`Joins` — A Concurrency Library

Claudio Russo and Nick Benton
crusso,nick@microsoft.com
Microsoft Research Ltd, 7JJ Thomson Ave, Cambridge, United Kingdom

September 1, 2006

1 Introduction

The research language Cω [7] promised C$^\#$ 1.x users a more pleasant world of concurrent programming. Cω presents a simple, declarative and powerful model of concurrency - *join patterns* - applicable both to multithreaded applications and to the orchestration of asynchronous, event-based distributed applications.

With the new Generics feature of C$^\#$ 2.0 (and the .NET runtime in general), we can now provide join patterns as a .NET library rather than a language extension. We call this library `Joins` for short, though it is actually provided by the .NET *assembly* `Microsoft.Research.Joins.dll`.

Encoding language features in a library has some obvious drawbacks, restricting the scope for both optimization and static checking — but it also has some advantages. The `Joins` library is language neutral; it can be used by C$^\#$ but also by VB and other .NET languages. A library's source code is easier to read and modify than a compiler, promoting experimentation. A library can be more dynamic: the `Joins` library already supports solutions that are more difficult to express with the declarative join patterns of Cω (see Section 4.5). Despite its additional flexibility, the `Joins` implementation is reasonably efficient and takes advantage of the same basic optimizations performed by the Cω compiler. The library offers a simple interface that makes it particularly easy to translate Cω programs to C$^\#$. However, it can also be used to write concurrent applications from scratch, without the crutch of Cω source code.

The purpose of this to document is to provide a tutorial introduction to *both* Cω's concurrency constructs and the `Joins` library. Section 2 starts with a brief introduction to join patterns; we use Cω since it remains attractive and is somewhat simpler to understand. Section 3 introduces the `Joins` library by example, showing how to re-express the Cω programs of Section 2 as vanilla C$^\#$ 2.0 code that references the library. Section 4 presents tutorial applications of join patterns, expressed both as Cω and `Joins` code. Section 5 provides a concise, yet precise, description of the `Joins` library, as a reference manual. Section 6 gives a brief overview the demos provided with the `Joins` distribution. Appendix A contains a rigorous specification of Cω's concurrency constructs.

The first author is responsible for the `Joins` library (and its bugs) but the Cω material described here is derived from Nick Benton's original unpublished manuscript "Cω Concurrency Constructs".
Note: to distinguish source languages, we will display Cω code in a *shadowed* frame,

```
async Email(string msg);
```

and C$^\#$ 2.0 code (referencing the `Joins` library) in a *plain* frame:

```
Asynchronous.Channel<string> Email;
```

Bugs: please report any bugs in `Joins` or mistakes in this document to `crusso@microsoft.com`.
Download: both Cω and the `Joins` library are available as separate downloads from

`http://research.microsoft.com/research/downloads`

2 Background: Cω's concurrency constructs

Cω extends the C# 1.2 programming language with new asynchronous concurrency abstractions. The new constructs are a mild syntactic variant of those previously described under the name 'Polyphonic C#' [3, 4]. Similar extensions to Java were independently proposed by von Itzstein and Kearney [6]. Cω actually combines Polyphonic C# with a rich new data programming model described elsewhere.

In Cω, methods can be defined as either *synchronous* or *asynchronous*. When a synchronous method is called, the caller is blocked until the method returns, as is normal in C#. However, when an asynchronous method is called, there is no result and the caller proceeds immediately without being blocked. Thus from the caller's point of view, an asynchronous method is like a `void` one, but with the useful extra guarantee of returning immediately. We often refer to asynchronous methods as *messages*, as they are a one-way communication from caller to receiver (think of posting a letter rather as opposed to asking a question and waiting for an answer during a face-to-face conversation).

By themselves, asynchronous method declarations are not particularly novel. Indeed, .NET already has a widely-used set of library classes which allow any method to be invoked asynchronously (though note that in this standard pattern it is the caller who decides to invoke a method asynchronously, whereas in Cω it is the callee (defining) side which declares a particular method to be asynchronous). The significant innovation in Cω is the way in which method bodies are defined.

In most languages, including C#, methods in the signature of a class are in bijective correspondence with the code of their implementations – for each method which is declared, there is a single, distinct definition of what happens when that method is called. In Cω, however, a body may be associated with a *set* of (synchronous and/or asynchronous) methods. We call such a definition a *chord*[1], and a particular method may appear in the header of several chords. The body of a chord can only execute once *all* the methods in its header have been called. Thus, when a chorded method is called there may be zero, one, or more chords which are enabled:

- If no chord is enabled then the method invocation is queued up. If the method is asynchronous, then this simply involves adding the arguments (the contents of the message) to a queue. If the method is synchronous, then the calling thread is blocked.
- If there is a single enabled chord, then the arguments of the calls involved in the match are de-queued, any blocked thread involved in the match is awakened, and the body runs.
- When a chord which involves only asynchronous methods runs, then it does so in a new thread.
- If there are several chords which are enabled then an unspecified one of them is chosen to run.
- Similarly, if there are multiple calls to a particular method queued up, we do not specify which call will be de-queued when there is a match.

2.1 Example: A Simple Buffer

Here is the simplest interesting example of a Cω class:

```
public class Buffer {
   public async Put(string s);

   public string Get() & Put(string s) {
      return s;
   }
}
```

[1] extending the musical theme of C#

This class contains two methods: a synchronous one, `Get()`, which takes no arguments and returns a string, and an asynchronous one, `Put(s)`, which takes a string argument and (like all asynchronous methods) returns no result. The class definition contains two things: a declaration (with no body) for the asynchronous method, and a chord. The chord declares the synchronous method and defines a body (the return statement) which can run when *both* the `Get()` and `Put(s)` methods have been called.

Now assume that producer and consumer threads wish to communicate via an instance `b` of the class `Buffer`. Producers make calls to `b.Put(s)`, which, since the method is asynchronous, never block. Consumers make calls to `b.Get()`, which, since the method is synchronous, will block until or unless there is a matching call to `Put(s)`. Once `b` has received both a `Put(s)` and a `Get()`, the body runs and the argument to the `Put(s)` is returned as the result of the call to `Get()`. Multiple calls to `Get()` may be pending before a `Put(s)` is received to reawaken one of them, and multiple calls to `Put(s)` may be made before their arguments are consumed by subsequent `Get()`s. Note that:

1. The body of the chord runs in the (reawakened) thread corresponding to the matched call to `Get()`. Hence no new threads are spawned in this example.

2. The code which is generated by the class definition above is completely thread safe. The compiler automatically generates the locking necessary to ensure that, for example, the argument to a particular call of `Put(s)` cannot be delivered to two distinct calls to `Get()`. Furthermore (though it makes little difference in this small example), the locking is fine-grained and brief - chorded methods to not lock the whole object and are not executed with "monitor semantics".

3. The reader may wonder how we know which of the methods involved in a chord gets the returned value. The answer is that it is always the synchronous one, and there can be at most one synchronous method involved in a chord.

In general, the definition of a synchronous method in Cω consists of more than one chord, each of which defines a body which can run when the method has been called *and* a particular set of asynchronous messages are present. For example, we could modify the example above to allow `Get()` to synchronize with calls to either of two different `Put(s)` and `Put(n)` methods:

```
public class BufferTwo {
   public async Put(string s);
   public async Put(int n);

   public string Get()
     & Put(string s) { return s;}
     & Put(int n) { return n.ToString();}
}
```

Now we have two asynchronous methods (which happen to be distinguished by type rather than name) and a synchronous method which can synchronize with either one, with a different body in each case.

2.2 Example: A One-place Buffer

The previous example showed how to define a buffer of unbounded size: any number of calls to `Put(s)` could be queued up before matching a `Get()`. We now define a variant in which only a single data value may held in the buffer at any one time:

```
public class OnePlaceBuffer {
   private async Empty();
   private async Contains(string s);
```

```
    public OnePlaceBuffer() {
        Empty();
    }
    public void Put(string s) & Empty() {
        Contains(s);
        return;
    }
    public string Get() & Contains(string s) {
        Empty();
        return s;
    }
}
```

The public interface of `OnePlaceBuffer` is similar to that of `Buffer`, although the `Put(s)` method is now synchronous, since it can now block in the case that there is already an unconsumed value in the buffer.

The implementation of `OnePlaceBuffer` makes use of two private asynchronous messages: `Empty()` and `Contains(string s)`. These are used to carry the state of the buffer and illustrate a very common programming pattern in Cω: note that we have made no use of fields. The way in which this works can be best understood by reading the constructor and the two chords in a simple declarative manner:

- When a new buffer is created, it is initially `Empty()`.
- If you call `Put(s)` on an `Empty()` buffer then it subsequently `Contains(s)` and the call to `Put(s)` returns.
- If you call `Get()` on a buffer which `Contains(s)` then the buffer is subsequently `Empty()` and `s` is returned to the caller of `Get()`.
- Implicitly: In all other cases, calls to `Put(s)` and `Get()` block.

It is easy to see that the constructor establishes, and both the chords maintain, the invariant that there is always exactly one `Empty()` or `Contains(s)` message pending on the buffer. Thus the two chords can be read as a fairly direct specification of a finite state machine.

3 The `Joins` Library

In Cω, classes that declare (a)synchronous method joined in chords implicitly declare a set of communication channels. An asynchronous method has a backing queue of pending method calls. A synchronous method has a backing queue of waiting threads. The state of the queues is summarized by a bit vector and protected by a hidden lock. Invoking an (a)synchronous method executes some specialized scheduling code that decides, given the current state and the declared chords, which, if any, chord gets to fire, either on the current or any waiting thread. Thus each object (or, for purely static methods, class) includes its own scheduling logic. Instead of relying on a central scheduling thread, threads that invoke chorded methods each spend a little time helping to schedule each other.

In the `Joins` library, the scheduling logic that would be compiled into the corresponding Cω class or instance has a separate, first-class representation as an object of the special `Join` class. The `Join` class provides a mostly declarative, type-safe mechanism for defining thread-safe asynchronous and synchronous communication channels and patterns. `Join` objects can be used to coordinate concurrently executing local threads or remote processes. Instead of (a)synchronous methods, as in Cω, the communication channels are special delegate values obtained from a common `Join` object. Communication and/or synchronization takes place by invoking these delegates, passing arguments and optionally waiting for return values. The allowable communication patterns as well as their effects are defined using *join patterns*: bodies of code whose execution is guarded

by linear combinations of channels. The body, or *continuation*, of a join pattern is provided by the user as a (typically anonymous) delegate that can manipulate external resources protected by the `Join` object.

3.1 Example: A Simple Buffer (`Samples\Buffer`)

Using the `Joins` library, we can implement the Cω `Buffer` class of Section 2.1 as follows:

```
using Microsoft.Research.Joins;
public class Buffer {
  // Declare the (a)synchronous channels
  public readonly Asynchronous.Channel<string> Put;
  public readonly Synchronous<string>.Channel Get;
  public Buffer() {
    // Allocate a new Join object for this buffer
    Join join = Join.Create();
    // Use it to initialize the channels
    join.Initialize(out Put);
    join.Initialize(out Get);
    // Finally, declare the patterns(s)
    join.When(Get).And(Put).Do(delegate(string s) {
       return s;
    });
  }
}
```

The code declares a buffer class with two fields of special delegate types. The `Put` field contains an asynchronous channel of type `Asynchronous.Channel<string>` whose delegate `Invoke` method takes a string argument and returns void (immediately). The `Get` field contains contains a synchronous channel of type `Synchronous<string>.Channel` whose delegate `Invoke` method returns a string but takes no argument. Both fields are initially `null`. The constructor allocates a new `Join` object, `join`, using the factory method `Create`. The `join` object is a private scheduler for the buffer. The constructor then calls `Initialize` on `join`, passing the locations of each of the channels: this assigns two new delegate values into the fields, each obtained from the `join` object.[2] Finally we declare the Cω chord by declaring a pattern on the `join` object, passing the synchronous channel `Get` to `When` and `And`ing it with the asynchronous channel `Put`. The pattern is completed by invoking `Do`, passing the continuation for this pattern, expressed here as an anonymous delegate. The continuation expects exactly one argument (the argument to `Put`); the continuation's return value is returned to the caller of `Get`. Notice that the body of the anonymous delegate corresponds to the body of the original Cω chord.

This is certainly longer than the Cω code, but if we ignore the boilerplate calls to `Create` and `Initialize`, which always follow the same pattern, then what remains retains the declarative flavour of the original Cω code.

```
class Buffer {
  public readonly Asynchronous.Channel<string> Put;
  public readonly Synchronous<string>.Channel Get;
  public Buffer() { ...
    join.When(Get).And(Put).Do(delegate(string s) {
       return s;
    });
  }
}
```

[2] The `Initialize` method is both generic and overloaded: it takes an `out` parameter just so that we can use it for different channel types and still rely on C#'s and VB's type inference to avoid specifying type arguments.

Moreover, client code of the Cω and C# buffers is (often) syntactically identical. Given a buffer `b`, clients invoke `b.Put(s);` to send a string `s` and `b.Get()` to receive one. Of course, the references to `Get` and `Put` members are compiled slightly differently, just invoking a method in the Cω client, but reading a field and then invoking its delegate value in the C# client.

To define a synchronous method with more than one body, as in the Cω class `BufferTwo`, one simply declares multiple join patterns with separate calls to `When`:

```
class BufferTwo {
  public readonly Asynchronous.Channel<string> Put;
  public readonly Asynchronous.Channel<int> PutInt;
  public readonly Synchronous<string>.Channel Get;
  public Buffer() {
    Join join = Join.Create();
    join.Initialize(out Put);
    join.Initialize(out PutInt);
    join.Initialize(out Get);
    join.When(Get).And(Put).Do(delegate(string s) {
       return s;
    });
    join.When(Get).And(PutInt).Do(delegate(int n) {
       return n.ToString();
    });
  }
}
```

Notice that the second continuation takes an integer (not a string): the type of a continuation is statically determined by the types of the channels joined in the pattern. Unfortunately, we had to translate the second Cω method `Put(n)` to `PutInt(n)`, changing its name, since C# does not support overloading on fields.

3.2 Example: A One-place Buffer (`Samples\OnePlaceBuffer`)

Translating the one-place buffer of Section 2.2 is easy:

```
  public class OnePlaceBuffer {
    private readonly Asynchronous.Channel Empty;
    private readonly Asynchronous.Channel<string> Contains;
    public readonly Synchronous.Channel<string> Put;
    public readonly Synchronous<string>.Channel Get;
    public OnePlaceBuffer() {
      Join j = Join.Create();
      j.Initialize(out Empty);
      j.Initialize(out Contains);
      j.Initialize(out Put);
      j.Initialize(out Get);
      j.When(Put).And(Empty).Do(delegate(string s) {
        Contains(s);
      });
      j.When(Get).And(Contains).Do(delegate(string s) {
        Empty();
        return s;
      });
      Empty();
    }
  }
```

To protect a buffer's invariant, we translate the private Cω `Empty` and `Contains` messages to private fields of asynchronous channel types. Just like its Cω counterpart, the constructor initially

establishes the invariant by calling `Empty()`, but only after initializing the channels and join patterns.

This example also demonstrates two new channel types. The asynchronous channel type `Asynchronous.Channel` (not to be confused with `Asynchronous.Channel<A>`) of the `Empty` field is a special delegate type whose `Invoke` method takes zero arguments (and returns void). As in Cω, nullary messages use a more efficient counter instead of a queue of argument values to record pending invocations. To encode the synchronous `Put(s)` method of the Cω class, that simply returns control, not a value, to its caller, we use an instance of the synchronous channel type `Synchronous.Channel<A>`: this channel type has a delegate `Invoke` method that returns `void` and takes one argument of type `A`.

4 Concurrency Tutorials

4.1 Spawning Threads (`Samples\SpawningThreads`)

As well as supporting chords on synchronous method declarations, Cω also allows the definition of purely asynchronous chords. These appear as separate class members using the keyword `when` followed by a chord of asynchronous methods. The body of such a chord will execute in a new thread. In particular, calling an asynchronous method that occurs as the sole method in a chord gives an easy way to start a new thread with parameters. For example,

```
using System;
using System.Threading;

public class ThreadSpawner {
  static async Spawn(string s);
  when Spawn(string s) {
    while(true) {
      Console.WriteLine(s);
      Thread.Sleep(1000);
    }
  }
  static void Main() {
    Spawn("thread␣one");
    Spawn("thread␣two");
  }
}
```

prints

```
thread one
thread two
thread one
...
```

Whenever the method `Spawn()` is called, the call returns immediately and the body is executed in a new thread.

The `Joins` library offers similar functionality. If a join pattern commences with an asynchronous channel, i.e. the argument to `When` is asynchronous, then its continuation spawns a new thread:

```
public class ThreadSpawner {
  static readonly Asynchronous.Channel<string> Spawn;
  static ThreadSpawner() {
    Join join = Join.Create();
    join.Initialize(out Spawn);
    join.When(Spawn).Do(delegate(string s) {
```

```
      while (true) {
        Console.WriteLine(s);
        Thread.Sleep(1000);
      };
    });
  }
  static void Main() {
    Spawn("thread␣one");
    Spawn("thread␣two");
  }
}
```

Here, we had to introduce an explicit *static* constructor to set up the static channel and pattern. Although implicit in the Cω source, the Cω compiler actually generates a static constructor too, to allocate a message queue for method `Spawn`.

Both in Cω and in `Joins`, an asynchronous chord may involve more than just one message, delaying thread creation until/unless all messages have been received. For instance:

```
async Electron();
async Positron();
when Electron() & Positron() {
   Photon(); Photon();
}
```

```
readonly Asynchronous.Channel Electron, Positron;
...
join.When(Electron).And(Positron).Do(delegate {
  Photon(); Photon();
});
```

4.2 Using Asynchronous Messages For State (`Samples\Counter`)

Objects which have mutable state and which may be accessed from multiple threads usually have to ensure that methods which read or write that state are executed with some form of mutual exclusion. In C$^{\#}$, the simplest way of achieving this is to use the `lock` statement [3] [4] as in

```
public class Counter {
  long i;
  public Counter() {
    i = 0;
  }
  public void Inc() {
    lock (this) {         // acquire the lock on this object
      i = i + 1;          // update the state
    }                     // release the lock
  }
  public long Value() {
    lock (this) {
      return i;           // release the lock & return state
    }
  }
}
```

Of course, the above still works in Cω, but there are also alternatives. Here is one:

[3] or a `System.Runtime.CompilerServices.MethodImplAttribute`

[4] or explicit calls to `System.Threading.Monitor.Enter/Exit`

```
public class Counter {
  long i;
  private async mylock();
  public Counter() {
    i=0;
    mylock();
  }
  public void Inc() & mylock() { // acquire the lock
    i = i+1;                     // update the state
    mylock();                    // release the lock
  }
  public long Value() & mylock() {
    long r = i;
    mylock();
    return r;
  }
}
```

In this case we have used a private asynchronous message `mylock()` to protect access to the field `i`. When the `Inc()` method is called, it will block until/unless there is a waiting `mylock`. When there is, then the call proceeds and the `mylock()` message is consumed, so further calls to `Inc()` or `Value()` from other threads will block until another `mylock()` is sent. Thus the thread which called `Inc()` has exclusive access to the field whilst it does the increment. Once it has finished, it sends another `mylock()` to allow a subsequent call access. Similar reasoning applies to `Value()`. Observe that there is always at most one call to `mylock()` pending on a counter.

With `Joins`, this becomes:

```
using Microsoft.Research.Joins;

public class Counter {
  long i;
  private readonly Asynchronous.Channel mylock;
  public readonly Synchronous.Channel Inc;
  public readonly Synchronous<long>.Channel Value;

  public Counter() {
    Join join = Join.Create();
    join.Initialize(out mylock);
    join.Initialize(out Inc);
    join.Initialize(out Value);
    join.When(Inc).And(mylock).Do(delegate
    { // acquire the lock
        i = i + 1;  // update the state
        mylock();  // release the lock
    });
    join.When(Value).And(mylock).Do(delegate
    { // acquire the lock
        long r = i;  // read the state
        mylock();    // release the lock
        return r;    // return state;
    });
    i = 0;
    mylock();
  }
}
```

Going a step further, we can eliminate the field entirely. Instead, we pass the state of the counter around as an argument to a private asynchronous message:

```
public class Counter {
  private async mystate(long i);

  public Counter {
    mystate(0);
  }
  public void Inc() & mystate(long i) { // acquire the state
    mystate(i+1); // reissue the state
  }
  public long Value() & mystate(long i) { // acquire the state
    mystate(i); // reissue the state
    return i;
  }
}
```

The translation to C# using `Joins` is easy:

```
using Microsoft.Research.Joins;

public class Counter {
  private readonly Asynchronous.Channel<long> mystate;
  public readonly Synchronous.Channel Inc;
  public readonly Synchronous<long>.Channel Value;
  public Counter() {
    Join join = Join.Create();
    join.Initialize(out mystate);
    join.Initialize(out Inc);
    join.Initialize(out Value);
    join.When(Inc).And(mystate).Do(delegate(long i)
    {  // acquire the state
      mystate(i + 1); // reissue the state
    });
    join.When(Value).And(mystate).Do(delegate(long i)
    { // acquire the state
      mystate(i); // reissue the state
      return i;
    });
    mystate(0);
  }
}
```

In such a trivial case, using join patterns to implement mutual exclusion this way is probably a bad idea - it is much the same length as the version which uses `lock`, is less efficient and has more opportunities for error (a typical example is forgetting to resend the state-carrying message at the end of a method). However, in cases where one wants more fine-grained locking on parts of the state of an object, or to block and awaken particular threads when certain conditions are established (i.e. situations in which the C# programmer would use `Notify()` and `Pulse()`), this style of programming becomes preferable.

4.2.1 Example: A Reader-Writer Lock (`Samples\ReaderWriter`)

Controlling concurrent access to a shared, mutable resource is a classic problem. Clients request, and later release, either read-only or read-write access to the resource. To preserve consistency but minimize waiting, one usually wishes to allow either any number of readers or a single writer, but not both, to have access at any one time. The code below implements a reader-writer lock in Cω. Writers call `Exclusive()`, do some writing and then call `ReleaseExclusive()`. Readers call `Shared()`, do some reading and then call `ReleaseShared()`:

```
public class ReaderWriter {
   private async Idle();
   private async Sharing(int n);
   public ReaderWriter() {
     Idle();
   }
   public void Exclusive() & Idle() {  }

   public void ReleaseExclusive() { Idle(); }

   public void Shared()
   & Idle() { Sharing(1);}
   & Sharing(int n) { Sharing(n+1);}

   public void ReleaseShared() & Sharing(int n) {
     if (n == 1) Idle(); else Sharing(n-1); }
}
```

Again, we use private messages, `Idle()` and `Sharing(n)` (where `n` is the current number of readers), to carry the state. And once more, there is a simple declarative reading of the constructor and chords which allows one to understand the implementation:

- When the lock is created there are no readers or writers, so it is `Idle()`.
- If a writer requests `Exclusive()` access and the lock is `Idle()` then he may proceed (but the lock is no longer `Idle()`).
- If a writer indicates he is finished by calling `ReleaseExclusive()` then the lock is `Idle()` again.
- If a reader requests `Shared()` access and the lock is `Idle()` then the lock moves to the state `Sharing(1)`, meaning there is now one reader, and he may proceed.
- If a reader requests `Shared()` access and there are currently n readers, then there are now $n+1$ readers, and the new one may proceed. The state transitions from `Sharing(n)` to `Sharing(n+1)`.
- If a reader indicates he is finished by calling `ReleaseShared()` and there were previously n readers, then if n was 1 then lock is `Idle()` again, otherwise there are now $n-1$ shared readers (`Sharing(n-1)`). In either case, the reader who has just relinquished access may proceed with whatever else he has to do.

Assuming that all the clients obey the request-release protocol, the invariant is that there is at most one pending private async message representing the state:

$$none \leftrightarrow Idle() \leftrightarrow Sharing(1) \leftrightarrow Sharing(2) \leftrightarrow Sharing(3) \leftrightarrow \ldots$$

Operationally, it may help to think about what does *not* appear in the code. There is, for example, no chord which is applicable if a client calls `Exclusive()` when there is a pending `Sharing(n)` message. In such a case, the client will block until all the readers have released their access and an `Idle()` message has been sent.[5]

This is trivial to implement using `Joins`:

```
using Microsoft.Research.Joins;

public class ReaderWriter {
```

[5]There is a question about fairness here - see the Polyphonic C# paper [4] for a modification to this example which enforces a simple form of fairness between readers and writers.

```
    private readonly Asynchronous.Channel Idle;
    private readonly Asynchronous.Channel<int> Sharing;
    public readonly Synchronous.Channel Exclusive, ReleaseExclusive;
    public readonly Synchronous.Channel Shared, ReleaseShared;

    public ReaderWriter() {
      Join j = Join.Create();
      j.Initialize(out Idle);
      j.Initialize(out Sharing);
      j.Initialize(out Exclusive);
      j.Initialize(out ReleaseExclusive);
      j.Initialize(out Shared);
      j.Initialize(out ReleaseShared);

      j.When(Exclusive).And(Idle).Do(delegate { });
      j.When(ReleaseExclusive).Do(delegate { Idle(); });

      j.When(Shared).And(Idle).Do(delegate { Sharing(1); });
      j.When(Shared).And(Sharing).Do(delegate(int n)
      {
        Sharing(n + 1);
      });
      j.When(ReleaseShared).And(Sharing).Do(delegate(int n)
      {
        if (n == 1) Idle(); else Sharing(n - 1);
      });

      Idle();
    }
  }
}
```

4.2.2 Example: A Bounded Buffer (Samples\BoundedBuffer)

The 'Getting Started' section gives examples of an unbounded buffer (for which producers are never blocked) and a single-slot buffer (at most one value may be held in the buffer). Here we show how one might program a general n-place buffer in Cω. There are several approaches one could take, but the following seems the neatest:

```
public class BoundedBuffer {
  private async Token();
  private async Value(object o);

  public BoundedBuffer(int size) {
    for(int i=0;i<size;i++) {
       Token();
    }
  }
  public void Put(object o) & Token() {
    Value(o);
  }
  public object Get() & Value(o) {
    Token();
    return o;
  }
}
```

When an n-slot buffer object is created, it sends itself n `Token()` messages. Calls to `Put(o)` block until there is a `Token()` available, consume it and send a `Value(o)` message containing the object which was put in the buffer. Calls to `Get()` block until there is a `Value(o)` message available, consume it and produce a fresh `Token()`. The invariant here, of course, is that the number of `Token()` messages plus the number of `Value(-)` messages always equals the size of the buffer.

Cω does not support Generics, but in our translation to `Joins`, we may as well take the opportunity to make the `BoundedBuffer` a strongly-typed, generic class `BoundedBuffer<T>`:

```
using Microsoft.Research.Joins;

public class BoundedBuffer<T> {
  private readonly Asynchronous.Channel Token;
  private readonly Asynchronous.Channel<T> Value;
  public readonly Synchronous.Channel<T> Put;
  public readonly Synchronous<T>.Channel Get;

  public BoundedBuffer(int size) {
    Join join = Join.Create();
    join.Initialize(out Token);
    join.Initialize(out Value);
    join.Initialize(out Put);
    join.Initialize(out Get);
    join.When(Put).And(Token).Do(delegate(T t)
    {
      Value(t);
    });
    join.When(Get).And(Value).Do(delegate(T t)
    {
      Token();
      return t;
    });
    for (int i = 0; i < size; i++) {
      Token();
    }
  }
}
```

4.3 Asynchronous Communication

The previous section describes how private asynchronous messages may be used to carry state. Here we show how to orchestrate asynchronous messages between different objects.

4.3.1 Semaphores (`Samples\Semaphore`)

A semaphore exposes two methods, `Wait()`, which is synchronous, and `Signal()`, which is asynchronous. Calls to `Wait()` block until there has been a matching call to `Signal()`:

```
public class Semaphore {
  public async Signal();

  public void Wait() & Signal() {
  }
}
```

For example, here is one way to structure a program which spawns two new threads and then waits for them both to finish:

```
public class MyApp {
  static async task1(int arg, Semaphore s);
  static async task2(int arg, Semaphore s);

  public static void Main() {
    Semaphore s = new Semaphore();
    task1(1,s);
    task2(2,s);
    // do something in main thread
    s.Wait(); // wait for one to finish
    s.Wait(); // wait for another to finish
    Console.WriteLine("both␣tasks␣finished");
  }
  when task1(int arg, Semaphore s) {
     // do something
     s.Signal(); // say we've finished
  }
  when task2(int arg, Semaphore s) {
     // do something else
     s.Signal(); // say we've finished
  }
}
```

With `Joins`, the `Semaphore` class is easily defined:

```
using Microsoft.Research.Joins;
public class Semaphore {
  public readonly Asynchronous.Channel Signal;
  public readonly Synchronous.Channel Wait;
  public Semaphore() {
    Join join = Join.Create();
    join.Initialize(out Signal);
    join.Initialize(out Wait);
    join.When(Wait).And(Signal).Do(delegate { });
  }
}
```

Unfortunately, the client code requires some modest contortions:

```
public class MyApp {
  static readonly Asynchronous.Channel<Pair<int, Semaphore>> Task1, Task2;
  public static void Main2() {
    Semaphore s = new Semaphore();
    Task1(new Pair<int, Semaphore>(1, s));
    Task2(new Pair<int, Semaphore>(2, s));
    // do something in main thread
    s.Wait(); // wait for one to finish
    s.Wait(); // wait for another to finish
    Console.WriteLine("both␣tasks␣finished");
  }

  static MyApp() {
    Join join = Join.Create();
    join.Initialize(out Task1);
    join.Initialize(out Task2);
    join.When(Task1).Do(delegate(Pair<int, Semaphore> p) {
        // do something with p.Fst
        p.Snd.Signal();
    });
```

```
    join.When(Task2).Do(delegate(Pair<int, Semaphore> p) {
        // do something else with p.Fst
        p.Snd.Signal();
    });
  }
}
```

Unlike Cω, whose (a)synchronous methods can take any numbers of arguments, `Joins` only supports channels that take at most one argument. To pass multiple values, we must wrap them in some form of tuple. Here, we used generic pairs, defined in the `Joins` library:

```
[Serializable]
public struct Pair<A, B> {
  public readonly A Fst; public readonly B Snd;
  public Pair(A Fst, B Snd) { this.Fst = Fst; this.Snd = Snd; }
}
```

To pass more than 2 arguments, the user could either declare their own tuple type, or just nest generic pairs. To match a Cω method signature directly, the user can always introduce a public wrapper method that take n arguments and forwards them to a private channel wrapped up as an n-tuple.[6]

4.3.2 Asynchronous call and return patterns (`Samples\AsyncService`)

Calling an asynchronous method is a one-way fire and forget operation. How then does one get results back from asynchronous calls? Firstly, one must explicitly tell the receiver of the call what do to with the result. This is often done by passing an extra parameter along with the arguments to the call. This extra argument can be of some application specific agreed type (for example, a buffer into which the result is to be put or the name of a file into which the result is to be written) or, more generally, a delegate (callback) which should be invoked with the result. In terms of the analogy between calling an asynchronous method and sending a letter, the extra parameter corresponds to the "return address" which one adds to the letter's actual content. Thus the interface to a typical service which can be invoked asynchronously and which returns a result asynchronously might look like this:

```
public delegate async MyCallback(object result);

public interface IService {
  public async Service(object parameter, MyCallback cb);
}
```

where the implementation of `Service` will do some computation with the parameter and then invoke the callback with a result.

The second interesting aspect of asynchronous programming is that messages arrive at unpredictable times and in unpredictable orders. Amongst other things, this means that correlating requests and responses can require some extra programming, as it may not otherwise be clear what a given response refers to. The correlation information can take the form of extra explicit tokens passed to, and returned back from, a service (corresponding to "your ref: whatever" or "please quote order number nnn in future correspondence" in a letter). Alternatively, the correlation tokens can be passed more implicitly by creating new target objects for the callbacks sent with different requests (this is a bit like adding "Dept. XYZnn" to the return address of a letter). One also often wishes to coordinate multiple asynchronous messages, for example performing an action

[6] In a distributed application, the wrapper method for an asynchronous channel should be marked with the `OneWay` attribute so that remote invocations of the wrapper retain the intended non-blocking semantics. The `Joins` library ensures that the target methods of `Asynchronous.Channel` and `Asynchronous.Channel<A>` channels are declared `OneWay`.

once a number of replies have been received or a certain time has passed. For example, here's how one might invoke two asynchronous services and subsequently wait for replies from both of them:

```
class JoinTwo {
  public MyCallback Firstcb;
  public MyCallback Secondcb;
  async First(object fst);
  async Second(object snd);

  public JoinTwo() {
    Firstcb  = new MyCallback(First);
    Secondcb = new MyCallback(Second);
  }

  public void Wait(out object x, out object y)
  & First(object fst)
  & Second(object snd) {
    x = fst; y = snd;
  }
}

class Client {
  public static void Main(string[] args) {
    IService s1 = ... ;
    IService s2 = ... ;
    JoinTwo j = new JoinTwo();
    s1.Service(10, j.Firstcb);
    s2.Service(30, j.Secondcb);
    ...                    // do something useful in the meantime...
    object x,y;
    j.Wait(out x, out y); // wait for both results to come back
    ...                    // do something with them
  }
}
```

`JoinTwo` is a library class which encapsulates the pattern of waiting for two responses. The client creates an instance of `JoinTwo`, passes its callbacks along with the arguments to two services and then carries on with its own processing. Later, it calls `Wait()` which will block until/unless both of the callbacks have been invoked. Note that the correlation here is of the 'target address' (demultiplexed) form, rather than the 'explicit token' (multiplexed) form; this is usually the preferred style in Cω, since synchronization behaviour cannot depend directly on message contents.

Writing classes like `JoinTwo` which wait for just one result (Select) or more complex conditions (e.g. at least m out of n) is straightforward.

With `Joins` we might define the `IService` interface as follows:

```
interface IService<R, A> {
  Asynchronous.Channel<Pair<A, Asynchronous.Channel<R>>> Service { get;}
}
```

Here, we have taken the opportunity to abstract over the return and argument types `R` and `A` and avoided introducing a separate `async` callback type by just using `Asynchronous.Channel<R>`. Since interfaces cannot have fields, we instead expose the channel as a (readonly) property. The channel itself receives the argument and reply callback as a pair of values.

Here's a trivial service to illustrate the pattern:

```
public class SimpleService : IService<string, int> {
  readonly Asynchronous.Channel<Pair<int,Asynchronous.Channel<string>>>
    Service;
```

```
  Asynchronous.Channel<Pair<int,Asynchronous.Channel<string>>>
    IService<string,int>.Service {
    get { return Service; }
  }

  public SimpleService() {
    Join j = Join.Create();
    j.Initialize(out Service);
    j.When(Service).Do(delegate(Pair<int,Asynchronous.Channel<string>> p)
    {
      string result = p.Fst.ToString(); // do some work with p.Fst
      p.Snd(result); // send the result on p.Snd
    });
  }
}
```

Fortunately, the `Joins` version of `JoinTwo` is almost nicer than the Cω one: there is no need to wrap the callback messages as delegates — channels *are* delegates. Since channels cannot use `out` parameters `Wait` returns a `Pair<R1,R2>` of results.

```
public class JoinTwo<R1, R2> {
  public readonly Asynchronous.Channel<R1> First;
  public readonly Asynchronous.Channel<R2> Second;
  public readonly Synchronous<Pair<R1, R2>>.Channel Wait;
  public JoinTwo(){
    Join j = Join.Create();
    j.Initialize(out First);
    j.Initialize(out Second);
    j.Initialize(out Wait);
    j.When(Wait).And(First).And(Second).Do(
      delegate(R1 r1, R2 r2)
      {
        return new Pair<R1, R2>(r1, r2);
      });
 }
}
```

The C# client code is similar, but waits for a pair of results:

```
using Request = Pair<int, Asynchronous.Channel<string>>;
class Client {
  public static void Main() {
    IService<string, int> s1, s2;
    s1 = new SimpleService();
    s2 = new SimpleService();
    JoinTwo<string, string> j = new JoinTwo<string, string>();
    s1.Service(new Request(10, j.First));
    s2.Service(new Request(30, j.Second));
    ...                    // do something useful in the meantime...
    // wait for both results to come back
    Pair<string, string> result = j.Wait();
    ...                    // do something with them
  }
}
```

4.3.3 Active Objects (`Samples\ActiveObjects`)

One common pattern of asynchronous programming is that of active objects, or actors. Active objects communicate by asynchronous messages which are processed sequentially by a thread of

control which is specific to that object. One way of programming active objects in Cω is by inheritance from an abstract base class:

```
public abstract class ActiveObject {
  protected bool done = false;
  private async start();
  protected abstract void ProcessMessage();

  public ActiveObject() {
    start();
  }

  when start() {
    while (!done) {
      ProcessMessage();
    }
  }
}
```

When an instance of `ActiveObject` is created, the `start()` method is called: this spawns a new thread that repeatedly calls the abstract synchronous method `ProcessMessage()`. Subclasses of `ActiveObject` then override `ProcessMessage()`, joining it in a sequence of chords with each of the asynchronous messages which they wish to process. For example, a message redistributor object in an event-based application might look like:

```
public interface EventSink {
  async Post(string message);
}
public class Distributor : ActiveObject, EventSink {
  private ArrayList subscribers = new ArrayList();
  private string myname;

  public async Subscribe(EventSink sink);
  public async Post(string message);

  protected override void ProcessMessage()
  & Subscribe(EventSink sink) {
    subscribers.Add(sink);
  }
  & Post(string message) {
    foreach (EventSink sink in subscribers) {
      sink.Post(myname + ":" + message);
    }
  }
  public Distributor(string name) {
    myname = name;
  }
}
```

A `Distributor` object can receive asynchronous `Subscribe` and `Post` messages. Each `Post` message is resent to each of the current subscribers. No locking is required for access to the current subscriber list since, although messages can arrive at any time, they are processed strictly sequentially by the `Distributor`'s message-loop thread (without blocking their sender, of course).

Since threads are a relatively expensive resource on the CLR, having many instances of this `ActiveObject` class around at once will be inefficient. However, ActiveObjects are not built in to Cω - they're just one example of the sort of thing one can build easily with the Cω concurrency constructs. Implementing variants of this pattern, such as families of objects that share a custom threadpool implementation, is also straightforward.

With `Joins`, we can define the abstract class like this:

```
using Microsoft.Research.Joins;

public abstract class ActiveObject {
  protected bool done = false;
  private readonly Asynchronous.Channel Start;
  protected Synchronous.Channel ProcessMessage;
  protected Join join;

  public ActiveObject() {
    join = Join.Create();
    join.Initialize(out ProcessMessage);
    join.Initialize(out Start);
    join.When(Start).Do(delegate
    {
      while (!done) ProcessMessage();
    });
    Start();
  }
}
```

To support Cω style subclassing, we expose the `Join` object as a protected field. Now subclasses of `ActiveObject`, like `Distributer`, can use the field to initialize new channels and define new patterns.

```
public interface EventSink {
  Asynchronous.Channel<string> Post { get; }
}
public class Distributor : ActiveObject, EventSink {
  private ArrayList subscribers = new ArrayList();
  private string myname;

  public readonly Asynchronous.Channel<EventSink> Subscribe;
  public readonly Asynchronous.Channel<string> Post;

  Asynchronous.Channel<string> EventSink.Post { get { return Post; } }

  public Distributor(string name) {
    join.Initialize(out Post);
    join.Initialize(out Subscribe);
    join.When(ProcessMessage).And(Subscribe).Do(
      delegate(EventSink sink)
      {
        subscribers.Add(sink);
      });
    join.When(ProcessMessage).And(Post).Do(
      delegate(string message)
      {
        foreach (EventSink sink in subscribers) {
          sink.Post(myname + ":" + message);
        }
      });
    myname = name;
  }
}
```

4.3.4 Futures (Samples\Futures)

A *future* is an old concurrency abstraction used to represent the value of a concurrent computation. Creating a future from a computation spawns a new thread to execute the computation in parallel. When the current (or another) thread actually needs the value of the computation it calls a method on the future to wait until/unless the computation is done. Between creating the future, and obtaining its value, the current thread is free to perform other tasks.

Generic futures with explicit waiting are simple to code up with Joins:

```
using Microsoft.Research.Joins;

public class Future<T> {
  public delegate T Computation();
  public readonly Synchronous<T>.Channel Get;

  private readonly Asynchronous.Channel Execute, Done;
  private T Value;
  public Future(Computation comp) {
    if (comp == null) throw new ArgumentNullException();
    Join j = Join.Create();
    j.Initialize(out Get);
    j.Initialize(out Done);
    j.Initialize(out Execute);
    j.When(Get).And(Done).Do(delegate
    {
      Done(); // reissue Done to allow multiple Gets
      return Value;
    });
    j.When(Execute).Do(delegate
    {
      Value = comp();
      comp = null; // discard comp to avoid space leak
      Done();
    });
    Execute();
  }
}
```

Consider computing the maximum of an integer array a. Using a future we can split this into computing the maximum of the bottom and top halves, with the latter executing concurrently:

```
int[] a = ...
Future<int> topMax =
  new Future<int>(delegate { return Max(a, a.Length / 2, a.Length); });
int bottomMax = Max(a, 0, a.Length / 2);
int max = (bottomMax <= topMax.Get()) ? topMax.Get() : bottomMax;
```

For the final comparison, the first call to topMax.Get() may block waiting for the computation to finish, but the second will return immediately.

Modifying class Future<T> to cope with exceptions thrown by comp, cancelation of comp or to use a thread pool is easy.

4.4 Distributed Applications & Remoting (Samples\RemoteActiveObjects)

In Cω, we could easily make any ActiveObject remotable, say for distributed applications, by modifying ActiveObject to inherit from MarshallByRefObject.

```
public abstract class ActiveObject : MarshalByRefObject { ...
}
```

This works nicely because Cω "channels" are implemented as (a)synchronous *methods*, so sending a message or request to a remote object is just a remote method invocation. The Cω compiler even marks `async` methods with the `System.Runtime.Remoting.Messaging.OneWay` attribute, so that asynchronous message sends are genuinely non-blocking.

With the `Joins` code, simply inheriting from `MarshalByRefObject` will work, but not the way we want. Suppose the client tries to post to some remote distributor `d`:

```
d.Post(msg);
```

In the `Joins` code, `Post` is a *field* containing a delegate value, not a method, so this is equivalent to executing:

```
Asynchronous.Channel<string> channel = d.Post;
channel(msg);
```

This code first synchronously reads a channel from a remote field before asynchronously invoking the channel. Compared to the corresponding Cω fragment, invoking `d.Post(msg)` performs an additional synchronous call, just to obtain the channel. Of course, invoking a synchronous channel in this way would also cost an additional roundtrip.

Our recommended solution is to modify the code to more closely match the Cω solution, using public *methods* `Subscribe` and `Post` that forward their invocations to private *channels* `subscribe` and `post`. The forwarding methods for these asynchronous channels should, of course, be marked `OneWay`.

```
using System.Collections;
using System.Runtime.Remoting.Messaging;
using Microsoft.Research.Joins;

public abstract class ActiveObject : MarshalByRefObject { ...
}
public interface EventSink {
  [OneWay]
  void Post(string message);
}
public class Distributor : ActiveObject, EventSink {
  private ArrayList subscribers = new ArrayList();
  private string myname;
  private Asynchronous.Channel<EventSink> subscribe;
  private Asynchronous.Channel<string> post;
  [OneWay]
  public void Post(string s) {
    post(s);
  }
  [OneWay]
  public void Subscribe(EventSink s) {
    subscribe(s);
  }
  public Distributor(string name) {
    join.Initialize(out post);
    join.Initialize(out subscribe);
    join.When(ProcessMessage).And(subscribe).Do(
    delegate(EventSink sink)
    {
      subscribers.Add(sink);
    });
    join.When(ProcessMessage).And(post).Do(
    delegate(string message)
    {
      foreach (EventSink sink in subscribers) {
```

```
        sink.Post(myname + ":" + message);
      }
    });
    myname = name;
  }
}
```

Nevertheless, it sometimes does make sense to transmit a channel as a first-class value, for example, to hand out an (asynchronous) callback to some remote service. For these scenarios, the `Joins` library guarantees that:

Channels are remotable: the target object of any channel delegate is always an instance of `MarshalByRefObject`;

Asynchronous channels are OneWay: the target method of an asynchronous channel always carries the `OneWay` attribute.

4.5 Beyond Cω: Dynamic Joins (`Samples\JoinMany`)

Section 4.3.2 presented a `JoinTwo` class that waits for precisely two replies, but what if we want to wait for any number of replies, where that number is only known at runtime. In Cω, a particular class can only declare a fixed number of (a)synchronous methods and chords so we have to encode the variable number of replies somehow, perhaps by sharing a single reply channel and counting the number of outstanding replies in a separate private message.

Inspired by a similar abstraction in the CCR [5], the `Joins` library lets you initialize, and join with, *arrays* of asynchronous channels. Since the size of an array can be determined at runtime, this supports *dynamic* join patterns.

Here is a simple example of a class that waits for n replies of type R.

```
using Microsoft.Research.Joins;

public class JoinMany<R> {
  private readonly Asynchronous.Channel<R>[] Responses;
  public readonly Synchronous<R[]>.Channel Wait;
  public Asynchronous.Channel<R> Response(int i) {
    return Responses[i];
  }
  public JoinMany(int n) {
    Join j = Join.Create(n + 1);
    j.Initialize(out Responses, n);
    j.Initialize(out Wait);
    j.When(Wait).And(Responses).Do(delegate(R[] results)
    {
      return results;
    });
  }
}
```

The class declares an array, `Responses`, of response channels, each carrying a value of type R. An object o `= new JoinMany`(n) requires $n+1$ channels: n asynchronous response channels, o`.Responses[i]` $(0 \leq i < n)$, and one synchronous channel, o`.Wait`. The constructor `Create`s a `Join` object supporting $n+1$ channels; it then `Initialize`s the response channels field with an array of n distinct channels and declares a pattern that waits on all the channels in this array. The continuation of the pattern receives all of the responses as a separate array (also of size n) of results of type R. The consumer calls o`.Wait()`, blocking until/unless all responses have arrived; producer i just posts her result r on o`.Response(`i`)(`r`)`, asynchronously. Here, we have taken the precaution of hiding the array in a private field to prevent external updates.

If we are only interested in waiting for multiple signals, not values, we can use this simpler, non-generic, `JoinMany` class:

```
using Microsoft.Research.Joins;

public class JoinMany {
  private readonly Asynchronous.Channel[] Responses;
  public readonly Synchronous.Channel Wait;
  public Asynchronous.Channel Response(int i) {
    return Responses[i];
  }
  public JoinMany(int n) {
    Join j = Join.Create(n + 1);
    j.Initialize(out Responses, n);
    j.Initialize(out Wait);
    j.When(Wait).And(Responses).Do(delegate(){ });
  }
}
```

Because `Responses` is declared as array of (parameter-less) `Asynchronous.Channel` channels, the pattern's continuation receives no argument from joining with `Responses`. Indeed, the continuation does nothing but return control to the caller of `Wait`.

5 `Joins` Library Reference

Programs that use the `Joins` library should reference the assembly

`Microsoft.Research.Joins.dll`

located in the `Assemblies` directory of the installation and, optionally, use the namespace:

`using Microsoft.Research.Joins;`

All of the types described here reside in the `Microsoft.Research.Joins` namespace.

A new `Join` instance j is allocated by calling an overload of factory method `Join.Create`.

`Join` j = `Join.Create`(); or
`Join` j = `Join.Create`(*size*);

The second overload takes an integer *size* argument. It is used to explicitly bound the number of channels supported by `Join` instance j. An omitted *size* argument defaults to 32. The *size* argument provides the value for the constant, readonly property j`.Size`.

A `Join` object notionally owns a set of asynchronous and synchronous *channels*, each obtained by calling an overload of method `Initialize`, passing the location, *channel*(s), of a channel or array of channels using an `out` argument:[7]

j`.Initialize(out` *channel*);
j`.Initialize(out` *channels*, *length*);

The second form takes an integer *length* argument and initializes the location *channels* with an an *array* of *length* distinct, asynchronous channels.

Asynchronous channels are instances of these (nested) delegate types:

`public delegate void Asynchronous.Channel`();
`public delegate void Asynchronous.Channel<`A`>`(A a);

[7]Languages that do not support `out` parameters can use alternative methods `CreateChannel()` and `CreateChannels()` provided by classes `Asynchronous`, `Synchronous`, `Synchronous<`R`>`.

Synchronous channels are instances of these (nested) delegate types:

```
public delegate R Synchronous<R>.Channel<A>(A a);
public delegate R Synchronous<R>.Channel();
public delegate void Synchronous.Channel<A>(A a);
public delegate void Synchronous.Channel();
```

Notice that the outer class of a channel, `Asynchronous`, `Synchronous` or `Synchronous<`R`>`, should be read as a modifier that specifies its blocking behaviour and optional return type.

The various channel flavours support zero or one arguments of type A, and zero or one results of type R. If required, multiple arguments or results can be passed using user-declared tuple types or the provided generic `Pair<`A, B`>` type.

When a synchronous channel is invoked, the caller is blocked until the delegate returns (void or some value). When an asynchronous channel is invoked, there is no result and the caller proceeds immediately without being blocked.

Apart from its channels, a `Join` object notionally owns a set of *join patterns*. Each pattern is defined by invoking an overload of the instance method `When` followed by zero or more invocations of instance method `And` followed by a final invocation of instance method `Do`. Thus a pattern definition typically takes the form:

$$j\texttt{.When}(a_1)\texttt{.And}(a_2)\cdots\texttt{.And}(a_n)\texttt{.Do}(d);$$

Alternatively, using an anonymous delegate for d:

$$j\texttt{.When}(a_1)\texttt{.And}(a_2)\cdots\texttt{.And}(a_n)\texttt{.Do(delegate}(P_1\ p_1,\ldots,P_m\ p_m)\{\ldots\});$$

Here, the initial argument a_1 to `When`(a_1) may be a synchronous or asynchronous channel or an array of asynchronous channels. Each subsequent argument a_i to `And`(a_i) (for $i > 1$) must be an asynchronous channel or an array of asynchronous channels; it cannot be a synchronous channel. The argument d to `Do`(d) is a *continuation* delegate that defines the body of the pattern. Although it varies with the pattern, the type of the continuation is always an instance of one of the following delegate types:[8]

```
public delegate R Continuation(P1 p1, ..., Pm pm);
public delegate void Continuation(P1 p1, ..., Pm pm);
```

The precise type of the continuation d, including its number of arguments, is determined by the sequence of channels guarding it. If the first argument a_1 in the pattern is a synchronous channel with return type R, then the continuation's return type is R; otherwise the return type is `void`.

The continuation receives the arguments of the joined channel invocations as delegate parameters $P_1\ p_1,\ldots,P_m\ p_m$, for $m \leq n$. The presence and types of any additional parameters $P_1\ p_1,\ldots,P_m\ p_m$ varies according the type of each argument a_i joined with invocation `When`(a_i)/`And`(a_i) (for $1 \leq i \leq n$):

- If a_i is of type `Synchronous<`R`>.Channel`, `Synchronous.Channel`, `Asynchronous.Channel` or `Asynchronous.Channel`[], then `When`(a_i)/`And`(a_i) adds no parameter to delegate d.
- If a_i is of type `Synchronous<`R`>.Channel<`P`>`, `Synchronous.Channel<`P`>`, or `Asynchronous.Channel<`P`>` for some type P then `When`(a_i)/`And`(a_i) adds one parameter p_j of type $P_j = P$ to delegate d.

[8] These are actually nested delegate types:

```
public delegate R JoinPatterns.JoinPattern<R>.OpenPattern<P1, ..., Pm>.Continuation(P1 p1, ..., Pm pm);
public delegate void JoinPatterns.JoinPattern.OpenPattern<P1, ..., Pm>.Continuation(P1 p1, ..., Pm pm);
```

so their apparently free type parameters are bound by an enclosing class.

- If a_i is an array of type `Asynchronous.Channel<`P`>`[] for some type P then `When`(a_i)/`And`(a_i) adds one parameter p_j of array type $P_j = P$[] to delegate d.

Parameters are added to d from left to right, in increasing order of i. In the current implementation, a continuation can receive at most $m \leq 8$ parameters. If necessary, it is possible to join more than 8 generic channels by calling method `AndPair`(a_i) instead of `And`(a_i). `And`(a_i) modifies the last argument of the new continuation to be a pair consisting of the last argument of the previous continuation and the new argument contributed by ai.

A join pattern associates a set of (a)synchronous channels with a body d. The body of a pattern can only execute once all the channels guarding it have been invoked. Thus, when a channel is invoked there may be zero, one, or more patterns which are enabled:

- If no pattern is enabled then the channel invocation is queued up. If the channel is asynchronous, then this simply involves adding the arguments (the contents of the message) to an internal queue. If the channel is synchronous, then the calling thread is blocked, joining a notional queue of threads waiting on this channel.
- If there is a single enabled join pattern, then the arguments of the calls involved in the match are de-queued, any blocked thread involved in the match is awakened, and the body of the pattern is executed in one of the threads.
- When a join pattern that involves only asynchronous channels is enabled, then it runs in a new thread. If there are several enabled patterns, then an unspecified one is chosen to run.
- Similarly, if there are multiple invocations of a particular (a)synchronous channel queued up, which call will be de-queued when there is a match is unspecified.

The current number of channels initialized on j is available as readonly property j`.Count`; its value is bounded by j`.Size`. Any invocation of j`.Initialize` that would cause j`.Count` to exceed j`.Size` throws `JoinException`.

Join patterns must be well-formed, both individually and collectively. Executing `Do`(d) to complete a join pattern will throw `JoinException` if d is `null`, the pattern repeats an asynchronous channel, an (a)synchronous channel is `null` or *foreign* to this pattern's `Join` instance, the join pattern is *redundant*, or the join pattern is *empty*. A channel is foreign to a `Join` instance j if it was not allocated by some call to j`.Initialize`. A pattern is redundant when the set of channels joined by the pattern subsets or supersets the channels joined by an existing pattern on this `Join` instance. A pattern is empty when the set of channels joined by the pattern is the empty set (emptiness can only arise from joining empty arrays of channels).

6 Concurrency Demos

These demos can all be found in the `Demos` subdirectory of the `Joins` installation directory.

6.1 Dining Philosophers (`Demos\DiningPhilosophers`)

This demo implements an animated solution to the classic dining philosophers problem. It demonstrates the use of asynchronous messages to carry state (in the `Room` and `Fork` classes, for example) as well as asynchronous messages to coordinate different threads (the `EndMove()` message, for example). There is one thread for each philosopher and another thread (the GUI thread) which runs the animation using a timer.

6.2 Santa Claus (`Demos\Santa`)

This demo implements a solution to a concurrent programming problem about Santa and his reindeer. For a full account of the problem and its Cω solution, please see the associated paper [2].

6.3 Lift Controller (`Demos\LiftController`)

This demo animates the lift controller algorithm described in Chapter 14 of the ERLANG book [1], adding some simulated people to drive the animation. The demo uses one active object (thread) for each lift, lift cabin, floor, and person; the GUI thread runs the animation using a timer. This demo was written from scratch, without relying on existing Cω sources.

6.4 TerraServer Client (`Demos\TerraClient`)

This demo implements a client to the TerraServer web service, which provides USGS aerial imagery (see `http://terraservice.net`). It requires a working connection to the internet (and the web service to be running) in order to work.

Type a city and state name into the text boxes, click one of the buttons and an aerial photograph will be displayed (this may take 30 or 40 seconds, even with a fast connection). TerraServer stores imagery as small 'tiles', several of which must be retrieved and pasted together to make a whole picture. If you click 'Synchronous' then the individual tiles are requested and retrieved one at a time from the server. If you click 'Asynchronous' then asynchronous requests for all the tiles are fired off together and the results are then processed as they arrive. Making the requests asynchronously is usually significantly faster, as shown by the time displays under each button.

The demo illustrates 'impedance matching' between the usual .NET asynchronous calling pattern and the library's asynchronous channels.

6.5 Stock Trading Application (`Demos\StockApp`)

This demo illustrates distributed programming using `Joins` with .NET Remoting. It comprises a command-line server application and a GUI client application. Multiple clients (possibly running on different machines) can connect to an instance of the server and make bids and offers for stocks. The server matches up buyers and sellers and notifies them of the outcome when there is a trade.

References

[1] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996.

[2] N. Benton. Jingle bells: Solving the Santa Claus problem in Polyphonic C$^\#$, `http://research.microsoft.com/~nick/santa.pdf`, March 2003.

[3] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C$^\#$. In B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, number 2374 in LNCS. Springer-Verlag, June 2002.

[4] N. Benton, L. Cardelli, and Cédric Fournet. Modern concurrency abstractions for C$^\#$. *ACM Transactions on Programming Languages and Systems*, 26:769–804, September 2004. An extended abstract appears in ECOOP'02 (LNCS 2374).

[5] G. Chrysanthakopoulos and S. Singh. An asynchronous messaging library for C$^\#$. In *Synchronization and Concurrency in Object-Oriented Languages (SCOOL), OOPSLA 2005 Workshop*. UR Research, October 2005.

[6] G. S. Itzstein and D. Kearney. JoinJava, `http://joinjava.unisa.edu.au`.

[7] Microsoft Research. Cω, `http://research.microsoft.com/Comega`.

A Cω Concurrency Constructs Reference

A.1 Syntax

This section gives a nearly-formal specification of the extensions to the C# grammar associated with Cω concurrency constructs.

A.1.1 Asynchronous methods

Asynchronous methods have a return type of `async`, which is a new keyword in Cω (similar to `void`). Each asynchronous method in a class must have an explicit *async-declaration* of the form

$$\textit{attributes}_{opt}\ \textit{method-modifiers}_{opt}\ \texttt{async}\ \textit{member-name}\ \texttt{(}\ \textit{formal-parameter-list}_{opt}\texttt{);}$$

*Async-declaration*s can appear in interfaces and one can also declare asynchronous delegate types:

$$\textit{attributes}_{opt}\ \textit{delegate-modifiers}_{opt}\ \texttt{delegate async}\ \textit{identifier}\ \texttt{(}\ \textit{formal-parameter-list}_{opt}\texttt{);}$$

A.1.2 Chords

Chords (or synchronization patterns) define the behaviours of asynchronous methods and of those (ordinary) synchronous methods which synchronize with asynchronous ones. The definition *method-declaration* in the C# grammar is replaced by the following in Cω:

$$\textit{sync-method-header}\ \textit{sync-chord}^{+}$$

where *sync-method-header* is

$$\textit{attributes}_{opt}\ \textit{method-modifiers}_{opt}\ (\texttt{void}|\textit{type})\ \textit{member-name}\ \texttt{(}\ \textit{formal-parameter-list}_{opt}\texttt{)}$$

and *sync-chord* is

$$(\texttt{\&}\ \textit{async-header})^{*}\ \textit{method-body}$$

where *async-header* is

$$\textit{member-name}\ \texttt{(}\ \textit{formal-parameter-list}_{opt}\texttt{)}$$

There are also another form of *class-member-declaration* in Cω, viz. *async-chord*, which is defined by

$$\texttt{when}\ \textit{async-header}\ (\texttt{\&}\ \textit{async-header})^{*}\ \textit{method-body}$$

where `when` is another new keyword.

A.1.3 Well-formedness conditions

- Asynchronous methods may not have `ref` or `out` parameters.
- Static and non-static methods may not be mixed within a chord.
- The *member-name* in any *async-header* within a class declaration must correspond to an *async-declaration* of the same name and parameter types within that class.
- Although a given asynchronous method (identified by name and parameter types) may occur in more than one *sync-chord* and/or *async-chord*, all the asynchronous methods within a single chord must be distinct. Within a single class, a particular synchronous method may only occur in a single *method-declaration*.
- For each *sync-chord* in a given *method-declaration*, the formal parameters to the synchronous method and the formal parameters in each *async-header* in the chord must all be distinct. Similarly, the parameters in each *async-header* in an *async-chord* must all be distinct.

- The bound variables in each *method-body* in a *method-declaration* are the formal parameters of the synchronous method unioned with those of all the *async-header*s in the enclosing *sync-chord*. Similarly, the bound variables of a *method-body* in an *async-chord* are all the parameters of the associated *async-header*s.
- The returned value (if any) of each *method-body* in a *method-declaration* must match the declared return type of the synchronous method being defined. Method bodies in *async-chord*s may not return a value.
- In a `struct` definition, only static methods may occur in non-trivial chords.
- If a (synchronous or asynchronous) method declaration in a class C is marked `override` then any method which appears in a chord with the *overridden* method (in the superclass where that is declared) must also be overridden in the class C.

This last restriction means that, for example, the following is ill-formed:

```
public class C {
  virtual async g();

  virtual void f() & g() {
    ...
  }
  ...
}

public class D : C {
  override void f() {
    ...
  }
  ...
}
```

Since in `D`, we have overridden the definition of `f()` in `C` but have not also overridden `g()` with which it appears in a chord. (If `g()` were non-virtual then the definition of `C` alone would generate a warning, since `f()` would be declared `virtual` but could never be legally overridden.)

A.2 Typing

We treat `async` as a subtype of `void` and allow *covariant return types* just in the case of these two (pseudo)types. Thus

- an `async` method may override a `void` one,
- a `void` delegate may be created from an `async` method, and
- an `async` method may implement a `void` method in an interface

but not conversely. This design makes intuitive sense (an `async` method *is* a `void` one, but has the extra property of returning 'immediately') and also maximizes compatibility with existing C# code (superclasses, interfaces and delegate definitions) making use of `void`.

A.3 Semantics

An asynchronous method will return (essentially) immediately to its caller without blocking and with no result. Asynchronous methods are automatically given the `OneWay` attribute, so that calls to them using Remoting are genuinely non-blocking.

Each chord involves a non-empty set of methods. A chord is *enabled* on an object (or class, in the case of static chords) when at least one *unconsumed* call has been made to each of those

methods on that object (or class). When an asynchronous method is called on an object (or class) and no chord is thereby enabled, the call is queued. When a synchronous method is called on an object () and no chord is enabled, the calling thread is blocked and queued. When at least one chord is enabled on an object (or class) then (an unspecified) one of them will be chosen to fire, which *consumes* (dequeues) one (unspecified) call of each of the methods in the chord and executes the body of the chord with the formal parameters bound to the arguments of those calls. In the case of a synchronous chord, the body is executed in the reawoken thread which made the consumed synchronous call. In the case of an asynchronous chord, the body is executed in a new thread.