or system of processes).

V

VCR: View-Centric Reasoning, a different approach to observing a system that allows for multiple observers, unreliable observers, and parallel event sets.

Appendix B

Brief Introduction to Haskell

Haskell is a statically-typed lazily-evaluated functional programming language. This appendix provides the necessary background information on Haskell to understand this chapter. We explain how to read Haskell types and some basic syntax, before explaining the concept of monads.

B.1 Types

We precede every Haskell function with its type signature. The format for this is:

```
functionName :: typeOfParameter1 -> typeOfParameter2 ->
                ... -> resultType
```

The statement `x :: t` should be read as: `x` has type `t`. Each parameter is separated by an arrow, so that `a -> b` is a function that takes a parameter of type `a`, and returns type `b`. This function arrow is right associative; all Haskell functions can be concieved as taking exactly one argument, and usually yield a further function as a result.

Any type beginning with an upper-case letter is a specific type, whereas any type beginning with a lower-case letter (by convention, single lower-case letters are used, most frequently `a`) is a type variable. For example, this is the type of the `map` function, that applies a transformation to a list:

```
map :: (a -> b) -> [a] -> [b]
```

This takes a function transforming a value of type `a` into type `b`, and maps from a list of type `a` to the corresponding list of type `b`. The `a` and `b` type variables can be the same (and frequently are).

B.2 Functions

The simplest form of a Haskell function definition consists of the function name, followed by a variable for each of its parameters, then an equals sign and the definition of the function. For example, this function adds the squares of two numbers:

```
addSquares :: Int -> Int -> Int
addSquares x y = (x * x) + (y * y)
```

This function can be called by simply following the function with parameters, separated by spaces: `addSquares 3 5`. There is also an infix notation for function calls. The prefix function call `addSquares 3 5` may also be written `3 `addSquares` 5`; the infix form is created by using backquotes around the function name.

Anonymous functions can be formed using the notation `\x -> x * x`; this is a function that takes one parameter and squares it. It is instructive to note that our previous `addSquares` definition is equivalent to:

```
addSquares :: Int -> Int -> Int
addSquares = \x y -> (x * x) + (y * y)
```

It is also equivalent to the following form which is most similar to the classic lambda calculus form:

```
addSquares :: Int -> Int -> Int
addSquares = \x -> \y -> (x * x) + (y * y)
```

B.3 Data Types

As well as primitive data types (`Int`, etc), Haskell contains algebraic data types, sometimes known as discriminated unions in other languages. An algebraic data type can have several different tags (these must begin with a capital letter, as must type names):

```
data DayOfWeek
  = Monday | Tuesday | Wednesday | Thursday
  | Friday | Saturday | Sunday
```

These tags can also have types following them, for example:

```
data Point = Point Int Int
data Shape = Rectangle Point Point | Circle Point Int
```

Valid values of these types are:

```
Point 3 5
Circle (Point 1 2) 6
```

Types can also be parameterised: much as the function map can be used on different types, so can our more general definition of our Point and Shape types:

```
data Point a = Point a a
data Shape a = Rectangle (Point a) (Point a) | Circle (Point a) a
```

Tuples and lists can be thought of as parameterised types such as these, but with some special syntax support:

```
data [a] = [] | a : [a]
```

B.4 Type-classes

Many programming languages support ad-hoc polymorphism: the ability to specialise the same function differently for different types. C++ and Java have function overloading, for example. Haskell provides type-classes for this purpose. Here, for example, is the Show type-class[1]:

```
class Show a where
  show :: a -> String
```

This declares a new type-class called Show, parameterised by one type (here named a). This declares that any type that is a member of the type-class Show must implement one method, show, which takes one argument of that type and returns a String.

This type-class constraint can then be used in the type signatures of any functions that require a type to belong to a type-class. This is done by adding a statement of constraints before a '=>' arrow preceding the type, for example:

```
showUpperCase :: Show a => a -> String
showUpperCase x = map toUpper (show x)
```

This is one of the most useful features of Haskell's type-classes: functions can be parameterised by the type-class instance for a type.

Some of the standard Haskell type-classes, such as Show, can be derived automatically by the compiler:

[1] Two further methods of Show are omitted for clarity.

```
data DayOfWeek
  = Monday | Tuesday | Wednesday | Thursday
  | Friday | Saturday | Sunday
  deriving (Show)
```

Another standard type-class is Eq, which defines the equality operator:

```
class Eq a where
  (==) :: a -> a -> Bool
```

B.5 Monads

Monads are one of the best-known features of Haskell. Monads allow a common pattern of computation to be captured in a Haskell type-class instance. It is important in this thesis to understand the basic principles of monads, and thus an explanation is provided here.

We will motivate monads by starting with a specific example. The standard Maybe data type is defined as follows:

```
data Maybe a = Just a | Nothing
```

This is a wrapper around a type that permits the possibility of having no value. It can be thought of as similar to references in Java that either contain a value, or are null, but here the distinction is more explicit.

The Maybe type is useful for return values from functions that may or may not succeed, for example the lookup function that looks up a value in a list of pairs:

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Suppose that we have two such lists: studentNumbers :: [(Name,StudentNumber)] and examMarks :: [(ExamNumber,Mark)]. We wish to find whether a student passed (achieved at least 40) from their name, admitting the possibilities that they may not have a student number and/or an exam mark:

```
passed :: Name -> Maybe Bool
passed name = case lookup name studentNumbers of
  Just studentNumber ->
    case lookup studentNumber examMarks of
      Just mark -> Just (mark >= 40)
      Nothing -> Nothing
  Nothing -> Nothing
```

This code is very repetitive. It can be shortened with the use of a function we will call bind:

```
bind :: Maybe a -> (a -> Maybe b) -> Maybe b
bind (Just x) f = f x
bind Nothing _ = Nothing

passed :: Name -> Maybe Bool
passed name = lookup name studentNumbers `bind`
  (\studentNumber -> lookup studentNumber examMarks `bind`
    \mark -> Just (mark >= 40)
  )
```

Monads take this idea and generalise it to other parameterised types:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

Bind has now become the operator >>=, and return has been added to wrap values in the monad. There is also a simpler sequence operator where the result of the first operation does not affect the second:

```
(>>) :: Monad m => m a -> m b -> m b
(>>) a b = a >>= (\_ -> b)
```

This definition is standard Haskell – the only language support that Haskell provides for monads is a more palatable **do** notation that is desugared into our earlier code. In **do** notation, monadic actions are listed in order, with an optional "result <-" on the left-hand side to name the result of the computation. A **do** notation version of our erlier function is:

```
findMark :: Name -> Maybe ExamMark
findMark name = do studentNumber <- lookup name studentNumbers
                   mark <- lookup studentNumber examMarks
                   return (mark >= 40)
```

The bind operator composes effectful operations, by taking a value in a monadic wrapper, and feeding it into a further computation that gives back a value inside the monadic wrapper. Maybe is a short-cutting error monad; once Nothing is encountered, all further operations are ignored, and Nothing is the result of the whole computation. As well as other error monads, common uses for monads include

the emulation of mutable state, and the IO monad for sequencing operations that interact with the outside world.

It is possible to define functions for common monadic operations, that will work across all monads. We will see these functions used repeatedly throughout this thesis. For example, there is the mapM function, which is the monadic version of map:

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM _ [] = return []
mapM f (x:xs) = do x' <- f x
                   xs' <- mapM f xs
                   return (x' : xs')
```

There is also the function forever:

```
forever :: Monad m => m a -> m b
forever m = m >> forever m
```

Its return type can be set to any type – since it never returns anything, its return type (within the monadic wrapper) can be any type.

Appendix C

Select From Guards Implementation

This appendix contains the lengthy definition of selectFromGuards, which is used in chapter 3 to define the choice operator, bridging the gap to the choie-resolution algorithms explained in chapter 5. The selectFromGuards function takes a list of guards and associated actions (or "bodies"), selects one from the list according to the semantics of choice given in chapter 3 and executes the associated body:

```
data Guard = TimeoutGuard (IO (STM ()))
           | SkipGuard
           | EventGuard (STM ()) [Event]

selectFromGuards :: [(Guard, IO a)] -> IO a
```

C.1 Non-Event Guards

The majority of the complexity of this selectFromGuards function comes from mixing skip guards, timeout guards and event guards together. To begin with, we show a version that ignores event guards completely and selects from timeout guards and skip guards:

```
selectFromNonEventGuards :: [(Guard, IO a)] -> IO a
selectFromNonEventGuards items
  = do enabled <- sequence (mapMaybe enable items)
       body <- atomically (foldr orElse retry enabled)
       body
```

```
  where
    enable :: (Guard, b) -> Maybe (IO (STM b))
    enable (SkipGuard, x) = Just (return (return x))
    enable (TimeoutGuard g, x) = Just (fmap (>> return x) g)
    enable (EventGuard _ _) = Nothing
```

Skip guards have an inner STM action that simply succeeds. Timeout guards run their initialisation action g (which is in the IO monad as it may need to access the current time), and we modify the STM action to return the guard's associated value. We initialise all guards (the sequence call in the first line of the **do** block), and then perform a transaction that waits until the first guard in the list is ready.

C.2 Event Guards

The previous definition ignored event guards and showed how to select from skip and timeout guards. In this section, we show the opposite – a function that selects from event guards, ignoring timeout and skip guards. We will use these primitives defined in chapter 5:

```
newtype SignalValue = Signal Int
data OfferSet = ...

makeOfferSet :: [(SignalValue, STM (), Set Event)] -> IO OfferSet

data EnabledActions = EnabledActions
 {waitForEvent :: STM SignalValue, retractOffers :: STM (Maybe SignalValue)}

enableEvents' :: OfferSet -> STM (Either SignalValue EnabledActions)
```

To recap: makeOfferSet makes an OfferSet from a list of offers. Each offer has an associated SignalValue (really an index into the list of guards), an STM action to execute if the offer is chosen, and a set of events which must all be synchronised on in order to complete the offer. The enableEvents' function takes this OfferSet and attempts to complete an offer. If it can complete an offer immediately, it returns the SignalValue associated with the completed offer, tagged as Left. If if it cannot complete immediately, it returns a structure with an STM action that waits for an offer to complete (waitForEvent) and an STM action that will attempt to retract all the offers, but indicate whether one had just recently been chosen before the retraction could take place (retractOffers).

Here, then, is our implementation that only chooses between event guards:

```
selectFromEventGuards :: [(Guard, IO a)] -> IO a
selectFromEventGuards items
 = do result <- makeOfferSet eventGuards >>= atomically . enableEvents'
      case result of
        -- Completed immediately; execute associated body:
        Left sig -> runBody sig
        -- Offers have been made; wait for one:
        Right acts -> do
          sig <- atomically (waitForEvent acts)
          runBody sig
  where
    guards = map fst items
    runBody (Signal n) = snd (items !! n)
    eventGuards = [ (Signal n, act, fromList es)
                  | (n, EventGuard act es) <- zip [0..] guards]
```

This function is straightforward; it makes an offer-set and then offers on all the events. If one completes immediately it executes the associated body, otherwise it waits (in a second transaction, to allow other processes to see the results of the first transaction) for one to complete and *then* executes the associated body.

C.3 Mixing Event and Non-Event Guards

The previous sections have demonstrated a short function to select between non-event guards only, and a short function to select between event guards only. Unfortunately, the function to select from a mixed list of the two is longer.

To begin with, we adapt selectFromNonEventGuards into the following function:

```
waitNonEventGuards :: [(Guard, b)] -> Maybe (STM b) -> IO b
waitNonEventGuards guards extra
  = do enabled <- sequence (mapMaybe enable guards)
       atomically $ foldr orElse retry (maybe id (:) extra enabled)
  where
    enable :: (Guard, b) -> Maybe (IO (STM b))
    enable (SkipGuard, x) = Just (return (return x))
    enable (TimeoutGuard g, x) = Just (fmap (>> return x) g)
    enable (EventGuard _ _) = Nothing
```

The two differences are firstly that the guards are associated with values which are returned directly (rather than having bodies which are executed immediately), and secondly the function takes an optional extra action which will be tried before all others.

With this, we can define the selectFromGuards function as follows:

```
selectFromGuards :: [(Guard, IO a)] -> IO a
selectFromGuards items
  | null eventGuards = waitNonEvent
  | otherwise = do
      os <- makeOfferSet (maybe id take earliestReady eventGuards)
      bodyOrActs <- atomically (tryEvents os)
      case bodyOrActs of
        Left body -> body
        -- Events will have been enabled; wait for everything:
        Right acts -> waitForAll acts
  where
    tryEvents os = do
      r <- enableEvents' os
      case (r, earliestReady) of
        -- An event completed
        (Left sig, _) -> return (Left (getBody sig))
        -- No event completed, and there is a ready skip guard,
        -- so disable the events and execute the normal guards.
        (Right acts, Just _) -> do Nothing <- retractOffers acts
                                   return (Left waitNonEvent)
        -- No event completed, and no skip guards.
        (Right acts, Nothing) -> return (Right acts)

    waitForAll acts = do
      (wasEvent, body) <- waitNonEventGuards
        [(g, (False, x)) | (g, x) <- items]
        (Just (do sig <- waitForEvent acts
                  return (True, getBody sig)))
      if wasEvent
        then body -- It was an event, execute body and we're done
        else -- Another guard fired, but we must check in case
             -- we have meanwhile been committed to taking an event:
          do possEvent <- atomically (retractOffers acts)
```

```
            case possEvent of
              -- An event overrides our non-event choice:
              Just sig -> getBody sig
              -- Go with the original option, no events became ready:
              Nothing -> body

    guards = map fst items
    getBody (Signal n) = snd $ items !! n
    eventGuards = [ (Signal n, act, fromList es)
                  | (n, EventGuard act es) <- zip [0..] guards]
    earliestReady = findIndex isSkipGuard guards
      where
        isSkipGuard SkipGuard = True
        isSkipGuard _ = False
    waitNonEvent = do body <- waitNonEventGuards items Nothing
                      body
```

The beginning of this function checks to see if there are any guards involving events. If not (it is just a choice between skip, stop and timeout guards) then it just waits for these non-event guards and executes the associated body, using waitNonEvent that is defined at the end of the **where** clause.

If there are event guards, we begin by scanning through the list of guards (see earliestReady) for the first skip guard, as we do not want to look at any event guards beyond the earliest definitely-ready (skip) guard. This is not merely an optimisation; our semantics require that a skip guard takes precedence over all guards later in the list.

We then make our offers on the events (at the start of tryEvents) and see if any can complete immediately. Recall that enableEvents' returns a Left item if an event completed immediately; in this case we simply execute the body associated with the guard (after the transaction). If no event was immediately available, but there was a later skip guard ready, we disable all the events (removing our offers to synchronise on them) and execute the non-event guards after the transaction. We know that if we retract the offers in the same transaction as we make the offer, it is impossible that one has become ready mid-transaction (due to the atomicity guarantee on STM transactions).

The third case is that no guards (event or non-event) are yet ready; in this case (see waitForAll) we wait for the non-event guards or an event guard to be ready. If this wait is ended by an event guard, we execute the associated body.

If it is ended by a non-event guard (which must be a timeout guard), we check to see if an event guard fired at almost exactly the same time. If it did, we take the event instead of the timeout (this is permitted by the semantics, and is not obvious to the user as it means the event must have fired almost simultaneously with the timeout), because it is not possible to backtrack on our offer to engage the event after it has been accepted.

Appendix D

Full Semantics and API

This appendix gathers together the final full semantics for CHP (including changes required to accommodate poison), and the core API, for reference.

D.1 Complete API

In this section we aggregate the core API from the previous sections.

D.1.1 Core

```
data CHP a

instance Monad CHP
instance Functor CHP
instance Applicative CHP

runCHP :: CHP a -> IO a

liftIO_CHP :: IO a -> CHP a
```

D.1.2 Concurrency

```
parallel  :: [CHP a] -> CHP [a]
parallel_ :: [CHP a] -> CHP ()
(<||>) :: CHP a -> CHP b -> CHP (a, b)
```

D.1.3 Choice and Conjunction

```
instance Alternative CHP where
  (<|>) :: CHP a -> CHP a -> CHP a
  empty :: CHP a

alt :: [CHP a] -> CHP a

stop :: CHP a
skip :: CHP ()

(<&>) :: CHP a -> CHP b -> CHP (a, b)
every :: [CHP a] -> CHP [a]
```

D.1.4 Barriers and Enrolling

```
data PhasedBarrier a
data Unenrolled c a

newPhasedBarrier' :: [phase] -> CHP (Unenrolled PhasedBarrier phase)
syncBarrier :: PhasedBarrier phase -> CHP phase
currentPhase :: PhasedBarrier phase -> CHP phase

class Enrollable b where
  enroll :: Unenrolled b p -> (b p -> CHP a) -> CHP a

instance Enrollable PhasedBarrier
instance Enrollable Chanin
instance Enrollable Chanout
```

D.1.5 Channels

```
data Chanin a
data Chanout a

readChannel :: Chanin a -> CHP a
writeChannel :: Chanout a -> a -> CHP ()

class Channel r w where
  newChannelRW :: CHP (r a, w a)

instance Channel Chanin Chanout -- one-to-one
```

```
instance Channel (Shared Chanin) Chanout -- one-to-any
instance Channel Chanin (Shared Chanout) -- any-to-one
instance Channel (Shared Chanin) (Shared Chanout) -- any-to-any
instance Channel (Unenrolled Chanin) Chanout -- one-to-many
instance Channel (Unenrolled Chanin) (Shared Chanout) --any-to-many
instance Channel Chanin (Unenrolled Chanout) --many-to-one
instance Channel (Shared Chanin) (Unenrolled Chanout) --many-to-any
instance Channel (Unenrolled Chanin) (Unenrolled Chanout) --many-to-many

data Shared c a
claim :: Shared c a -> (c a -> CHP b) -> CHP b
```

D.1.6 Poison

```
onPoisonTrap :: CHP a -> CHP a -> CHP a
onPoisonRethrow :: CHP a -> CHP b -> CHP a

throwPoison :: CHP a

class Poisonable a where
  poison :: a -> CHP ()

instance Poisonable (Chanin a)
instance Poisonable (Chanout a)
instance Poisonable (PhasedBarrier phase)

finallyCHP :: CHP a -> CHP b -> CHP a
```

D.2 Semantics

The final operational semantics for CHP (including poison) are gathered together into figures 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62 and 63.

$$
\begin{array}{rrcl}
 & b, c, x, y & \in & \textit{Variable} \\
\text{Values} & V & ::= & \backslash x \to M \mid l \mid con\ M_1 \ldots M_n \mid \texttt{return}\ M \mid M \texttt{>>=}\ N \\
 & & & \mid \texttt{readChannel}\ c \mid \texttt{extReadChannel}\ c\ M \mid \texttt{writeChannel}\ c\ x \\
 & & & \mid \texttt{extWriteChannel}\ c\ M\ x \mid \texttt{endComm}\ c \mid \texttt{syncBarrier}\ b \\
 & & & \mid \texttt{currentPhase}\ b \mid \texttt{enroll}\ b\ M \mid \texttt{unenroll}\ b \\
 & & & \mid \texttt{liftIO\_CHP}\ M \mid M\ \texttt{<||>}\ N \mid x \sqcup y \mid \texttt{skip} \mid \texttt{stop} \\
 & & & \mid M\ \texttt{<|>}\ N \mid M\ \texttt{<\&>}\ N \mid \texttt{newChannelRW} \mid \texttt{claim}\ c \\
 & & & \mid \texttt{newPhasedBarrier'}\ x \mid \texttt{throwPoison} \mid \texttt{poison}\ x \\
\text{Terms} & M, N, O & ::= & x \mid V \mid M\ N \mid \ldots \\
\text{Evaluation Contexts} & \mathbb{E} & ::= & [\cdot] \mid \mathbb{E} \texttt{>>=}\ M \mid \texttt{finallyCHP}\ \mathbb{E}\ M \mid \texttt{onPoisonTrap}\ \mathbb{E}\ M
\end{array}
$$

Figure 50: The final syntax of values and terms.

$$\{\mathbb{E}[\texttt{return N >>= M}]\} \Rightarrow \{\mathbb{E}[\texttt{M N}]\} \qquad \text{(LUNIT)}$$

$$\{\mathbb{E}[\texttt{M >>= return}]\} \Rightarrow \{\mathbb{E}[\texttt{M}]\} \qquad \text{(RUNIT)}$$

$$\frac{x \notin fn(N) \cup fn(O)}{\{\mathbb{E}[\texttt{(M >>= N) >>= O}]\} \Rightarrow \{\mathbb{E}[\texttt{M >>= (\x -> N x >>= O)}]\}} \qquad \text{(BIND)}$$

$$\frac{\varepsilon[M] = V \qquad M \not\equiv V}{\{\mathbb{E}[M]\} \Rightarrow \{\mathbb{E}[V]\}} \qquad \text{(FUN)}$$

Figure 51: The final basic transition rules for monadic and functional code.

$$P \parallel Q \equiv Q \parallel P \qquad \text{(COMMUTE)}$$

$$P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R \qquad \text{(ASSOC)}$$

$$\nu x.\nu y.P \equiv \nu y.\nu x.P \qquad \text{(SWAP)}$$

$$(\nu x.P) \parallel Q \equiv \nu x.(P \parallel Q), \quad x \notin fn(Q) \qquad \text{(EXTRUDE)}$$

$$\nu x.P \equiv \nu y.P[y/x], \quad y \notin fn(P) \qquad \text{(ALPHA)}$$

$$\frac{P \xrightarrow{\alpha} Q}{P \parallel R \xrightarrow{\alpha} Q \parallel R} \qquad \text{(PAR)}$$

$$\frac{P \xrightarrow{\alpha} Q \qquad x \notin \alpha}{\nu x.P \xrightarrow{\alpha} \nu x.Q} \qquad \text{(NU)}$$

$$\frac{P \equiv P' \qquad P' \xrightarrow{\alpha} Q' \qquad Q \equiv Q'}{P \xrightarrow{\alpha} Q} \qquad \text{(EQUIV-L)}$$

$$\frac{P \equiv P' \qquad P' \Rightarrow Q' \qquad Q \equiv Q'}{P \Rightarrow Q} \qquad \text{(EQUIV-F)}$$

$$\frac{P \Rightarrow P' \qquad P' \xrightarrow{\alpha} Q}{P \xrightarrow{\alpha} Q} \qquad \text{(FREE)}$$

$$\frac{P \Rightarrow Q \qquad Q \Rightarrow R}{P \Rightarrow R} \qquad \text{(FTRANS)}$$

Figure 52: Final structural congruences and structural transitions.

$$\frac{u \notin fn(\mathbb{E}) \qquad v \notin fn(\mathbb{E}) \qquad u \neq v}{\{\mathbb{E}[M \texttt{<||>} N]\}_t \xrightarrow{\{\}} \nu u.\nu v.(\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{M\}_u \mid\mid \{N\}_v)} \quad \text{(S-PAR)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{\texttt{return x}\}_u \mid\mid \{\texttt{return y}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{return (x, y)}]\}_t \quad \text{(E-PAR-RR)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{\texttt{throw e}\}_u \mid\mid \{\texttt{return y}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw e}]\}_t \quad \text{(E-PAR-TR)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{\texttt{return x}\}_u \mid\mid \{\texttt{throw e}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw e}]\}_t \quad \text{(E-PAR-RT)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{\texttt{throw e}\}_u \mid\mid \{\texttt{throw f}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw e}]\}_t \quad \text{(E-PAR-TT-L)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{\texttt{throw e}\}_u \mid\mid \{\texttt{throw f}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw f}]\}_t \quad \text{(E-PAR-TT-R)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{\texttt{return x}\}_u \mid\mid \{\texttt{throwPoison}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throwPoison}]\}_t \quad \text{(E-PAR-RP)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{\texttt{throwPoison}\}_u \mid\mid \{\texttt{return y}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throwPoison}]\}_t \quad \text{(E-PAR-PR)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{\texttt{throwPoison}\}_u \mid\mid \{\texttt{throwPoison}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throwPoison}]\}_t \quad \text{(E-PAR-PP)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{\texttt{throw e}\}_u \mid\mid \{\texttt{throwPoison}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw e}]\}_t \quad \text{(E-PAR-TP)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid \{\texttt{throwPoison}\}_u \mid\mid \{\texttt{throw e}\}_v \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throw e}]\}_t \quad \text{(E-PAR-PT)}$$

$$\{\mathbb{E}[u \sqcup v]\}_t \mid\mid [t \lightning e] \xrightarrow{\{\}} \{\mathbb{E}[u \sqcup v]\}_t \mid\mid [u \lightning e] \mid\mid [v \lightning e] \quad \text{(PAR-ASYNC)}$$

Figure 53: Operational semantics for parallel composition. The $\sqcup$ notation is invented purely for describing the semantics, and is not visible to the programmer. Note that, unlike the Haskell function forkIO, the thread identifiers are not made visible to the programmer.

$$\{\mathbb{E}[\texttt{skip}]\} \xrightarrow{\{\}} \{\mathbb{E}[\texttt{return ()}]\} \quad \text{(SKIP)}$$

$$\{\mathbb{E}[\texttt{return x <|> N}]\} \xrightarrow{\{\}} \{\mathbb{E}[\texttt{return x}]\} \quad \text{(CHLRET)}$$

$$\frac{\{M\} \not\rightarrow \{M'\} \quad M \not\equiv (\texttt{A <|> B})}{\{\mathbb{E}[\texttt{(M <|> return x)}]\} \xrightarrow{\{\}} \{\mathbb{E}[\texttt{return x}]\}} \quad \text{(CHRRET)}$$

$$\texttt{(M <|> N) <|> O} \Rightarrow \texttt{M <|> (N <|> O)} \quad \text{(CHASSOC)}$$

$$\frac{\{M\} \xrightarrow{\alpha} \{M'\} \quad M \not\equiv (\texttt{A <|> B})}{\{\mathbb{E}[\texttt{M <|> N}]\} \xrightarrow{\alpha} \{\mathbb{E}[\texttt{M'}]\}} \quad \text{(CHLEFT)}$$

$$\frac{((\{M\} \not\rightarrow \{M'\} \wedge \{N\} \xrightarrow{\alpha} \{N'\}) \vee (\{M\} \overset{\{\}}{\not\rightarrow} \{M'\} \wedge \{N\} \xrightarrow{\alpha} \{N'\} \wedge \alpha \neq \{\})) \quad M \not\equiv (\texttt{A <|> B})}{\{\mathbb{E}[\texttt{M <|> N}]\} \xrightarrow{\alpha} \{\mathbb{E}[\texttt{N'}]\}} \quad \text{(CHRIGHT)}$$

$$\frac{\{M\} \xrightarrow{\alpha} \{M'\} \quad \{N\} \xrightarrow{\beta} \{N'\}}{\{\mathbb{E}[M \texttt{ <\&> } N]\} \xrightarrow{\alpha \cup \beta} \{\mathbb{E}[M' \texttt{ <||> } N']\}} \quad \text{(CONJ)}$$

Figure 54: The semantics for the choice and conjunction operators. The condition $M \not\equiv$ (A <|> B) on many of the rules forces the choice to be rearranged into a right-associative form (via CHASSOC) before proceeding. The condition $\{M\} \not\rightarrow \{M'\}$ is only satisfied when M cannot make any transition, while $\{M\} \overset{\{\}}{\not\rightarrow} \{M'\}$ is satisfied when M can make a transition with a non-empty set of events or M cannot make any transition.

$$\frac{c \notin fn(\mathbb{E})}{\{\mathbb{E}[\texttt{newChannelRW :: CHP (Chanin a, Chanout a)}]\} \xrightarrow{\{\}} \nu c.\{\mathbb{E}[\texttt{return } (c_R,\ c_W)]\}} \text{(NEW-CHANNEL)}$$

$$\frac{c \notin fn(\mathbb{E})}{\{\mathbb{E}[\texttt{newChannelRW :: CHP (Shared Chanin a, Chanout a)}]\}_t \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return } (c_R,\ c_W)]\}_t \mid\mid \langle c_R \rangle)} \text{(NEW-CHAN-SRW)}$$

$$\frac{c \notin fn(\mathbb{E})}{\{\mathbb{E}[\texttt{newChannelRW :: CHP (Chanin a, Shared Chanout a)}]\}_t \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return } (c_R,\ c_W)]\}_t \mid\mid \langle c_W \rangle)} \text{(NEW-CHAN-RSW)}$$

$$\frac{c \notin fn(\mathbb{E})}{\{\mathbb{E}[\texttt{newChannelRW :: CHP (Shared Chanin a, Shared Chanout a)}]\}_t \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return } (c_R,\ c_W)]\}_t \mid\mid \langle c_R \rangle \mid\mid \langle c_W \rangle)} \text{(NEW-CHAN-SRSW)}$$

$$\frac{b \notin (keys(B) \cup fn(\mathbb{E}))}{\{\mathbb{E}[\texttt{newPhasedBarrier' } h]\} \mid\mid \beta(B) \xrightarrow{\{\}} \nu b.\{\mathbb{E}[\texttt{return } b_U]\} \mid\mid \beta(B \triangleleft \{b \mapsto (0, h)\})} \text{(NEW-BP)}$$

$$\frac{c \notin (keys(C) \cup fn(\mathbb{E}))}{\{\mathbb{E}[\texttt{newChannelRW :: CHP (Unenrolled Chanin a, Chanout a)}]\} \mid\mid \gamma(C) \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return } (c_{UR}, c_W)]\} \mid\mid \gamma(C \triangleleft \{c \mapsto 0\}))} \text{(NEW-BROAD)}$$

$$\frac{c \notin (keys(C) \cup fn(\mathbb{E}))}{\{\mathbb{E}[\texttt{newChannelRW :: CHP (Chanin a, Unenrolled Chanout a)}]\} \mid\mid \gamma(C) \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return } (c_R, c_{UW})]\} \mid\mid \gamma(C \triangleleft \{c \mapsto 0\}))} \text{(NEW-REDUCE)}$$

Figure 55: Final operational semantics for channel and barrier creation.

$$\frac{c \in P}{\nu c.(\{\mathbb{E}[\texttt{readChannel c}]\} \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{throwPoison}]\} \mid\mid \Omega(P))} \text{(READ-P)}$$

$$\frac{c \in P}{\nu c.(\{\mathbb{E}[\texttt{writeChannel c}]\} \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{throwPoison}]\} \mid\mid \Omega(P))} \text{(WRITE-P)}$$

$$\frac{c \in P}{\nu c.(\{\mathbb{E}[\texttt{extReadChannel c m}]\} \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{throwPoison}]\} \mid\mid \Omega(P))} \text{(EXT-READ-P)}$$

$$\frac{c \in P}{\nu c.(\{\mathbb{E}[\texttt{extWriteChannel c n}]\} \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{throwPoison}]\} \mid\mid \Omega(P))} \text{(EXT-WRITE-P)}$$

$$\frac{c \in P}{\nu c.(\{\mathbb{E}[\texttt{endComm c}]\} \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{throwPoison}]\} \mid\mid \Omega(P))} \text{(END-COMM-P)}$$

Figure 56: The final semantics for communicating on poisoned channels.

$$\frac{b \in P}{\nu b.(\{\mathbb{E}[\texttt{syncBarrier b}]\} \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\texttt{throwPoison}]\} \mid\mid \Omega(P))} \quad \text{(SYNC-P)}$$

$$\frac{b \in P}{\nu b.(\{\mathbb{E}[\texttt{currentPhase b}]\} \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\texttt{throwPoison}]\} \mid\mid \Omega(P))} \quad \text{(CUR-P)}$$

$$\frac{x \in P}{\nu x.(\{\mathbb{E}[\texttt{enroll x M}]\} \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu x.(\{\mathbb{E}[\texttt{throwPoison}]\} \mid\mid \Omega(P))} \quad \text{(ENROLL-P)}$$

$$\frac{x \in P}{\nu x.(\{\mathbb{E}[\texttt{unenroll x}]\} \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu x.(\{\mathbb{E}[\texttt{throwPoison}]\} \mid\mid \Omega(P))} \quad \text{(UNENROLL-P)}$$

Figure 57: The final semantics for enrolling, unenrolling and synchronising on barriers in the presence of poison.

$$(\{\mathbb{E}[\texttt{claim x m}]\}_t \mid\mid \langle x \rangle) \xrightarrow{\{\}} \{\mathbb{E}[\texttt{m x `finallyCHP` release x}]\}_t \quad \text{(CLAIM)}$$

$$\{\mathbb{E}[\texttt{release x}]\}_t \xrightarrow{\{\}} (\{\mathbb{E}[\texttt{return ()}]\}_t \mid\mid \langle x \rangle) \quad \text{(RELEASE)}$$

Figure 58: The final operational semantics for claiming and releasing shared channels.

$$\frac{c \notin P}{\nu c.(\{\mathbb{E}_1[\texttt{readChannel}\ \ c_R]\}_t \mid\mid \{\mathbb{E}_2[\texttt{writeChannel}\ \ c_W\ \ \texttt{x}]\}_u \mid\mid \Omega(P)) \xrightarrow{\{c.x\}} \nu c.(\{\mathbb{E}_1[\texttt{return x}]\}_t \mid\mid \{\mathbb{E}_2[\texttt{return ()}]\}_u \mid\mid \Omega(P))} \text{(BASIC-COMM-P)}$$

$$\frac{(b \mapsto (k,h)) \in B \quad b \notin P}{\nu b.(\{\mathbb{E}[\texttt{enroll}\ \ b_U\ \ \texttt{M}]\} \mid\mid \beta(B) \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\texttt{M}\ \ b_E\ \ \texttt{`finallyCHP`}\ \ \texttt{unenroll}\ \ b_E]\} \mid\mid \beta(B \triangleleft \{b \mapsto (k+1,h)\}) \mid\mid \Omega(P))} \text{(ENROLL-BP-P)}$$

$$\frac{(b \mapsto (k,h)) \in B \quad b \notin P}{\nu b.(\{\mathbb{E}[\texttt{unenroll}\ \ b_E]\} \mid\mid \beta(B) \mid\mid \Omega(P)) \xrightarrow{\{\}} \{\mathbb{E}[\texttt{return ()}]\} \mid\mid \beta(B \triangleleft \{b \mapsto (k-1,h)\})} \text{(UNENROLL-BP-P)}$$

$$\frac{(b \mapsto (k, o : h : h')) \in B \quad b \notin P}{\nu b.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{syncBarrier}\ \ b_E]\}_{t_i} \mid i \in \{1..k\}\} \mid\mid \beta(B) \mid\mid \Omega(P)) \xrightarrow{\{b\}} \nu b.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{return}\ \ h]\}_{t_i} \mid i \in \{1..k\}\} \mid\mid \beta(B \triangleleft \{b \mapsto (k, h : h')\}) \mid\mid \Omega(P))} \text{(SYNC-BP-P)}$$

$$\frac{(b \mapsto (k, h : h')) \in B \quad b \notin P}{\nu b.(\{\mathbb{E}[\texttt{currentPhase}\ \ b_E]\} \mid\mid \beta(B) \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu b.(\{\mathbb{E}[\texttt{return}\ \ h]\} \mid\mid \beta(B) \mid\mid \Omega(P))} \text{(CUR-BP-P)}$$

Figure 59: The final operational semantics for communication on normal channels and enrolling/synchronising on barriers.

$$\frac{c \notin P}{\{\mathbb{E}_1[\texttt{(extReadChannel } c_R \texttt{ M)}]\}_t \mid\mid \{\mathbb{E}_2[\texttt{(extWriteChannel } c_W \texttt{ N)}]\}_u \xrightarrow{\{\tau\}} \{\mathbb{E}_1[\texttt{(readChannel } c_R \texttt{ >>= M) `finallyCHP` endComm } c]\}_t \mid\mid \{\mathbb{E}_2[\texttt{(N >>= writeChannel } c_W\texttt{) `finallyCHP` endComm } c]\}_u} \quad \text{(EXT-COMM-P)}$$

$$\frac{c \notin P}{\{\mathbb{E}_1[\texttt{(readChannel } c_R\texttt{)}]\}_t \mid\mid \{\mathbb{E}_2[\texttt{(extWriteChannel } c_W \texttt{ N)}]\}_u \xrightarrow{\{\tau\}} \{\mathbb{E}_1[\texttt{readChannel } c_R]\}_t \mid\mid \{\mathbb{E}_2[\texttt{N >>= writeChannel } c_W]\}_u} \quad \text{(EXT-COMM-W-P)}$$

$$\frac{c \notin P}{\{\mathbb{E}_1[\texttt{(extReadChannel } c_R \texttt{ M)}]\}_t \mid\mid \{\mathbb{E}_2[\texttt{(writeChannel } c_W\texttt{)}]\}_u \xrightarrow{\{\tau\}} \{\mathbb{E}_1[\texttt{(readChannel } c_R \texttt{ >>= M) `finallyCHP` endComm } c]\}_t \mid\mid \{\mathbb{E}_2[\texttt{writeChannel } c_W \texttt{ `finallyCHP` endComm } c]\}_u} \quad \text{(EXT-COMM-R-P)}$$

$$\frac{c \notin P}{\{\mathbb{E}_1[\texttt{endComm c}]\}_t \mid\mid \{\mathbb{E}_1[\texttt{endComm c}]\}_u \xrightarrow{\{\}} \{\mathbb{E}_2[\texttt{return ()}]\}_t \mid\mid \{\mathbb{E}_2[\texttt{return ()}]\}_u} \quad \text{(END-COMM2-P)}$$

Figure 60: The final operational semantics for extended communication on channels.

$$\frac{(c \mapsto k) \in C \quad c \notin P}{\begin{array}{c}\nu c.(\{\mathbb{E}[\texttt{enroll}\ \ c_U\ \ \texttt{M}]\} \mid\mid \gamma(C) \mid\mid \Omega(P)) \\ \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{M}\ \ c_E\ \ \texttt{`finallyCHP`}\ \ \texttt{unenroll}\ \ c]\} \mid\mid \gamma(C \triangleleft \{c \mapsto k+1\}) \mid\mid \Omega(P))\end{array}} \text{(ENROLL-BR-P)}$$

$$\frac{(c \mapsto k) \in C \quad c \notin P}{\begin{array}{c}\nu c.(\{\mathbb{E}[\texttt{unenroll}\ \ c]\} \mid\mid \gamma(C) \mid\mid \Omega(P)) \\ \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return}\ \ \texttt{()}]\} \mid\mid \gamma(C \triangleleft \{c \mapsto k-1\}) \mid\mid \Omega(P))\end{array}} \text{(UNENROLL-BR-P)}$$

$$\frac{(c \mapsto k) \in C \quad c \notin P}{\begin{array}{c}\nu c.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{readChannel}\ \ c]\}_{t_i} \mid i \in \{1..k\}\} \mid\mid \{\mathbb{E}_0[\texttt{writeChannel c x}]\}_u \mid\mid \gamma(C) \mid\mid \Omega(P)) \\ \xrightarrow{\{c\}} \nu c.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{return x}]\}_{t_i} \mid i \in \{1..k\}\} \mid\mid \{\mathbb{E}_0[\texttt{return ()}]\}_u \mid\mid \gamma(C) \mid\mid \Omega(P))\end{array}} \text{(COMM-BROAD-P)}$$

$$\frac{(c \mapsto k) \in C \quad c \notin P}{\begin{array}{c}\nu c.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{writeChannel c}\ x_i]\}_{t_i} \mid i \in \{1..k\}\} \mid\mid \{\mathbb{E}_0[\texttt{readChannel c}]\}_u \mid\mid \gamma(C) \mid\mid \Omega(P)) \\ \xrightarrow{\{c\}} \nu c.(\Big|\Big|\{\{\mathbb{E}_i[\texttt{return ()}]\}_{t_i} \mid i \in \{1..k\}\} \mid\mid \{\mathbb{E}_0[\texttt{return (combine}\ x_1 \ldots x_k\texttt{)}]\}_u \mid\mid \gamma(C) \mid\mid \Omega(P))\end{array}} \text{(COMM-REDUCE-P)}$$

Figure 61: The final operational semantics for communication on broadcast and reduce channels.

$$\{\mathbb{E}[\texttt{throwPoison} >>= M]\} \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throwPoison}]\} \qquad \text{(BIND-POISON)}$$

$$\{\mathbb{E}[\texttt{throwPoison `onPoisonTrap` M}]\} \xrightarrow{\{\}} \{\mathbb{E}[M]\} \qquad \text{(POISON-TRAP)}$$

$$\nu c.(\{\mathbb{E}[\texttt{poison c}]\} \mid\mid \Omega(P)) \xrightarrow{\{\}} \nu c.(\{\mathbb{E}[\texttt{return ()}]\} \mid\mid \Omega(P \cup \{c\})) \qquad \text{(POISON)}$$

$$\{\mathbb{E}[\texttt{throwPoison <|> N}]\} \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throwPoison}]\} \qquad \text{(CHLPOIS)}$$

$$\frac{\{M\} \not\rightarrow \{M'\} \quad M \not\equiv (\texttt{A <|> B})}{\{\mathbb{E}[(\texttt{M <|> throwPoison})]\} \xrightarrow{\{\}} \{\mathbb{E}[\texttt{throwPoison}]\}} \qquad \text{(CHRPOIS)}$$

Figure 62: The final operational semantics for poison.

$$\mathsf{runCHP}\ \mathsf{p} \equiv \{\mathsf{p}\} \mid\mid \beta(\{\}) \mid\mid \gamma(\{\}) \mid\mid \Omega(\{\})$$

Figure 63: The final operational semantics for the top-level runCHP call.

Appendix E

Example Proofs

In this appendix we give a few examples of using the semantics of CHP (given in appendix D) to prove some of the laws of CHP given in section 3.10.

E.1 Notation

Here is an example of the notation we will use for exploring the transitions a piece of code can make:

$$\text{by SKIP} \quad \xrightarrow{\{\}} \quad \begin{array}{l} \mathsf{skip} \\ \mathsf{return}\ () \end{array}$$

On the left we state which rule was used. Rules that are transitions appear with a transition arrow ($\xrightarrow{\{c\}}$). Rules that use equivalences ($\equiv$) or structural transitions ($\Rightarrow$) have no such arrow, and are implicitly applied using the rules EQUIV-L, EQUIV-R, FREE or TRANS (given in figure 52) which allows them to be chained with the next standard transition. Similarly, we use PAR and NU automatically, to apply transitions to a process inside a parallel composition or ν scope. We use ***bold italic*** headers whenever different possibilities are being explored.

E.2 Equality

We take equality of entire systems (i.e. those within a runCHP call) to mean equality of traces (where a trace is a sequential list of transition event-sets), after filtering out all τ events and then all empty sets of events. The CSP calculus has

further notions of equating failures and divergences [Roscoe, 1997], but we do not explore those here. Proper consideration of the semantics of a full program would probably instead equate the sequences of IO actions which might be performed, but we focus here on the semantics of the CHP aspects of a program.

Although trace equality is sufficient for entire systems, in this appendix we will prove properties of particular small parts of a program. Equality in this setting (notated as $\cong$) requires checking the aforementioned trace equality, but also checking that the unfiltered traces have the same initial transition. This is necessary to make sure that if the program part is composed in a choice (a situation where the range of transitions each choice can make affect which rules can fire) the behaviour is the same.

As an example of an equality that holds, consider:

```
skip >> skip ≅ skip
```

The left-hand side has the following behaviour:

```
                                skip >> skip
by definition of (>>)           skip >>= \_ -> skip
by SKIP              --{}-->    return () >>= \_ -> skip
by LUNIT                        (\_ -> skip) ()
by FUN                          skip
by SKIP              --{}-->    return ()
```

The right-hand side is simply:

```
                    skip
by SKIP  --{}-->    return ()
```

The initial transition is the same for both (the empty set: $\{\}$), and ignoring empty set transitions, the sequence of labels is the same, so the equality holds. Now consider instead the false equality claim that:

```
skip >>= writeChannel c ≅ writeChannel c ()
```

The left-hand side has the following behaviour:

```
                                skip >>= writeChannel c
by SKIP              --{}-->    return () >>= writeChannel c
by LUNIT                        writeChannel c ()
If there is a reader:
by BASIC-COMM        --{c}-->   return ()
```

The right-hand side is simply:

writeChannel c ()

If there is a reader:

by BASIC-COMM $\xrightarrow{\{c\}}$ return ()

Their filtered traces are the same, but the unfiltered traces begin with a differently labelled transition, which reveals the difference between the two sides: skip >>= writeChannel c will be always ready in a choice, but the readiness of writeChannel c () depends on the reader being ready.

E.3 Parallel Composition

We will prove the rule: p <|> skip $\cong$ skip >> p. We begin by considering the behaviour of the left-hand side[1]:

p <|> skip

by S-PAR $\xrightarrow{\{\}}$ $\nu u.\nu v.(\{u \sqcup v]\}_t \mid\mid \{\mathsf{p}\}_u \mid\mid \{\mathsf{skip}\}_v)$

by SKIP $\xrightarrow{\{\}}$ $\nu u.\nu v.(\{u \sqcup v]\}_t \mid\mid \{\mathsf{p}\}_u \mid\mid \{\mathsf{return}\ ()\}_v)$

If p $\xrightarrow{\alpha}$ **return ()*:***

$\xrightarrow{\alpha}$ $\nu u.\nu v.(\{u \sqcup v]\}_t \mid\mid \{\mathsf{return}\ ()\}_u \mid\mid \{\mathsf{return}\ ()\}_v)$

by E-PAR-RR $\xrightarrow{\{\}}$ return ((), ())

If p $\xrightarrow{\alpha}$ **throw e*:***

$\xrightarrow{\alpha}$ $\nu u.\nu v.(\{u \sqcup v]\}_t \mid\mid \{\mathsf{throw\ e}\}_u \mid\mid \{\mathsf{return}\ ()\}_v)$

by E-PAR-TR $\xrightarrow{\{\}}$ throw e

If p $\xrightarrow{\alpha}$ **throwPoison*:***

$\xrightarrow{\alpha}$ $\nu u.\nu v.(\{u \sqcup v]\}_t \mid\mid \{\mathsf{throwPoison}\}_u \mid\mid \{\mathsf{return}\ ()\}_v)$

by E-PAR-PR $\xrightarrow{\{\}}$ throwPoison

If thread t ***receives an asynchronous exception:***

$\nu u.\nu v.(\{u \sqcup v]\}_t \mid\mid \{\mathsf{p}\}_u \mid\mid \{\mathsf{return}\ ()\}_v) \mid\mid [t \lightning e]$

by PAR-ASYNC $\xrightarrow{\{\}}$ $\nu u.\nu v.(\{u \sqcup v]\}_t \mid\mid \{\mathsf{p}\}_u \mid\mid \{\mathsf{return}\ ()\}_v) \mid\mid [u \lightning e] \mid\mid [v \lightning e]$

by IO semantics $\nu u.\nu v.(\{u \sqcup v]\}_t \mid\mid \{\mathsf{throw\ e}\}_u \mid\mid \{\mathsf{throw\ e}\}_v)$

by E-PAR-TT-L $\xrightarrow{\{\}}$ throw e

or E-PAR-TT-R

[1] We show the semantics for when the right-hand side skip branch of the parallel composition makes the transition first – the results are identical if it makes the transition last, but we do not repeat the full reasoning again for this case.

We now consider the right-hand side:

		`skip >> p`
by definition of (`>>`)		`skip >>= \_ -> p`
by SKIP	$\xrightarrow{\{\}}$	`return () >>= \_ -> p`
by LUNIT		`(\_ -> p) ()`
by FUN		`p`
If `p` $\xrightarrow{\alpha}$ **`return ()`*:***		
	$\xrightarrow{\alpha}$	`return ()`
If `p` $\xrightarrow{\alpha}$ **`throw e`*:***		
	$\xrightarrow{\alpha}$	`throw e`
If `p` $\xrightarrow{\alpha}$ **`throwPoison`*:***		
	$\xrightarrow{\alpha}$	`throwPoison`
If thread t ***receives an asynchronous exception:***		
		$\{\ \texttt{p}\ \}_t \mid\mid [t \lightning e]$
by IO semantics		`throw e`

Examination of each of the cases reveals that the semantics are equal in each case. (Recall from section 3.10 that our laws assume that the user processes do not trap or indefinitely mask asynchronous exceptions, hence the transitions from the asynchronous exceptions directly to `throw e`.)

E.4 Choice

We will prove the rule: `skip <|> p` $\cong$ `skip`. We first consider the left-hand side:

		`skip <|> p`
by SKIP and CHLEFT:	$\xrightarrow{\{\}}$	`return ()`

In a way this seems trivial. The key to this law is not which transition *can* fire, but rather all the transitions that *cannot* fire, no matter what `p` is. If `p = return x`, rule CHRRET cannot fire because its pre-condition is not satisfied (the left-hand side *can* make a transition). Equally, if `p = throwPoison`, rule CHRPOISON cannot fire because its pre-condition is not satisfied. Finally, rule CHRIGHT cannot fire because its pre-condition is again not satisifed – the left-hand side being able to make a $\xrightarrow{\{\}}$ transition invalidates CHRIGHT's pre-condition.

The right-hand side of our rule is then trivially equivalent to the left:

		`skip`
by SKIP	$\xrightarrow{\{\}}$	`return ()`

E.5 Conjunction

We will prove the rule: (skip >> p) <&> (skip >> q) ≅ p <||> q. We consider the left-hand side:

$$(\mathsf{skip} \gg \mathsf{p}) \mathbin{<\&>} (\mathsf{skip} \gg \mathsf{q})$$

by SKIP, SKIP and CONJ $\xrightarrow{\{\}}$ $\mathsf{p} \mathbin{<||>} \mathsf{q}$

We have thus rearranged the left-hand side to the right-hand side. The extra observations needed to prove equality are as follows. The filtered traces must be equivalent, since our application of CONJ did not add anything to the filtered trace. The first transition of the left-hand side is $\xrightarrow{\{\}}$, and since the only rule that can fire on the right-hand side is S-PAR, which also has a $\xrightarrow{\{\}}$ transition, the initial transitions are also the same. Therefore the equality holds.

E.6 Choice and Poison

This section is not a proof of equality, but rather explores the different behaviours involving poisoned channels. We consider the choice:

writeChannel c () <|> writeChannel d ()

This behaviour can transition under eight different conditions:

1. Both channels are unpoisoned, and there is only an available reader for c.
2. Both channels are unpoisoned, and there is only an available reader for d.
3. Both channels are unpoisoned, and there are available readers for c and d.
4. Channel c is poisoned and there is no available reader for d.
5. Channel c is poisoned and there is an available reader for d.
6. Channel d is poisoned and there is no available reader for c.
7. Channel d is poisoned and there is an available reader for c.
8. Both channels are poisoned.

There cannot be an available reader for a poisoned channel, because once the channel is poisoned no communication can take place (this is formalised by the precondition on the BASIC-COMM-P rule). We explore each of the above conditions below:

writeChannel c ()
<|> writeChannel d ()

Both unpoisoned, reader for c:

by CHLEFT and BASIC-COMM-P $\xrightarrow{\{c\}}$ return ()

Both unpoisoned, reader for d:

by CHRIGHT and BASIC-COMM-P $\xrightarrow{\{d\}}$ return ()

Both unpoisoned, reader for c and d, option 1:

by CHLEFT and BASIC-COMM-P $\xrightarrow{\{c\}}$ return ()

Both unpoisoned, reader for c and d, option 2:

by CHRIGHT and BASIC-COMM-P $\xrightarrow{\{d\}}$ return ()

c is poisoned, no reader for d:

by CHLEFT and WRITE-P $\xrightarrow{\{\}}$ throwPoison

c is poisoned, reader for d:

by CHLEFT and WRITE-P $\xrightarrow{\{\}}$ throwPoison

d is poisoned, no reader for c:

by CHRIGHT and WRITE-P $\xrightarrow{\{\}}$ throwPoison

d is poisoned, reader for c:

by CHLEFT and BASIC-COMM-P $\xrightarrow{\{c\}}$ return ()

The cases without poison are straightforward and symmetric: either channel can complete if available. The cases involving poison are not symmetric. If c (i.e. the item on the left-hand side of choice) is poisoned then it will always take precedence over the right-hand side, just as skip would. Since the left-hand side can make an empty-set transition to throwPoison via WRITE-P, it effectively "suppresses" the right-hand side as the rule CHRIGHT then cannot fire. If, alternatively, d is poisoned then it will only be chosen if c cannot be written to, as the left-hand side is still preferred if available.

Bibliography

Armstrong, J., R. Virding, and M. Williams (1993). *Concurrent Programming in Erlang*. Prentice Hall.

Atkey, R. (2009). Parameterised notions of computation. *Journal of Functional Programming 19*(3–4), 335–376.

Barnes, F., P. Welch, and A. Sampson (2005). Barrier synchronisations for occam-pi. In H. R. Arabnia (Ed.), *Proceedings of the 2005 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2005)*, Las Vegas, Nevada, USA, pp. 173–179. CSREA press. ISBN: 1-932415-58-0.

Barnes, F. R. M. and C. G. Ritson (2009). Checking process-oriented operating system behaviour using CSP and refinement. In *PLOS 2009*. ACM. To Appear.

Barnes, F. R. M. and P. H. Welch (2002). Prioritised Dynamic Communicating Processes - Part I. In *Communicating Process Architectures 2002*, pp. 321–352.

Benton, N., L. Cardelli, and C. Fournet (2004). Modern Concurrency Abstractions for C#. *ACM Trans. Program. Lang. Syst. 26*(5), 769–804.

Bjørndalen, J. M., B. Vinter, and O. Anshus (2007). PyCSP – Communicating Sequential Processes for Python. In *Communicating Process Architectures 2007*, pp. 229–248.

Boehm, H. and N. Maclaren (2006). `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2016.html` (visited September 2009).

Boehm, H.-J. (2005). Threads cannot be implemented as a library. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, pp. 261–268. ACM.

Boehm, H.-J. and S. V. Adve (2008). Foundations of the c++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, New York, NY, USA, pp. 68–78. ACM.

Brown, N. C. C. (2004). C++CSP Networked. In I. R. East, D. Duce, M. Green, J. M. R. Martin, and P. H. Welch (Eds.), *Communicating Process Architectures 2004*, pp. 185–200.

Brown, N. C. C. (2007). C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In *Communicating Process Architectures 2007*, pp. 183–205.

Brown, N. C. C. (2008). Communicating Haskell Processes: Composable explicit concurrency using monads. In *Communicating Process Architectures 2008*, pp. 67–83.

Brown, N. C. C. (2009). Automatically generating CSP models for Communicating Haskell Processes. In *Automated Verification of Concurrent Systems (AVoCS) 2009*.

Brown, N. C. C. (2010). Conjoined events. In *Advances in Message Passing 2010*.

Brown, N. C. C. (2011a). Combinators for message-passing in haskell. In R. Rocha and J. Launchbury (Eds.), *Practical Aspects of Declarative Languages*, Volume 6359 of *Lecture Notes in Computer Science*, pp. 19–33. Springer-Verlag Berlin / Heidelberg.

Brown, N. C. C. (2011b). The InterleaveT abstraction: Alternative with flexible ordering. *The Monad Reader* (17).

Brown, N. C. C. and M. L. Smith (2008). Representation and Implementation of CSP and VCR Traces. In *Communicating Process Architectures 2008*, pp. 329–345.

Brown, N. C. C. and M. L. Smith (2009). Relating and Visualising CSP, VCR and Structural Traces. In P. H. Welch et al. (Ed.), *Communicating Process Architectures 2009*.

Brown, N. C. C. and P. H. Welch (2003). An Introduction to the Kent C++CSP Library. In J. F. Broenink and G. H. Hilderink (Eds.), *Communicating Process Architectures 2003*, pp. 139–156.

Burgin, M. and M. L. Smith (2006). Compositions of concurrent processes. In F. Barnes, J. Kerridge, and P. Welch (Eds.), *Communicating Process Architectures 2006*, pp. 281–296. IOS Press.

Cascaval, C., C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee (2008). Software transactional memory: Why is it only a research toy? *Communications of the ACM 51*(11), 46–58.

Chakravarty, M. M. T., G. Keller, R. Lechtchinsky, and W. Pfannenstiel (2001). Nepal – nested data-parallelism in haskell. In *Euro-Par 2001*, pp. 524–534. Springer-Verlag.

Chalmers, K. and S. Clayton (2006). CSP for .NET Based on JCSP. In F. R. M. Barnes, J. M. Kerridge, and P. H. Welch (Eds.), *Communicating Process Architectures 2006*, pp. 59–76.

Chaudhuri, A. (2009). A concurrent ML library in concurrent Haskell. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, New York, NY, USA, pp. 269–280. ACM.

Chitil, O., C. Runciman, and M. Wallace (2000). Tracing and Debugging of Lazy Functional Programs - A Comparative Evaluation of Three Systems. In M. Mohnen and P. Koopman (Eds.), *Draft Proceedings of the 12th International Workshop on Implementation of Functional Languages*, Aachen, Germany, pp. 47–62. Aachener Informatik-Bericht 00-7, RWTH Aachen.

Cleaveland, J. C. and R. C. Uzgalis (1977). *Grammars for Programming Languages*. Elsevier.

Dijkstra, E. W. (1971). Hierarchical Ordering of Sequential Processes. *Acta Informatica 1*(2), 115–138.

Donnelly, K. and M. Fluet (2006). Transactional events. *SIGPLAN Not. 41*(9), 124–135.

East, I. R. (2003). The Honeysuckle programming language: an overview. *IEE Proc.-Softw. 150*(2), 95–107.

Efron, B. and R. J. Tibshirani (1993). *An Introduction to the Bootstrap*. Chapman & Hall/CRC.

Fähndrich, M., M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi (2006). Language Support for Fast and Reliable Message-based Communication in Singularity OS. In *EuroSys '06: Proceedings of the 2006 EuroSys conference*, New York, NY, USA, pp. 177–190. ACM Press.

Formal Systems (Europe) Ltd (1997). Failures-divergence refinement: FDR2 manual.

Fournet, C. and G. Gonthier (2000). The join calculus: A language for distributed mobile programming. In *In Proceedings of the Applied Semantics Summer School (APPSEM), Caminha*, pp. 268–332. Springer-Verlag.

Fraser, K. and T. Harris (2007). Concurrent programming without locks. *ACM Trans. Comput. Syst. 25*(2), 5.

Freitas, L. and J. Woodcock (2009). Fdr explorer. *Formal Aspects of Computing 21*, 133–154.

Gill, A. (2009). Type-safe observable sharing in haskell. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, New York, NY, USA, pp. 117–128. ACM.

Hammond, K. (1994). Parallel functional programming: an introduction. In H. V. Hong (Ed.), *Parallel Symbolic Computation PASCO, 1994: Proceedings of the First International Symposium*, River Edge, NJ, USA, pp. 181–193. World Scientific Publishing Co., Inc.

Harris, T., S. Marlow, S. Peyton-Jones, and M. Herlihy (2005). Composable memory transactions. In *PPoPP '05*, pp. 48–60. ACM.

Harris, T. and S. Singh (2007). Feedback directed implicit parallelism. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, New York, NY, USA, pp. 251–264. ACM.

Hewitt, C., P. Bishop, and R. Steiger (1973). A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pp. 235–245.

Hilderink, G. H. (2005). *Managing Complexity of Control Software through Concurrency*. Ph. D. thesis, Laboratory of Control Engineering, University of Twente.

Hilderink, G. H. and J. F. Broenink (2003). Sampling and timing a task for the environmental process. In J. F. Broenink and G. H. Hilderink (Eds.), *Communicating Process Architectures 2003*, pp. 111–124.

Hilderink, G. H., J. F. Broenink, W. Vervoort, and A. W. P. Bakkers (1997). Communicating Java Threads. In A. W. P. Bakkers (Ed.), *Proceedings of WoTUG-20: Parallel Programming and Java*, pp. 48–76.

Hill, S. (1995, mar). Parallel Imperative Functional Programming. In P. Nixon (Ed.), *Proceedings of WoTUG-18: Transputer and occam Developments*, pp. 33–46.

Hoare, C. (2007). Fine-grain concurrency. In *Communicating Process Architectures 2007*, pp. 1–19.

Hoare, C. A. R. (1985). *Communicating Sequential Processes*. Prentice-Hall.

Hudak, P., J. Hughes, S. P. Jones, and P. Wadler (2007). A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, New York, NY, USA, pp. 12–1–12–55. ACM.

Hunt, G., J. Larus, M. Abadi, M. Aiken, P. Barham, M. Fähndrich, C. Hawblitzel, O. Hodson, S. Levi, N. Murphy, B. Steensgaard, D. Tarditi, T. Wobber, and B. Zill (2005). An Overview of the Singularity Project. Technical report, Microsoft Research.

INMOS (1985). *occam 2 Reference Manual*. Prentice-Hall.

Jones, S. L. P. and P. Wadler (1993). Imperative functional programming. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, New York, NY, USA, pp. 71–84. ACM.

Kerridge, J. M. (2005). Groovy Parallel! A Return to the Spirit of occam? In *Communicating Process Architectures 2005*, pp. –.

Kiselyov, O. and C.-c. Shan (2008). Lightweight monadic regions. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, New York, NY, USA, pp. 1–12. ACM.

Lehmberg, A. and M. N. Olsen (2006). An Introduction to CSP.NET. In F. R. M. Barnes, J. M. Kerridge, and P. H. Welch (Eds.), *Communicating Process Architectures 2006*, pp. 13–30.

Leuschel, M. and M. Fontaine (2008). Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. *ICFEM 2008*.

Marlow, S., S. P. Jones, A. Moran, and J. Reppy (2001). Asynchronous exceptions in haskell. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, PLDI '01, New York, NY, USA, pp. 274–285. ACM.

Martin, J., I. East, and S. Jassim (1994). Design Rules for Deadlock Freedom. *Transputer Communications 3*(2), 121–133. John Wiley and Sons. 1070-454X.

Martin, J. and P. Welch (1996). A Design Strategy for Deadlock-Free Concurrent Systems. *Transputer Communications 3*(4), 215–232. John Wiley and Sons. 1070-454X.

McBride, C. and R. Paterson (2008). Applicative programming with effects. *J. Funct. Program. 18*(1), 1–13.

Milner, R. (1999). *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press.

Moores, J. (1999). CCSP – A Portable CSP-Based Run-Time System Supporting C and occam. In B. M. Cook (Ed.), *Proceedings of WoTUG-22: Architectures, Languages and Techniques for Concurrent Systems*, pp. 147–169.

Peyton Jones, S. (2001). Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pp. 47–96. IOS Press. Revised 2002–2010.

Peyton Jones, S. (2003). Haskell 98 language and libraries: the Revised Report. *Journal of Functional Programming 13*(1).

Peyton Jones, S. L., A. Gordon, and S. Finne (1996). Concurrent Haskell. In *Symposium on Principles of Programming Languages*, pp. 295–308. ACM Press.

Reisig, W. (1985). *Petri Nets—An Introduction*, Volume 4 of *EATCS Monographs on Theoretical Computer Science*. Berlin, New York: Springer-Verlag.

Reppy, J. H. (1995). First-class synchronous operations. In *TPPP '94: Proceedings of the International Workshop on Theory and Practice of Parallel Programming*, pp. 235–252. Springer-Verlag.

Ritson, C. G., A. T. Sampson, and F. R. M. Barnes (2009). Multicore Scheduling for Lightweight Communicating Processes. In J. Field and V. T. Vasconcelos (Eds.), *Coordination Models and Languages, 11th International Conference, COORDINATION 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, Volume 5521 of *Lecture Notes in Computer Science*, pp. 163–183. Springer.

Ritson, C. G. and P. H. Welch (2010). A process-oriented architecture for complex system modelling. *Concurrency and Computation: Practice and Experience 22*, 965–980.

Robison, A. (2007). `http://software.intel.com/en-us/blogs/2007/11/30/volatile-almost-useless-for-multi-threaded-programming/` (visited September 2009).

Roscoe, A. (1997). *The Theory and Practice of Concurrency*. Prentice-Hall.

Roscoe, A. W. (1994). *Model-checking CSP*, Chapter 21. Prentice-Hall.

Roscoe, A. W. and N. Dathi (1987). The pursuit of deadlock freedom. *Information and Computation 75*(3), 289–327.

Roscoe, A. W. and C. A. R. Hoare (1988). The laws of occam programming. *Theor. Comput. Sci. 60*(2), 177–229.

Runciman, C., M. Naylor, and F. Lindblad (2008). Smallcheck and Lazy Smallcheck: automatic exhaustive testing for small values. In *Haskell '08: Proceedings of the first ACM SIGPLAN symposium on Haskell*, pp. 37–48. ACM.

Schneider, S., A. Cavalcanti, H. Treharne, and J. Woodcock (2006). A Layered Behavioural Model of Platelets. In Michael G. Hinchey (Ed.), *ICECCS-2006*, Stanford, California, pp. 98–106. IEEE.

Simpson, J. and C. L. Jacobsen (2008). Visual process-oriented programming for robotics. In *Communicating Process Architectures 2008*, pp. 365–380.

Smith, M. L. (2000). *View-Centric Reasoning about Parallel and Distributed Computation*. Ph. D. thesis, University of Central Florida.

Smith, M. L. (2004). Focussing on Traces to Link VCR and CSP. In I. R. East, D. Duce, M. Green, J. M. R. Martin, and P. H. Welch (Eds.), *Communicating Process Architectures 2004*, pp. 353–360.

Sputh, B. and A. R. Allen (2005). JCSP-Poison: Safe Termination of CSP Process Networks. In *Communicating Process Architectures 2005*.

Stewart, D. (2009). Domain specific languages for domain specific problems (position paper). In *LACSS Workshop on Non-Traditional Programming Models*.

Sufrin, B. (2008). Communicating Scala Objects. In F. R. M. Barnes, J. F. Broenink, A. A. McEwan, A. Sampson, G. S. Stiles, and P. H. Welch (Eds.), *Communicating Process Architectures 2008*, pp. –.

Sulzmann, M., E. S. L. Lam, and P. V. Weert (2008). Actors with multi-headed message receive patterns. In *COORDINATION 08: Proc. 10th Intl. Conf. Coordination Models and Languages*, pp. 315–330.

Vasudevan, N., S. Singh, and S. A. Edwards (2008). A deterministic multi-way rendezvous library for haskell. In *Parallel and Distributed Processing (IPDPS), 2008*.

Wadler, P. (1989). Theorems for free! In *FPCA '89: Proceedings of the fourth international conference on Functional Programming languages and Computer Architecture*, New York, NY, USA, pp. 347–359. ACM.

Welch, P. (1989a). Emulating Digital Logic using Transputer Networks (Very High Parallelism = Simplicity = Performance). *International Journal of Parallel Computing 9*. North-Holland.

Welch, P., F. Barnes, and F. Polack (2006a). Communicating complex systems. In M. G. Hinchey (Ed.), *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, Stanford, California, pp. 107–117. IEEE. ISBN: 0-7695-2530-X.

Welch, P., N. Brown, B. Sputh, K. Chalmers, and J. Moores (2007). Integrating and Extending JCSP. In A. A. McEwan, S. Schneider, W. Ifill, and P. Welch (Eds.), *Communicating Process Architectures 2007*, pp. 349–370.

Welch, P., G. Justo, and C. Willcock (1993). Higher-Level Paradigms for Deadlock-Free High-Performance Systems. In R. Grebe, J. Hektor, S. Hilton,

M. Jane, and P. Welch (Eds.), *Transputer Applications and Systems '93, Proceedings of the 1993 World Transputer Congress*, Volume 2, Aachen, Germany, pp. 981–1004. IOS Press, Netherlands. ISBN 90-5199-140-1.

Welch, P. H. (1989b). Graceful termination – graceful resetting. In A. W. P. Bakkers (Ed.), *OUG-10: Applying Transputer Based Parallel Machines*, pp. 310–317.

Welch, P. H. (1997). Java Threads in Light of occam/CSP (Tutorial). In A. W. P. Bakkers (Ed.), *Proceedings of WoTUG-20: Parallel Programming and Java*, pp. 282–282.

Welch, P. H. (1998). Java Threads in the Light of occam/CSP. In *Architectures, Languages and Patterns for Parallel and Distributed Applications*, Volume 52 of *Concurrent Systems Engineering Series*, pp. 259–284. WoTUG: IOS Press.

Welch, P. H. and F. R. M. Barnes (2005). Communicating mobile processes: introducing occam-pi. In *25 Years of CSP*, Volume 3525 of *Lecture Notes in Computer Science*, pp. 175–210. Springer Verlag.

Welch, P. H., F. R. M. Barnes, and F. A. C. Polack (2006b). Communicating complex systems. In M. G. Hinchey (Ed.), *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, Stanford, California, pp. 107–117. IEEE. ISBN: 0-7695-2530-X.

Welch, P. H., N. C. C. Brown, J. Moores, K. Chalmers, and B. H. C. Sputh (2010). Alting barriers: synchronisation with choice in Java using JCSP. *Concurrency and Computation: Practice and Experience 22*(8).