CSP for .NET Based on JCSP.

Conference Paper · January 2006
Source: DBLP

CITATIONS
3

READS
44

2 authors, including:

Kevin Chalmers
Edinburgh Napier University
35 PUBLICATIONS **155** CITATIONS

SEE PROFILE

Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

CSP for .NET Based on JCSP

Kevin CHALMERS and Sarah CLAYTON
School of Computing, Napier University, Edinburgh, Scotland, EH10 5DT
{k.chalmers, s.clayton}@napier.ac.uk

Abstract. We present a CSP framework developed for the .NET platform, building upon the ideas developed for the JCSP library. Discussing the development of the core functionality and then onto extra features in .NET that can be taken advantage of, we have created an initial platform that can provide simple development of CSP style process networks. However, we demonstrate that the Microsoft .NET implementation is more resource hungry for multi-threaded applications than other approaches considered in this paper.

Keywords. CSP, .NET Framework, JCSP, Performance Evaluation

Introduction and Motivation

JCSP [1, 2] has long been the mainstay of cross platform CSP style development. More recently, the Transterpreter [3] has brought the occam language to a number of new platforms, and hopefully a number of new users. However, what appears to be lacking is a library that is not only cross-platform, but also cross-language, and the Microsoft's .NET Framework addresses some of these issues.

The original aim and motivation of this work was to develop architecture more specific to .NET for the development of process based system, and that not only are the basic constructs developed for JCSP replicable in .NET, but also in a language independent manner. As time went on, it became apparent that a number of features within .NET could be exploited within a CSP implementation, adding to the overall simplicity of the library. The outcomes of comparison tests with JCSP are surprising however. In Section 1 we provide some background into .NET and the JCSP framework. In Section 2 we describe the core functionality of CSP .NET. Section 3 a discussion on the development of a typed channel is presented and Section 4 describes using methods as processes. In Section 5 we update the existing plug and play components to utilize .NET features, before moving onto performance comparisons in section 6. Section 7 draws conclusions in comparison to other platforms, before we move onto possible future work in Section 8.

1. Background

JCSP, which has recently been released under an open source license (www.jcsp.org), provides a set of basic features allowing the creation of occam style process networks locally and distributed across a network. Constructs such as channels and parallels have been developed, and they emulate the simplicity of such an approach to concurrent and parallel systems. This package has already been implemented in .NET by compiling JCSP source code under J#, but there are some disadvantages to this approach as we shall discuss in Section 2 and illustrated in the results gathered in Section 6.

1.1 What is .NET?

The roots of .NET go back to 1997. In an attempt to unify its many code management schemes, Microsoft set about creating a Common Language Runtime (CLR) that would cater for the diverse programming languages marketed by the company. Contrary to Sun's primary aim with Java, of providing a language that was portable onto many hardware platforms, Microsoft's aim was to provide a single code management and runtime software platform for many languages. As will be discussed later in this section, .NET is delivered with support for a number of languages (C#, C++, J# and Visual Basic) and has been extended to support other languages (e.g. Python).

In an effort to encourage the adoption of .NET on non-Windows platforms, and generalize its use, Microsoft submitted C# and the Common Language Infrastructure to the European Computer Manufacturers Association (ECMA) for development into a standard. This was ratified in 2003. Since then, .NET has been ported onto various platforms with the open source implementation Mono (Linux, Solaris, Mac OS X, Windows, and UNIX). Another open source implementation also exists called the Shared Source Common Language Infrastructure, codenamed ROTOR, which is currently in its second version.

1.2 .NET vs. Java

There are striking similarities between Java and C# [4, 5]. They are both derived from C, both have garbage collection, both compile to a form of bytecode, both execute using Just In Time (JIT) compilers and are both object orientated. In practice, they also compare well in terms of performance [6]. C# also has a number of features not present in Java, the main ones considered here are properties, indexers, delegates, operator overloading and generics.

One striking omission from current comparisons appears to be multithreaded tests, something we hope to address in part later in Section 6. C# has a far richer set of concurrency constructs than Java. While Java supports concurrency at the object level, C# provides a number of static classes and objects to implement thread safe synchronization. A far from exhaustive list of these is:

- `Monitor` – restricts access to blocks of code
- `Semaphore` – controls access to a pool of resources
- `Mutex` – implements mutual exclusion and is designed for inter-process sharing
- `WaitHandle` – `AutoResetEvent` and `ManualResetEvent` are derived from this base class. These are basic wait and signal primitives.

The `Monitor` class provides two static methods, `Enter()` and `Exit()`, to lock portions of code. The `lock` keyword can entirely hide the operation of a `Monitor` class, in a similar method to Java's `synchronized` keyword. The main constructs used within this .NET implementation of CSP are `Monitor` and `AutoResetEvent`.

A number of other attempts have been made to augment the C# concurrency model. Güntensperger and Gutknecht [7] implemented a simple channel based system similar to CSP with Active C#. However, their attempt involved making alterations to the underlying compiler, without making use of the existing features of the language. As we shall see, C# is more than capable, on its own, of implementing CSP constructs.

1.2.1 .NET Features

We have already mentioned a number of extra features in .NET (properties, indexers, etc) and here we provided some basic examples which will help in the discussion later.

Properties are intrinsic to both Visual Basic and Borland Delphi. These formalize the widely used getter/setter pattern, simplifying code. As the example shows below, three new keywords have been introduced: `get`, `set` and `value`. Lines 2-5 shows the Java equivalent of the C# {8-12}.

```
01  // Java Code
02  public void setNumber(int number) {
03     this.number = number; }
04  public int getNumber() {
05     return number; }
06
07  // C# equivalent
08  public int Number {
09     get {
10        return number; }
11     set {
12        number = value; } }
```

When called in code, this behaves as:

```
13  int aNumber = aClass.Number;
14  aClass.Number = 42;
```

Indexers are built on properties and allow objects to be treated as arrays.

```
15  public class Dinner {
16     private Philosopher[] philosophers;
17     public Philosopher this[int index] {
18        get {
19           return philosophers[index]; }
20        set {
21           if (value != null)
22              philosophers[index] = vale; } } }
```

This would be called in code as:

```
23  Dinner dinner = new Dinner();
24  Dinner[0] = new Philosopher("Schopenhauer");
25  Dinner[1] = new Philosopher("Hegel");
```

Delegates were initially developed for Visual J++. These can be thought of as type safe, object encased function pointers, and solve problems that would otherwise require interfaces in Java or function pointers in C++. The system of event handling in C# is built on delegates, formalization of events being one of the language's key innovations. Delegates that declare, handle and trigger events are tagged with the `event` keyword. Any number of delegates can be added and removed from an event property, using the += and -= operators. In .NET 2.0, anonymous delegates were introduced and these are explained more fully in Section 4.

Operator overloading has long been a feature of C++. The creators of Java had severe reservations about this, and have refused to implement it. However, this is a core language feature of C# and is used widely, for example, in assigning delegates to events.

1.3 Supported Languages

Although originally having a few languages that could be compiled into MSIL (Microsoft Intermediate Language - .NET bytecode), due to the open standard of CLR a number of

projects have been undertaken to allow a much broader range of implementations to operate within the .NET environment. A few of the languages in this (quite comprehensive) list are given in Section 1.3.2. First of all however, we shall look at the core languages that are part of the standard .NET framework.

1.3.1 Core Languages

The core language of the .NET Framework is C# as it provides a high level language equivalent to the underlying MSIL. C# has already been covered in Section 1.2.

Next is the Microsoft mainstay of Visual Basic, now called Visual Basic .NET. The general semantics and syntax have really not changed from Visual Basic 6, except now the .NET Framework class library is used. Functionally, Visual Basic provides the closest comparison to C#, being able to perform almost all tasks with only a few minor omissions.

J# is the Java based language for .NET, and comes with the majority of the Java API as well as still having the .NET class library to fall back on. This should not give the impression that a J# program will run in a Java Virtual Machine, as J# programs, like the rest of .NET, still compile into MSIL. This also means that existing Java class libraries need to be recompiled with J# before they can be used within .NET. Other points to consider are Java's lack of support for key .NET features such as properties and operator overloading. These features are available to J# from already compiled .NET libraries (functionality would be lost otherwise) but the method of implementation leaves a lot to be desired. Java friendly wrappers are provided to J#, for example a property Prop in C# becomes two methods `set_Prop` and `get_Prop`. Thankfully, this is handled by the underlying framework and therefore does not have to be implemented by the developer.

C++ is also a core language in .NET, allowing programs that are both native and MSIL compliant to be developed. It could easily be argued that this is overkill by Microsoft – if you want native code C++ is a better option, if you want MSIL then use C#.

The final language, although unlikely to be used for application development, is MSIL itself. .NET comes with an assembler that will compile MSIL into the equivalent executable files and libraries if need be.

1.3.2 Additional Languages

Table 1. Additional .NET Languages

Language	Supplier	Implementation
Ada	United States Air Force Academy	Ada to MSIL, MSIL to Ada
Cobol	Fujitsu	Fully incorporated
Fortran	Lahay Computer Systems	Fully incorporated
Lisp	Microsoft	Compiles to MSIL
Perl	Active State	Compiles to MSIL
Pascal	Queensland University of Technology	Fully incorporated
Prolog	Edinburgh University	Fully incorporated
Python	Microsoft	Can use MSIL libraries
Small Talk	Small Script LLC	Fully incorporated
SML	Microsoft \ Cambridge University	Fully incorporated

Although the core languages are the ones supported and developed by Microsoft as standard, due to the nature of the underlying MSIL other organisations have developed various language implementations that compile under .NET. Ranging from Perl to Ada, these languages either integrate fully with .NET or compile into MSIL themselves so they

can be executed in a .NET platform. Table 1 summarises the more interesting languages, a fuller list of which can be found at `www.dotnetpowered.com/languages.aspx`.

Other variants of these languages exist and the ones mentioned here have no preference over any other. What Table 1 illustrates is that support for the .NET Framework is growing, and coupled with projects such as Mono, which provides Linux support for the CLR, it can be seen that the .NET framework may have some potential. By allowing both cross platform and cross language capability, Microsoft has provided a framework that builds upon the good elements (as well as the bad) of the Java platform in a language independent manner.

1.4 JCSP Package

JCSP provides a set of packages that allow occam like channel and process structures for the Java platform. Ignoring the later Network Edition for now, there are four basic internal packages: `lang`, `plugnplay`, `util` and `awt`. The `lang` package contains the basic constructs required for a CSP based implementation (processes, channels, parallel, etc), as well as integer based versions of these channels. Everything in .NET is considered an object, so the need for integer only channels is removed.

The plug and play package (and its integer based child) provides a basic set of process components that can be plugged together to create various systems. Examples of these include Numbers (generates the natural numbers), Delta (replicates its input channel across its outputs) and Plex (multiplexes its inputs into one output).

The `util` package aids in the creation of buffers that can be added to channels, thereby removing the normal synchronizing read and write operations. Within JCSP these channels are usually used in conjunction with the `awt` package's Active Components, preventing the created GUI giving the impression that it had locked as it deals with an event. These channels can also be used between standard processes providing a level of buffering between them if required.

JCSP has previously been used within the .NET platform by compiling the source with J#. There are a few issues with this. Firstly JCSP uses the C# keywords in and out to represent the input and output ends of a channel, meaning the @ symbol must be used before any occurrence of in or out. The creation of a new numbers process becomes in this case `Numbers(chan.@out()).`

Secondly JCSP was written for Java specifically, and therefore does not take advantage of some of the extra features of the .NET Framework. In particular, properties, indexers and parameters can be taken advantage of, and adding these features into an implementation that is aimed more specifically at the .NET Framework seems desirable.

One final issue with J# is that although it is developed and distributed by Microsoft, it is not truly a core language of .NET, requiring extensions to run on a standard Windows platform (due to the Java API). The .NET platform aimed at more resource constrained devices (the .NET Compact Framework) has no support for J# at all, only a download for the standard framework is provided.

2. CSP .NET Core Functionality

To build the core functionality, three key elements need to be developed: channels, a parallel construct and an alternative construct. A simple interface to allow an object to be a process is also required, although this is trivial.

2.1 Channels

The most important feature to develop within C# is a working channel. If channels can be developed, then it is quite possible to have a simple system using standard threads that communicate using the channels. Due to the similarities between C# and Java it is possible to follow the same basic model of a single channel object that has a method of supplying an input and output end which are actually the same channel object. For example, the underlying `One2OneChannel` in JCSP is also an `AltingChannelInput` and a `ChannelOutput`. This is of course dangerous as it is possible for a process to try and write to a channel that it should treat as an input channel. Therefore another level of abstraction is provided, giving the impression that a channel supplies an input and output end, but is itself neither of these.

Before examining the structure of the .NET channels, let us first look at how the read and write operations are designed in JCSP.

```
26  public void write(Object obj) {
27     synchronized(rwMonitor) {
28        hold = value;
29        if (empty)
30           empty = false;
31        else {
32           empty = true;
33           rwMonitor.notify();
34        }
35        rwMonitor.wait();
36     }
37  }
38  public Object read() {
39     synchronized (rwMonitor) {
40        if (empty) {
41           empty = false;
42           rwMonitor.wait();
43        }
44        else
45           empty = true;
46        rwMonitor.notify();
47        return hold;
48     }
49  }
```

It is quite possible to replicate this in C#, replacing the Java specific commands (`notify` for example) with the equivalent in C# (`pulse`), but there are two problems with this. Firstly, the full range of monitor methods are not available in the .NET Compact Framework (which this initial version is aimed at), and secondly it does not take full advantage of the extra concurrency constructs that are available. As previously mentioned, a simple wait and signal construct is available in the form of the `AutoResetEvent` and this can be used to provide a simpler channel. The channel requires just two `AutoResetEvents`, put and get, both set to false (non-signaled) to begin with. First let us look at the write operation.

2.1.1 Write

During a write operation on a channel, only three steps need take place: put the object in the channel `hold` {51}, release `get` {52} and wait on `put` {53}. In C# this is:

```
50  public void Write(object obj) {
51     hold = obj;
52     get.Set();
53     put.WaitOne(); }
```

Initially it is always possible to perform a write to a channel, and then the get is set allowing the corresponding read to be performed. The process that is writing then waits on the put event which needs to be released by the corresponding read operation.

2.1.2 Read

Similarly to the write operation, read only involves three steps: wait on `get` {55}, release `put` {57} and return the channel `hold` {58}. However, due to the underlying reference semantics of C#, this in fact takes four lines of code:

```
54  public object Read() {
55     get.WaitOne();
56     object toReturn = hold;
57     put.Set();
58     return toReturn; }
```

The read will initially always wait on the AutoResetEvent `get` until the corresponding write has occurred. Then, as we are dealing with a reference to the object in the `hold`, we create a new reference to it. `put` is released allowing the write to continue and the new reference to the `hold` is returned.

2.1.3 Channel Properties

As stated, `in` and `out` are keywords within C#, so using `in()` to get the input end is not possible. As C# has component style properties built into the language, we can use this instead to provide relevant input and output ends as `chan.In` and `chan.Out`. Capitalizing removes the problem of in and out being recognized as keywords, and follows the .NET coding convention.

```
59  public ChannelInput In {
60     get { return this; } }
61  public ChannelOutput Out {
62     get { return this; } }
```

Only `get` methods within the property are developed, as we do not wish to change the `In` and `Out` property. These property methods are no different to the `in()` and `out()` methods in Java, it just gives the impression that the channel ends are in fact a property of the overall channel.

Various differences in these operations are necessary in implementations of shared channel ends (Any2One, One2Any, Any2Any) and `AltingChannelInput` ends, which we do not discuss here. The alting end will be discussed further in Section 2.3, when we discuss the Alternative construct.

2.2 Parallel

Whereas the channels can be made simpler in implementation than JCSP channels, the Parallel needs no such modifications as it is a very well engineered construct. Some additions can be made to take advantage of the `params` keyword to provide a variable number of parameters in the constructor. This removes the need to create an array of

processes before creating the Parallel, as in JCSP, as the underlying framework deals with this for us. It is now possible to call `new Parallel(process, …)` providing one or more processes to the constructor.

2.3 Alternative

Conceptually, this is considered one of the hardest constructs to both create and envision. Basically, a select operation on an alternative performs the following four tasks:

- Enable the Guards
- If no guards are enabled wait on a signal
- Disable the enabled Guards
- Return the selected Guard

Therefore a simplified select method looks like:

```
63  public int Select() {
64     EnableGuards();
65     if (selected == -1)
66        signal.WaitOne();
67     DisableGuards();
68     return selected; }
```

The `EnableGuards` method iterates through all the provided Guards checking to see if they are ready by calling an `Enable` method on them. Here, we shall simplify this to starting at the first Guard in the list and iterating towards the last. In reality, we determine which Guard to start at based on the type of select called (Pri or Fair).

```
69  private void EnableGuards() {
70     for (int i = 0; i < guards.Length; i++) {
71        if (guards[i].Enable(this)) {
72           selected = i;
73           return; } }
74     selected = -1; }
```

If any Guard is found to be ready {71}, then selected is set to its index {72} and the method returns {73}, otherwise selected is set to -1 {74}. The `Enable` method is implemented by each Guard in a slightly different manner (for example a Skip Guard always returns true). An `AltingChannelInput` end looks like:

```
75  private bool Enable(Alternative alt) {
76     lock (rwMonitor) {
77        if (isEmpty)
78           this.alt = alt;
79        return !isEmpty; } }
```

A synchronized block of code must be entered {76} before checking if the channel is empty {77}. If it is, the alternative registers with the channel {78} and finally, the method returns the status of the channel {79}.

This adds three new properties to our channels, both of which significantly change the read and write operations. The actual form of an One2One channel becomes:

```
80  public void Write(object obj) {
81     lock (rwMonitor) {
82        isEmpty = false;
```

```
83         hold = obj;
84         if (alt != null)
85            alt.Notify(); }
86      get.Set();
87      put.WaitOne(); }
88   public object Read() {
89      get.WaitOne();
90      isEmpty = true;
91      object toReturn = hold;
92      put.Set();
93      return toReturn; }
```

The `rwMonitor` is required to ensure safe usage of the `isEmpty` field. The `alt` field relates to the Alternative construct that the channel end belongs to, and is set when `Enable` is called. `isEmpty` relates to the state of the channel, and is checked during the `Enable` and `Disable` calls. `Disable` on this channel simply sets the `alt` to `null`, and returns the current state of the channel. The `Notify` method for the Alternative sets the signal so that the Alternative can fire.

The `DisableGuards` method does exactly the same as the `EnableGuards` method but in reverse, starting from the selected guard if there is one.

```
94   public void DisableGuards() {
95      if (selected == -1) {
96         for (int i = guards.Length - 1; selected >= 0; i--) {
97            if (guards[i].Disable())
98               selected = i; } }
99      else {
100        for (int i = selected; selected >= 0; i++)
101           guards[i].Disable(); } }
```

First of all, a check is made to see if a guard was previously selected during the enable operation {95}. If there was not, the guards are iterated through in reverse, and if `Disable` returns true then selected is set to the guard's index {96-98}. Otherwise, if a guard was selected, each guard that was previously enabled is simply disabled and `selected` kept at the same value. This operation is also simplified, as account would have to be taken for fair select also.

The Disable operation on each guard is also dependant on their individual operations. Here, only the `One2OneChannel` operation is given.

```
102 public bool Disable() {
103    lock (rwMonitor) {
104       alt = null;
105       return !isEmpty; } }
```

A locked piece of code is entered {103} then the `alt` within the channel set to null {104}. The method then returns whether the channel is empty or not {105}.

2.4 Channel Lists

Unlike Java, C# provides methods of operator overloading, and in particular the use of array syntax is interesting. Microsoft has allowed objects to be treated as arrays (indexers) built on the properties concept. For example, consider a `MyClass` which has a collection of `MyInnerClass` objects. `MyClass` methods can be called (`MyClass.Method()`), as can the inner classes methods (`MyClass[3].InnerMethod()`). It is also possible to treat these inner objects as properties (`MyClass[1] = new MyInnerClass()`). By utilizing dynamic

collections and checking the value of the index, it is possible to have an expanding set of internal classes available within the index.

This can be used for channel lists, an idea also used in the Groovy implementation [8]. As well as Channel Lists, we can also develop Channel Input Lists and Channel Output Lists, the latter two both properties of the overall Channel List object. This means a collection of channels can be created, and all the input or output ends accessed at once as a property, or even individually. For example, to access a single channel from a Channel List call `list[1]` and to access all the input ends of the list call `list.In`.

Although not any more efficient as behind the scenes the normal array operations are being carried out, syntactically this is cleaner and easy to understand. Also, due to operator overloading within C#, it is possible to use the line `chans += CSP.CreateOne2One()` to add a new channel to the list or use `-=` to remove one.

3. Generic Channels

The Channel List feature may be a nice addition, but it does not really add anything that cannot be done in standard JCSP. In occam, it is possible to type a channel (e.g. `CHAN INT`), but apart from integer channels, this is missing in JCSP. As everything in .NET is considered an object this could be regarded as not a big issue from an implementation point of view, as primitive data does not have to be wrapped in another object. However, a stronger typed channel does have its advantages, and we can use generics to accomplish this.

3.1 Generics

Current versions of .NET and Java share in common the implementation of generics, a form of parametric polymorphism. That is, all objects that return a value can be parameterized on type, as can constructors and function parameters.

The concept of generics has existed for a long time, and was implemented as templates in C++ but has only recently been adopted in .NET and Java (versions 2.0 and 1.5 respectively). They appear to have many advantages, not least in terms of improving code readability, type safety, performance, and reducing errors [9]. Reportedly C# code using generics is 20% faster than equivalent code without generics [10].

Applying generics to channels is relatively simple. As a channel only contains a single sent object, making this typed requires only making the internal object typed as well as the read and write operations. A typed `One2OneChannel`'s read and write operations become:

```
106 public T Read() {
107    get.WaitOne();
108    T toReturn = hold;
109    put.Set();
110    Return toReturn; }
111 public void Write(T obj) {
112    hold = obj;
113    get.Set();
114    put.WaitOne(); }
```

Providing these new channel types allows the following code to be written:

```
115 One2OneChannel<int> chan = CSP.CreateOne2One<int>();
116 ChannelInput<int> input = chan.In;
117 int I = input.Read();
```

There is no need to cast the returned object as we know it will be an `int`. And unlike the current JCSP implementation, we do not need to create different versions of channels for each primitive type. This addition is also quite possible in the Java implementation.

3.2 Benefits

The most apparent benefit of this method is the type safety it provides. It is not necessary to cast the read object to a particular type and hope it is correct, or to check the type before using it. The second benefit is performance. Converting an object to the base class and then back to the original takes time. This alleged benefit will be analyzed from a channel performance perspective in Section 6.

4. Anonymous Methods

A new feature of .NET 2.0 is anonymous methods, which allow methods without a method header. To understand this in its actual context, we will first return to the idea of delegates.

4.1 Delegates

Delegates are the successor to function pointers in C, and are possible as .NET treats methods as first class objects. An example of a delegate can be given in respect to thread creation in .NET. A `CSProcess` interface requires a run method, and to execute this in a separate thread involves:

```
118 ThreadStart start = new ThreadStart(process.Run);
119 Thread thread = new Thread(start);
120 thread.Start();
```

The delegate is created {118} by giving the `ThreadStart` object the method to run (`process.Run`). This delegate object is then given to the thread so it knows what to execute {119}. As of .NET 2.0 the creation of the `ThreadStart` delegate is not actually necessary, and a Thread can be constructed by calling `new Thread(process.Run)`.

4.2 Process Delegate

As a delegate can be used for thread creation, the same is possible for process creation, allowing their creation without full scale objects. To create a delegate, a template must be provided to describe how the delegate is used:

```
121 public delegate void Process();
```

This marks a process as being a void method requiring no parameters. Adding the ability to use the process delegate within a Parallel allows the code:

```
122 static One2OneChannel chan = CSP.CreateOne2One();
123 static Process Producer() {
124    for (int i = 0; i < 10; i++)
125       chan.Out.Write(i); }
126 static Process Consumer() {
127    for (int i = 0; i < 10; i++)
128       Console.WriteLine(chan.In.Read()); }
129 static Main() {
130    new Parallel(Producer, Consumer).Run(); }
```

First a channel is created in the global context {122} and then two processes are defined {123-128}. The main method then runs these two inside a Parallel {130}.

Two problems with this technique are that the channel has to be created globally so it can be used within the two process methods, and that parameters cannot be passed into the processes. This is where anonymous methods come into play, as it allows the returning of a section of code within a method as a delegate.

```
131 static Process Producer(ChannelOutput output) {
132    return delegate() {
133       for (int i = 0; i < 10; i++)
134          output.Write(i); } ; }
135 static Process Consumer(ChannelInput input) {
136    return delegate() {
137       for (inr i = 0; i < 10; i++)
138          Console.WriteLine(input.Read()); } ; }
139 static Main() {
140    One2OneChannel chan = CSP.CreateOne2One();
141    new Parallel(Producer(chan.Out), Consumer(chan.In)).Run(); }
```

The three main differences here are in the method headers, the creation of the channel within the main body and use of the anonymous method {132 & 136}. Wrapping the internal body of the process inside the `return delegate()` section provides this functionality.

Although not the most elegant looking code, it does allow the development of processes without creating full scale objects with constructors. This isn't the real purpose of anonymous methods; in actuality they remove the need for creating a full scale method for a small delegate.

4.3 Process Wrapper

A problem now exists that may involve a `Process` delegate being used within a `Parallel` that has `CSProcess` objects. The `params` keyword means we expect a list of either `Process` delegates or `CSProcess` objects, not a mixture. To overcome this, a `ProcessWrapper` has been added, which wraps a `Process` delegate within a `CSProcess`. This removes the need to update the internal functionality of the `Parallel` as each passed in delegate within the constructor can be wrapped and treated like a `CSProcess`.

5. Plug and Play Components

One of the best features of JCSP is the Plug and Play Components which allow construction of systems by connecting these processes together. There are a number of these and each provides a different piece of functionality. Replicating this package in .NET seems therefore a worthwhile undertaking, so first let us look at how JCSP uses them.

5.1 The JCSP Way

Within JCSP, components are joined together using previously created channels. For a simple example, we will examine a system that displays the natural numbers to the screen, one every second.

```
142 One2OneChannel[] chan = Channel.createOne2One(2);
143 Numbers num = new Numbers(chan[0].out());
144 FixedDelay delay = new FixedDelay(1000, chan[0].in(), chan[1].out());
```

```
145 Printer print = new Printer(chan[1].in(), "", "\n");
146 CSProcess[] processes = {num, delay, print};
147 new Parallel(processes).run();
```

First two channels are created {142}, and these are used to connect a `Numbers`, `FixedDelay` and `Printer` process {143-145}. The delay is set to 1000 milliseconds {144} and the postfix of each line set to the newline character {145}. These three processes are placed in a `CSProcess` array {146} and then run in parallel {147}.

This method can be exactly replicated within C#, but it is possible to take advantage of properties here. This can help in the visualization of the process network structure for beginners.

5.2 The CSP .NET Way

C# is not only an OO style language, but component based as well. As the processes in plug and play are generally referred to as components, it becomes plausible to treat them as such in .NET. This allows the following code to be used instead:

```
148 Numbers num = new Numbers();
149 FixedDelay delay = new FixedDelay();
150 Printer print = new Printer();
151 num.Out = delay.In;
152 delay.Delay = 1000;
153 delay.Out = print.In;
154 print.Postfix = "\n";
155 new Parallel(num, delay, print).Run();
```

First the three processes are created {148-150}. Then the `Numbers` process is plugged into the delay process {151}. The delay is set to 1000 milliseconds {152} and then plugged into the `Printer` process {153}. The postfix is set to newline {154} and the three processes run in parallel {155}.

The only way to achieve this in Java is to make all the relevant channel attributes of the processes public; or create getter and setter methods. The former is undesirable, and the latter really does not simplify code understanding (`num.setOut(delay.getIn())` for example).

The most obvious question is “where are the channels?” In simple terms, these are created within the `get` and `set` methods for the properties. Channels are either created or set dependant on whether the property is retrieved or set, happening behind the scenes so that the CSP .NET user does not have to concern themselves with their creation.

6. Performance Comparison

The generic tests to compare the performance of two different CSP derived platforms are *CommsTime* and *StressedAlt* [11]. Comparing this implementation to an occam implementation or a C++CSP implementation would be slightly unfair however, so instead a comparison between .NET and Java will be made, with the Transterpreter [3] being given as the high mark for a implementation running on a Virtual Machine.

A number of different tests can be undertaken, based on the types of channels (generic and normal) and the types of process (delegate and normal) being used. It is also possible to test two different compilations on the .NET platform, comparing JCSP compiled under .NET with J# (J#CSP) as well as JCSP compiled under Java 1.5.

Due to the dynamic nature of the anonymous methods, the delegate methods approach is expected to be the worst performing of all the tests, with the others being roughly equal. The test bed used was a 3 GHz Pentium 4 with 512 MB of RAM running Windows XP.

6.1 CommsTime Test

CommsTime involves three processes, a `Prefix`, `Delta` and `Successor`; connected together to create the Natural numbers (essentially the `Numbers` process within plug and play). These are connected to a custom `Consumer` process that records the time taken to produce a number from the processes. For this test we gave a 1000 cycle warm up time, and then read 1 million numbers and worked out the average time from this.

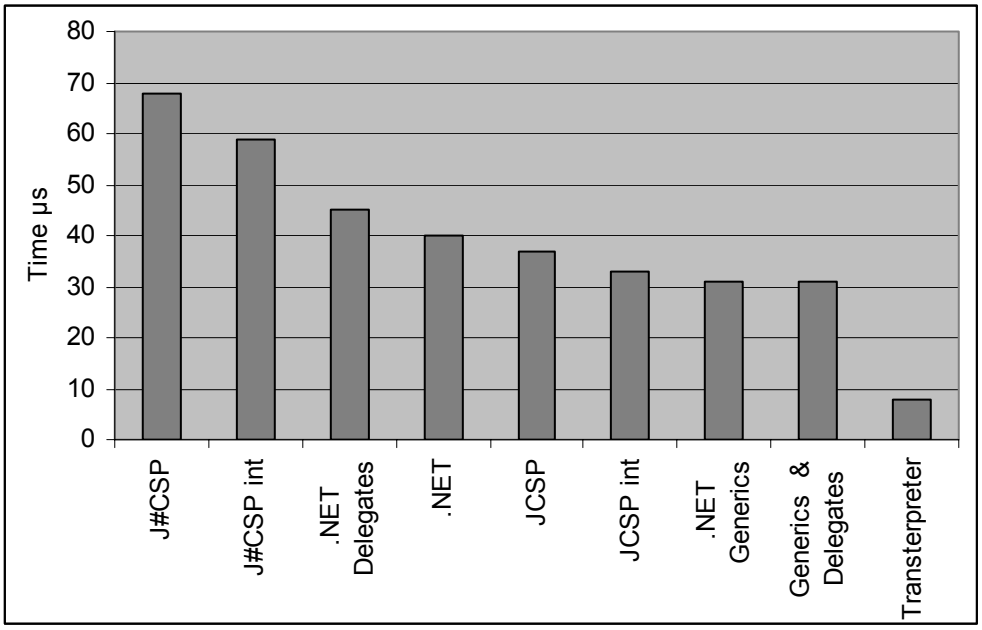

Figure 1. *CommsTime* Comparison

As expected, CSP .NET and JCSP show very little difference; Java performing better when normal channels are used and .NET when Generic channels are compared to int channels. J#CSP performance is significantly different however, and this is most likely due to it being Microsoft's interpretation of Java, and may use wrappers to utilize the .NET Framework underneath. One surprise is the efficiency of the delegate processes. When using normal channels the performance is as expected. However, typed channels provide the same performance as the normal processes with typed channels.

6.2 StressedAlt

The *StressedAlt* test involves an Alternative selecting fairly over many `Any2OneChannel`s, there being multiple writers to each of the multiple channels. If the Alternative is working fairly then the numbers should increase together. The time measurement is taken by performing 1 million select operations, and working out the average time per select. The tests run ranged from 10 channels each with 10 writers, to 20 channels each with 100 writers. The results are presented in Table 2.

Table 2. *StressedAlt* Results

Platform	Time µs					
	10x10	**20x20**	**20x50**	**50x20**	**20x70**	**20x100**
JCSP	8	9	10	10	10	10
J#CSP	14	20	27	21	29	-
CSP .NET	14	29	29	30	33	-

What is immediately apparent is that JCSP by far outperforms the .NET implementations by a considerable margin, and also that Java can cope with many more processes at one time. The most likely explanation for this is that .NET utilizes managed threads. Each thread that is created takes up a significant amount of resources in comparison to a Java thread. To contrast, .NET can create around 1700 threads on the test platform before an out of memory exception occurs, whereas Java can create about 7000.

This shortcoming is far more drastic than this however, as the 1700 .NET threads require over 2 Gbytes of page file; the 7000 Java threads barely registered 600 Mbytes, and it was the .NET virtual machine that ran out of memory. It would appear that although both platforms are comparable in single threaded performance, the .NET threading model can be seen as extremely inefficient in comparison. This is nothing to do with the method of development within CSP .NET as the J#CSP platform suffered the same performance problems.

The other major difference is between the standard .NET implementation and J#CSP at certain points. It would appear that J#CSP is more efficient when the number of processes that share a channel is less. CSP .NET performs comparably well when the number of processes sharing a channel increases, and this is most likely due to differing algorithms within the Alternative's select operation. This will need to be investigated further to determine a better performing approach.

7. Conclusions

Although not suitable for applications which require a large number of threads, .NET does provide a platform with some extra features which add to the functionality as well as the simplicity of understanding. The real benefit is that a library written in one .NET language can be used in a multitude of other languages. The extra capabilities such as indexers and properties also allow a cleaner looking language.

.NET also provides the same mobile code abilities as Java, although not on its reduced framework aimed at mobile devices. Looking at all the features available, it is possible to build up a comparison map of the various CSP implementations: KRoC, the Transterpreter, C++CSP, JCSP and CSP .NET. There are a number of attributes that are important considerations when choosing such a platform – simplicity, performance, mobility features, portability and system integration. Mapping the five implementations gives us Table 3.

Table 3. CSP Implementation Map

Rank	Simplicity	Performance	Mobility	Portability	Integration
1	KRoC	KRoC	JCSP	JCSP	C++ CSP
2	Transterpreter	C++ CSP	C++ CSP	Transterpreter	CSP .NET
3	CSP .NET	Transterpreter	KRoC	CSP .NET	JCSP
4	JCSP	JCSP	CSP .NET	C++ CSP	KRoC
5	C++ CSP	CSP .NET	Transterpreter	KRoC	Transterpreter

Simplicity is judged on how simplistic the structure of the language is, especially in relation to CSP semantics. occam-π (KRoC and the Transterpreter) are naturally very easy to understand as its concurrency model is founded upon CSP. CSP .NET follows these due to constructs such as Channel Lists and the usage of the `params` keyword; and, then, JCSP due to its similarities. C++ CSP comes last due to its less intuitive feel, as well as the necessity to utilize pointers and memory allocation, although this is more a reflection on the C++ language itself.

Performance is fairly obvious, being based on such aspects as context switch time and memory requirements. KRoC is by far the best performing framework, followed by C++ CSP. The Transterpreter leads the virtual machine based implementations, whereas CSP .NET and JCSP, with their garbage collection and other "safety" features are last. As the *StressedAlt* test has shown, .NET is less efficient than Java for multithreaded applications, and this is an area that needs serious addressing. As such, .NET is not a viable platform for serious research in the area of CSP-based communicating process languages and libraries in its current form, as occam provides the power and JCSP provides the platform heterogeneity. In fact, it could be argued that .NET is not fit for multithreaded purposes, as 1700 lightweight processes is not a significant amount. Although ranked just below Java, the gap is large.

Mobility relates to the usage of mobile processes and mobile channels, and the required code loading within the framework. JCSP is by far the most suitable at present, having networked mobile process and channel capabilities. C++ CSP follows due to the nature of the C++ language and object serialization capabilities. KRoC (occam-π) has mobility features due to mobile channels and mobile processes, but these are not (yet) as good as those of JCSP and C++ CSP. CSP .NET has basic single machine mobility (due to references) but as no network capability has been developed yet, it remains at the lower end. Potentially, CSP .NET could rival JCSP in mobility features, but not yet. The Transterpreter sits at the bottom of the scale as it has not yet implemented the full range of mobility mechanisms of occam-π.

The portability of the system relates directly to how many different platforms the resulting code can be executed on. Java, being implemented on everything from mobile phones to web servers, puts JCSP at the top of this table. The Transterpreter comes next as it has been implemented on a number of platforms, and can potentially rival JCSP, as can CSP .NET. Currently .NET is only available as a full scale product on Windows based machines, including the Pocket PC. Even though a Linux version is being undertaken with the Mono project this is still not complete. C++ is of course native, and therefore needs recompiled for every platform it wishes to run in, whereas KRoC is (currently) pretty much Linux specific and loses out here because of this.

The final property, integration, is measured by how well the framework can integrate with the system it is running. C is used by all the other frameworks (apart from C++) to implement any integration that is not available within the standard API. The other four can be shuffled to suit our own interpretation of integration, and here we give CSP .NET the next place due to its ability to call native methods without a wrapper, as well as the size of the .NET API. JCSP has a large API so is next, then KRoC and the Transterpreter due to their smaller API's. This last column is subjective however, and should not be read as a definitive answer.

This tells us three things:

1. If you want performance and simplicity, choose KRoC.
2. If you want mobility and portability, choose JCSP.
3. If you want system integration, choose C++ CSP.

What we have shown is that the same underlying principles and constructs found in JCSP are replicable in .NET, providing a language independent library for process based systems. The main concern is the scaling of performance when a large number of threads are required. These issues can possibly be overcome by using a different implementation of the .NET runtime in the form of Mono which is something that needs investigated.

8. Future Work

There are two main features lacking in CSP .NET in comparison to its closest relative JCSP. First there are no Active Components that use channels to send events instead of having to create event handlers and listeners. Adding this feature can lead to easier to develop GUIs, and judging by the number of components available within .NET this would be a large undertaking.

The second feature missing is network capabilities, and this is also a major undertaking. CSP .NET has the same potential as JCSP in this area, having code loading capabilities as well as object serialization. A possible worthwhile step in this would be to create a framework that would allow CSP .NET to communicate with JCSP, allowing a cross platform distributed systems framework. The possibility of this is questionable, but using J# as a form of buffer may make it possible.

Further work investigating the .NET threading model is of prime importance, especially if it can address the poor performance. Java has been criticized for its threading model [12] and we believe the same criticisms can be leveled at .NET. Analysis as to the behavior of the underlying concurrency constructs as well as testing on other .NET platforms (Mono) needs to be undertaken.

Acknowledgements

We acknowledge the help of Jon Kerridge, in both introducing us to the joy of using parallel techniques, and aiding us in developing a simpler underlying channel. Although quite dubious to begin with, we hope that we have persuaded him that not all languages beginning with C are unusable.

References

[1] P. H. Welch and J. M. R. Martin, "A CSP Model for Java Multithreading," in P. Nixon and I. Ritchie (Eds.), *Software Engineering for Parallel and Distributed Systems,* pp. 114-122. IEEE Computer Society Press, June 2000.

[2] P. H. Welch, J. R. Aldous, and J. Foster, "CSP Networking for Java (*JCSP.net*)," in P. M. A. Sloot, C. J. Kenneth Tan, J. J. Dongarra, and A. G. Hoekstra (Eds.), *Proceedings of International Conference Computational Science – ICCS 2002, Lecture Notes in Computer Science 2330,* pp. 695-708. Springer Berlin / Heidelberg, 2002.

[3] C. L. Jacobson and M. C. Jadud, "The Transterpreter: A Transputer Interpreter," in I. R. East, D. Duce, M. Green, J. M. R. Martin, and P. H. Welch (Eds.), *Communicating Process Architectures 2004 (WoTUG-27),* IOS Press, Amsterdam, The Netherlands, 2004.

[4] D. Gruntz, "C# and Java: The Smart Distinctions," *Journal of Object Technology*, 1(5), pp. 163-176, 2002. Available from http://www.jot.fm/issues/issus_2002_11/article4.pdf.

[5] S. S. Chandra and K. Chandra, "A Comparison of Java and C#," *Journal of Computing Science in Colleges*, 20(3), pp. 238-254, Consortium of Computing Sciences in Colleges, USA, 2005.

[6] J. Singer, "JVM versus CLR: a comparative study", in *PPPJ '03: Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java, ACM International Conference Proceeding Series*, 42, pp. 167 – 169, Computer Science Press, Inc, NY, USA, 2003.

[7] R. Güntensperger and J. Gutknecht, "Active C#," in V. Skala (Ed.), *.NET Technologies' 2004*, pp. 49 - 56. Available at http://dotnet.zcu.cz/NET_2004/NET_2004.htm.
[8] J. Kerridge, K. Barclay, and J. Savage, "Groovy Parallel! A Return to the Spirit of occam?," in J. Broenink, H. Roebbers, J. Sunter, P. H. Welch, and D. Wood (Eds.), *Communicating Process Architectures 2005 (WoTUG- 28),* IOS Press, Amsterdam, The Netherlands, 2005.
[9] A. Kennedy and D. Syme, "Design and implementation of generics for the .NET Common Language Runtime", *SIGPLAN Notices,* 36(5), pp. 1-12, 2001.
[10] J. Clark, "Introducing Generics in the CLR", in *MSDN Magazine,* 21, 2006. Available at http://msdn.microsoft.com/msdnmag/issues/06/00/NET/default.aspx.
[11] N. C. Brown and P. H. Welch, "An Introduction to the Kent C++CSP Library," in J. F. Broenink and G. H. Hilderink (Eds.), *Communicating Process Architectures 2003 (WoTUG-26).* IOS Press, Amsterdam, The Netherlands, 2003.
[12] P. B. Hansen, "Java's Insecure Parallelism," *ACM SIGPLAN Notices,* 34(4), pp. 38-45, 1999.