Communicating Process Architectures 2008
Susan Stepney, Fiona Polack, Alistair McEwan, Peter Welch, and Wilson Ifill (Eds.)
IOS Press, 2008

Experiments in Translating CSP||B to Handel-C

Steve Schneider [a,1], Helen Treharne [a] Alistair McEwan [b] and Wilson Ifill [c]
[a] *University of Surrey*
[b] *University of Leicester*
[c] *AWE Aldermaston*

Abstract. This paper considers the issues involved in translating specifications described in the CSP||B formal method into Handel-C. There have previously been approaches to translating CSP descriptions to Handel-C, and the work presented in this paper is part of a programme of work to extend it to include the B component of a CSP||B description. Handel-C is a suitable target language because of its capability of programming communication and state, and its compilation route to hardware. The paper presents two case studies that investigate aspects of the translation: a buffer case study, and an abstract arbiter case study. These investigations have exposed a number of issues relating to the translation of the B component, and have identified a range of options available, informing more recent work on the development of a style for CSP||B specifications particularly appropriate to translation to Handel-C.

Keywords. Handel-C, CSP||B, translation, formal development

Introduction

This paper investigates the translation of rigorous models written in CSP||B into the Handel-C programming language, as a route to hardware implementation. CSP||B [22] is a formal method integrating CSP for the description of control flow and the B-Method for the handling of state. It is supported by industrial-strength tools [9,3,6,12] which enable verification of CSP and B models, separately and together, through model-checking and proof. It is particularly suited for applications where control and data considerations are both critical for correct behaviour. Areas where CSP||B has been applied include a file transfer protocol [7], dynamic properties of information systems [8], and modelling platelets in blood clot modelling [20]. It has also been used within the AWE funded project 'Future Technologies for System Design' at the University of Surrey, which is concerned with formal approaches to co-design. An element of that project has been the investigation of a development methodology which takes CSP||B formal models to hardware implementation, via a translation to Handel-C. Handel-C [17,5] is a programming language designed for compilation to hardware, in particular FPGAs. The translation from CSP||B to Handel-C is a key link in the project's development methodology, which aims to refine requirements down to implementation, via formally verified specifications. CSP||B is used as the modelling language, since it enables formal analysis with respect to high level properties. Low level programming languages closer to code, such as Handel-C, are not sufficiently abstract to support such analysis, so our ideal is to refine the CSP||B models (once they are verified) to Handel-C code to provide an appropriate implementation.

[1]Corresponding Author: *Steve Schneider, University of Surrey, Guildford, Surrey, GU2 7XH, UK.* E-mail: `S.Schneider@surrey.ac.uk`.

There is existing work on translating CSP to Handel-C [21,16] and the translation of the CSP controllers of our CSP||B description is broadly in line with those. There is also work on translating B to Handel-C [11] through use of annotations to indicate the control flow. However, translations of CSP and B combined models have not previously been considered, and this is the gap that this paper begins to address. The novelty of the approach we are taking is the combined translation of control and state, which is not considered in the other authors' previous work, and the new issues being considered are the translation of the state aspects of a CSP||B description within the context of the CSP translation. Data is often a key aspect of a system description, and our approach introduces the ability to incorporate a significant state component at the abstract level, and through the translation.

The paper describes two exploratory case studies that have been carried out to investigate issues that arise in this translation. The translations so far have been carried out by hand. The first case study is of a buffer, where the control component looks after the input and output to the buffer, and the state component is used to store the data passing through the buffer. The second case study is of an abstract arbiter: a component that tracks a set of requests, and then chooses one element of the set. This case study is inspired by an AMBA bus case study [2,15] which makes use of an arbiter component to resolve contention for the bus. We investigate the use of channels and of signals to carry the requests. Some experimentation was done to see how fast (in terms of clock cycles) the inputting of a signal could be: instead of reading the signal into the CSP controller and then calling the corresponding B operation when appropriate, a further translation was considered in which the line could be read and the operation called within the same clock cycle.

In both case studies we begin with a CSP||B description, and then consider how it can be translated to Handel-C. The approach taken is relatively systematic in terms of how a control loop should be translated, and where the Handel-C implementations of the B machine should appear. Some parts of the specification have to be translated in a less generic way because of the specific requirements of Handel-C. Indeed, some of the motivation for these investigations is to identify where this becomes necessary.

1. Background

1.1. *CSP||B*

The formal method CSP||B is a combination of the process algebra CSP [10,19] and the state-based B-Method [1], designed to model systems which are rich both in interaction (CSP) and in data (B-Method).

CSP describes systems in terms of *processes*, which interact by synchronising on common events. Processes are defined in terms of the patterns of *events* that they can perform. A process language enables processes to be described. It includes channel input $c?x \rightarrow P(x)$, channel output $c!v \rightarrow P$, interleaved parallel $P \mid\mid\mid Q$, termination SKIP, sequential composition $P;\ Q$, and recursive definitions $N \mathrel{\widehat=} P$. The language also supports choice, synchronised parallel, abstraction, mutual recursion, and channel communications that have both input and output, though these are not used in this paper. Tool support [9,12] enables model-checking for refinement, deadlock, divergence, and temporal logic properties.

The B-Method describes systems in terms of *abstract machines*, which encapsulate state and operations. It is supported by tool support which at the formal modelling level enables proof of internal consistency and refinement [6,3], and model-checking [12]. The tool support also provides for fully formal development. An abstract machine will define state variables, initialisation, invariants on the state, and operations which can be used to read or update the state (or both). Abstract machines can be proven internally consistent with respect to their invariants: that every operation, called within its precondition, is guaranteed to preserve

```
a:= 0;
b:= 1;
par {
  a:=2;
  b:= a;
}
```

Figure 1. A parallel assignment

the invariant. Of particular relevance to the work presented here is the use of simultaneous assignments: the construction $x := E \parallel y := F$ evaluates both E and F in the original state, and then simultaneously updates both x and y. This is the same treatment of parallel assignment as provided by Handel-C. Static information (e.g. user defined types, constants, auxiliary functions) can also be defined within abstract machines. The B-method is a design methodology and supports data refinement: changing the way data is represented so the model can be developed in more detail and progress towards implementation. Such changes are written as *refinement machines*, which encapsulate the changed data representation and its relationship to the original abstract machine.

CSP||B [22] combines CSP processes and B machines by treating B machines as processes that interact through the performance of their operations: an operation op passing a value x will interact with a process which has op as a channel which carries x. The typical unit is a *controlled component* consisting of a CSP process in parallel with a B machine. We think of the CSP process as governing the control flow of the combination, determining the next possible operations or communication; and the B machine maintains the state, and is controlled and updated by the CSP process. For example, in Section 2 we will see a $CELL$ machine (Figure 2) controlled by a process $CONTROLLER1$ (Figure 3). The combination is written $CONTROLLER1 \parallel CELL$. There is also a general approach to establishing consistency between the CSP and the B parts of a description: that operations are called only within their preconditions.

1.2. Handel-C

Handel-C [5] is a clock-synchronous programming language reminiscent of occam [14,18]: it offers co-routines, concurrency and communication, and it is intended for programming applications onto reconfigurable hardware, specifically Field Programmable Gate Arrays (FPGAs). Its aim is to provide an interface to hardware that looks and feels like a traditional imperative programming language, while allowing the programmer to exploit the natural concurrency inherent in a hardware environment. It is not a hardware description language; it is a high level programming language with a clean, intuitive semantics, with a syntax based on C including extensions for concurrency and communication. The extensions are based on similar constructs in occam [14] and have a theoretical basis in CSP.

Handel-C differs from occam in two main respects. Firstly, parallel assignments to state variables are synchronous: all assignments take place on the rising edge of the hardware clock. Secondly, shared access to variables is permitted between processes.

The code fragment in Figure 1 demonstrates how concurrency interacts with the clock in Handel-C. The first two lines of this program are executed sequentially, with the semicolon marking a clock tick: after the second clock tick, a will have the value 0, while b will have the value 1. The program then enters the parallel block. This block takes one clock cycle to execute: both assignments happen simultaneously. Once it has completed, b will have been assigned the *initial* value of a (the value held in a before the rising clock edge). On the rising clock edge, both assignments happen: a takes on the value 2, while b takes on the value 0; the clock then ticks and the statement terminates.

The synchronous nature of assignment gives a very well defined notion of timing in the language: an assignment takes a clock cycle, and expression evaluation takes no time. This is achieved in implementation as follows: when a program is loaded onto an FPGA, a calculation is required to set the clock speed of the hardware. The maximum clock speed permissible by any one application is the maximum combinatorial cost of the longest expression evaluation, along with the assignment cost. Therefore, any one application may be clocked at a different speed from any other on the same FPGA, and the same application may be able to achieve a different speed on a different FPGA.

Shared access to variables also needs to be handled with care. As may be expected in a hardware environment, if two (or more) processes attempt to write to a register at the same time, the result is undefined. There is no resolution offered by Handel-C to this: should a programmer insist on writing co-processes that both write to a shared variable, it is the programmer's responsibility to deal with any non-determinism arising. In the Handel-C development environment DK3.1 [13] used for the experiments reported in this paper, the compiler emits a warning if the same variable appears on the left hand side of an assignment in two or more parallel blocks.

Handel-C provides *channels* which allow concurrent processes to interact through synchronous communication. Channels behave as they do in CSP and occam: communication occurs when both sender and receiver are ready to engage in it, and a value is passed from sender to receiver. The type of the channel defines the possible values that can pass along it.

Handel-C also provides *signals* as a mechanism for communication between concurrent processes. Processes assign values to signals, which can be read by other processes in the same clock cycle. A signal will hold the value assigned to it just for the period of the clock cycle, after which the value will no longer be available. In this sense a signal behaves as a wire. A signal can also have a default value, which it holds when it has not been assigned any other value.

1.3. Translation to Handel-C

When approaching the translation to Handel-C, we integrate the CSP and the B aspects of the description by embedding the code corresponding to the B into the translation of the CSP controller. This can be done in one of three ways:

- include the translation of the operation directly where it appears in the CSP controller. This will involve substituting the actual for the formal parameters directly in the code that is being included;
- use Handel-C macros to encapsulate the translation of the operation;
- use Handel-C functions to encapsulate the translation of the operation.

The first choice is direct, and it may result in replicating code where an operation is called a number of times in the body of a controller. The second choice avoids the replication of code, but at the cost of generating fresh hardware for each instance of a macro call in the code. The main difference between these two choices is to do with readability of the code, and separation of concerns. The third choice provides the best chance of obtaining a separate translation of an entire B machine which is then accessed from the controller. It allows reuse of hardware by generating one implementation of each function. However, multiple instances of a function cannot be called in parallel, in contrast to macros. Hence each of the approaches brings different benefits and it is clear that there will be trade-offs in deciding which to deploy in any given situation.

```
MACHINE          CELL
VARIABLES        xx
INVARIANT        xx : NAT
INITIALISATION   xx := 0
OPERATIONS
   set(yy) = PRE yy : NAT
             THEN xx := yy
             END;

  yy <-- get =  BEGIN yy := xx END

END
```

Figure 2. The CELL machine

$$CONTROLLER1 \mathrel{\widehat{=}} bufin?Value \rightarrow set!Value \rightarrow$$
$$get?Stored \rightarrow bufout!Stored \rightarrow CONTROLLER1$$

Figure 3. CSP controller for the CELL machine

2. Case Study I: Buffers

The first case study is concerned with simple buffers-style behaviour. This is an artificial example chosen to drive the initial exploration. Thus it includes a B machine to manage some simple state, and control written in CSP to access and update that state. The purpose of this example is to see how such a description translates.

In the first instance, the B machine controlling the information consists of a memory cell in which a single value can be set or retrieved. We consider two controllers for this machine. The first controller simply accepts values from some external source, puts them into the cell, and then reads the value from the cell and outputs it. This case study exposes the issues involved in producing Handel-C corresponding to a CSP||B specification. Issues emerged on declarations, header files, Handel-C types, and on how to construct an appropriate harness to best interact with the resulting hardware.

The second controller consists of two concurrent processes: an input process responsible for setting the state, and an output one responsible for reading the state and passing it externally. The processes do not synchronise with each other, but both made use of operations of the memory cell.

In the final case study of this family, the B machine tracks not just a single value, but an array (of bits or booleans) in which each element of the array can be flipped independently by the operation called with the corresponding index.

2.1. CSP||B description of a simple buffer

The first buffer consists of a piece of state in a B machine used to capture the value to be stored in the buffer given in Figure 2, and a CSP controller operating a store/fetch cycle given in Figure 3. Capturing a single piece of state using a B machine, and controlling it with a CSP process, forms a simple example of interaction with state and control in the CSP||B specification.

Two operations exist in this B machine. The first, `set`, takes a natural number as a parameter and sets the value of local state. The second, `get`, returns the value of the local state.

The first CSP controller is a recursive process that reads in a value from the environment on the channel *bufin*, and then calls the operation in the B machine to set the value. It then reads the value back from the B machine on the channel get, before outputting the value to the environment on the channel *bufout*. The process then recurses.

2.2. Handel-C translation of a simple buffer

The code presented in this section is the Handel-C translation of the CSP||B specification $CONTROLLER1 \parallel CELL$.

The B machine essentially describes the kind of state for which the machine is responsible. Thus its translation to Handel-C will consist of the state declarations (and any other declarations included in the machine definition), and its initialisation to ensure it begins in the correct state. In this case study, we will be including the translation of an operation directly in the place where it is called within the CSP description, so translation of operations will occur within the CSP translations.

The machine has one state variable: xx. Variables in Handel-C must have a specified bit width. In this case study, we have decided to use 3-bit values.

```
#define WORD_TYPE unsigned int
#define WORD_WIDTH 3
WORD_TYPE WORD_WIDTH xx;
```

The following channel declarations provide input and output channels.

```
  chan WORD_TYPE WORD_WIDTH bufout;
  chan WORD_TYPE WORD_WIDTH bufin;
```

The following macros are used to implement atomic statements in the specification. Their use in this case study helps clarify which type of CSP specification statements are implemented, and how they are implemented. The general approach would be to use macros for B operations corresponding to CSP events. Here they are all atomic (input, output, and assignment), but in principle they could be more complex.

The macro procedure CHANINPUT takes a channel name X and a variable Y, and performs the Handel-C channel input X?Y. The macro does not type the parameters, although they are checked at compile-time. The macro CHANOUTPUT does the same for a CSP channel output.

The macro BASSIGN takes a variable Y, and assigns to it the value X; the same type-checking rules apply. Furthermore, it is assumed that the necessary variable parameters have been declared in advance.

```
  macro proc CHANINPUT(X,Y) { X?Y; }
  macro proc CHANOUTPUT(X,Y) { X!Y; }
  macro proc BASSIGN(X,Y){ Y = X; }
```

In the main source file, the initialisation xx := 0 from the INITIALISATION clause of the CELL machine (Figure 2) is translated to Handel-C as an assignment. This is followed by the implementation of the CSP controller.

```
void main(void)
{
  /* B machine initialisations */
  BASSIGN(0, xx);
  /* CSP MAIN controller */
  SimpleBuffer(bufin,bufout);
}
```

The implementation of the controller declares two local variables Stored and Value for the two variables used in the description of $CONTROLLER1$.

The CSP loop is translated to Handel-C by translating each of the events in the description of $CONTROLLER1$ in turn:

$$
\begin{aligned}
In &\mathrel{\widehat=} bufin?Value \rightarrow set!Value \rightarrow In \\
Out &\mathrel{\widehat=} get?Stored \rightarrow bufout!Stored \rightarrow Out \\
CONTROLLER2 &\mathrel{\widehat=} In \mid\mid\mid Out
\end{aligned}
$$

Figure 4. a second CSP controller for CELL

- $bufin?Value$ is translated as a channel input.
- $set!Value$ represents a call of the `set` operation in the CELL machine, and so the body of that operation in the B description is translated, as an assignment of `Value` to `xx`.
- $get?Stored$ represents a call of the `get` operation in the CELL machine, and is translated as an assignment of `xx` to `Stored`.
- $bufout!Stored$ is translated as a channel output: $bufout$ is a channel rather than an operation call.
- The overall recursive definition is translated to a `do` loop.

The result is as follows:

```
macro proc SimpleBuffer(bufin, bufout)
{
  /* local CSP state */
  WORD_TYPE WORD_WIDTH Stored;
  WORD_TYPE WORD_WIDTH Value;

  do {
    CHANINPUT(bufin, Value);
    BASSIGN(Value,xx);
    BASSIGN(xx, Stored);
    CHANOUTPUT(bufout, Stored);
    } while(1);
}
```

2.3. A two-part controller

A more complicated case study is to manipulate the state in the B machine using two interleaved controllers: a process In handling input, and a process Out handling output. These are given in Figure 4, together with the resulting controller $CONTROLLER2$ which combines them with interleaving parallel. This does not provide a buffer in CSP||B because the two component controllers can proceed independently, and at different rates. However, our translation to Handel-C will result in their clock-synchronous execution, providing the behaviour of a buffer containing an initial value.

Given that there is no communication and synchronisation between the input and output processes, the implementation of $CONTROLLER2 \parallel CELL$ is straightforward. Each is implemented using the same approach as previously, and then the results are run in parallel. We obtain the following:

```
macro proc InterleaveBuffer(bufin, bufout) {
  /* local CSP state */
  WORD_TYPE WORD_WIDTH Stored;
  WORD_TYPE WORD_WIDTH Value;

  par {
    do {CHANINPUT(bufin,Value);
        BASSIGN(Value, xx);
```

```
MACHINE BOOLARRAY
VARIABLES   xx
INVARIANT   xx : NAT --> 0..1
INITIALISATION  xx := NAT * { 0 }
OPERATIONS
      set(yy) = PRE yy : NAT
                THEN xx(yy) := 1 - xx(yy)
                END;

yy <-- get = BEGIN
              yy := xx
            END
END
```

Figure 5. The BOOLARRAY machine

$$In \mathrel{\widehat{=}} bufin?Value \rightarrow set!Value \rightarrow In$$

$$Out \mathrel{\widehat{=}} get?Stored \rightarrow bufout!Stored \rightarrow Out$$

$$CONTROLLER3 \mathrel{\widehat{=}} In \mathrel{|||} Out$$

Figure 6. A controller for the BOOLARRAY machine

```
       } while(1);

    do {BASSIGN(xx, Stored);
        CHANOUTPUT(bufout, Stored);
       } while(1);
  }}

void main(void)
{
  BASSIGN(0, xx);
  InterleaveBuffer(bufin,bufout);
}
```

The CSP description does not in fact necessarily behave as a buffer, because In and Out are independent and could proceed at different rates. However, their synchronous implementation in Handel-C means that they execute at the same rate, matching outputs to inputs, and yielding a buffer. This is a refinement of the behaviour encapsulated in the CSP||B description.

2.4. A boolean function

This next case study increases the complexity of the B machine. Instead of a natural number, we store a function from natural numbers (indexes) to booleans. This case study introduces more complexity into the implementation as the local state is not just a simple variable. The machine is given in Figure 5. Here the controller accepts bit-wise updates to the array on the input cycles, but outputs the entire array on each output cycle. The associated controller is given in Figure 6.

In the following implementation of $CONTROLLER3 \parallel BOOLARRAY$, we must restrict to a finite domain for the function: in fact three bit integers, the set 0..7.

```
#define WORD_TYPE unsigned int
#define WORD_WIDTH 3
```

```
#define ARRAY_WIDTH 8
unsigned int 1 xx[ARRAY_WIDTH];
```

Input to this buffer is an index, and the bit stored at this index is flipped. The buffer output is the sequence of values stored in the array. As this is sent over a channel it must be a single word of 8 bits. The following code achieves this—but it is not the simple assignment that may have been naïvely expected. An array of 8 bits is of a different type to an 8-bit word so xx cannot be assigned to $Stored$ directly.

```
macro proc BoolArray(bufin, bufout) {
  /* local CSP state */
  unsigned int WORD_WIDTH Value;
  unsigned int ARRAY_WIDTH Stored;

  par {
    do {
      CHANINPUT(bufin,Value);
      BASSIGN(!xx[Value], xx[Value]);
    } while(1);

    do {
      BASSIGN(xx[7]@xx[6]@xx[5]
              @xx[4]@xx[3]@xx[2]
              @xx[1]@xx[0], Stored);
      CHANOUTPUT(bufout, Stored);
    } while(1);
  }
}
```

2.5. *Issues*

In this section, we highlight the issues uncovered in implementing this specification.

1. Variable size in bits.
   In the B machine, the local state was expressed as a natural number. In a Handel-C program, each variable must be declared in terms of a finite size in bits. This is an issue with any state-based development—and the solution normally is to retrench [4] to the initial abstract specification and rewrite it with acceptable bounds. Retrenchment provides a controlled and traceable way of allowing changes to the specification in the light of implementation considerations. This is the solution that was adopted in this case.
   An orthogonal issue concerns the use of signed bits: a number can be represented in *signed* or *unsigned* format; and the behaviour of program code is highly dependent on this choice. Therefore, we expect that a useful further investigation would be to refine data types in the B machine into bitwise implementations, thereby ensuring that all operations consider signed bit issues.
2. Preconditions are a specification statement, not an implementation one.
   Preconditions are specification statements, not implementation ones. In analyzing a CSP||B specification, one normally proves that the CSP controller never calls a B operation in a state that would violate a precondition. Therefore the precondition need not be implemented: the fact that the precondition is always respected is discharged during analysis and there is no requirement to translate any operation preconditions into Handel-C. Preconditions are therefore dropped during translation.
3. Main processes
   CSP processes are frequently defined—and may easily be analyzed—without declaring a starting point. For instance, the behaviour of the recursive processes $Ex \mathrel{\widehat{=}} a \rightarrow$

$b \rightarrow Ex2$ and $Ex2 \mathrel{\widehat{=}} c \rightarrow d \rightarrow Ex$ have quite clear behaviour; but it is not explicit where *execution* begins. Programs however have a very clearly defined starting point for execution—in Handel-C this is given by a `main` function. When writing a CSP||B specification for implementation, one must consider where execution will begin and how this can translate into a `main` function. One useful solution to this is the one adopted by the ProB tool [12]—to require that one of the processes is named `Main`.

4. Variable scope and introduction from channel inputs
   Programming languages usually have to be explicit about when they introduce storage and declare variables. However, CSP is not so explicit. Another issue thrown up by this case study concerned the introduction and management of storage introduced by CSP channel inputs.
   The input communication $a?x \rightarrow P$ introduces the variable x, and then writes to it from the channel a. This is similar to a `var x` declaration just before the $a?x$ input. The process $P \mathrel{\widehat{=}} a?x \rightarrow P$ therefore introduces a copy of x each time it recurses, but the scope of the local variable declaration includes all recursive calls. Thus in CSP this amounts to a nesting of local variable declarations. However, the tail recursive nature of the process definition makes it sufficient in an implementation to declare x only once (as would be natural in Handel-C or occam), and to re-use it on each cycle, since each nested declaration supersedes the previous one. Alternatively it could be declared afresh on each cycle. In any case translation to Handel-C would preclude dynamic generation of new local variables, since there would need to be a bound on the hardware state, so a constraint such as the need for recursions to be tail recursive would need to be present to enable translation.
5. Type conversion
   The boolean buffer highlighted an issue of type conversion that became apparent because of the necessity of compile-time constants in bit indexing. We would normally expect that type converters would be introduced at a formal level when data-refining abstract data type; but the appearance of this issue in this case study means that we should be aware of it in further investigations.

3. Case study II: Simple Arbiter

The AMBA AHB 2.0 specification [2] for the AMBA bus makes use of an arbiter component to choose between competing requests for a bus. This case study is an idealisation of the arbiter function, tracking a set of requests, and then choosing between them. Our focus is on issues such as the storing of local state, interaction of local state with timing, and the distinction between signals and channels, and the impact this has on specifications and implementations.

The case study uses a machine which maintains a *set* of values (4 bit numbers in fact). This is an abstract data structure used in the specification of the AMBA arbiter within the 'Future Technologies for System Design' project, and so it was a natural candidate for exploration. The machine allows two operations: `add` which adds a value to the set; and `choose` which selects an arbitrary element from the set (the statement `yy :: ss` is a nondeterministic assignment of any element of the set `ss` to the variable `yy`), and also resets the set to $\{0\}$. A default element is always in the set. A more abstract specification could be less determined when no element has been added to the set, for example deadlocking, or providing any arbitrary value. However, the AMBA case study specifies the use of a default element, and so we follow this approach.

The translation of the machine will be done it two steps: it first needs to be refined within the B-Method in order to bring it to a point closer to implementation and thus appropriate for translation. The resulting refinement machine is then used in the translation to Handel-C.

```
MACHINE        SetChoice
CONSTANTS      NAT4
PROPERTIES     NAT4 = 0..15
VARIABLES      ss
INVARIANT      ss <: NAT4 & 0 : ss
INITIALISATION ss := {0}
OPERATIONS
yy <-- choose = BEGIN yy :: ss
                   || ss := { 0 }
               END;

    add(xx) = PRE xx : NAT4
              THEN ss := ss  \/ { xx }
              END
END;
```

Figure 7. The SetChoice machine

The controller for this machine reads bits on a number of lines concurrently, and for any line, if its value is 1 then it should be added to the set. When these have all occurred, then the choice is read and passed as output.

This case study demonstrates the clear difference between treating the input line as a signal, and treating it as a channel. Both translations are provided. The difference in timing behaviour and hence the resulting behaviour between reading signals and reading channels is exposed by the test cases.

Some experimentation was also done to see how fast (in terms of clock cycles) the inputting of a signal could be: instead of reading the signal into the CSP controller and then calling the corresponding B operation when appropriate, a further translation was considered in which the line could be read and the operation called on the same clock cycle.

3.1. CSP||B description of a simple arbiter

The abstract state contains a set, ss, containing natural numbers in the range 0..15. Elements may be added into the set using add. An element may be chosen from the set non-deterministically using the operation choose, at which time the set is also reset to its default value $\{0\}$. The machine is given in Figure 7. Note that the B semantics of parallel assignment means that yy is chosen from the original value of the set ss while ss is simultaneously updated.

This machine is not ready for implementation in Handel-C because of the presence of non-determinism, and also of the presence of a set-valued variable. The non-determinism in the choose operation needs to be resolved: we decide as a refinement step to choose the maximum value in the set. Furthermore, we need to refine abstract sets into something more concrete, as well as the operations performed on the set. We achieve this as a data refinement mapping the set of values to an array of bits, indexed by the values. The resulting refinement is encapsulated within B as a REFINEMENT machine, and given in Figure 8. Note the linking invariant stating (in ascii) that $ss = arr^{-1}(\!|\ \{1\}\ |\!)$. This states that the set ss is the same as the inverse image of the array (considered as a function) arr on the set $\{1\}$: in other words, the set of all locations in arr that contain the value 1.

In this refinement machine, the set is represented by an array of bits: if a given element is a member of the set, the value of the bit at that index is 1, otherwise is it 0. The first element of the array is always set to 1, implementing the default value. We choose some refinement of the abstract specification of choose, and (arbitrarily) select the highest-valued member: the function max selects the element with the highest index. This function will be defined in

```
REFINEMENT SetChoiceR
REFINES    SetChoice
VARIABLES  arr
INVARIANT  arr : 0..15 --> 0..1
           & ss = arr~ [{ 1 }] & 0 |-> 1 : arr
INITIALISATION  arr := (1..15 * { 0 })
                       \/ { 0 |-> 1 }
OPERATIONS
yy <-- choose =
         BEGIN yy := max(arr~[{ 1 }] )
                     || arr := { 0 |-> 1 }
                        \/ (1..15 * { 0 })
         END;

       add(xx) = BEGIN arr(xx) := 1 END

END;
```

Figure 8. The refinement machine SetChoiceR

$$\begin{aligned} \mathit{Read}(x) = \mathit{line}.x?b \rightarrow\ & \textbf{if}\ (b = 1) \\ & \textbf{then}\ \mathit{add}!x \rightarrow \mathit{SKIP} \\ & \textbf{else}\ \mathit{SKIP} \end{aligned}$$

$$\begin{aligned} \mathit{CONTROLLER} = & (|||\ x : 0..15 \bullet \mathit{Read}(x)); \\ & \mathit{choose}?y \rightarrow \mathit{out}!y \rightarrow \mathit{CONTROLLER} \end{aligned}$$

Figure 9. The CSP controller for SetChoice

the implementation. Finally, the add operation sets the bit of the given parameter to true. The resulting machine is now appropriate for translation to Handel-C.

The controller is given in Figure 9. The process Read listens on the channel $\mathit{line}.x$, where x is an index. If the boolean value communicated on that line is true, the result is stored. The process $\mathit{CONTROLLER}$ replicates Read for each of the lines—there are 16, as the specification is concerned with 4 bit numbers. After reading each line, a candidate is chosen by the B operation choose, and output to the environment.

3.2. Handel-C translation of a simple arbiter using channels

In this section we present Handel-C implementations of the simple arbiter. Several implementations are presented. The first, is a straightforward channel implementation. The second uses signals instead of channels in a naïve way, simply using them in place of the channels in the previous description—but we see that this approach does not work. The third also uses signals, but is an adaptation that performs everything in a single cycle. We discuss what an abstract specification and controller of this version may look like, and use this to demonstrate that a different specification is written for signals than for channels.

The state in the B machine is declared as a 16-element array, where each element is a single bit.

```
unsigned int 1 arr[16];
```

In this implementation, the B machine initialisation and operation choose are declared as Handel-C functions. The operation add is declared as a macro procedure: this will in-line the hardware everywhere it is called in source code.

```
void Init();
macro proc add(unsigned int 4 xx);
void choose(unsigned int* yy);
```

The initialisation takes no parameters, and returns no values. The add function takes a 4-bit integer corresponding to the array element to be set to 1. Each interleaved *Read* calls add, and so there will be a copy for each interleaved call. This would be necessary even if add were declared as a function: the possibility of up to 16 simultaneous calls of add would require 16 copies of add to be created. This arises because we wish to call a number of them concurrently within a single cycle.

The choose operation in the B machine takes no parameters, but returns a result. In this implementation we use reference parameters to the function call to return results, instead of a return value of the function implementation. This is because it is possible in a B machine to specify an operation with more than one return type, and the simplest way to implement this in a translation is with reference parameters.

Initialisation sets the value of every element in the array as described in the B machine. All elements are set in parallel so initialisation takes only one clock cycle.

```
void Init() {
  par {
    arr[0]= 1;
    par (i=1; i<16; i++) { arr[i]= 0; }
  }
}
```

The add macro is defined as follows:

```
macro proc add(unsigned int 4 xx)
                 { arr[xx] = 1; }
```

The specification did not state how max would be implemented. Normally a development would include this information. In this case, we have (arbitrarily) implemented max using a number of nested conditional statements that assigns the value stored by the reference parameter.

```
macro proc max(yy) {
  if (arr[15]==1) { *yy=15; } else {

      ⋮

  if (arr[1]==1) { *yy=1; } else {
     *yy = 0;
  } ... }
}
```

The implementation of choose resets the array, in parallel with returning the result of calling max (on the original value of the array). Note here that the reset operation uses the Init function which achieves the resetting of the array, conserving hardware.

```
void choose(unsigned int* yy) {
  par { max(yy);
        Init();
        }
  }
}
```

We now consider the implementation of the controller process. We declare an array of 16 channels corresponding to the input lines into the simple arbiter, and a single output channel.

```
chan unsigned int 1 line[16];
chan unsigned int 4 out;
```

The main process initializes the state in the B machine, and then runs a process monitoring the clock for debugging purposes in parallel with a process `SIM_DRIVER` (defined below) that drives the Handel-C simulator and the main CSP controller for the case study.

The implementation of the controller replicates one copy of the *Read* process for every input line, then calls the B operation `choose`, outputs the result and iterates.

```
void main() {
  unsigned int 4 y;
  Init();
  par {
    SIM_DRIVER();
    do {
      par (i=0; i<16; i++) { Read(i); }
      choose(&y);
      out!y;
    } while(1);
      /* CHANNELS */
  }
}
```

The *Read* process performs an input on its indexed channel, and branches on a conditional. In this case, the *SKIP* in the specification has been implemented using a `delay` to ensure that both branches of the choice take the same time.

```
macro proc Read(x) {
  unsigned int 1 b;
  line[x]?b; if (b) { add(x); }
             else { delay; }
}
```

The process that drives the simulator reads in an input word and writes indexed bits to individual lines on alternate cycles. In parallel with this, it listens on the output line from the arbiter, and writes this to simulator output, again on alternate cycles.

```
macro proc SIM_DRIVER() {
  unsigned int 16 in;
  unsigned int 4 result;
  par {
    do { input?in;
         par (i=0; i<16; i++) { line[i]!in[i]; }
    } while(1);
    do { out?result; output!result; } while(1);
  }
}
```

3.3. Handel-C (incorrect) translation of a simple arbiter using signals

In this section, we describe an attempted implementation of the CSP controller as above; but this implementation uses Handel-C signals in place of channels. This gives different behaviour, and it illustrates that channels cannot simply be replaced by signals.

The only noticeable change to the main function is that the output signal line is assigned to, rather than treated like a channel communication. `Init`, `add`, `max`, and `choose` are unchanged.

```
void main() {
  unsigned int 4 y;
  Init();
  par {
    SIM_DRIVER();
    do {
      par (i=0; i<16; i++) { Read(i); }
      choose(&y);
      out = y;
    } while(1);
  }
}
```

The Read process is changed in that it reads from a signal rather than a channel. This has a significant impact on the behaviours of the program. Reading from the signal is guaranteed to take a single cycle, and to take on *only the value written to the signal on that cycle*, whereas the channel communication is guaranteed to block until the channel is written to—and the value written is persistent until it is read.

```
macro proc Read(x) {
  unsigned int 1 b;
  b = line[x]; if(b) {add(x); }
               else { delay; }
}
```

Very contrasting behaviour is observed running this version compared to the previous. Specifically, the channel version produces the expected output, and the signals version does not. This is because there is no persistence in a signal: it only holds a value for single cycle, so unless we can be certain that it will be read on the exact cycle it is written to, the communication cannot be reasoned about. In the above code, the signal is *never* read from on the cycle it is written to.

In the next section, we show an implementation, using signals, that produces the output we would expect. We demonstrate where this is a very different implementation to the above, leading us to conclude that translating to signals requires a different approach to using channels.

3.4. An improved signal implementation

In this section, we present an implementation using signals that produces the expected output. The interesting aspect of this implementation is in the way it does things differently to the channel implementation. The different structure is required to ensure that the value of the input signals are read to, and output signals written to, on every cycle, to ensure that the value is always meaningful to the environment.

The main loop of the program writes to the signal out on every cycle. To guarantee this, the choose method must take only a single cycle to execute. This was not possible in previous implementations as the signal was assigned to, and then local state was assigned to, requiring two clock cycles.

```
void main() {
  unsigned int 4 y;
  Init();
  par {
    SIM_DRIVER();
    do {
      par {
        fastchoose(&y);
        out = y;
      }
    } while(1);
  }
}
```

The `choose` method—here named `fastchoose` to distinguish it from the previous `choose`—must occupy only one clock cycle if the timing statement mentioned above is to hold. To achieve this, we observe that the arbiter can make a decision about the input signals without storing them in local state (the B machine). Losing this assignment saves a clock cycle. The only clock cycle consumed is the one that assigns the result of the operation to the reference parameter representing the return value of the call.

```
void fastchoose(unsigned int* yy) {
  if (line[15]==1) { *yy=15; } else {

      ⋮

  if (line[1]==1) { *yy=1; } else {
    *yy = 0;
  } ... }
}
```

The read process changes also, such that it only consumes a single clock cycle. Instead of storing the value of the signal in local state, the conditional branches on the signal value. A statement like this could not be written in such a way as to consume a single clock cycle if a channel communication were used: a pipelining implementation with a cycle latency would be needed.

```
macro proc Read(x) {
  if (line[x]) { add(x); }
  else { delay; } }
```

The process driving the simulation is also different, and highlights the point above about needing a single cycle latency for a channel communication. The process reads in a value from the simulator, and on the next cycle writes it to the application signals. Signals are written to, and read from, on every cycle.

```
macro proc SIM_DRIVER() {
  unsigned int 16 in;
  par {
    do {input?in;           } while(1);
    do {par (i=0; i<16; i++)
        { line[i] = in[i]; }} while(1);
    do {output!out;      }    while(1);
  }
}
```

3.5. Issues

1. Macro procedures and function calls: in-lining code
   An issue that requires consideration is that of implementing B operations and CSP processes as either function calls or macro procedures. Use of macro procedures is generally more robust as a single piece of hardware is created for each call; whereas use of function calls is more efficient as hardware is reused. Care needs to be taken where parallel access to function calls may happen.
2. Return values from B operations
   A B operation may have several return values. In this case study we have demonstrated that return values can be implemented using reference parameters in Handel-C; therefore functions returning multiple values do not present a problem for us.

3. Implementation of B methods in single cycle
   The model of time assumes that a B operation takes a single cycle to execute. However, it is possible that a specification of a B operation may involve sequential composition; depending on the number of compositions, this can translate into a Handel-C operation that takes some other amount of time. Consideration of the timing semantics of the Handel-C operation is important; our solution to this problem is reflected in the language that we believe suitable for translation.
4. Resolving underspecification
   In the case study above, the implementation of `choose` using `max` was one of many possibilities. This could be seen as a problem of *underspecification*—it was clear what it was supposed to do, but we had not stated *how* it should be achieved. Underspecification issues need to be considered before the translation process can begin because the target language does not contain underspecification constructs.
5. Implementing $SKIP$
   Implementing $SKIP$ is an interesting issue. In this case study, we decided to insert a delay statement (e.g. in the translation of *Read*) to solve combinatorial logic issues where the CSP process terminated, to ensure that the branches either side of a choice took the same time. With more diverse choices, a delay to match the longest branch would be required. However, this may not always be appropriate or necessary.
   Currently, the issue of how to handle $SKIP$ is an open question. Clearly the language suitable for translation needs to include $SKIP$; but it cannot always translate to a delay statement. For instance, in CSP, $SKIP;\ SKIP = SKIP$; however, in Handel-C `delay; delay;` $\neq$ `delay`. The problem here is not just the loss of equivalences; the translation of $SKIP$ will depend on its context.
6. Timing of signal versus channel.
   The case study also illustrated the difference between signals and channels. Simply changing channel definitions to signal definitions (and resulting code fragments) produced an implementation that unsurprisingly did not work. A signal implementation that did work was very different in nature to the channel implementation.

4. Discussion

This paper has considered two case studies in translating CSP||B to Handel-C. The intention was to understand how to treat the state component in the translation process, and to consider a progression of state specifications: single items, arrays, and abstract sets. The CSP controllers we have considered have been in a particular form: recursive sequential processes without choice, and parallel combinations of these. Further CSP operations, including choice, synchronising parallel, and hiding, were not considered in these case studies and remain as subjects for further work.

A number of standard refinement issues arose in the buffer case study, including the need for retrenchment in the implementation of natural numbers, and the dropping of preconditions when implementing operations. More specifically to our translation, the use of local variables in CSP recursive calls has also arisen in the context of CSP to Handel-C translations [21,16], and the approach taken by them is also appropriate in our case in the presence of the B. In the third buffer we have also seen the need for type conversion in refinement in order to achieve a suitable implementation, where we wanted to allow multiple updates to the array within a single clock cycle. In all these case studies we observed how the B component of the specification is translated: to a declaration and initialisation of all the machine state, and with the bodies of the operations translated directly at the point of operation calls.

In the arbiter case study, we have explored the use of channels and of signals to pass information, and have observed the different considerations arising in these two approaches, in particular with regard to timing considerations. In the case of signals, the translation of the controller process had to be hand-crafted to keep all the necessary activity within a single cycle and avoid loss of signals. The case study brought out the question of implementation of operations to macro procedures or to function calls. Implementing operations as macro procedures creates a separate piece of hardware for each call, and so these can be run in parallel; use of function calls only creates one piece of hardware for each operation, allowing reuse for different calls, and hence is more space efficient. However, concurrent calls to the same operation are not possible, so the use of function calls might not always be appropriate, and depends on the context. The case study also demonstrated how outputs to operations can be implemented by using reference parameters, enabling the implementation of multiple outputs allowed for B operations.

These case studies have exposed a considerable amount of detail that needs to be clarified for a translation to Handel-C. The type information in the abstract model needs to be extracted from the various places where it resides: in the B machines' invariants (state), preconditions (operation input), or simply implicit (operation outputs); and in CSP channel declarations (inputs and output). This information needs to be distilled so it can appear (suitably refined) in the Handel-C translation. Other necessary clarifications include the concrete implementation of the abstract types used in the specification; whether CSP communications are to be implemented as signals or channels; whether operations are translated in-line, as macros, or as functions; treatment of $SKIP$ (i.e. should all paths through an operation take the same time); and local variable declaration and scope. This level of detail would need to be provided manually, or extracted in some way from the abstract model, in order to make any automated or machine-assisted translation possible.

This paper has been concerned with initial explorations into the translation, so has not been concerned with questions of formal correctness. Ultimately correctness of the translation would require a relationship to be established between the semantics of CSP||B and that of Handel-C, and the considerations of the details listed above come into play. The development methodology currently being developed is considering the use of an intermediate CSP||B description using a restricted subset of CSP||B which is more readily translatable to Handel-C, and whose semantics is more closely aligned with Handel-C. The benefit of this approach is that the abstract CSP||B and the restricted CSP||B are within the same semantic framework, enabling a formal relation to be established between them. It also means that only the restricted subset of CSP||B needs a translation to Handel-C. To establish correctness of the translation we would aim to establish a simulation relationship between restricted CSP||B and its Handel-C translation.

The consideration of which aspects of CSP||B translate most readily into Handel-C has also resulted in the more recent development of a clocked style for CSP||B specifications particularly appropriate to translation. The core functionality of the machine is bound up in a single operation. Inputs to the machine are either treated separately through input operations to set the state, or else incorporated as inputs to the core operation. Outputs are provided through separate output operations, to make them available to other components independently of the core operation. Our approach now is to develop the machine and identify the appropriate operations in tandem with the development of the control loop, which will typically cycle on input, core operation, and output. This approach to component specification and translation was investigated through the development of a larger case study within the Future Technologies project, and is the subject of ongoing research.

Acknowledgements

This work was funded under the AWE ‘Future Technologies for Systems Design’ project at the University of Surrey, and was carried out when Alistair McEwan was at the University of Surrey. We are grateful to David Pizarro for advice on Handel-C, and to the anonymous referees for their careful reading and insightful comments.

References

[1] J-R. Abrial. *The B-Book: Assigning Programs to Meaning.* Cambridge University Press, 1996.
[2] ARM. AMBA specification v2.0. Technical Report 0011A, ARM, 1999.
[3] B-Core. *B-Toolkit.*
[4] R. Banach, M. Poppleton, C. Jeske, and S. Stepney. Engineering and theoretical underpinnings of retrenchment. *Science of Computer Programming*, 67:301–329, 2007.
[5] Celoxica Ltd. Handel-C language reference manual. 2004.
[6] Clearsy. *Atelier-B.*
[7] Neil Evans and Helen Treharne. Investigating a file transfer protocol using CSP and B. *Software and System Modeling*, 4(3):258–276, 2005.
[8] Neil Evans, Helen Treharne, Régine Laleau, and Marc Frappier. Applying CSP||B to information systems. *Software and System Modeling*, 7(1):85–102, 2008.
[9] Formal Systems (Europe) Ltd. *Failures-Divergences Refinement: FDR2 Manual*, 1997.
[10] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.
[11] W. Ifill and S. Schneider. A step towards refining and translating B control annotations to Handel-C. In *Communicating Process Architectures*, pages 399–424. IOS Press, 2007.
[12] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *FM 2003: The 12th International FME Symposium*, pages 855–874, 2003.
[13] Celoxica Ltd. Design suite dk 3.1. 2005.
[14] INMOS Ltd. occam *Programming manual.* Prentice-Hall, 1984.
[15] A. A. McEwan and S. Schneider. Modelling and analysis of the AMBA bus using CSP and B. In *Communicating Process Architectures*, pages 379–398. IOS Press, 2007.
[16] M. Oliveira and J. Woodcock. Automatic generation of verified concurrent hardware. In *International Conference on Formal Engineering Methods, ICFEM 2007*, LNCS 4789, pages 286–306. Springer, 2007.
[17] I. Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, (12(1)):87–107, 1996.
[18] A.W. Roscoe and C.A.R. Hoare. Laws of occam programming. *Theoretical Computer Science*, 60:177–229, 1988.
[19] S. Schneider. *Concurrent and Real-time Systems: The CSP Approach.* John Wiley and Sons, 1999.
[20] S. Schneider, A. Cavalcanti, H. Treharne, and J. Woodcock. A layered behavioural model of platelets. In *ICECCS*, pages 98–106, 2006.
[21] S. Stepney. CSP/FDR2 to Handel-C translation. Technical Report YCS-2002-357, University of York, June 2003.
[22] H. Treharne and S. Schneider. Communicating B machines. In *ZB2002: 2nd International Conference of B and Z users*, LNCS 2272, pages 416–435. Springer, 2002.