Communicating Process Architectures 2004
Ian East, Jeremy Martin, Peter Welch, David Duce, and Mark Green (Eds.)
IOS Press, 2004

An Automatic Translation of CSP to Handel-C

Jonathan D. PHILLIPS
452½ East 700 North, Logan, UT 84321
jdphillips@cc.usu.edu

G. S. STILES
Utah State University, 4120 Old Main Hill, Logan, UT 84322-4120
dyke.stiles@ece.usu.edu

Abstract. We present tools that convert a subset of CSP into Handel-C code. Handel-C was derived from the original occam concurrency language, but has a syntax similar to the standard C programming language. It compiles to produce files to program an FPGA. We thus now have a process that can directly generate hardware from a verified high-level description. The CSP to Handel-C translator makes use of the Lex and Yacc programming tools. The Handel-C code produced is functional, but not necessarily optimized for all situations. The translator has been tested using several CSP scripts of varying complexity. The functionality of the resulting Handel-C programs has been verified with simulations, and two scripts were further checked with successful implementations on an FPGA.

1. Introduction

This paper describes tools to automate the conversion of a set of concurrent, interactive software processes into equivalent hardware. The tools are based on the development of an algorithm that automatically translates CSP [1] scripts into the Handel-C [2] language. A major benefit of creating specific hardware to implement software algorithms that are easily parallelized is a substantial increase in the throughput of the system; increased speed is vital in many real-time electronics applications, especially digital signal processing. CSP provides robust, reliable channels for inter-process communication, and CSP scripts can be checked to ensure absence of deadlock and – within certain limits – to verify correct implementation of formal specifications of the algorithms. Furthermore, automated translation of software into hardware, specifically in the case of large concurrent applications, can greatly reduce development time.

Communicating Sequential Processes, or CSP, is an algebra designed for describing and reasoning about synchronizations and communications between processes. Systems of processes that are described using CSP can be checked for non-determinism, freedom from deadlock and livelock, and correctness with respect to a formal specification through the use of the software tools FDR and ProBE [3]. The verified CSP script can then be used as a basis for implementing the algorithm in a high level language, or can be used as a guide to design hardware to perform the same task. In the case of this project, the verified CSP script is translated into Handel-C, which is then compiled to produce files for programming an FPGA device.

This automated verification and translation decreases the time generally spent on manually designing and testing an FPGA based on a CSP algorithm. Complicated systems with many processes and communicating channels can be easily implemented with minimal (or zero) errors in channel or process naming and channel synchronization.

The remainder of this paper is dedicated to the specifics of this project. The importance and relevance of this translation tool is discussed, and previous work along similar lines is reviewed. We will define the subset of CSP that was included and discuss the details of the translation tool, which is based on Lex and Yacc. The paper then presents examples of translations of simple and complex CSP scripts. The paper concludes with the successful simulations of the scripts using the Handel-C compiler and the results of two complete runs on an actual FPGAs.

2. Background

2.1 Project Significance

This project is an attempt to combine the mathematical strengths of CSP with the hardware-producing capabilities of Handel-C. Automating the translation of a CSP script into a Handel-C file minimizes the probability of human error. When complex systems of interacting processes are modelled in CSP, and then manually translated into a hardware or software programming language, it is easy for a human to confuse variable or channel names, improperly order processes, omit critical code synchronizations, and commit other syntactical or logical errors. Since CSP and Handel-C both have well-defined syntax and grammar rules, creating a compiler or translator to automate the conversion from one to the other immediately eliminates many errors that could otherwise occur during the translation process.

2.2 Previous Work

The area of hardware-software interaction has been addressed by several recent projects at Utah State University. John Campbell [6] developed and implemented algorithms for mapping regions of software into hardware based on dynamic behavior of the software. The mapping, based on profile information gathered in test executions of the software, configured hardware functions with high rates of communication in close proximity. He also investigated theoretical extensions of the algorithm for dynamic/run time caching of functionality to hardware.

Initial work on the automated conversion of CSP was done by Zhou and Stiles [7], where concurrent CSP scripts were translated into equivalent sequential scripts; this tool was developed in Mathematica©. The work most closely related to this paper is Raju, Rong, and Stiles [8], where CSP scripts were automatically translated into concurrency-capable, channel-oriented versions of Java and C (CTJ [9, 10], JCSP [11, 12], and CCSP [13]); the translation tools were Mathematica and C augmented with string-handling libraries.

Current research at the University of Surrey [14] is looking at the translation of occam directly into FPGA code. Preliminary results indicate that success has been obtained in modeling basic logic gates.

2.3 Project Details

The creation of an application-specific FPGA from a CSP specification involves several steps. The overall process is depicted in fig. 1. First, the CSP script must be verified for

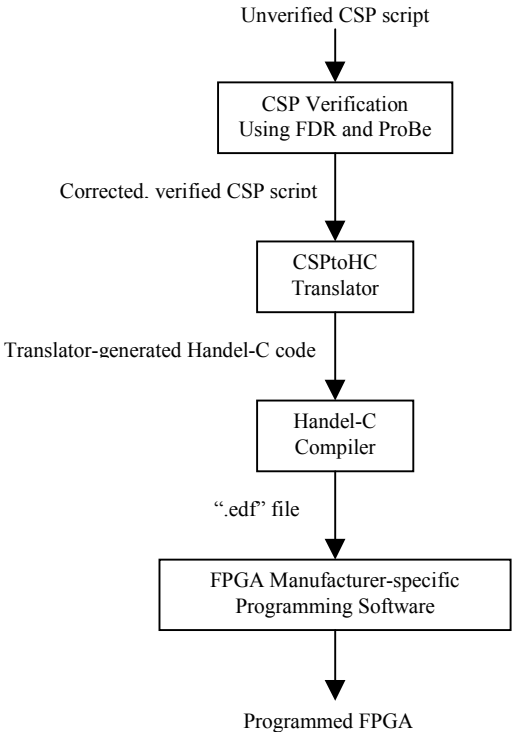

Figure 1: System overview.

correctness with respect to a specification, including absence of deadlock and livelock, using the ProBE and FDR tools. Once the CSP script is correct, it is passed to the CSPtoHC translator, which produces a Handel-C program based on the CSP design.

Handel-C is based on a subset of the traditional C programming language, with added features included to facilitate various facets of parallel hardware creation. Integers are the only data type that is allowed. Integers of any specific size can be declared, since unneeded bits would create inefficient hardware. A `par` construct allows for the parallel execution of instructions. The `prialt` construct behaves similarly to a switch or select statement, where the cases are triggered by data reception on an input channel. In addition to the traditional array construct, RAMs and ROMs can also be created, filled with data, and manipulated in similar fashion.

Handel-C programs can be simulated at the cycle level, using the Handel-C software, to observe how the hardware will respond. After simulation, the Handel-C compiler can produce an output file that can be fed into an FPGA-programming package, which ultimately results in a programmed FPGA that performs the functionality originally described in the CSP script.

This project concentrates mainly on the CSPtoHC translator. It is assumed that CSP scripts passed to the translator will have been verified for correctness previously, and thus the specifics of ProBE and FDR will not be discussed here. The Handel-C simulator will be employed extensively to demonstrate that the results match the desired output. An FPGA will be programmed with two of the more complex codes to show that the hardware does correspond as specified and to measure the performance.

The process of designing the CSPtoHC translator, which was done with the help of the Lex and Yacc utilities, will be described next. A brief overview on how to use the tool will also be provided.

3. The CSP to Handel-C Translator

3.1 CSP Subset Selection

CSP has considerable expressive power. However, a simple subset of CSP can be chosen that will serve to adequately model many real-world applications. Table I shows the subset that has been selected for translation in this compiler, and also lists the equivalent operation in Handel-C. (The CSP notation is expressed in the machine-readable version CSPM that is required by the tools).

Table 1: Handel-C Translation of CSPM Features

Feature	CSP	Handel-C
Comments	`-- …`	`// …`
	`{- … -}`	`/* … */`
Channel Declarations	`channel`	`chan, chanin, chanout`
Channel Operations	`in?x`	`in?x;`
	`out!x`	`out!x;`
Integer Declarations	`implied`	`int 8 x;`
External Choice	`[]`	`prialt {…}`
Synchronous Parallel	`[|{|…|}|]`	`par {…}`
Recursion	`P = … → P`	`while(1) {…}`

CSP is an algebra concerned primarily with the interactions of multiple processes that communicate with each other via events. However, operations that are significant and internal to a process – such as complex mathematical expressions – may not be represented in CSP in an easy-to-translate form. Such operations are, however, of vital importance to the functionality of the FPGA, so a method has been developed for embedding arbitrary Handel-C statements in the CSP script as comments. For example:

```
{-MacroC calculation
      x = f(x);
      y = g(y);
EndMacroC-}
```

Assume that this CSP comment is associated with the following CSP script:

```
Ex = in?x -> in?y -> calculation -> out!x -> out!y -> Ex
```

The C statements included in the comment would be inserted in the Handel-C code anywhere that the term `calculation` was found. Thus a type of macro has been developed, where C code can be embedded as a comment in the CSP script and automatically inserted in the correct place in the Handel-C output file.

3.2 Lexical Analysis of CSP

The first step in creating any sort of a compiler-type program is to identify the tokens that are acceptable as input. The Lex utility makes the recognition of these tokens easy; the utility uses a list of tokens, along with a list of instructions on the actions that should take

place when a specific token is found. For example, we present a piece of the translator's Lex file including the tokens "->" and "channel":

```
"->"        {return PREFIX;}              /* PREFIX token */

channel     {                             /* CHANNEL keyword */
                  var_dec = 1;
                  return CHANNEL;
            }
```

Note that in the case of the channel keyword, a variable is set for use in another part of the program before the token type is returned.

The complete list of tokens that the translator recognizes is very large and complex, with several tokens necessitating the call of various functions to set flags, store strings in memory, and perform a variety of other operations. In most cases, tokens are separated by white space, which is defined as any combination of spaces, tabs, and newline characters.

3.3 Grammatical Analysis of CSP

The lexical analysis breaks the input stream into tokens based on defined rules. This list of tokens is not very useful in itself, but when it is combined with a syntactical parser, it becomes invaluable. The Yacc utility is provided to help with the implementation of parsers. Code written using Yacc interacts with lexer code to produce a complete compiler.

Writing code using Yacc is similar to Lex. Once again, a list of allowable constructs is created. This time, however, combinations of tokens are represented. For example, here is part of the listing of possible combinations that the translator program will allow:

```
statement:  PROCESS '=' expression
      |     CHANNEL variable_list
      |     MACRO
      |     C_VAR
      |     COMMENT
      ;
```

This list signifies that a statement is one of five things: a process definition, a channel declaration, a macro, a C variable declaration, or a comment. These key words are provided to the Yacc program by the lexical analyzer described above.

As is the case with the lexer, the parser calls functions and set flags when specific constructs are recognized. If the lexer provides combinations of tokens that are not recognized as valid by the parser, a syntax error is generated.

The use of Lex and Yacc greatly improves the process of adding additional tokens or constructs to the translator. New features can be added simply by placing the appropriate tokens in the token list, and constructs in the construct list.

3.4 Code Design

The first step of the code design process is to determine which parts of CSP to accept. An appropriate subset has been selected, as shown in Table I. Next, a mapping of CSP to Handel-C was performed. Where there is a one-to-one correspondence between a CSP feature and a Handel-C feature, the translation process is simply a substitution of one character string for another.

The translation process is not always this simple, however. For example, the CSP "=" operator serves to assign an event or list of events to a process. In Handel-C, the events are listed sequentially, but the process name is meaningless as far as code execution is concerned. The list of events assigned to a specific process are stored in a linked list of processes, so that the individual processes can be combined properly when the inter-process interaction definitions are found later in the CSP script.

Another example of a complicated conversion involves the CSP synchronous parallel construct. A list of processes that would run in parallel in Handel-C requires the use of the `par` construct.

The translation of simple recursions required the creation of infinite `while` loops. A recursive process in CSP is written as:

```
GetVal = input?x -> GetVal
```

In Handel-C, this is represented as:

```
while(1)
{
      input?x;
}
```

This requires setting a flag in the process structure to indicate that the process in question is recursive. On another note, variables declared in Handel-C are visible to the entire function in question. Care should be taken in the CSP design to avoid the reuse of variable names, even in different processes, unless the intention is to allow these variables to be shared by the entire Handel-C program.

3.5 Using the CSPtoHC Translator

Consideration will now be given to the technique of using the `CSPtoHC` translator to produce Handel-C code. When a CSP script has been duly tested and proven reliable using the ProBE and FDR tools, the translator can be invoked to perform the conversion. The syntax expected by the `CSPtoHC` translator is as follows:

```
CSPtoHC <CSP filename> <Handel-C filename>
```

The CSP file must exist, or an error will occur and the program will exit. If the Handel-C file specified already exists, it cannot be overwritten, and an error will occur. If there are not exactly two files specified when the compiler is invoked, it will exit.

A typical `CSPtoHC` compiler session might appear as follows:

```
io% ./CSPtoHC Commstime.csp Commstime.hcc
Process PrefixInt0 found.
Process IdInt found.
Process Delta2Int found.
Process Consume found.
Process SuccessorInt found.
Process System found.
Translation successful -- File Commstime.hcc created.
```

Notice that `Commstime.csp` is the input file, and `Commstime.hcc` will be created as the output file. As the translation takes place, processes that have been identified are listed one by one. This process listing can be used as a means of checking to ensure that the compiler has actually found all the processes included in the CSP script. Once the translation succeeds, the user is alerted to this fact and notified that the output file has been created and is ready for use.

4. Testing The Translator

4.1 Simple Simulations

To test the functionality of the CSPtoHC translator, several simple programs were first employed to check the correctness of the different constructs. These simple CSP scripts and the Handel-C equivalent code are discussed in this section.

4.1.1 UpHandler

`UpHandler` is a very simple process that repeatedly reads in an integer on channel `input` and then exports the integer on channel `output`. The CSP script for this process is:

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

The resulting Handel-C code appears as follows:

```
// This is a test file for a simple process with recursion
void main(void)
{
      chanin input;
      chanout output;
      int 8 UpHandler_flag, x;

      // UpHandler
      while(UpHandler_flag == 0)
      {
            input?x;
            output!x;
      }
}
```

There are several things to note about this translation. First, the CSPM comment is correctly translated into a C-style comment. The CSP channels are separated into input and output channels, and declared accordingly. The recursion is detected, and is properly represented through the use of the while loop. Observe that all `while` loops are given a loop flag, which is automatically initialized to zero in the C code, so that the programmer can manually insert a loop termination condition later on. (Alternatively the CSP programmer could insert an `if-the-else` clause in the CSP version, which would be correctly translated in Handel-C). The process name (UpHandler), is included in the Handel-C file as a comment. Semicolons have been inserted in the appropriate places. Tabs have been added to improve the readability of the code. Lastly, `x` is determined to be a variable, and is appropriately declared as an integer.

4.1.2 StopAndWait

The `StopAndWait` [15] protocol consists of two processes called `Send` and `Recv` that synchronize to establish a path of controlled data flow. In fig. 2, it can be seen that the `Send` process accepts data from an input channel, sends the data out on channel `mid`, and then waits for a response from channel `ack` before allowing further input. `Recv` reads data from channel `mid`, outputs it on channel `out`, and then sends an `ack` back to process `Send`.

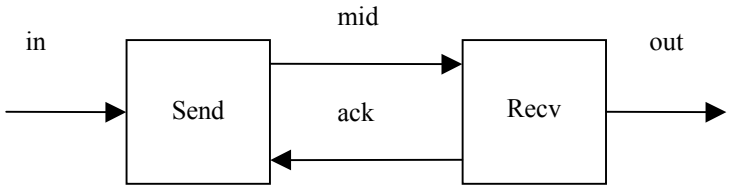

Figure 2: The `StopAndWait` protocol.

The CSP script that describes this system is:

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

Executing the CSPtoHC compiler on this script produces the following Handel-C file:

```
// Stop and Wait Protocol
void main(void)
{
     chan ack, mid;
     chanin in;
     chanout out;
     int 8 Recv_flag, y, Send_flag, x;

     par   // System
     {
           // Send
           while(Send_flag == 0)
           {
                 in?x;
                 mid!x;
                 ack?x;
           }

           // Recv
           while(Recv_flag == 0)
           {
                 mid?y;
```

```
                out!y;
                ack!y;
            }
        }
    }
```

The main feature is the use of the `par` construct, where `Send` and `Recv` are required to operate in synchronous parallel. The CSP channels are once again sorted as input, output, or bi-directional channels, based on the "?" and "!" tokens. One other factor of note is the length of the Handel-C program compared to the length of the original CSP script: the Handel-C program is about five times the length of the corresponding CSP script. Similar expansion of code was noted in Java and C translations of CSP [8].

4.1.3 ManyToOne

The last of the simple algorithms to be presented is the `ManyToOne` algorithm. This program acts as a simple multiplexer or switch. Four input channels are mapped to one output channel. The input channels are serviced by a first-in, first-out algorithm, where only one value at a time is output. Figure 3 shows how the channels interact.

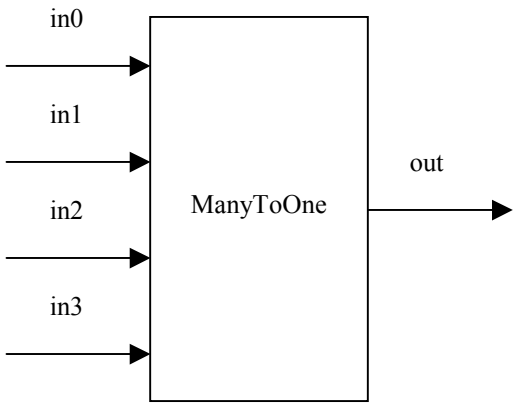

Figure 3: The `ManyToOne` algorithm.

The CSP description of this algorithm introduces the external choice operator:

```
-- This program maps four inputs to one output

channel in0, in1, in2, in3, out
MtoO = in0?x -> out!x -> MtoO
[] in1?y -> out!y -> MtoO
[] in2?z -> out!z -> MtoO
[] in3?w -> out!w -> MtoO
```

When this script is translated, the following Handel-C code is produced:

```
// This program maps four inputs to one output
void main(void)
{
     chanin in3, in2, in1, in0;
     chanout out;
     int 8 w, z, y, MToO_flag, x;

     // MToO
     while(MToO_flag == 0)
     {
          prialt
          {
               case in0?x:
               out!x;
               break;

               case in1?y:
               out!y;
               break;

               case in2?z:
               out!z;
               break;

               case in3?w:
               out!w;
               break;
          }
     }
}
```

This Handel-C program makes use of the `prialt` operator – Handel-C's implementation of CSP's external choice. Notice that there is a `prialt` case setup for each input channel, and once an input is read, that input value must be output before any other input channel can be serviced. Since all processes are part of the same C function, distinct variable names must be used in each case.

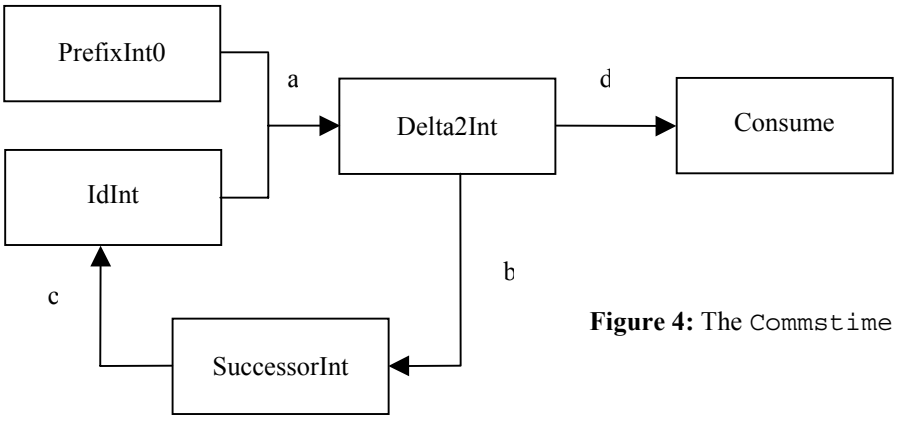

Figure 4: The `Commstime` algorithm.

4.2 Larger Programs

4.2.1 Commstime

The first example is `Commstime`, which is a commonly used timing benchmark [10]. The `Commstime` system is shown in fig. 4. The system starts with the number 0, and the number is circulated through the processes, being incremented by one each time it passes through the `SuccessorInt` process. Thus, the incrementing numbers are observed as they arrive at `Consume`, and the throughput of the system can be calculated.

The CSP script that corresponds to this figure is shown below. Notice the inclusion of a macro `incw` to define the addition operation present in the `SuccessorInt` process (in this case the simple arithmetic operation could be included in the channel output: `c!(w+1)`).

```
{-MacroC incw
w = w + 1;
EndMacroC-}

channel a, b, c, d, e

PrefixInt0 = a!0 -> Skip
IdInt = c?x -> a!x -> IdInt
Delta2Int = a?y -> d!y -> b!y -> Delta2Int
Consume = d?z -> e!z -> Consume
SuccessorInt = b?w -> incw -> c!w -> SuccessorInt
System = (SuccessorInt
      [|{| |}|] Consume
      [|{| |}|] Delta2Int
      [|{| |}|] IdInt
      [|{| |}|] PrefixInt0)
```

The equivalent Handel-C code:

```
void main(void)
{
      chan d, c, b, a;
      chanout e;
      int 8 SuccessorInt_flag, w, Consume_flag, z,
            Delta2Int_flag, y, IdInt_flag, x;

      par   // System
      {
            // PrefixInt0
            {
                  a!0;
            }

            // IdInt
            while(IdInt_flag == 0)
            {
                  c?x;
                  a!x;
            }

            // Delta2Int
            while(Delta2Int_flag == 0)
            {
                  a?y;
```

```
                d!y;
                b!y;
            }

            // Consume
            while(Consume_flag == 0)
            {
                d?z;
                e!z;
            }

            // SuccessorInt
            while(SuccessorInt_flag == 0)
            {
                b?w;
                w = w + 1;
                c!w;
            }
        }
    }
```

Upon analysis, it can be seen that there are four recursive processes and one non-recursive process that execute in parallel. Synchronization between processes is accomplished by using channels. The throughput of this system is easy to measure. Using the Handel-C simulator, the number of clock cycles it takes for the value output on channel `e` to increment can be observed (the current clock cycle is listed on the left of each line, and the value of each variable at that time is shown):

```
 Compiled :   0 gates,   0 inverters,  11 latches,  84 others
Optimised :   0 gates,   0 inverters,  11 latches,  72 others
 Expanded : 164 gates,  35 inverters,  58 latches,   8 others
Optimised :  76 gates,  13 inverters,  58 latches,   8 others

0: w=0 z=0 y=0 x=0
1: w=0 z=0 y=0 x=0
2: w=0 z=0 y=0 x=0
2: Ready to accept output from `e' ? (y/n) y
2: Output from channel `e' = 0
3: w=0 z=0 y=0 x=0
4: w=1 z=0 y=0 x=0
5: w=1 z=0 y=0 x=1
6: w=1 z=0 y=1 x=1
7: w=1 z=1 y=1 x=1
7: Ready to accept output from `e' ? (y/n) y
7: Output from channel `e' = 1
8: w=1 z=1 y=1 x=1
9: w=2 z=1 y=1 x=1
10: w=2 z=1 y=1 x=2
11: w=2 z=1 y=2 x=2
12: w=2 z=2 y=2 x=2
12: Ready to accept output from `e' ? (y/n) y
12: Output from channel `e' = 2
```

Outputs are ready from channel `e` on clock cycles 2, 7, and 12. Thus, we can deduce that it takes five clock ticks for the system to loop one time. In addition to the current clock cycle, the simulator indicates the value held by each variable register on each clock cycle. Statistics are also given for the number of gates, inverters, latches, and other constructs that would be used if actual hardware was created. These numbers are helpful when determining the size of the FPGA that will be used to create the corresponding hardware.

4.2.2 Sieve of Eratosthenes

Another common benchmark program is the sieve of Eratosthenes, which was developed by the Greek mathematician Eratosthenes around 200 BC. This algorithm is an efficient method for calculating prime numbers, and lends itself well to pipelined parallelism. The diagram for a three-stage sieve is in fig. 5. This version passes all numbers through, attaching a tag (`primeout`) indicating whether the output is prime or non-prime.

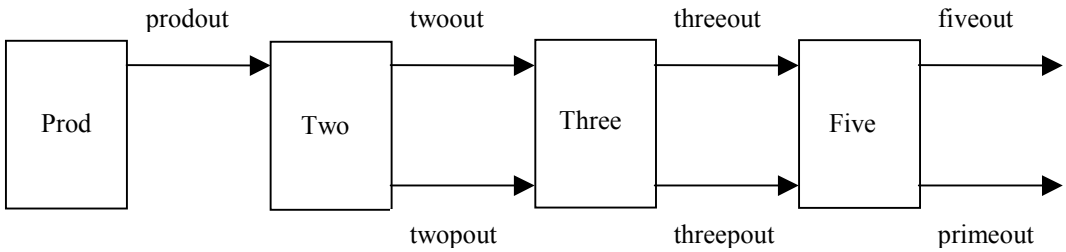

Figure 5: The sieve of Eratosthenes.

This model checks each input (`prodout`) to the pipe for multiples of two, three, and five, and thus will find all prime numbers up to five squared, or 25. Parallelism is possible, because each stage in the pipeline can be working on a different number at any given time. There are two channels for communication between most of the processes – one to pass the number being tested, and one to pass the status of that number as prime or non-prime. The CSP description of this system follows. A future version of the translator will allow if-then-else constructs in CSP, which would allow a much more efficient CSP script to be used for this algorithm.

```
-- An implementation of the Sieve of Eratosthenes that calculates
-- all prime numbers between 2 and 25.  This is done by marking
-- all numbers that are multiples of 2, 3, or 5 as non-prime (0).

--VarC int 8 i
--VarC int 8 j
--VarC int 8 k

{-MacroC incx
x++;
if(x > 25) x = 0;
EndMacroC-}

{-MacroC check2
g = 1;
for(i=4; i<=25; i=i+2)
{
  if(i == d) g = 0;
}
EndMacroC-}

{-MacroC check3
for(j=6; j<=25; j=j+3)
{
  if(j == e) h = 0;
}
EndMacroC-}

{-MacroC check5
```

```
for(k=10; k<=25; k=k+5)
{
  if(k == f) p = 0;
}
EndMacroC-}

channel prodout, twoout, threeout, fiveout, primeout
channel twopout, threepout

Prod = prodout!x -> incx ->  Prod
Two = prodout?d -> check2 -> twoout!d -> twopout!g -> Two
Three = twoout?e -> twopout?h -> check3 -> threeout!e ->
      threepout!h -> Three
Five = threeout?f -> threepout?p -> check5 -> fiveout!f ->
      primeout!p -> Five

Sieve = (Five || Three || Two || Prod)
```

When this CSP file is given to the CSPtoHC translator, the following output is obtained:

```
// An implementation of the Sieve of Eratosthenes that calculates
// all prime numbers between 2 and 25.  This is done by marking
// all numbers that are multiples of 2, 3, or 5 as non-prime (0).

void main(void)
{
      chan threepout, twopout, threeout, twoout, prodout;
      chanout primeout, fiveout;
      int 8 i;
      int 8 j;
      int 8 k;
      int 8 Five_flag, p, f, Three_flag, h, e, Two_flag, g, d,
            Prod_flag, x;
      par   // Sieve
      {
            // Prod
            while(Prod_flag == 0)
            {
                  prodout!x;
                  x++;
                  if(x > 25) x = 0;
            }

            // Two
            while(Two_flag == 0)
            {
                  prodout?d;
                  g = 1;
                  for(i=4; i<=25; i=i+2)
                  {
                        if(i == d) g = 0;
                  }
                  twoout!d;
                  twopout!g;
            }

            // Three
            while(Three_flag == 0)
            {
                  twoout?e;
                  twopout?h;
```

```
                        for(j=6; j<=25; j=j+3)
                        {
                              if(j == e) h = 0;
                        }
                        threeout!e;
                        threepout!h;
                  }

                  // Five
                  while(Five_flag == 0)
                  {
                        threeout?f;
                        threepout?p;
                        for(k=10; k<=25; k=k+5)
                        {
                              if(k == f) p = 0;
                        }
                        fiveout!f;
                        primeout!p;
                  }
            }
      }
```

The Handel-C code contains four parallel processes that interact via channels to keep track of the current number, and whether or not that number is prime. Simulating this file yields the following hardware specifications:

```
  Compiled :   0 gates,   0 inverters,  24 latches, 190 others
 Optimised :   0 gates,   0 inverters,  21 latches, 161 others
  Expanded : 476 gates, 179 inverters, 117 latches,  30 others
 Optimised : 160 gates,  30 inverters, 110 latches,  30 others.
```

The simulator produces the following timing listing for the number 3 is prime:

```
65: i=26 j=6 k=30 p=1 f=2 h=1 e=3 d=4 x=4 g=1
66: i=26 j=9 k=30 p=1 f=2 h=1 e=3 d=4 x=5 g=1
67: i=4 j=12 k=30 p=1 f=2 h=1 e=3 d=4 x=5 g=1
68: i=4 j=15 k=30 p=1 f=2 h=1 e=3 d=4 x=5 g=0
69: i=6 j=18 k=30 p=1 f=2 h=1 e=3 d=4 x=5 g=0
70: i=8 j=21 k=30 p=1 f=2 h=1 e=3 d=4 x=5 g=0
71: i=10 j=24 k=30 p=1 f=2 h=1 e=3 d=4 x=5 g=0
72: i=12 j=27 k=30 p=1 f=2 h=1 e=3 d=4 x=5 g=0
73: i=14 j=27 k=30 p=1 f=3 h=1 e=3 d=4 x=5 g=0
74: i=16 j=27 k=30 p=1 f=3 h=1 e=3 d=4 x=5 g=0
75: i=18 j=27 k=10 p=1 f=3 h=1 e=3 d=4 x=5 g=0
76: i=20 j=27 k=15 p=1 f=3 h=1 e=3 d=4 x=5 g=0
77: i=22 j=27 k=20 p=1 f=3 h=1 e=3 d=4 x=5 g=0
78: i=24 j=27 k=25 p=1 f=3 h=1 e=3 d=4 x=5 g=0
79: i=26 j=27 k=30 p=1 f=3 h=1 e=3 d=4 x=5 g=0
79: Ready to accept output from `fiveout' ? (y/n) y
79: Output from channel `fiveout' = 3
80: i=26 j=27 k=30 p=1 f=3 h=1 e=4 d=4 x=5 g=0
80: Ready to accept output from `primeout' ? (y/n) y
80: Output from channel `primeout' = 1
```

It takes 15 clock cycles for the sieve to determine that 3 is a prime. Since this is a pipelined system, each of the three stage processes is working on a different number at any given time, and the speedup achieved would approach three times that of a non-pipelined version if the times for each stage were equal (in the future more efficient version).

4.3 Commstime in Hardware

We next produce actual hardware for two of these simulations, and then check that hardware to ensure that it does indeed match the CSP specification. The first program is the `Commstime` routine; it is more complicated than the simpler algorithms, and yet provides an easy-to-observe output – a continuously incrementing 8-bit number.

The hardware used in this test was a Celoxica RC200 development board [16]. The FPGA provided on this board is a Xilinx XC2V1000 [17], which includes 11,520 logic cells and 720 KB of RAM. This is ample room for the implementation of the `Commstime` algorithm.

The process of placing the Handel-C code on the FPGA is somewhat complicated. The CSPtoHC compiler translates CSP scripts into Handel-C code for the simulator. If an actual hardware build is required, the Handel-C file must be modified a little; a clock source must be defined, and any channels that communicate with external devices must be renamed and mapped to valid FPGA port pins. The beginning section of the modified `Commstime` Handel-C code is

```
#define PAL_TARGET_CLOCK_RATE 2000000
#include "pal_master.hch"

void main(void)
{
      chan d, c, b, a;
      int 8 e;

      interface bus_out() WriteData(int 8 DataOut = e) with
            {data = {"M3","P2","P1","N2","N1","M2","M1","R2"}};
```

Notice that a 2 MHz clock is defined, and the channel formally labeled output channel `e` is now a `bus_out` construct, mapped to eight I/O pins on the FPGA.

Once these modifications have been made, the Handel-C source code is opened in the Celoxica DK editor, where it can be compiled to an ".edif" file (among other options). The ".edif" file then needs to be fitted to a chip, where it is placed, routed, and tested, and a "bit" file is produced. Using the RC200 development board greatly simplifies this procedure, since the DK builder automatically performs the placement and routing routines, and produces an output file specifically configured for the RC200 board. The Celoxica FPU utility is then employed to program the FPGA.

Once the FPGA is programmed, the output can be observed by connecting a logic analyzer to the output pins of the FPGA. The results obtained are displayed in fig. 6 (remember that `Commstime` is a benchmark program used to observe the speed in which different processes connected by channels interact). The output of `Commstime` is an 8-bit number that continuously increments. The results obtained in fig. 6 confirm that this is indeed the case.

The least-significant bit, bit 0, changes every time the 8-bit number is incremented, which occurs every 2.5 microseconds. Going back to the earlier simulation of the `Commstime` algorithm, we predicted that each cycle would take 5 clock ticks. Since the FPGA is running at 2 MHz, that equates to a clock period of 500 ns. If the cycle truly takes 5 clock ticks, that would be 500 ns times 5, or 2.5 microseconds; thus, the simulator and the hardware agree precisely.

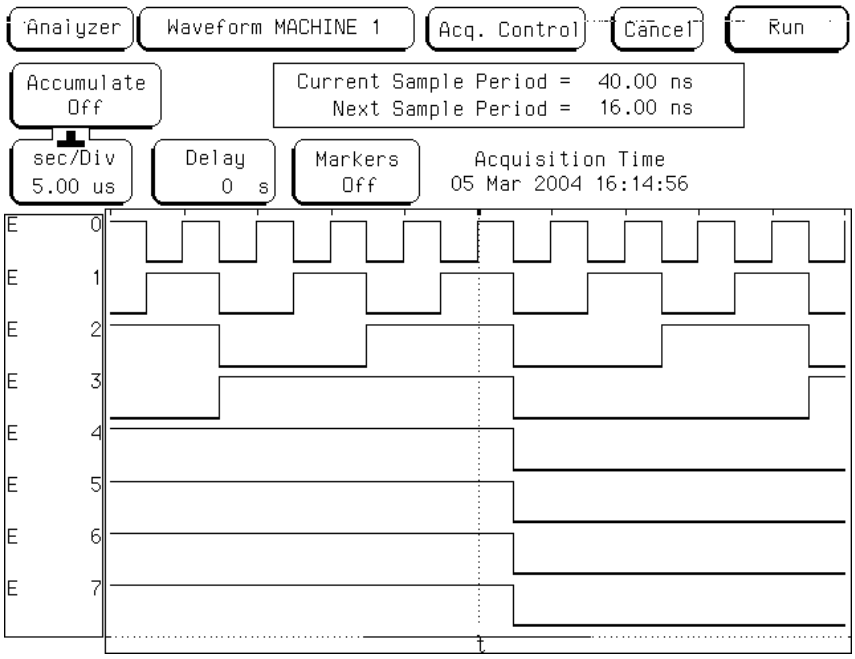

Figure 6: `Commstime` output from logic analyzer.

4.4 Sieve of Eratosthenes in Hardware

The second Handel-C program that was taken to the FPGA stage was the sieve of Eratosthenes. The same steps were taken to program the FPGA in this case as were in the Commstime example. The Handel-C code was modified to provide a clock source, and output signals were mapped to pins on the FPGA. The modified portion of the Handel-C source code is shown here:

```
// An implementation of the Sieve of Eratosthenes that calculates
// all prime numbers between 2 and 25.  This is done by marking
// all numbers that are multiples of 2, 3, or 5 (which is the
// square root of 25) as non-prime.

#define PAL_TARGET_CLOCK_RATE 2000000
#include "pal_master.hch"

void main(void)
{
      chan threepout, twopout, threeout, twoout, prodout;
      int 8 y, z;

      interface bus_out() WriteData(int 8 DataOut = y) with
            {data = {"M3","P2","P1","N2","N1","M2","M1","R2"}};
      interface bus_out() WritePrime(int 8 PrimeOut = z) with
            {data = {"T2","R4","R3","P3","P4","N4","N3","M4"}};
```

A logic analyzer was once again connected to the output pins of the programmed FPGA to observe the operations taking place. Figure 8 shows the resulting output. The top five data lines show the number being output from process Five, while the bottom line indicates whether that number is prime. Reading straight from the output, the numbers 0, 1, 2, 3, 5, 7, 11, 13, 17, 19, and 23 are marked as primes, all the others are not. It can be concluded that this implementation of a 3-stage sieve of Eratosthenes is correct. (Note that 0, not a prime, is marked as prime by the algorithm; added complexity could avoid this.)

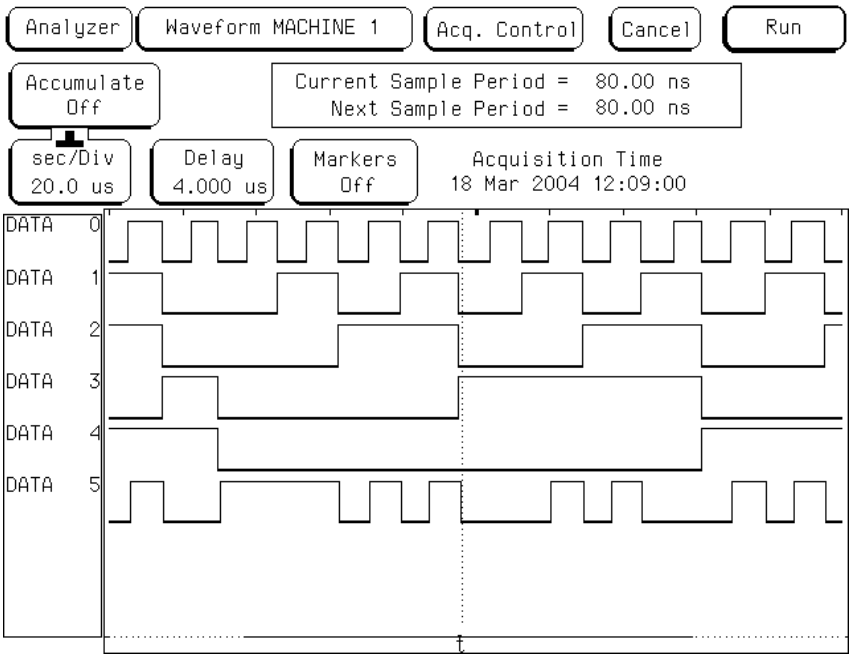

Figure 8: Sieve of Eratosthenes output from logic analyzer.

5. Summary and Conclusions

5.1 Successes

A translator has been constructed that correctly converts a useful subset of the CSP process description language into C code that can be implemented directly in FPGAs. Simulations and actual FPGA implementations verify that the hardware created does indeed match the original CSP specification.

5.2 Possible Improvements

There are many facets of CSP and Handel-C that have not been represented here. The most prominent is that of the introduction of discrete timing to a set of processes, which is critical for real-time systems.

A second improvement that could be provided is that of intra-process verification. CSP does an excellent job of modeling and verifying the external communications between processes, but using other tools, such as Promela [18], could improve the specification and verification of the internal operations.

A more user-friendly interface, possibly some sort of graphical user interface (GUI), could be developed to increase usefulness of the translator and provide more options and feedback. Options could also be added to compile the CSP script into simulation-style or release-style Handel-C.

Optimization of the Handel-C code is also possible. The Handel-C code produced by the compiler is functional, but there are many ways that the code could be rewritten to enhance and optimize its functionality. Optimization could occur at the CSP level, during the CSP to Handel-C conversion, or after the Handel-C has been translated.

5.3 Future Work

This project is just a small step in the field of studying the relationships and interactions between software and hardware. This project has significant room for improvement, including adding additional features of CSP to the allowable input and making the CSPtoHC translator more user-friendly.

Another area for future research would be comparing the efficiency of the generated Handel-C code with Handel-C code that was written by hand. A comparison of the efficiency of Handel-C with hardware languages such as VHDL, Verilog, or ABEL might prove interesting.

There is significant room for optimization in the Handel-C code. As mentioned previously, the focus during this project was to create Handel-C that was robust and functional, with efficiency being a lesser goal. Now that the functionality has been proven, more effort can be put towards making the code more efficient.

References

[1] C.A.R. Hoare, "Communicating sequential processes," *CACM*, vol. 21(8), pp. 666-667, Aug. 1978.

[2] Celoxica, "Handel-C Software-Compiled System Design." Available: `http://www.celoxica.com/methodology/handelc.asp`

[3] Formal Systems (Europe) Ltd, "Formal Systems Software." Available: `http://www.fsel.com/software.html`

[4] J. R. Levine, T. Mason, and D. Brown. Lex and Yacc. Sebastopol, CA: O'Reilly, 1992.

[5] STMicroelectronics. "Welcome to ST." Available: `http://www.st.com`

[6] J. Campbell, "*Efficient automatic mapping of hardware to software*," PhD dissertation, Utah State University, Logan, UT. 2002.

[7] W. Zhou and G. S. Stiles, "The automated serialization of concurrent CSP scripts using Mathematica," presented at the Communicating Process Architectures Conference, Enschede, the Netherlands, 2000.

[8] V. Raju, L. Rong, and G. S. Stiles, "Automatic conversion of CSP to CTJ, JCSP, and CCSP," presented at the Communicating Process Architectures Conference, Enschede, the Netherlands, 2003.

[9] G. Hilderink, J. Broenink, W. Vervoort, and A. Bakkers, "Communicating java threads," in *Proceedings of WoTUG 20: Parallel Programming and Java*, A. Bakkers, Ed. Amsterdam, the Netherlands: IOS Press, 1997, pp. 48-76.

[10] University of Twente, "CSP for Java." Available: `http://www.rt.el.utwente.nl/`

[11] 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*, H. R. Arabnia, Ed. Athens, GA: CSREA Press, 2000, pp. 51-57.

[12] University of Kent, "Communicating Sequential Processes for Java (JCSP)." Available: `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`

[13] Quickstone Technologies Ltd, "CCSP." Available: `http://www.quickstone.com/xcsp/ccspnetworkedition/`

[14] R. M. A. Peel and J. W. H. Feng, "Steps in the Verification of an Occam-to-FPGA Compiler". University of Surrey. Guildford, Surrey, U.K.

[15] S. Schneider. Concurrent and Real-Time Systems: The CSP Approach. Chichester, UK: Wiley, 1999.

[16] Celoxica, "Products: RC200 Development Board." Available: `http://www.celoxica.com/products/boards/rc200.asp`

[17] Xilinx, "Xilinx Products and Services." Available: `http://www.xilinx.com/xlnx/xil_prodcat_product.jsp`

[18] Promela, "Concise Promela Reference." Available: `http://spinroot.com/spin/Man/Quick.html`