Cω is an extension of the C# language with two new concurrency constructs based on the join calculus: asynchronous methods and chords. We will describe both of them and we will apply to solve some variants of the Producer-Consumer Problem.

Keywords: Producer-Consumer Problem, asynchronous methods, chords

1. Cω Language Overview

The Cω language was designed in 2002 by a team from Microsoft Research led by Nick Benton and Luca Cardelli. The new language is described by Microsoft as a strongly typed, data-oriented programming language to bridge semi-structured hierarchical data (XML), relational data (SQL) and the .NetCTS (Common Type System). Additionally, Cω extends C# with new asynchronous concurrency abstractions, based on the join calculus [2,5].

The language provides a powerful, and yet simple, model of concurrency which is applicable both to multithreaded applications running on a single machine and to the large event-based applications communicating over a wide area network.

The .NET execution engine provides a multi-threaded execution environment with synchronization based on locks [4]. The C# language includes a lock statement, which obtains the mutex associated with a given object during the execution of a block. Also, the .NET libraries implement a couple of traditional concurrency control primitives such as semaphores, mutexes and reader/writer locks, as well as an asynchronous programming model based on delegates. However, there is a serious mismatch between the 70's model of concurrency on a single machine (shared memory, threads, synchronization based on mutual exclusion) and the asynchronous, message-based style that one uses for programming web-based applications and services.

Conventional methods, inherited from C# language, are synchronous, so the caller must wait until the call is completed. In opposite, if a method is declared asynchronous then any call to it is guaranteed to complete immediately. Asynchronous methods never return a result or throw an exception.

The asynchronous methods are declared by using the async keyword instead of void in the header of the method. The call of an asynchronous method is like sending a message or posting an event. Since asynchronous methods have to return immediately, they can be used to schedule for execution a method which needs a long time to be completed. This usage is actually rather rare in Cω. More commonly, asynchronous methods are defined using chords, as described below, and do not necessarily require new threads.

A chord consists of a header and a body. The header is a set of methods separated by ‘&’. The declaration of a chord must to include at most one synchronous method (which must to be first written in the header) and at least one asynchronous method. The body is only executed once all the methods in the header have been called. Method calls are implicitly queued up until/unless there is a matching chord.

Let's consider, for example, the class below [2]:

```csharp
public class Buffer
{
    public async Put(string s);
    public string Get() & Put(string s) {
        return s;
    }
}
```

1303
The code above defines a class `Buffer` with two instance methods, which are jointly defined in a single chord. The method `string Get()` is a synchronous method taking no arguments and returning a string, and method `async Put(string s)` is asynchronous, so it returns no result and takes one string argument.

If `b` is a instance of class `Buffer` and the call of the synchronous method `b.Get()` occurs, then we have are two possibilities:

1. if there was previously an unmatched call to `b.Put(s)` then there is now a match, so the pending `Put(s)` is dequeued and the body of the chord runs;
2. if there are no previous unmatched calls to `b.Put(s)` then the call to `b.Get()` is blocked until another thread supplies a matching `Put()`.

Conversely, if a call to the asynchronous method `b.Put(s)` occurs, the caller never waits, but there are two possible behaviors with regard to the other threads:

1. if there was previously an unmatched call to `b.Get()` then there is a match now, so the pending call is dequeued and its associated blocked thread is awakened to run the body of the chord;
2. if there are no pending calls to `b.Get()` then the call to `b.Put(s)` is simply queued up until one arrives.

2. The Producer-Consumer Problem for a Running Belt with Finite Length

Firstly, we review the terms of this variant of the Producer-Consumer Problem: "A producer makes objects and places them, one by one, at an extremity of a bounded running belt. At the other extremity of the same belt the consumer takes objects, one by one too, and uses them. The producer cannot place an object on a full belt and the consumer cannot take an object from an empty belt. At any time, either the producer or the consumer can access the belt."

The program below solves this problem, using the concurrency constructs which are provided by the language C#, starting with the theoretical solution described in [1,3]:

```csharp
public class Belt {
    private async FreePlaces();
    private async OccupiedPlaces(int x);

    public Belt(int length) {
        for (int i=0; i<length; i++) FreePlaces();
    }

    public void Put(int x) & FreePlaces(){
        //First chord
        Console.Write("P " + x + " ");
        OccupiedPlaces(x);
    }

    public int Get() & OccupiedPlaces(int x) { //Second chord
        Console.Write("C " + x + " ");
        FreePlaces();
        return x;
    }
}

public class Test {
    private static Belt b = new Belt(3);
}
```
private static Random r = new Random();

private static async Producer() {
    int i = 10;
    while (i < 100) {
        Thread.Sleep(r.Next(10));
        b.Put(i++);
    }
}

private static async Consumer() {
    while (true) {
        b.Get();
        Thread.Sleep(r.Next(30));
    }
}

public static void Main()
{
    Producer();
    Consumer();
}

The Producer and the Consumer act in the following manner:
1. the constructor of Belt class makes a number of calls of the asynchronous method FreePlaces equal with the length of the belt;
2. the Producer can place an object on the belt, calling the synchronous method Put, if the belt isn't full. In this case the body of the first chord is executed, which will make a call of the method OccupiedPlaces to update the configuration of the belt;
3. the Consumer can take an object from the belt, calling the synchronous method Get, is the belt isn't empty. In this case the body of the second chord is executed, which will make a call of the method FreePlaces to update the configuration of the belt.

3. The Producer-Consumer Problem for a Running Belt with Infinite Length
Firstly, we review the terms of this variant of the Producer-Consumer Problem: "A producer makes objects and places them, one by one, at an extremity of an infinite running belt. At the other extremity of the same belt the consumer takes objects, one by one too, and uses them. The consumer cannot take an object from an empty belt. At any time, either the producer or the consumer can access the belt."

The program below solves this problem, starting with the theoretical solution described in [1,3]:

using System;
using System.Threading;

public class Belt
{

In this variant, the Producer can make any number of calls of the asynchronous method \textit{Put} (due to the infinite length of the belt). When the Consumer will invoke the synchronous method \textit{Get} the body of the chord from class \textit{Banda} and he will get one object from the belt. It's easy to see that the body of the chord can be executed only after a minimum one call of the method \textit{Put}, so the Consumer cannot try to take an object from an empty belt.

4. Conclusions
The concurrency is an important factor in the behavior and performance of modern programs: these type of programs are difficult to design, write and debug. Moreover, most popular programming languages treat concurrency not as a language feature, but as a collection of external libraries and this fact has undesirable consequences. The \textit{Cco} language advantage is that it has such features embedded in the language, so the
compiler can analyze them, and can produce better code and warn programmers of potential and actual problems.

On the other hand, asynchronous events and message passing are more and more used at all levels of software systems and very often we find situations where there are many asynchronous messages to be handled concurrently, and where many threads are used to handle them. Threads are still an expensive resource on most systems, so the it's a big advantage to use the messages and threads behind a language mechanism, because a compiler may optimize the generated code.

For these reasons we suppose (and hope) that the $C\omega$ language will be used on a larger scale in the development of distributed applications.

**Bibliography**