Communicating Process Architectures – 2003
Jan F. Broenink and Gerald H. Hilderink (Eds.)
IOS Press, 2003

An Introduction to the Kent C++CSP Library

Neil BROWN and Peter WELCH

Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, England.
neil@twistedsquare.com, P.H.Welch@kent.ac.uk

Abstract. This paper reports on a CSP library for C++, developed over the past year at the University of Kent. It is based on the OO-design and API of JCSP and the lightweight algorithms of KRoC **occam**, with extensions to exploit specific C++ capabilities (such as templates). Both user-level and operating system threads are used to provide a range of implementation options and semantics (e.g. for managing blocking system calls intelligently) that run efficiently under either Windows or Linux. The library is presented from the user's point of view, mainly by way of a tutorial. Implementation details are also outlined and some benchmark results given. The performance of C++CSP is between that of KRoC **occam** and JCSP — fairly close to KRoC.

1 Background

This year sees the 25th anniversary of C.A.R. Hoare's paper describing Communicating Sequential Processes (CSP) [1, 2], a new programming model for dealing with parallelism and concurrency that has a strong mathematical basis. Supported by a range of direct implementation technologies, CSP avoids most of the fundamental problems inherent in concurrent design – allowing it to be used with confidence by systems engineers with only a modest amount of training and no specialist mathematical knowledge.

Despite this, no mainstream operating systems or programming languages have built-in support for CSP mechanisms. For concurrency, they offer only unstructured varieties of threads and optional locking mechanisms (such as semaphores or monitors). They provide no support to enforce correct and efficient use of those locks – and cannot be made to provide it. Received wisdom[3] is that this concurrency is hard, should only be used as a last resort (for performance reasons) and only by very experienced designers and programmers. That received wisdom is correct. But CSP turns this around.

Of course, **occam**[4, 5] provides the standard for CSP implementation against which others must be measured – but **occam** is not (yet) a mainstream language. For C, CCSP[6] is a library and run-time kernel that provides the occam model. JCSP[7] and CTJ[8] are CSP/**occam** libraries for Java. This paper reports on C++CSP: a CSP library for C++ that combines an Object Oriented API (derived from JCSP, but with modifications to take advantage of specific C++ capabilities, such as templates) with fast kernel support (along the lines of CCSP and **occam**).

C++CSP runs native under either Windows or Linux/Unix. Applications, therefore, have direct access to all standard Windows – or Linux/Unix – libraries.

This paper provides a user's introduction to the library, describes some of the internal workings and provides early performance figures. It assumes readers have familiarity with the basic ideas and mechanisms of CSP, **occam**, JCSP and C++.

2 Library Details

At the heart of the CSP paradigm are processes and channels – processes being pieces of code that can be executed in sequence or in parallel and channels being the means by which they communicate.

Like JCSP before it, C++CSP adopts an object-oriented approach to implementing CSP. A process is a subclass of `CSProcess`, and channels are classes, such as the standard one-to-one unbuffered channel `One2OneChannel`. Channels are templated classes (the mechanism for generic programming in C++) that can be used to communicate any assignable type, be it a primitive, a "pod" class or a more complex class. In a typical scenario, a parent process creates a channel and passes a reading end (also a class – `Chanin`) to one process and a writing end (again, a class – `Chanout`) to another process. That way, neither child process can access the channel directly to try and obtain a class representing the opposite end of the channel.

As in JCSP, a new process is implemented by declaring a new class as a child of the `CSProcess` class and implementing the `void run()` method (though in C++CSP this is protected rather than public[1]). Unfortunately, it is necessary to pass to the constructor of `CSProcess` the size of the stack needed for the process. This will be explained later.

The C++CSP library provides:

- processes (run either in sequence or in parallel);
- channels (either or both of the reading and writing ends may be *shared*; also, they may be *buffered* using a variety of library or user-defined buffers);
- parallel channel communications (a special case of parallelism included for convenience);
- `ALT`ernatives;
- *buckets* and *barriers*[9];
- *mobile* objects with the same semantics as their (modern) **occam** counterparts;
- easy and secure channel *poisoning* for clean shutdown;
- support for Microsoft Windows (98 and later);
- support for Linux and most other Unix-variants.

3 Tutorial

3.1 The Semantics of Assignment and Communication

Channel communication is just a distributed form of assignment – a value is computed in one process and assigned to a variable in another (after synchronising the two processes of course). In occam, there is the equivalence:

```
FOO x, y:                              FOO x, y:
...                                    ...
x := y              equals             CHAN FOO c:
                                       PAR
                                         c ! y
                                         c ? x
```

[1]This means that processes can only be run by the library itself and cannot be run directly by the user, due to problems with stacks that will be explored later in section 4.2.

Therefore, the semantics of communication must be compatible with the semantics of assignment.

In C++, the meaning of the assignment operator, =, can be overridden to mean whatever the programmer wishes it to mean... and this will be reflected in the meaning of communication in the C++CSP library. So, if a class overrides assignment to mean "shallow copying", a shallow copy of an object will be sent across a C++CSP channel. If a class has not overridden its assignment operator, then a "member-wise" copy (the default for assignment of C++ objects) is received by the reading process. If the channel carries array types, only the pointers will be communicated, since in C++ an array is treated as a pointer. If the channel carries primitive types (int, float etc.), only their values will be communicated.

In this paper, most of the examples work with `int`-carrying channels. If `Bar` is a class, then `Bar`-carrying channels are handled with equal ease - examples are referred to in Sections 5.3 and 6.3.

3.2 Our First Program

We will now look at creating a simple program using the C++CSP library. There will be a writer process that writes 100 integers to a channel then exits, and a reader process that reads 100 integers from a channel and exits. These processes are declared as follows[2]:

```
class Writer : public CSProcess
{
private:
    Chanout<int> out; //writing end of the channel
protected:
    virtual void run(); //overrides the method in CSProcess
public:
    Writer(const Chanout<int>& o); //constructor
};

class Reader : public CSProcess
{
private:
    Chanin<int> in; //reading end of the channel
protected:
    virtual void run(); //overrides the method in CSProcess
public:
    Reader(const Chanin<int>& i); //constructor
};
```

The constructors are as follows (we will use an arbitrarily-sized large stack):

```
Writer::Writer(const Chanout<int>& o)
    :    CSProcess(65536),out(o)
{
}

Reader::Reader(const Chanin<int>& i)
    :    CSProcess(65536),in(i)
{
}
```

The above pieces of code are a general template for any process. Each process is a child of `CSProcess` and must implement the run method. They will have as class members

[2]In all sample code, it is assumed that the namespace of the library, `csp`, is in scope.

any channel ends they require; these are passed to the class constructor. As can be seen above, channel ends can be assigned for ease of use. Following the object-oriented paradigm, channel ends are the same regardless of what type of channel (buffered, shared, or both) they are attached to, allowing any channel type to be used with any process (bar a few exceptions regarding ALTernatives and shared channels). The run methods are given below:

```
void Writer::run()
{
    for (int i = 0;i < 100;i++)
    {
        out << i; //see note below
    }
}

void Reader::run()
{
    int n;
    for (int i = 0;i < 100;i++)
    {
        in >> n; //see note below
    }
}
```

Note: the << and >> operators are overriden so that the indicated above lines expand to `out.write(&i);` and `in.read(&n);` respectively. The ampersands are required because the implementation of our generic channels expects pointers to the source and destination for the communication. However only the integer *values* (not the pointers to the values) will be communicated in the above example.

These functions could hardly be simpler. Internally, input and output calls pass the source and destination respectively as pointers to avoid copying large data structures (pointers are chosen over references to help emphasise that point). Now all that is required is a parent function to glue the components together:

```
void function()
{
    Start_CSP();
    One2OneChannel<int> channel;
    Parallel(
        new Writer(channel.writer()),
        new Reader(channel.reader()),
    NULL);
    End_CSP();
}
```

The `Start_CSP()` and `End_CSP()` calls must be made at the start and end of the section of code in a program that uses CSP to properly initialise and shutdown the library. A channel is declared and its `writer()` and `reader()` methods are called to obtain writing and reading ends of the channel. These are passed to the constructors of the processes which in turn are passed to the vararg `Parallel` function which takes in a varying-size `NULL`-terminated list of processes to be run in parallel. The library takes care of deleting the processes when they have finished executing.

So there is a simple working CSP program. It doesn't print out greetings for the globe but it does what it is intended to do.

3.3 The Poisoned Chalice

At the moment our program is very rigid – the reader expects precisely 100 numbers before it exits. To make our program more flexible we might like to allow any number of inputs before it exits. One way of accomplishing this would be to use a special terminator value but this is a messy solution and not always viable if all integer values might be used in the communication. It can also cause massive deadlock headaches in the design of a shutdown poison path. For this frequently occuring need the library provides a poison mechanism.

Poisoning in other CSP frameworks such as **occam** usually involves poison protocols [10] – a dressed up terminator value solution. C++CSP allows you to poison a channel, a form of "stateful" poisoning. A process can set any channel that it has an end of to be poisoned. Subsequently, any attempts by any process to access this channel will cause a poison exception to be thrown. This can then be caught by the process that is attempting to access the channel, and dealt with by poisoning all channels it can access before exiting gracefully. So we can redesign our writer process as follows:

```
void Writer::run()
{
    for (int i = 0;i < 100;i++)
    {
        out << i;
    }
    out.poison(); //<-- added line
}
```

If we wanted to be cautious then we would include a try-catch block to check for a poison exception in the writer but for our small example it is unnecessary. So now we must modify the reader to keep inputting numbers until the channel becomes poisoned. In fact there is a "common process" supplied with the library that does what we require – the BlackHole process. The code is given here:

```
template <class DATA_TYPE>
class BlackHole : public CSProcess
{
    DATA_TYPE t;
    const Chanin<DATA_TYPE> channelIn;
protected:
    void run()
    {
        try
        {
            while(true)
                channelIn >> t;
        }
        catch(PoisonException)
        {
            channelIn.poison(); //You never know!
        }
    };
public:
    inline BlackHole(const Chanin<DATA_TYPE>& chIn)
        :    CSProcess(65536),channelIn(chIn)
    {
    };
};
```

This code is fairly self-explanatory, fitting the pattern of processes that has already been seen. When the black-hole gets poisoned it will drop to catching the poison exception. At this point (as referenced by the cryptic "you never know!" comment) it poisons all the channels it can. This is technically unnecessary because that must have been the channel that threw the poison exception originally, but it shows what is good practice – had this been an id process then it would had to have poisoned both its channels because there is no way to tell which channel threw the poison exception. Poisoning an already poisoned channel has no effect on the channel (it does not throw a poison exception either).

Replacing our reader process with this black-hole process will produce a program that has the same semantics but is more flexible. The change to the code for `function` is trivial:

```
void function()
{
    Start_CSP();
    One2OneChannel<int> channel;
    Parallel(
        new Writer(channel.writer()),
        new BlackHole<int>(channel.reader()),
    NULL);
    End_CSP();
}
```

3.4 Shared/Buffered Channels

Using a shared channel is simply a matter of declaring `Any2OneChannel` etc instead of the normal `One2OneChannel` and then giving reading and writing ends to the appropriate processes. This can be demonstrated by changing our program so that there are two black-hole processes listening on a one-to-any channel:

```
void function()
{
    Start_CSP();
    One2AnyChannel<int> channel; //change the channel type
    Parallel(
        new Writer(channel.writer()),
        new BlackHole<int>(channel.reader()),
        new BlackHole<int>(channel.reader()),
        // ^^^ - add an extra black hole
    NULL);
    End_CSP();
}
```

Likewise, buffered channels are used by declaring `One2OneChannelX` (note the X) or `One2AnyChannelX` etc and passing to the constructor a buffer object to specify how to buffer the channel (for example limited/unlimited FIFO buffering or overwriting buffering). This follows the mechanisms of JCSP. Making our shared channel buffered requires only the following change to its declaration:

```
Buffer<int> buffer(10);   // declare a buffer of the required type
One2AnyChannelX<int> channel(&buffer);
```

The channel takes its own copy of the buffer to use, to avoid the user passing the same buffer to multiple channels and many channels trying to use the same buffer.

3.5 Parallel Communication

In **occam**, processes may be run in parallel at granularities down to single statements. Here, as in JCSP and CCSP, real classes must be set up before processes may be run in parallel – which can be tedious and expensive. A common CSP/**occam** idiom where this happens is for performing primitive inputs/outputs in parallel, e.g.

```
PROC delta (CHAN INT in?, out.0!, out.1!)
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      PAR
        out.0 ! x
        out.1 ! x
:
```

To support this idiom, C++CSP provides a special `ParallelComm` class. To demonstrate its use, here is a *generic* `Delta` class for doing the same as the above **occam** process, but which can be applied for channels of any data type (using C++ templates), and for any user-defined number of parallel output channels:

```
template <class DATA_TYPE,unsigned OUTPUT_CHAN_AMOUNT = 2>
class Delta : public CSProcess
{
private:
    const Chanin<DATA_TYPE> channelIn;
    DATA_TYPE t;
    ParallelComm* parComm;
protected:
    void run()
    {
        while (true)
        {
            channelIn >> t;
            parComm->communicate();    // outputs t on all channels
                                       // (see below in the constructor)
        };
    };
public:
    inline Delta
        (
            const Chanin<DATA_TYPE>& chIn,
            const Chanout<DATA_TYPE>* const chOut
        )
        :
            CSProcess(65536),channelIn(chIn)
    {
        ParCommItem* pars[OUTPUT_CHAN_AMOUNT];          // the size of
        for (int i = 0;i < OUTPUT_CHAN_AMOUNT;i++)      // the output array
        {                                               // is determined
            pars[i] = chOut[i].parOut(&t);              // by the template
        }
        parComm = CSP_NEW ParallelComm(pars,OUTPUT_CHAN_AMOUNT);
    };
};
```

The poison-related portions of the run method have been omitted for the sake of simplicity. The run method is simple – continually input a data item and perform the parallel communication. This communication is set up in the constructor. An array of (pointers to) `ParCommItem` data types is created and initialised by calling the `parOut` method of each channel writing end to obtain a `ParCommItem` (that will output from the variable `t`).

To construct an instance of this process for integer channels and with, say, 10 outputs:

```
Chanin<int> in;
Chanout<int> out[10];

...  initialise in and out

Delta<int,10> myDelta(in,out);
```

Parallel communication is encapsulated in a class for reasons of expected efficiency and typing problems with implementing it using functions.

This mechanism is currently being developed to a more generalised system that will allow other common actions, such as syncing on a barrier or "falling into" a bucket to be used in a similar way, and this development is likely to lead to a name change to reflect this expansion from pure communications.

3.6 Alternatives

Being able to choose between channels or other events (Alternatives) is one of the fundamental primitives of CSP. As with **occam** and JCSP, C++CSP restricts this to choosing between channel inputs, timeouts and skips. Here is an example from our test suite – a process that repeatedly `ALT`s between two channels and a timeout:

```
class Alter : public CSProcess
{
    const Time t;
    Chanin<char> in0,in1;
protected:
    void run()
    {
        char c;
        Alternative alt(
            in0.inputGuard(&c),
            in1.inputGuard(&c),
            new RelTimeoutGuard(t),
        NULL);

        while (true)
        {
            switch(alt.priSelect())
            {
                case 0:               //input has taken place on in0
                    ...  deal with it
                    break;
                case 1 :              //input has taken place on in1
                    ...  deal with it
                    break;
                case 2:               //timeout expired
                    ...  perform action for timeout
                    break;
            }
```

```
            }
        };
    public:
        Alter(
            const Time& time,
            const Chanin<char>& c0,
            const Chanin<char>& c1
        )
            :   CSProcess(65536),t(time),in0(c0),in1(c1)
        {
        };
    };
```

The ALT is set up in the run method. The constructor takes a list of varying size of pointers to Guard objects that is NULL-terminated (as per the Parallel function). The inputGuard method of the channel reading end fetches an input guard (that will write into c). A “relative” timeout guard is constructed that will timeout after the time length t has elapsed since the beginning of the ALT (absolute timeout guards that expire after time t are also available). The Alternative class is executed using the priSelect method and then its return value (the zero-based index of the guard that fired) can be acted on. Note that there is a difference between this API and the JCSP version: in C++CSP, the guard action (e.g. input) is performed automatically by the select method of the Alternative (whereas, in JCSP, programming that action is the user’s responsibilty). There are arguments for both approaches – the current C++CSP mechanism is open for discussion.

C++CSP offers “PRI ALT”, “fair ALT” and “same ALT” capabilities. The difference between the three occurs when multiple guards are ready; PRI ALTs always pick the earliest guard in the list in this situation. A fair ALT evens out the priority over the course of many calls to the select method by least favouring the last activated guard, and a same ALT favours the last activated guard – useful for guards that will activate infrequently in short bursts. C++CSP also offers an *arbitrary* ALT, where the decision between multiple ready guards is non-deterministic.

4 Implementation Details

4.1 Concurrency/Scheduling

One of the main implementation details of any CSP framework is how the essential element of concurrency is provided. JCSP uses Java’s threads to run processes in parallel. KRoC[5, 11], Kent’s **occam** compiler, uses assembly language to switch workspaces (by altering the stack register). This means that JCSP does not have control over the context switches that occur (the JVM does), but KRoC does have control over the context switches. In other words KRoC is co-operatively multitasking.

C++CSP uses user-space threads to follow the KRoC model of being co-operatively multitasked. Since the library then has control over exactly when context switches occur, many optimisations can be made regarding the need for mutexes and similar synchronisation mechanisms for internal library data because only one process will ever be scheduled at any one time.

If this was the only mechanism used then C++CSP would not be able to take advantage of multiple processors at all as it would only use a single thread (besides which, the optimisations would all be invalid if multiple processors were in use).

One possibility for altering this would have been to create an OS-thread version along the lines of Vella et al. [12, 13]. C++CSP instead uses a many-to-many threading model. The library contains a function to spawn a process in a new thread. This new thread is initialised with a separate user-space kernel as described above and the specified process is run using this kernel in the new thread. Special channels for communicating between threads are available in the library, as is an inter-thread barrier that is used to synchronise processes in different threads (especially at the time of termination). Their algorithms are not discussed in this paper, only the algorithms for the user-space threading.

C++CSP runs under Microsoft Windows by utilising the little-known "fibers" lightweight user-space thread capabilities. For Unix variants, C++CSP uses the standard ANSI C function `setjmp` / `longjmp` on supported processors. This uses a single line of assembly language (the only one in the whole library) to manipulate the stack. It can also use the highly-portable (if a little slow) GNU `pth`[14] library to function under virtually every known Unix-variant.

The scheduler can also be ported to use other scheduling mechanisms (for example an assembly language based scheduler in an embedded device). This is because at its core the scheduler only needs three primitive operations – start process, stop process, and switch to specified process.

4.2 Stacks

Using user-level threads means that either the C++CSP library or a threading library that it uses must allocate the stack from the heap as a normal piece of memory. While Microsoft offer automatic stack-growing in their fibers library, such mechanisms are not provided under the other scheduling methods. This leads to the unfortunate situation where the user must guess the size of stack that a process needs (in a portable library!) and if it is too low, stack overflow and hence memory access violations will occur. Equally, setting too large a size can cause the program's memory requirements to bloat, especially if one process must be run thousands of times in parallel. Profiling tools could potentially be used to aid in getting the stack size right. A possible solution to this could be to automatically check for nearing the end of the stack area in memory and making the library expand the stack, but this would only be possible using the `setjmp` mechanism as the `pth` library does not allow for direct stack manipulation such as this. CCSP has this same problem (although **occam** does not – even for the latest version[5, 15] that includes run-time creation of processes and mobiles).

4.3 Channel Algorithms

The channel algorithms are much simplified by the fact that only one process can actually be running at any one time and that the library is in control of switching between processes. This removes the need for mutexes or monitors to be used with the channel data – it can be accessed directly.

The core of the channel algorithms work on the assumption that the channel is one-to-one – shared channels wrap these functions up with mechanisms to ensure that only one writer and one reader will actually be attempting to access the channel at any time (the other readers and writers must form a queue).

At any point in time, a one-to-one channel that is only being used for normal inputs and outputs can be in one of three states – empty, reader waiting or writer waiting. There will never be a situation where both are waiting because the second party to arrive will see the other waiting and perform the communication. From the point of view of one of the participating parties a channel can only be in two states – empty or another process waiting (since the channel is one-to-one they must be a reader for our writer or vice versa).

So the main data member of a channel is a handle to a process. It can either be a null handle indicating that the channel is empty/unused, or it will point to a waiting process. There should also be a data member to store the destination/source of the communication (for a waiting reader/writer respectively) so that the second process to arrive at the channel can perform the communication itself. So the pseudo-code of either process consists of:

- If the channel is empty:
  - Put our pointer in the channel, and our destination/source pointer
  - Suspend ourselves from the run queue and schedule another process
- Else:
  - Perform the transfer using the waiting process's destination/source pointer
  - Put the waiting process back on the run queue

A simple and elegant algorithm. Of course more complexities are required for the finished library due to the variety of different communication types (shared channels, buffered channels, `ALT`s, extended inputs, parallel communication, and various permissible combinations of these).

One particular problem with the above function is that if the waiting party is performing a parallel communication then they will be "woken up" (put back on the run queue) as soon as one of them completes, regardless of whether the remaining channel communications have completed. So we can use the concept of an "occupied count". Every process will have an occupied count (that basically counts the number of actions it is occupied with). During normal operation it is irrelevant but when the process is waiting on a channel (and therefore not on the run queue) it sets its occupied count to be the amount of communications that it is waiting to finish.

Instead of putting a waiting process straight back on the run queue, the occupied count of the process should be decremented and compared to zero – if it is zero then the process is put back on the run queue, otherwise it is waiting for other transfers and should be left off the run queue for now. Processes wishing to perform normal communications simply set their occupied count to 1 so they will be woken up as soon as the transfer completes, as before.

The last channel procedure that will be examined here is that of `ALT`ing. If a process wants to `PRI ALT` over many channels then it should first check the channels in order to see if a writer is waiting to output on any of them. If there is a waiting writer then the `ALT`ing process can complete the transfer, wake the writer and complete the `ALT`. Complications arise however if there are no waiting writer.

Clearly the `ALT`ing reader should wait on all these channels. If it uses the usual algorithm however then as soon as a writer finds the reader, it will complete the transfer and wake the reader. Should two writers arrive in quick succession (that is, the second arrives before the reader has run again), they will both complete the transfer – clearly an error as an `ALT`ing reader should only ever complete one transfer per `ALT`.

The solution to this is to have the reader place its handle in the channel as normal, but put a null pointer into the data destination address in the channel. Then any writer that finds this null pointer must not perform the transfer – instead it should overwrite the process handle in the channel with its own, set the source address and suspend itself as if the channel were empty, but before it reschedules, it should place the reader back on the run queue.

When the reader wakes up, it knows that *at least* one writer must be ready. It should check the channels (in reverse order, from least to highest priority), checking the process handles in the channels. If it finds its own handle then it should remove it (to make the channel unused again). If it finds a writers handle then it should record the index of the channel, as this will be the highest priority ready writer it has found. When all the channels have been checked, it can perform an input on the appropriate channel and then return. Any writer that it did not input from can be left as it is set-up ready for the next input on the channel.

This is the standard `ALT`ing algorithm, first applied by INMOS for the transputer's microcoded scheduler[16] and subsequently developed for non-transputer **occam** kernels[11] and JCSP[17].

5 Advanced Features

5.1 Extended Rendezvous

C++CSP has some features that extend the core CSP functionality. One idea borrowed from KRoC is that of the extended rendezvous[15]. This is the term coined for when a channel communication takes place between two processes, but the writing process is not free to continue until the reading process orders it (rather than free as soon as the communication has finished). This allows for interesting new functionality – for example, a semantically invisible *snooping* process, inserted into any channel (where the writer is not free until the *snooper* has successfully inspected the data and forwarded it to the reader). C++CSP allows these extended inputs to be performed both stand alone and inside an `ALT`. An example is given in section 6.3.

5.2 Blocking System Calls

As mentioned in the concurrency section, C++CSP allows processes to be placed in separate threads. On a single-processor machine using operating-system threads will be slower to perform normal computations than using the normal user-level threads. The difference arises in relation to blocking system calls (the most common of which are input/output operations).

If a normal process makes a blocking system call, it will cause the entire CSP kernel of that thread to freeze for an indefinite period of time. This could be tolerable in some cases but often this is totally unacceptable. The solution in C++CSP is fairly simple – write the code in a process with no regard for blocking, and use the functionality of the library to create this process in another thread. The process can block without freezing the original thread, and can still communicate with the processes in the original thread by using the special inter-thread channels.

It is hoped that features such as this and the fact that most GUIs and many toolkits are written in C/C++ will allow C++CSP to be used for GUIs – it has been noted before that CSP has a natural fit to graphical interfaces. Unfortunately most GUIs are not thread-safe, a classic example being Java's Swing[3]. Those built on C++CSP, however, will be easy to make thead-safe – simply by conforming to the usual (**occam**) *parallel useage* rules.

5.3 Mobiles

Mobiles [18] are a relatively new addition to the KRoC compiler. They are a compound data structure that has zero-aliasing enforced by the compiler, but they are still referred to by pointers. Whenever the mobile is send down a channel or involved in an assignment, the compiler transfers the pointer but changes the value of the source of the operation to be undefined, thereby ensuring that only one reference to the mobile exists at any one time.

C++CSP translates these semantics into a so-called "smart pointer" class. It holds a single pointer, and whenever an assignment is performed, the pointer in the source mobile is set to

NULL. The library uses the assignment operator inside its communications, so no special case is needed for sending mobiles down channels, they are used as every other object and the mobile semantics are inherently used.

5.4 KRoC Inter-operability

KRoC is an **occam** compiler developed by the University of Kent. It provides ultra-fast context switches using its own user-level kernel. One relatively new feature is its user-defined channel mechanism that allows custom channels to be provided to communicate almost seamlessly with the **occam** code. C++CSP uses this, along with its inter-program channels (that can communicate between any two programs on the same machine run by the same user), to allow C++CSP processes to communicate with **occam**. Both programs can use their own native channels, allowing for very easy inter-operability.

The inter-program channels only allow for one-to-one communication (with no ALTs) but 'id' processes can be used at either end to allow for any variety of communication to be performed.

6 Performance Measures And Tests

6.1 CommsTime

One benchmark[19] that is commonly used for CSP frameworks is the commstime benchmark. This consists of building a simple network with a prefix, successor, delta and custom recorder process to measure how fast it takes the network to produce each number. The results recorded on a Celeron 667 MHz. (256 MBytes SDR RAM) were:

CSP Framework	Time Per Iteration
occam (KRoC 1.3.3)	1.3 microseconds
C++CSP (setjmp)	5 microseconds
C++CSP (GNU pth)	75 microseconds
JCSP (JDK 1.4)	230 microseconds

Unsurprisingly, KRoC is the fastest of all the CSP implementations, due to its heavily optimised and CSP-centred nature. C++CSP is not far behind however – only a factor of 4, despite being high-level and portable. This is a very encouraging result that shows it can be within an order of magnitude of KRoC. The results for GNU pth are included here to show the time differences between the scheduling methods. It is disappointingly slow, given that the library is built using the same setjmp/longjmp functions that C++CSP uses for its own scheduling. Therefore the native C++CSP setjmp implementation is recommended for use over GNU pth wherever possible.

JCSP is two orders of magnitude behind KRoC and C++CSP. This is because it sits on top of whatever OS/JVM thread mechanisms are implementing Java's native threads. On our test platforms, these are based on non-cooperative multitasking with relatively large-grained time slicing. Whilst not particularly fast, this does bring other benefits – namely that blocking system calls do not block the rest of the parallel system.

6.2 Stressed ALT

Another test, from JCSP, is the stressed ALT test. This involves connecting p writing processes to n shared channels and getting one process to fair ALT over the n channels, meaning it will take input from a total of $p \times n$ processes. The library has successfully run this test with 200 processes on each of 100 shared channels, yielding a total of 20,000 processes trying to write to one (very stressed!) reader. This test succeeded, with the fair ALT performing correctly. Subsequently, timings were performed to compare with both KRoC and JCSP:

CSP Framework	Time for test A	Time for test B	Time for test C
occam (KRoC 1.3.3)	0.6 microseconds	0.7 microseconds	1 microsecond
C++CSP (setjmp)	3 microseconds	7 microseconds	10 microseconds
JCSP (JDK 1.4.3)	130 microseconds	200 microseconds	—

Test A ran with 10 channels (n), 10 writers on each (p). Test B ran with 20 channels, each with 100 writers. Test C used 100 channels, each with 200 writers. The times recorded are the average time for each successful input by the reading process. The machine used was the same Celeron 667 MHz.

KRoC's *fair ALTing* mechanism delivers unit time for each choice, regardless of the number of choices being checked — so long as the system is under stress (i.e. messages are always pending on each channel)[3]. The same applies for C++CSP and JCSP. The increase in timing from test A to C is the result of cache misses — the **occam** version of test A fitting entirely on to primary cache!

The times for KRoC and C++CSP are one order of magnitude apart, but it is pleasing to see that C++CSP appears to perform proportionally better on the small test (A).

The results for JCSP are, perhaps, unfairly penalised by the very small cache on the Celeron processor. For example, running Test A on a 233 MHz. Pentium 2, gave a timing of 80 microseconds. Unfortunately, that machine had only 64 MBytes of main memory, which was simply not enough for the 2001 Java threads required for Test B — the system thrashed its disc heavily and timings fluctuated wildly (above 300 microseconds). Test C requires 20,000 Java threads and the consequent disc thrashing brought both these machines to their knees.

6.3 Mandelbrot Fractal

To further demonstrate the abilities of the library, here is an abbreviated version of a system to produce a Mandelbrot fractal. The layout of the network will be very naive and simple – one process for each pixel of the Mandelbrot, with each feeding (via a shared channel) into a single "pixel drawer" process.

The pixel-drawer process could be implemented simply by inputting a pixel (coordinates and colour) and drawing it. The feeder channel could be made buffered but that would make little difference to this uni-processor demonstration. An alternative would be for the drawer process to use extended inputs. This means that outputting a pixel down the channel has the semantics of an imperative procedure call – once the output has completed, the pixel has been drawn. This alternative method is the one that is used below.

The pixel-drawer process listed below includes some synchronisation code to synchronise with the locking/unlocking of a frame buffer. Whenever the process receives an input on the sync channel (apart from the initial one), it waits for a second input; the first message

[3] This was not the case for the *transputer* implementation, which always delivered an $O(n)$ overhead for each choice, where n was the number of choices.

indicates that the frame buffer has been locked, the second that the frame buffer is now unlocked:

```
class PixelDrawer : public CSProcess
{
private:
    Chanin<Pixel> in;
    Chanin<bool> sync;
    Pixel p;
protected:

    static void DrawPixel(csp::CSProcess* process)
    {
        PixelDrawer* t = (PixelDrawer*)process;
        Pixel p = t->p;
        ...  draw the pixel
    };

    void run()
    {
        try
        {
            bool b;
            sync >> b;

            csp::Alternative alt(
                in.extInputGuard(DrawPixel,&p),
                sync.inputGuard(&b),
            NULL);

            while (true)
            {
                if (alt.priSelect() == 1)
                {
                    sync >> b;    // wait for screen to be unlocked
                };
            }
        }
        catch (PoisonException)
        {
            in.poison();
            sync.poison();
        }
    };
public:
    PixelDrawer(
        const csp::Chanin<Pixel>& i,
        const csp::Chanin<bool>& sy
    )
        :   CSProcess(65536),in(i),sync(sy),screen(scr)
    {
    };
};
```

The syntax for extended inputs in alternatives sets up the `extInputGuard` method of a `Chanin` with two parameters – the first is a function that takes a single parameter of type `CSProcess` and returns void, the second must be the destination address for the input. This function is executed for the extended part of the input (before the outputter is freed) and is passed the pointer to the process that is doing the input. In this example, no further action needs to be taken in response to the extended input guard – action is only needed in response to a `sync` input. The code for the main process is given here:

```
class Mandelbrot : public CSProcess
{
protected:
    void run()
    {
        One2OneChannel<Pixel> draw;
        One2OneChannel<bool> sync;

        Barrier barrier(1);
        // ^^^ - initialise with (this) one process already registered

        ...  initialise graphics

        spawnProcess(
            new PixelDrawer(draw.reader(),sync.reader()),
            &barrier
        );

        const int X = 160, Y = 120;

        CSProcess* mandels[X * Y];
        for (int i = 0;i < X;i++)
        {
            for (int j = 0;j < Y;j++)
            {
                mandels[(i * Y) + j] =
                    new MandelbrotPixel(draw.writer(),i,j);
            }
        }

        sync.writer() << true;    // lock the frame buffer

        Parallel(mandels,X*Y);
        // ^^^ - will only return when all the pixels have been drawn

        draw.writer().poison();
        // ^^^ - slight cheat, to poison the channel

        barrier.sync();           // unlock the frame buffer

        sleepFor(Seconds(30));
        // ^^^ - Simplistic way of displaying the image for a while

        ...  de-initialise graphics
    };
public:
    Mandelbrot() : CSProcess(65536*8)
    {
    };
};
```

The `MandelbrotPixel` process simply calculates the pixel colour for its given coordinates, sends the pixel down its channel and exits – omitted here for brevity. Graphics initialisation etc. code has also been omitted. In this example, the `PixelDrawer` process never actually recieves a `sync` message associated with the unlock – it is poisoned instead. The `spawnProcess` function forks off the specified process, taking as its second parameter a pointer to a `Barrier` (C++CSP's implementation of the KRoC/JCSP barrier primitive) that can then be `sync()`ed with to wait for the termination of the forked process.

7 Conclusion

C++CSP is a new but stable library that implements all common CSP paradigms and is highly portable. Possible future developments include more integrated network support, some GUI support and possibly other tools to aid in the development (such as an automatic process template generator). Following JCSP.net[20] and KRoC.net[21], C++CSP.net seems almost required!

The library is still in its infancy however and needs to cultivate a user base while ensuring (and proving) its stability and providing new functionality – without succumbing to so-called "feature-bloat". It is hoped that this library will prove useful to existing CSP users, and hopefully even introduce some C++ users to the useful concept of Communicating Sequential Processes.

The library will be made available on the web by the University of Kent shortly. Enquiries can be made to either author.

References

[1] C.A.R.Hoare. Communicating Sequential Processes. In *CACM*, volume 21, pages 666–677, August 1978.

[2] C.A.R.Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[3] Hans Muller and Kathy Walrath. Threads and Swing. `http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html`.

[4] INMOS LIMITED. Occam 2.1 Reference Manual. Technical Report, May 1995. Available at: `http://www.wotug.org/occam`.

[5] University of Kent at Canterbury. Kent Retargetable occam Compiler. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/kroc/`.

[6] J.Moores. CCSP – a Portable CSP-based Run-time System Supporting C and occam. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering series*, pages 147–168, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press.

[7] University of Kent at Canterbury. Java Communicating Sequential Processes. Available at: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`.

[8] Gerald H. Hilderink. CTJ (Communicating Threads for Java). `http://www.ce.utwente.nl/javapp/`.

[9] Peter H. Welch and David C. Wood. Higher Levels of Process Synchronisation. In A. Bakkers, editor, *Parallel Programming and Java, Proceedings of WoTUG 20*, volume 50 of *Concurrent Systems Engineering*, pages 104–129, Amsterdam, The Netherlands, April 1997. (WoTUG), IOS Press. ISBN: 90-5199-336-6.

[10] P.H.Welch. Graceful Termination – Graceful Resetting. In *Applying Transputer-Based Parallel Machines, Proceedings of OUG 10*, pages 310–317, Enschede, Netherlands, April 1989. Occam User Group, IOS Press, Netherlands. ISBN 90 5199 007 3.

[11] P.H.Welch and D.C.Wood. The Kent Retargetable occam Compiler. In *Proceedings of WoTUG 19*, volume 47, pages 143–166, March 1996.

[12] K.Vella and P.H.Welch. CSP/occam on Shared Memory Multiprocessor Workstations. In B.M.Cook, editor, *Architectures, Languages and Techniques for Concurrent Systems*, volume 57 of *Concurrent Systems Engineering Series*, pages 87–119, Amsterdam, the Netherlands, April 1999. WoTUG, IOS Press.

[13] Kevin Vella, Kurt Debattista, and Joseph Cordina. Cache-Affinity Scheduling for Fine Grain Multithreading. In James Pascoe, Peter Welch, Roger Loader, and Vaidy Sunderam, editors, *Communicating Process Architectures 2002*, volume 60 of *Concurrent Systems Engineering*, pages 321–352, IOS Press, Amsterdam, The Neterlands, September 2002. IOS Press.

[14] GNU. GNU `pth`. `http://www.gnu.org/software/pth/`.

[15] F.R.M.Barnes and P.H.Welch. Prioritised Dynamic Communicating and Mobile Processes. *IEE Proceedings-Software*, 150(2):121–136, April 2003.

[16] M.D.May, P.W.Thompson, and P.H.Welch. *Networks, Routers and Transputers*, volume 32 of *Transputer and occam Engineering Series*. IOS Press, 1993.

[17] Peter H. Welch and Jeremy M.R. Martin. Formal Analysis of Concurrent Java Systems. In Peter H. Welch and Andr W.P. Bakkers, editors, *Communicating Process Architectures 2000*, volume 58 of *Concurrent Systems Engineering series*, pages 275–301, Amsterdam, the Netherlands, 2000. IOS Press.

[18] F.R.M.Barnes and P.H.Welch. Mobile Data, Dynamic Allocation and Zero Aliasing: an occam Experiment. In *Communicating Process Architectures*. IOS Press, 2001.

[19] Roger M.A. Peel. A Reconfigurable Host Interconnection Scheme for Occam-Based Field Programmable Gate Arrays. In Alan G. Chalmers, Henk Muller, and Majid Mirmehdi, editors, *Communicating Process Architectures 2001*, volume 59 of *Concurrent Systems Engineering*, pages 179–192, IOS Press, Amsterdam, The Neterlands, September 2001. IOS Press.

[20] P.H.Welch, J.R.Aldous, and J.Foster. CSP networking for java (JCSP.net). In P.M.A.Sloot, C.J.K.Tan, J.J.Dongarra, and A.G.Hoekstra, editors, *Computational Science - ICCS 2002*, volume 2330 of *Lecture Notes in Computer Science*, pages 695–708. Springer-Verlag, April 2002.

[21] Mario Schweigler, Fred M.R. Barnes, and Peter H. Welch. Flexible, Transparent and Dynamic occam Networking With KRoC.net. In J.F.Broenink, editor, *Communicating Process Architectures – 2003*, volume 61 of *Concurrent Systems Engineering*, Amsterdam, The Netherlands, September 2003. (WoTUG), IOS Press.