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

Automatic Conversion of CSP to CTJ, JCSP, and CCSP

V. RAJU
2861 Tall Oaks Ct. #10, Auburn Hills, MI 48326
varsha_raju@hotmail.com

L. RONG
ASpire Technologies LTD, Hi-Tech Industrial Park, Shenzhen, Guangdong Province, P.R.China
ronglm@21cn.com

G. S. STILES[1]
Computing Laboratory, University of Kent, Canterbury, CT2 7NF, England
dyke.stiles@ece.usu.edu

Abstract. We present tools that automatically convert a subset of machine-readable CSP script to executable Java or C code. CSP is used to design and verify the correctness of large and complex systems of processes that interact only via explicit synchronous messages. These systems can be implemented in Java using CTJ or JCSP, packages that add CSP-like features to Java, or in CCSP, a package that adds similar features to standard C. Implementation of CSP systems can be tedious and error-prone when large numbers of processes and communications are involved, and sorting out errors in channel naming or the ordering of messages can be very time-consuming. The tools we have developed minimize such problems by converting the verified CSP descriptions of communicating processes directly into Java or C code, thus guaranteeing that channels are correctly named and the communications occur in the proper order. This process can significantly cut development time.

1 Introduction

The development of large concurrent applications can be a complex and time-consuming task. The use of formal approaches, such as Hoare's Communicating Sequential Processes (CSP: [1, 2, 3]) simplifies the problem by making the required synchronizations among processes explicitly dependent upon communications. Systems specified in the CSP script can be checked for correctness and freedom from deadlock and livelock with the tools FDR and ProBE [4]. Once verified, the script can then be the basis for developing an actual implementation in languages that directly support the CSP style of concurrency through channels, such as occam [5, 6], and in other languages with the help of packages that add CSP features, such as CTJ [7, 8] and JCSP [9, 10] for Java, CCSP [11] for C, and C++CSP [21] for C++.

The implementation itself can be problematic, however. If there are a large number of processes, connected by many channels, and a reasonably complex pattern of

[1] Permanent address: Utah State University, 4120 Old Main Hill, Logan UT 84322-4120

communications, it may take a substantial amount of time to get the application running correctly. Students compare the complexity to that of wiring by hand a large discrete logic system: there are so many connections that it is nearly impossible to get it right the first time around. The tools developed in this project are designed to avoid these problems by automatically generating the appropriate channel-based Java or C code from the verified CSP script. We have found that this can substantially cut the implementation time.

With this tool we now have a reasonably robust procedure for designing concurrent applications. We begin by specifying the desired behaviour (particularly the communication and synchronization patterns) in CSP. Next we use ProBE and FDR to verify that the implementation satisfies the specification and is free from deadlock and livelock; this typically requires a few iterations, but the turn-around time is faster than when working with actual Java or C code. Once the implementation has been verified we use the conversion tool to generate the appropriate code; this will include the necessary declarations and channel operations for the required communications – correctly named and in the proper order. Depending on the actual problem, additional sequential code may need to be added to the resulting routines. Testing follows and – if the specification was correct – is usually brief.

We are not aware of previous efforts to convert CSP directly into executable code. Many other systems have been implemented, however; e.g., a group at Stony Brook [22] has been developing tools to convert a graphical version of the process algebra CCS into executable Java and Ada'95 code, and at Edinburgh [23] tools are available that convert a stochastic process algebra into Ada.

The following sections discuss first the means of carrying out the conversions from the CSP script to Java or C code and their limitations. This will be followed by several examples of both simple and more complex applications. The paper will conclude with a summary and a discussion of future work.

2 The Conversion

This project originated as a follow-on to a previous exercise [12] that developed routines using Mathematica [13] to automatically convert concurrent CSP structures into completely equivalent sequential versions. The ultimate goal at that time was to build a family of Mathematica-based tools for the manipulation and analysis of CSP scripts. We thus started this project with a Mathematica conversion of CSP to CTJ.

2.1 CSP to CTJ via Mathematica

Mathematica is a very elaborate package, containing a great variety of tools covering the range from numerical and symbolic mathematics through graphics to sophisticated manipulation of strings. The pattern matching and list processing features, when used within Mathematica's procedural programming system, provide a fairly quick path to build a translator. This was the approach used to convert CSP script to CTJ code.

Mathematica is not, however, readily available at all institutions and is fairly expensive at the single-copy level. We thus looked for a more commonly available approach during the second phase of this project.

2.2 CSP to JCSP and CCSP via C++

We wished to implement these translators in C or C++. The bulk of the translation work requires string processing; there is, fortunately, a Standard Template Library (STL) in C++

publicly available [14] that contains some very useful routines in its *String* and *Vector* classes.

In standard C, a string is simply an array of characters that always includes a null terminator. In the C++ String class, string objects associate an array of characters with methods useful for managing and manipulating it. A string object also contains certain housekeeping information about the size and storage location of the data. C++ strings do not include a null terminator. C++ strings greatly reduce the likelihood of common and destructive C programming errors, such as overwriting array bounds and trying to access arrays through uninitialized or incorrectly valued pointers.

The Vector class provides methods for dynamically allocating and reclaiming storage as required. These are useful when working with, e.g., a CSP process, where we must keep track of such details as its name, its channel names and directions, and any internal structures it may have, such as external choice and if-then-else clauses. Different processes will have different amounts of information. Managing the storage required explicitly would be tedious and error-prone. The STL Vector class, however, manages storage automatically.

2.3 Features Converted

The choice of CSP features to convert was based largely on the content of programs that had been developed in our classes over the past few years. These programs tended to model various communication problems. Table I shows the elements that have been translated in the various versions. Semaphores are not explicitly included in CSP, but have been added here since they can easily be modeled in CSP and can be implemented in JCSP and CCSP. The CSP notation in Table I and the rest of the paper is in the machine-readable form used by ProBE and FDR [4].

Table I. CSP Features Translated

CSP	CTJ	JCSP	CCSP
comments: `--`	✓	✓	✓
comments: `{- ... -}`		✓	✓
declarations	✓	✓	✓
prefix: `->`	✓	✓	✓
integer data	✓	✓	✓
`chan ? data, chan ! data`	✓	✓	✓
`chan ? d1 . d2 . ..., chan ! d1 . d2 . ...`	✓	✓	✓
`if ... then ... else ...`	✓	✓	✓
external choice (alternative): `[]`	✓	✓	✓
synchronous (sharing) parallel: `[| {| ... |} |]`	✓	✓	✓
processes	✓	✓	✓
recursive processes	✓	✓	✓
semaphores		✓	✓

This set seems sufficient to put together a good variety of examples. The most glaring shortcoming, from an engineering standpoint, is the lack of floating-point data – but this will soon be remedied. Other features that would be useful are sequential composition, interrupts, timeouts, boolean guards, renaming, and replicated constructs.

2.4 Restrictions

There are two restrictions on the CSP syntax that have been included to simplify the conversion programs; all are compatible with ProBE and FDR. We require first that all CSP statements terminate with the semicolon, and we deal only with integer channels and variables.

3 Examples

We begin by looking at the translation of some of the basic components. This will be followed by increasingly complex examples of full programs.

3.1 Simple Fragments and Programs

We look first at a recursive process `UpHandler` that repeatedly reads an integer into the variable `x` and passes it on. The CSP description is:

```
-- This is a test file for a simple process with recursion;
channel input, output;
UpHandler = input?x -> output!x -> UpHandler;
```

The resulting CTJ code includes the class definition, declarations of variables and channels, the constructor, and the run method:

```
// This is a test file for a simple process with recursion;
import csp.lang.*;
import csp.lang.Integer;
import csp.lang.Process;
import csp.io.*;

public class UpHandler implements Process
{
   private integer x = new Integer();
   private boolean Running;
   private int Identity;
   private Channel_of_Integer input;
   private Channel_of_Integer output;

   public UpHandler ( int Identity,
                      private Channel_of_Integer input,
                      private Channel_of_Integer output)
   {
      this.Identity = Identity;
      this.input = input;
      this.output = output;
   }

   public void run()
   {
      Running = true;
      while(Running)
      {
         input.read(x);
         output.write(x);
      }
   }
}
```

The conversion tool detects the recursion and sets up a loop controlled by a variable `Running` that is set to `true` (and could be modified at runtime by a specific value of the channel input variable `x` if desired). We also automatically add a parameter `Identity` that proves useful when multiple instances of one process are used.

For a process to run an object has to be created and the `run()` method called for this object. This object is created in the main file in the `main()` method of the system class. Calling its `run()` method activates the object. In CTJ the translator generates:

```
...
import csp.lang.System;
public class system
{
   public static void main(String[] args)
   {
      // This is a test input file for a simple process
      private int Identity;
      final Channel_of_Integer input = new Channel_of_Integer();
      final Channel_of_Integer output = new Channel_of_Integer();

      ...

      UpHandler UpHandler =
         new UpHandler(Identity, input, output);
      UpHandler.run();
      java.lang.System.out.flush();
   }
}
```

The next example is the *Stop and Wait* flow-control protocol [3]. The processes *Send* and *Recv* cooperate to manage the flow of data from a channel *in* to a channel *out*. The *Send* process repeatedly accepts an input over the channel *in*, passes it along to the *Recv* process over the channel *mid*, then waits until it receives an *ack* signal from *Recv*. *Recv* does not send the *ack* until it has successfully passed the data to the *out* channel.

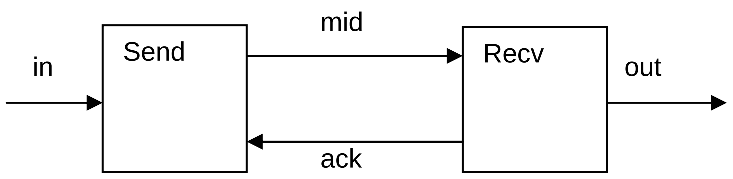

Figure 1. The Stop and Wait protocol.

The CSP description is straightforward:

```
-- Stop-and-Wait Protocol
channel in, mid, ack, out : T ;
Send = in ? x -> mid ! x -> ack ? x -> Send ;
Recv = mid ? y -> out ! y -> ack ! y -> Recv ;
System = Send [| {| mid, ack |} |] Recv ;
```

The complete code, now shown in JCSP:

```
import jcsp.lang.*;
import java.lang.*;
/*-- Stop-and-Wait Protocol */
class Send implements CSProcess
```

```
{
  One2OneChannelInt in;
  One2OneChannelInt ack;
  One2OneChannelInt mid;
  int x;
  boolean Running;

  public Send(One2OneChannelInt in, One2OneChannelInt ack,
              One2OneChannelInt mid)
  {
    this.in = in;
    this.ack = ack;
    this.mid = mid;
  }

  public void run()
  {
    Running = true;
    while (Running)
    {
      x = in.read();
      mid.write(x);
      x = ack.read();
    }
  }
}

class Recv implements CSProcess
{
  One2OneChannelInt mid;
  One2OneChannelInt out;
  One2OneChannelInt ack;
  int y;
  boolean Running;

  public Recv(One2OneChannelInt mid, One2OneChannelInt out,
              One2OneChannelInt ack)
  {
    this.mid = mid;
    this.out = out;
    this.ack = ack;
  }

  public void run()
  {
    Running = true;
    while (Running)
    {
      y = mid.read();
      out.write(y);
      ack.write(y);
    }
  }
}
```

```
public class Target
{
  public static void main (String[] args)
  {
    One2OneChannelInt  in = new One2OneChannelInt();
    One2OneChannelInt  mid = new One2OneChannelInt();
    One2OneChannelInt  ack = new One2OneChannelInt();
    One2OneChannelInt  out = new One2OneChannelInt();
    Send Send_init = new Send(in,ack,mid);
    Recv Recv_init = new Recv(mid,out,ack);
    Parallel System_init =
    new Parallel( new CSProcess[] { Send_init, Recv_init, });
    System_init.run();
    java.lang.System.out.flush();
  }
}
```

The conversion creates the *main* process (which declares the channels) and the *Send* and *Recv* processes, combines the last two processes in a *Parallel* construct, and starts things up. Demonstration of this program does require adding one process to supply data to the *in* channel and one to consume the data produced by the channel *out*.

The variable and channel names are fairly distinct in this example and the JCSP would be fairly easy to type in by hand. Use of the automatic translation from the verified CSP will, however, avoid any typographical errors.

A striking feature of this conversion is the explosion in the size of the code. The CSP source of four lines grows to 46 lines of Java (ignoring comments, blank lines, and lines with braces only). This ratio of about 10 to 1 results from the normal Java housekeeping, the fact that such things as channels are effectively defined in several places, and the simplicity of the individual processes: if there were more internal operations on the data items, e.g., the ratio would decrease.

We look next at the CCSP version of *Stop and Wait*:

```
#include <stdio.h>
#include "/usr/include/ccsp/csplib.h"
/*-- Stop-and-Wait Protocol */
void Send (Process *p , Channel *in, Channel *ack,
           Channel* mid)
{
  int x ;
  bool Running;
  Running = true;
  while (Running)
  {
    x = ChanInInt(in);
    ChanOutInt(mid, x);
    x =ChanInInt(ack);
  }
}
void Recv (Process *p , Channel *mid, Channel *out,
           Channel * ack)
{
  int y ;
  bool Running;
  Running = true;
```

```
    while (Running)
    {
      y =ChanInInt(mid);
      ChanOutInt(out, y);
      ChanOutInt(ack, y);
    }
  }

  void RunMain(Process* p)
  {
    Channel*  in  = ChanAlloc();
    Channel*  mid = ChanAlloc();
    Channel*  ack = ChanAlloc();
    Channel*  out = ChanAlloc();

    Process* Send_p = ProcAlloc(Send, 65536, 3, in, ack, mid);
    Process* Recv_p = ProcAlloc(Recv, 65536, 3, mid, out, ack);
    ProcPar(Send_p, Recv_p,  NULL);
  }
```

The C version is noticeably shorter – 26 rather than 46 lines – due primarily to the fact that the channels are not mentioned so often. Otherwise the C and the Java versions are equally easy to follow.

The next example illustrates the use of the CSP external choice construct to implement a multiplexer (`Mux_up`) in a system that routes packets among a number of processes. The packets contain message type, source, destination, and data components. The CSP:

```
  Mux_up =
        upsendout0?dest.data -> uptodown!1.0.dest.data -> Mux_up
      []
        upsendout1?dest.data -> uptodown!1.1.dest.data -> Mux_up
      []
        uprecack0?ack -> uptodown!0.0.0.ack -> Mux_up
      []
        uprecack1?ack -> uptodown!0.0.0.ack -> Mux_up
```

Each message has four components (all integers). We could define a new JCSP channel type to handle this, but here we simply send each component separately. We use the JCSP `Alternative` construct, followed by a `switch`, to implement the external choice.

```
  class Mux_up implements CSProcess
  {
    One2OneChannelInt upsendout0;
    One2OneChannelInt upsendout1;
    One2OneChannelInt uprecack0;
    One2OneChannelInt uprecack1;
    One2OneChannelInt uptodown;
    int data;
    int ack;
    int dest;
    int value1 = 1;
    int value0 = 0;
    boolean Running;

    public Mux_up( One2OneChannelInt upsendout0,
                   One2OneChannelInt upsendout1,
                   One2OneChannelInt uprecack0,
                   One2OneChannelInt uprecack1,
                   One2OneChannelInt uptodown)
    {
      this.upsendout0 = upsendout0;
      this.upsendout1 = upsendout1;
      this.uprecack0 = uprecack0;
      this.uprecack1 = uprecack1;
      this.uptodown = uptodown;
    }

    public void run()
    {
      int index;
      AltingChannelInputInt[] in = {upsendout0, upsendout1,
                                    uprecack0, uprecack1};
      Alternative alt = new Alternative (in);
      Running = true;
      while (Running)
      {
       index = alt.fairSelect ();
       switch(index)
       {
         case 0:
           dest = upsendout0.read();
           data = upsendout0.read();
           uptodown.write(value1);
           uptodown.write(value0);
           uptodown.write(dest);
           uptodown.write(data);
         break;
         case 1:
           dest = upsendout1.read();
           data = upsendout1.read();
           uptodown.write(value1);
           uptodown.write(value1);
           uptodown.write(dest);
           uptodown.write(data);
         break;
         case 2:
           ack = uprecack0.read();
           uptodown.write(value0);
           uptodown.write(value0);
           uptodown.write(value0);
           uptodown.write(ack);
         break;
         case 3:
           ack = uprecack1.read();
           uptodown.write(value0);
           uptodown.write(value0);
           uptodown.write(value0);
           uptodown.write(ack);
         break;
       }
      }
    }
  }
```

Once again we see the size of the code explode – but the reason here is the number of data items passed in each channel communication. Creation of a new channel type would simplify this example considerably.

3.2 Larger Programs

We look next at several full programs. The first, `Commstime`, is a timing benchmark in the JCSP library [10]. The system (Figure 2) consists of four processes: `PrefixInt`, `Delta2Int`, `Consume`, and `SuccessorInt`. `PrefixInt(0)` initially sends out a 0 over channel `a`, then forwards whatever it receives over channel `c`. `Delta2Int` copies its input value to `Consume` and `SuccessorInt`. `SuccessorInt` receives an integer from `Delta2Int`, adds 1 to it, and passes the result to `PrefixInt`. `Consume` collects and reports statistics. Because there is essentially no work in this system other than the message passing, we can use it to measure the message passing and context switch times.

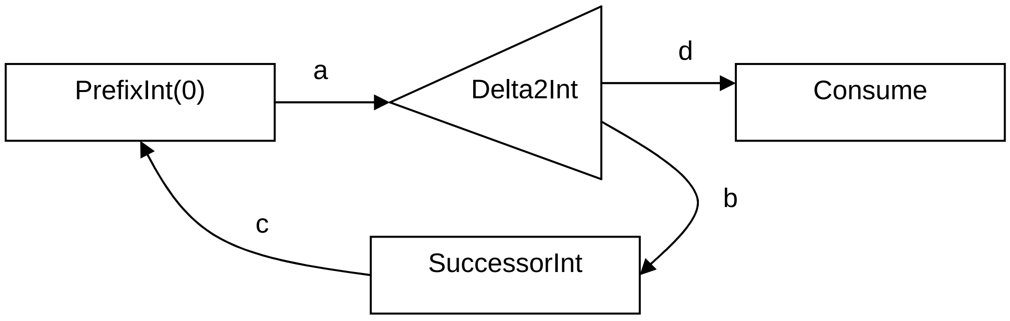

Figure 2. The `Commstime` benchmark.

The CSP description of `Commstime` - with the process names slightly modified:

```
channel a, b, c, d : Integer ;
MyInit = a ! 0 ;
MyPrefixInt = c ? x -> a ! x -> MyPrefixInt ;
MyDelta2Int = a ? x -> d ! x -> b ! x -> MyDelta2Int ;
MyConsume = d ? x -> MyConsume ;
MySuccessorInt = b ? x -> c ! x -> MySuccessorInt ;
System =          (MySuccessorInt
            [| {| |} |]
                        (MyConsume
                   [| {| |} |]
                              (MyDelta2Int
                        [| {| |} |]
                                    (MyPrefixInt
                              [| {| |} |]
                                    MyInit))));
```

The `System` process has been formatted to emphasize its structure. Its component processes are arranged in a horizontal tree, with the topmost operator farthest to the left. The components of this operator – either another simple or composite process – are both shifted right by one level; and so on. Thus the top-level operator is parallel sharing – `[|{||}|]` – and its components are `MySuccessorInt` and another parallel sharing structure. Note that channels are not explicitly required in the sharing operator by the JCSP tool since they can be determined from the definitions of the component processes.

`PrefixInt(0)`is implemented as a combination of `MyInit`, which delivers the intital 0, and `MyPrefixInt`, which simply forwards its input to its output. The JCSP implementation follows.

```
class MyInit implements CSProcess
{
  One2OneChannelInt a ;
  int value0 = 0;
  boolean Running;

  public MyInit( One2OneChannelInt a)
  {
    this.a = a;
  }

  public void run()
  {
    a.write(value0);
  }
}


class MyPrefixInt implements CSProcess
{
  One2OneChannelInt c ;
  One2OneChannelInt a ;
  int x ;
  boolean Running;

  public MyPrefixInt( One2OneChannelInt c, One2OneChannelInt a)
  {
    this.c = c;
    this.a = a;
  }

  public void run()
  {
    Running = true;
    while (Running)
    {
      x = c.read();
      a.write(x);
    }
  }
}

class MyDelta2Int implements CSProcess
{
  One2OneChannelInt a ;
  One2OneChannelInt d ;
  One2OneChannelInt b ;
  int x ;
  boolean Running;

  public MyDelta2Int( One2OneChannelInt a, One2OneChannelInt d,
                     One2OneChannelInt b)
  {
    this.a = a;
    this.d = d;
    this.b = b;
  }
  public void run()
  {
    Running = true;
    while (Running)
    {
      x = a.read();
      d.write(x);
      b.write(x);
    }
  }
}

class MyConsume implements CSProcess
{
  One2OneChannelInt d ;
  int x ;
  boolean Running;

  public MyConsume( One2OneChannelInt d)
  {
    this.d = d;
  }
  public void run()
  {
    Running = true;
    while (Running)
    {
      x = d.read();
    }
  }
}

class MySuccessorInt implements CSProcess
{
  One2OneChannelInt b ;
  One2OneChannelInt c ;
  int x ;
  boolean Running;

  public MySuccessorInt( One2OneChannelInt b, One2OneChannelInt c)
  {
    this.b = b;
    this.c = c;
  }

  public void run()
  {
    Running = true;
    while (Running)
    {
      x = b.read();
      c.write(x);
    }
  }
}
```

```
public class Target
{
  public static void main (String[] args)
  {
    /*Channel declaration*/
    One2OneChannelInt  a = new One2OneChannelInt();
    One2OneChannelInt  b = new One2OneChannelInt();
    One2OneChannelInt  c = new One2OneChannelInt();
    One2OneChannelInt  d = new One2OneChannelInt();
    MyInit MyInit_init = new MyInit(a);
    MyPrefixInt MyPrefixInt_init = new MyPrefixInt(c,a);
    MyDelta2Int MyDelta2Int_init = new MyDelta2Int(a,d,b);
    MyConsume MyConsume_init = new MyConsume(d);
    MySuccessorInt MySuccessorInt_init = new MySuccessorInt(b,c);
    Parallel System_init = new Parallel( new CSProcess[]
      {
        MyInit_init,
        MyPrefixInt_init,
        MyDelta2Int_init,
        MyConsume_init,
        MySuccessorInt_init,
      });
    System_init.run();
    java.lang.System.out.flush();
  }
}
```

The JCSP code in this case has about 80 significant lines, about eight times the number in the CSP description.

The last example models a virtual-channel system that connects two processes on an `Upstream` node to two processes on a `Downstream` node. Each process can send messages to any process on the other node; flow control is enforced on the sources by requiring the reception of an acknowledgement to a previous message before a following message is accepted. The top-level schematic is shown in Figure 3. All messages and acknowledgements must be multiplexed over a single channel in each direction. An expanded view of the `Upstream` node is shown in Figure 4.

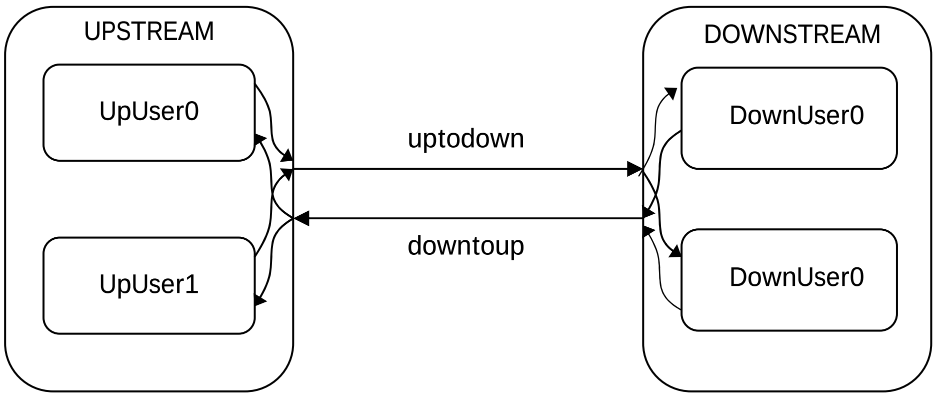

Figure 3. Top-level schematic of the virtual channel system.

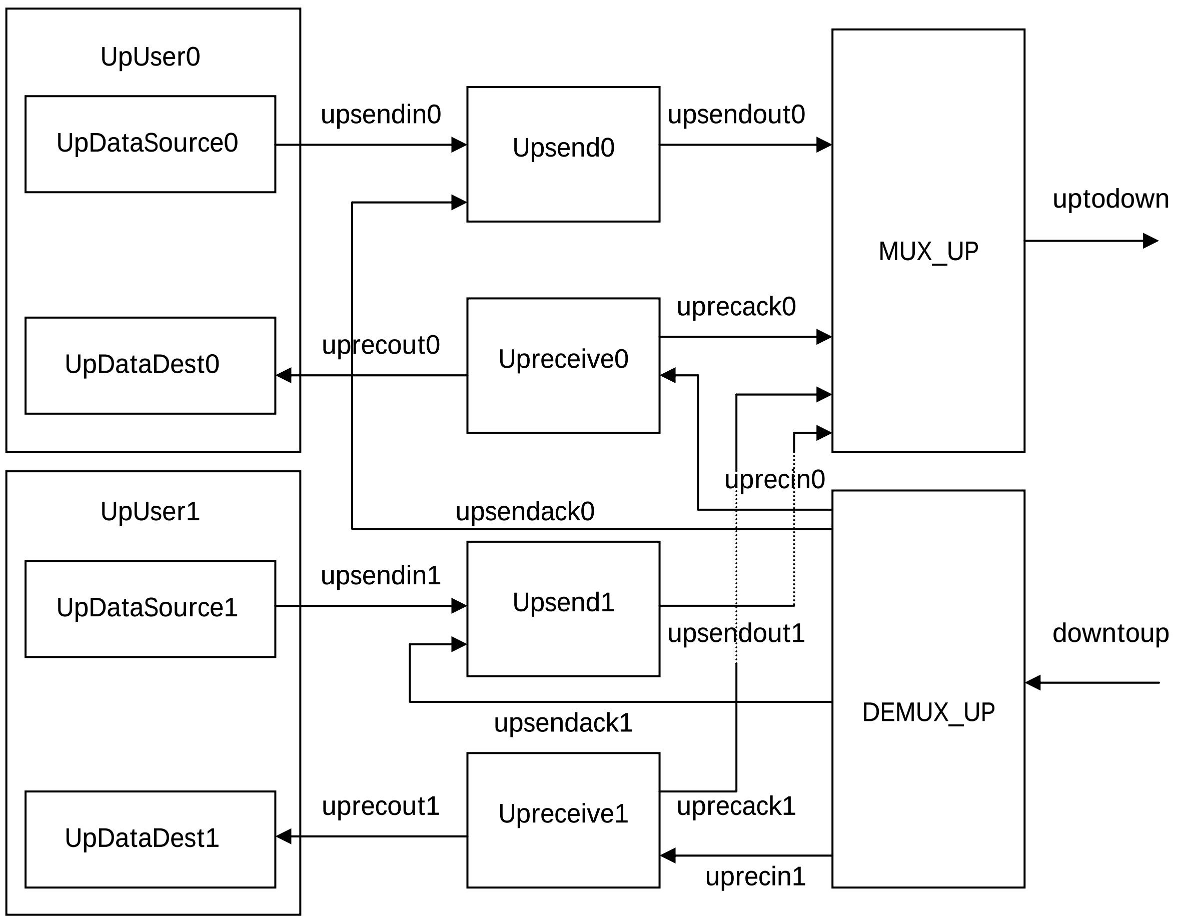

Figure 4. Detail of the Upstream node of Figure 3. Boxes represent processes, and arrows represent channels.

The CSP description of the virtual channel system:

```
-- Virtual channel system ;

channel upsendin0, upsendin1,upsendout0,upsendout1, downsendin0,
        downsendin1, downsendout0, downsendout1 : SourceData ;
channel upsendack0, upsendack1, downsendack0, downsendack1,
        uprecack0, uprecack1, downrecack0, downrecack1 : Ack ;
channel uprecin0,uprecin1,uprecout0,uprecout1,downrecin0,
        downrecin1,downrecout0,downrecout1 : DestData ;
channel downtoup, uptodown : InterData ;

Upsend0 =   upsendin0?dest.data -> upsendout0!dest.data ->
            upsendack0?ack -> upsend0;
Upsend1 =   upsendin1?dest.data -> upsendout1!dest.data ->
            upsendack1?ack -> upsend1;

Upreceive0 = uprecin0?source.data -> uprecout0!source.data ->
             ( if (source ==0)
               then uprecack0!0 -> upreceive0
               else uprecack1!1 -> upreceive0 );
```

```
Upreceive1 = uprecin1?source.data -> uprecout1!source.data ->
             ( if (source ==0)
               then uprecack0!0 -> upreceive1
               else uprecack1!1 -> upreceive1 );
Downsend0 = downsendin0?dest.data -> downsendout0!dest.data ->
            downsendack0?ack -> downsend0;
Downsend1 = downsendin1?dest.data -> downsendout1!dest.data ->
            downsendack1?ack -> downsend1;

Downreceive0 = downrecin0?source.data -> downrecout0!source.data ->
             ( if (source ==0)
               then downrecack0!0 -> downreceive0
               else downrecack1!1 -> downreceive0 );
Downreceive1 = downrecin1?source.data -> downrecout1!source.data ->
             ( if (source ==0)
               then downrecack0!0 -> downreceive1
               else downrecack1!1 -> downreceive1 );
-- if the data is the data we want to send from upstream data
-- source to downstream data destination, we insert a 1 in front of
-- it(also the number indicated data source). If it is ack signal
-- for downstream data source, we insert 0.0.0 in front of it;

Mux_up = upsendout0?dest.data -> uptodown! 1.0.dest.data -> Mux_up
         []
         upsendout1?dest.data -> uptodown! 1.1.dest.data -> Mux_up
         []
         uprecack0?ack -> uptodown! 0.0.0.ack -> Mux_up
         []
         uprecack1?ack -> uptodown! 0.0.0.ack -> Mux_up;
-- if testbit is 0 then the data in testbit.source.dest.data is
-- ack signal from upstream to downstream otherwise it is data send
-- from upstream to downstream;

Demux_down = uptodown ? testbit.source.dest.data ->
             (if(testbit == 0)
              then (if (data == 0)
                    then downsendack0!data -> Demux_down
                    else downsendack1!data -> Demux_down )
              else (if ( dest ==0 )
                    then downrecin0! source.data -> Demux_down
                    else downrecin1! source.data -> Demux_down ) );
-- if the data is the data we want to send from downstream data
-- source to upstream data destination, we insert a 1 in front of
-- it(also the number indicated data source). If it is ack signal
-- for upstream data source, we insert 0.0.0 in front of it;

Mux_down =  downsendout0?dest.data->downtoup! 1.0.dest.data ->
            Mux_down
           []
            downsendout1?dest.data->downtoup! 1.1.dest.data ->
            Mux_down
           []
            downrecack0?ack -> downtoup! 0.0.0.ack -> Mux_down
           []
            downrecack1?ack -> downtoup! 0.0.0.ack -> Mux_down;

-- if testbit is 0 then the data in testbit.source.dest.data is
-- ack signal from upstream to downstream otherwise it is data send
-- from upstream to downstream;
```

```
Demux_up = downtoup? testbit.source.dest.data ->
             ( if (testbit == 0)
              then (if (data == 0)
                    then upsendack0!data -> Demux_up
                    else upsendack1!data Demux_up )
              else (if ( dest ==0 )
                    then uprecin0!source.data Demux_up
                    else uprecin1!source.data Demux_up ) );
UpStream =        (Demux_up
            [| {| |} |]
                        (Mux_up
                  [| {| |} |]
                              (Upreceive1
                        [| {| |} |]
                                    (Upreceive0
                              [| {| |} |]
                                          (Upsend1
                                    [| {| |} |]
                                          Upsend0))))) ;

DownStream =      (Demux_down
            [| {| |} |]
                        (Mux_down
                  [| {| |} |]
                              (Downreceive1
                        [| {| |} |]
                                    (Downreceive0
                              [| {| |} |]
                                          (Downsend1
                                    [| {| |} |]
                                          Downsend0)))));
System =          UpStream
            [| {| |} |]
                  DownStream ;
```

The source files, with source and sink processes added, can be found on the paper web page [24]. Both implementations run correctly. The CCSP code runs about 400 lines, and the JCSP code about 560 lines; the CSP specification is about 60 lines. On a similar problem, with traffic moving in only one direction, a CSP specification of about 30 lines resulted in a CTJ implementation of about 160 lines.

3.3 Discussion

The first impression we have from this exercise is that the conversion from CSP to Java or C results in a 5-fold or so expansion of the code. Given that the CSP description of a reasonable application may run to hundreds or thousands of lines, the utility of automatic conversion is clear.

The last example (the virtual channel system) exhibits some of the features of larger systems that make hand-coding of the target code error-prone: there are large numbers of channels and processes that have nearly identical names. The names are explicitly chosen to be descriptive of the location of the channels or processes within the topology of the system, so we are reluctant to do otherwise. However, when it comes to typing in the code, it is rare that errors do not occur. The worst errors are those where an existing but incorrect channel or process is referred to; the compilers will not catch this mistake.

Over the past few years, final class projects have typically required a week or two to get the CSP script correct, and an equal amount of time to get the Java running correctly. The bulk of the Java effort was aimed at getting the wiring between the processes and the ordering of events correct. With the conversion tool described in this paper, the JCSP version of the virtual channel system above (last year's final project) was up and running within less than an hour of the verification of the CSP. The CCSP version came right behind.

At present the translators do not explicitly handle non-CSP code – such as common numerical or textual operations and assignments. These can be included within CSP comments, which will be converted into appropriate comments in the target code. These comments can then in turn be immediately converted in executable code.

Due to the nature of the conversion process, the structure of the Java or C code will closely follow that of the CSP script. Thus some time invested in a clean design of the CSP pays off directly in a well-laid out version of the much larger target code.

4 Summary and Further Work

4.1 Summary

We have developed simple tools that allow us to create applications automatically from verified descriptions in CSP. At this point the tools convert a small but useful subset of the standard CSP script into executable CTJ or JCSP versions of Java or CCSP versions of C code. The CSP features converted seem sufficient to implement applications based primarily on communication. The primary advantage of the tools is that they correctly implement the large amount of process and channel boiler-plate that is tedious and difficult to get correct by hand. Since applications may be five or more times larger than the CSP specifications, this is a significant gain that can greatly reduce development time. This project was relatively straightforward to carry out, and should be easy to expand. We expect to continue this work.

First, however, we plan to review our approach to the development of the conversion tools. Our initial intention was to extend a set of tools [12] for the manipulation of CSP scripts that had been implemented in Mathematica [13]; this would allow us to take advantage of Mathematica's large libraries of procedures for manipulating sets of strings and symbolic computation – with the hope that we would eventually have a single package to develop the CSP and then convert it directly to the target code once it had been verified. Mathematica is not, however, universally available, and during the second phase of this project (JCSP and CCSP) we developed an additional conversion tool based on C. In the next phase we plan to base the conversion tools on *lex* and *yacc* [e.g., 15] to gain added flexibility.

We have not yet run any performance tests on the translations. The examples completed thus far are composed nearly entirely of communications between processes, and the conversion tools order the communications just as they were ordered in the CSP – by design, since that is the part we really wish to have correct. Optimisation of communications would certainly be a worthwhile feature; this would probably require some interesting effort on extracting the communication patterns.

4.2 Future Work

There are a number of extensions and additional features planned. On the CSP side, we will extend coverage to include replication operators (e.g., on external choice), boolean guards and priorities on external choice, and sequential composition. The conversion of complex channel types (those formed from dotted combinations of more primitive CSP types) will be changed to generate appropriate objects in the target languages and channels to carry them. We will also allow specification in CSP (by annotated comments) of standard target channel types such as `int` and `float`.

A number of changes and additions on the target code side will be made. We should be able to include arbitrary fragments of target code in the CSP script by appropriately annotating standard CSP comments. This will give us the ability to manipulate data read over a channel before passing it along another channel; presently this would have to be done by adding code manually to the target after the conversion. Real-time clocks will be included, which will allow the implementation of time-out operations on channels. (Note that CSP originally included untimed timeouts, so we are not certain how this will be handled in the CSP script. Schneider's Timed CSP [3] does allow timed timeouts – as well as other timed features – so we may wish to extend our CSP coverage to Timed CSP.)

The recent addition of networking capabilities via JCSP.net [18] opens up a number of opportunities for distributed and Internet programming. It should be straightforward to trigger the generation of the JCSP.net code by including directives in CSP comments or by appropriately naming the channels themselves. This capability can also be applied to the CCSP version via CCSP.net [11]. This will allow us to specify in CSP applications that are to be distributed across networks, verify the correctness of the communication patterns, and then generate immediately the application code.

The same approach can be applied to embedded multiprocessor systems, such as might be used on a mobile robot with a processor driving each wheel. Once we know how communications are handled in the target code, the conversion tool can be modified accordingly. Systems could thus be quickly built on JStamp modules [19], which run Java byte code directly in hardware, or the Intel PXA250 ARM embedded processor [20] (which includes hardware support for wireless communication) in Java or C.

We are presently developing a new version of the tool that will convert CSP specifications into Handel-C [16]. This will give us a direct path from formal, verified specifications in CSP to implementation in FPGAs.

The three present versions convert directly into CTJ, JCSP, or CCSP. We hope to combine these into a single tool. This may be done by identifying a single intermediate language that can easily be converted into C and Java; this would also give us a smooth path to adding additional target languages. It may turn out, however, that the three conversions are so similar that an intermediate stage would add unneeded complexity.

We will continue to test these packages on increasingly complex applications. Suggestions are welcome.

Finally we note that this tool should work nicely in combination with Hilderink's [17] package that converts graphical diagrams into CSP. This will give us an automatic path from a graphical description of a CSP system directly to runnable code.

The translator programs and examples can be found on the paper web page [24].

References

[1] C. A. R. Hoare, Communicating sequential processes, *CACM*, 21(8), August 1978, pp. 666-677.

[2] A. W. Roscoe, *The Theory and Practice of Concurrency*, Prentice Hall, London, 1998.

[3] Steve Schneider, Concurrent and Real-Time Systems: The CSP Approach, Wiley, Chichester, 1999.

[4] ______, Formal Systems (Europe) Ltd.: http://www.fsel.com/software.html

[5] J. Galletly, *occam 2 – including occam 2.1*, UCL Press, London, 1996.

[6] ______, Kent Retargetable Occam Compiler (KRoC): http://wotug.ukc.ac.uk/kroc/

[7] Gerald Hilderink, Jan Broenink, Wiek Vervoort, and Andre Bakkers, Communicating java threads, in: *Proceedings of WoTUG 20: Parallel Programming and Java*, ed. A. W. P. Bakkers, IOS Press, Amsterdam, 1997, pp., 48-76.

[8] ______, CSP for Java: http://www.rt.el.utwente.nl/javapp/

[9] P. H. Welch, Process oriented design for Java: concurrency for all, in: *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, ed. H. R. Arabnia, CSREA Press, Athens GA, 2000, pp. 51-57.

[10] ______, Communicating Sequential Processes for Java^{TM} (JCSP): http://www.cs.ukc.ac.uk/projects/ofa/jcsp/

[11] ______, Quickstone Technologies Ltd.: http://www.quickstone.com /xcsp/ccspnetworkedition/

[12] W. Zhao, and G. S. Stiles, The automated serialization of concurrent CSP scripts using Mathematica, *Communicating Process Architectures 2000*, ed. P. H. Welch and A. W. P. Bakkers, IOS Press, Amsterdam, Sept. 2000, pp. 15-32.

[13] Mathematica Book Online: http://documents.wolfram.com.

[14] B. Eckel, *Think in C++ 2nd Edition, Volume2*, Upper Saddle River NJ, Prentice Hall, 2000.

[15] John R. Levine, Tony Mason, and Doug Brown, *lexx and yacc*, O'Reilly & Associates, Inc., Sebastopol CA, 1992.

[16] ______, Handel-C: http://www.celoxica.com/tech/handel-c/default.asp

[17] Gerald Hilderink, A graphical modeling language for specifying concurrency based on CSP, *Communicating Process Architectures 2002*, ed. James Pascoe, Roger Loader, and Vaidy Sunderam, IOS Press, Amsterdam, Sept. 2002, pp. 255-284.

[18] ______, Quickstone Technologies Ltd.: http://www.quickstone/xcsp/jcspnetworkedition/.

[19] ______, Systronix, http://jstamp.systronix.com/about.htm.

[20] ______, Intel, http://www.intel.com/design/pca/prodbref/298620.htm.

[21] N. C. C. Brown, An Introduction to the C++ CSP Library, *Communicating Process Architectures 2003*, ed. Jan Broenink, IOS Press, Amsterdam, Sept. 2003, pp. xx-yy.

[22] ______, http://www.cs.sunysb.edu/~concurr/features.html

[23] ______, http://www.dcs.ed.ac.uk/pepa/tools.html

[24] ______, http://www.ece.usu.edu/research/rtpc/projects/JavaCSP/#CSP_to_Java