A verified development of hardware using CSP||B

Alistair McEwan
Department of Computing
University of Surrey, Guildford, Surrey, U.K.
Email: a.mcewan@surrey.ac.uk

Steve Schneider
Department of Computing,
University of Surrey, Guildford, Surrey, U.K.
Email: s.schneider@surrey.ac.uk

Abstract— **In this paper, we show how a combination of the process algebra CSP and the state-based formalism B, combined into a single notation called CSP||B can be used in the formal development of hardware. The use of CSP||B and associated tools is demonstrated using a significant case study. This paper is the first recorded use of CSP||B in hardware development although it has been previously used for software.**

Development in CSP||B is both top-down and piece-wise: refinement is from an abstract sequential specification into a highly concurrent implementation. In introducing concurrency, additional requirements appear. Those requirements could be met by software, by a dedicated hardware component, or by custom hardware built on an FPGA. The piece-wise nature of a CSP||B development allow for this choice to be postponed while other components are implemented—possibly in different technologies. We use CSP||B to develop custom hardware on an FPGA, and to provide assurances that the final implementation meets both the requirements explicit in the initial specification and those which appear during development. The choice of where concurrency may be introduced in order to meet timing requirements, whilst still attaining reasonable area usage is guided by a knowledge of the application domain and the target FPGA platform. Correctness of each of the development steps is assured by refinement: proof obligations are discharged using a model-checker and a theorem prover.

I. Introduction

In this paper, we show how a combination of the process algebra CSP and the state-based formalism B, combined into a single notation called CSP||B (pronounced *CSP parallel B*) can be used in the formal development of reconfigurable hardware. The use of CSP||B and associated tools is demonstrated using a significant, realistic application. This paper is the first recorded use of CSP||B in hardware development although it has been previously used for software. The contribution of this paper may be summarised as follows:

- A demonstration of a formal CSP||B development, guided by engineering intuition and domain knowledge.
- Evidence that CSP||B forms a feasible technology upon which to build high assurance hardware systems.
- Examples of proof techniques and tool usage for CSP||B in giving these high levels of assurance.

An implementation is developed in *Handel-C*. A peculiar property of the implementation is that it is not what would naively have been expected: the design has been carefully chosen to exploit both the features of *Handel-C* and the features of the target FPGA. The goal of the development in this paper therefore is to start with an abstract specification describing the functionality required of the application, and through a process of stepwise refinement, end up with the implementation. In doing so, understanding of how CSP||B can contribute to, guide, and verify the engineers intuition in this design flow process is demonstrated.

The paper is structured as follows. The remainder of this section presents background in the technologies adopted. This is followed in section II by a description of the case study and the empirical results used to guide the formal development. An abstract specification is given in section III, and this is followed by automated verification of the functional properties in section IV. The CSP||B model is developed and refined into an appropriate implementation architecture in section V. A proof technique for justifying this development is given in section VI. Finally, some conclusions are drawn in section VII.

A. CSP

The process algebra Communicating Sequential Processes (CSP)[7], [16], [19] is a mathematical approach to the study of concurrency and communication. It is suited to the specification, design, and implementation of systems that continuously act and interact with their environment. CSP is a state-based approach to modelling—systems are characterised by the events in which they are willing to participate. The collection and interaction of these events form processes, which can be combined using the operators of CSP to describe more complex systems.

Example 1.1: Process refinement in CSP: $P \sqsubseteq_{FD} Q$

$$P \mathrel{\widehat{=}} a \rightarrow P \sqcap b \rightarrow P$$
$$Q \mathrel{\widehat{=}} b \rightarrow Q$$

□

Integral to CSP is the notion of *refinement*. A process Q is said to refine a process P if all observations possible of Q are also possible of P. In example 1.1, the process P offers to engage in either the event a or b, after which it recurses. Which of the two it chooses to offer is non-deterministic. For instance, one possibility is that it will always choose to refuse the event a. This is precisely the behaviour described by the process Q, and therefore Q can be regarded as a refinement of P, written $P \sqsubseteq_{FD} Q$. A customer who requested P would be content if Q were delivered, and in fact be none the wiser.

An important property of refinement is that if a behaviour is not possible of P, then it is also not possible of Q. For instance, if P can be shown to be deadlock-free then all refinements of P are certain to be also. Refinement is transitive.

Example 1.2: Transitivity of refinement

$$P \sqsubseteq_{FD} Q \wedge Q \sqsubseteq_{FD} R \Rightarrow P \sqsubseteq_{FD} R$$

$\square$

Failures Divergences Refinement (FDR) [9] is a tool for model-checking networks of CSP processes, checking refinements between processes, and allowing the proving or refuting of assertions about those processes. If assertions are incorrect, FDR presents a counter-example.

B. The B-method

The B-method[1], [20] is a formal approach to the specification and development of computer software. Much of the theory behind B draws from the Z notation[29], [23], and the Refinement Calculus[14], with an emphasis placed on structured specifications and automated tool support. This emphasis on tool support is the major difference between B and Z, and is really the characteristic signature of B.

A specification in B consists of a description of *abstract data types* known as *B Machines*— collections of data, operations possible on that data, and invariants constraining their legitimacy. These data types may be refined from abstract specifications through concrete models, to program code.

A typical development in B will be permeated with verification issues: showing consistency between machines, justifying decisions, and providing formal arguments proving their correctness. This structured, formal support to development has meant that it has accrued some popularity in industry.

C. Handel-C

Handel-C[3] is a language targeting Field Programmable Gate Arrays, reminiscent of `occam`[10]: co-routines, parallelism, and communication are all primitive to the language; the communication discipline is value passing with synchronous handshaking. *Handel-C* differs from `occam` in that parallel assignments to state variables all take place in synchrony on the leading edge of each clock cycle.

The clean, intuitive semantics of *Handel-C* mean that its programs may be modelled using CSP, augmented with a technique for describing abstract data types. For this reason, we adopt it as our target implementation language as it allows a straightforward route to the production of hardware.

A useful semantic feature of *Handel-C* is that channel communications and variable assignments take a clock cycle; and the longest combinatorial path of a single clock cycle sets the clock speed for a given piece of hardware. In modelling this, we delimit clock cycles in the CSP using the event *tock*.

D. Combining CSP and B: background and history

The B-method allows an engineer to describe abstract data types and operations: however it has no facilities for describing when an operation may be made available or blocked. Also, there is no notion of, or ability to reason about, concurrency. These facilities are offered by CSP—however, CSP is not very rich in terms of ability to describe abstract data types. Our motivation behind the development of CSP||B is that the integration of the two languages offers the benefits of both; moreover that it may offer facilities useful in a formal approach to hardware and software co-design.

An important consideration in integrating formal notations is that of tool support[12]. Our view is that an integration of notations should allow use of existing tools; this is now generally accepted to be valid in the formal methods community.

The tools which concern us most for CSP||B and in this paper are the model-checker for CSP FDR[9], and the development tool for B, B-Toolkit[4]. Existing results in literature on CSP||B have used these tools, and an important part of this paper is to demonstrate that this result extends to using CSP||B to develop hardware.

Other approaches to integrating CSP and B exist. These include [2], [8], and are not discussed in this paper. Several examples of using CSP||B have been published in literature, such as for instance [26], [6]—these examples have all been restricted to software developments. Previous works in the case study used in this paper—such as [5], contribute to modelling existing programs. This paper goes further in that it develops the implementation in reconfigurable hardware from an abstract specification that is not specific to hardware.

The opportunity to exploit large scale, and fine-grained, concurrency is one of the characteristic differences between software and hardware development. Previous work in CSP||B has not been driven by the exploitation of concurrency. Introducing, and verifying, concurrency in a reasoned manner is another major achievement of this paper.

II. Case study: an Internet packet filter

The case study upon which this paper is based is an Internet packet filter[12], [13]. It monitors traffic on a local Ethernet, flagging observations of predetermined source/destination address pairs. An important property of the device is it must not interfere with traffic of no concern to it: essentially, its presence should be effectively unobservable. This manifests itself in a number of ways. In particular it should not:

- block legitimate traffic;
- allow traffic to leave the filter before being inspected;
- constrain throughput unnecessarily;
- introduce inconsistent or unexpected delays.

Traffic is assumed to be transmitted as a byte-stream. The application should passively observe this stream, identify when it corresponds to an IP header, and investigate the addresses contained within. This is a non-trivial task: the stream is passing at a rapid rate, and the vast majority of data will be irrelevant. The device must be able to identify an IP header, perform necessary checksum calculations, extract addresses from the header, compare them to a dictionary, and return a result before it has left the stream; and this must be done with a minimum amount of interference to the stream.

A. Implementation architecture

The implementation design chosen consists of three processes. Firstly a process takes data from the network byte-wise, passes it through a pipeline large enough to contain an IP header, and returns data from the pipeline to the network.

Secondly, another process continually monitors this transitional state of the pipeline, checking on every shift if it corresponds to an IP header.

The third process is a lookup engine. When the pipeline corresponds to an IP header, the lookup engine should extract the source and destination addresses from the pipeline and compare them to a list of known banned addresses. If they exist in this list, it should flag this.

It is in this third component that some interesting requirements emerge. The speed at which the pipeline may be run is dictated both by the speed at which the presence of an IP header may be detected and the speed of the target network. Given that an IP header is 20 bytes long, if it is assumed that this detection never results in a false positive, then the time permissible for the addresses to be checked is the time taken for 20 pipeline shifts after the presence of an IP header is detected. This may be expressed in terms of wall-clock time, or in terms of clock ticks per shift.

This time has been calculated experimentally by measuring it on the FPGA. Therefore both an abstract specification and timing requirements may be given—both of which are crucial to a successful implementation. This paper is concerned only with the development of the lookup engine. This implementation is shown formally to be correct with respect to the abstract specification, and to meet the space and area requirements.

B. Content addressable memories

A Content Addressable Memory (CAM) is a device in which search operations are performed based on content, rather than on address. Retrieval of data is done by comparing a search term with the contents of the memory locations. If the contents of a location match the supplied data, a match is signalled. Typically, searching in a CAM can be performed in a time independent of the number of locations in the memory, which is where it differs to, for instance, a hash table.

Various CAM architectures, and associated speed/area cost trade-offs have been proposed[22], [15], [24]. Factors influencing the implementations chosen are the speed and area costs of the FPGA, and timing requirements of the application.

Conventional CAMs perform a *word parallel bit parallel* search on words in memory. While this offers the fastest lookups because of the fully parallel nature of the search, it has very high hardware costs due to the complexity of the comparator hardware.

The design adopted in this application is called a *Rotated ROM CAM* [13], figure 1. Each dictionary word has an associated comparator that iterates along the word, comparing relative positions with the search term. Its simplicity makes it an ideal CAM architecture to implement on an FPGA because it can be designed to be as wide as the number of words, and as deep as the width of the words, in the dictionary. The trade off is that while area costs reduce, the time to complete a search increases linearly with the length of the search term.

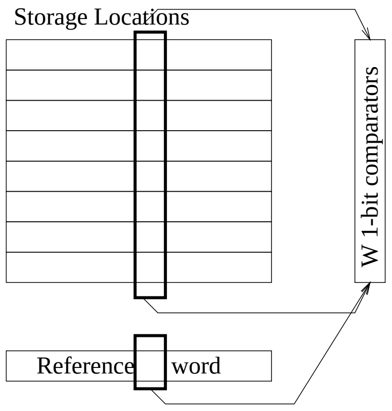

Fig. 1. A word parallel, bit serial CAM

Timing requirements are an important piece of information in choosing the CAM implementation. [13] demonstrates that the CAM is required to return a result within 20 *Handel-C* clock ticks of being loaded with the search term—this is due to the pipelining of the byte-stream through the application.

[13] further describes how the CAM can be built as ROMs of depth 16 bits, and width 2 bit, giving 32 bit words: each one of which corresponds to an IP address of interest. The search circuitry compares 2 bits at a time, meaning that 16 comparisons are required to compare the search term with a word in the dictionary. The circuitry assigns a value to a flag indicating whether a word matches the search term or not.

Experimentation with the FPGA gives the wall-clock time required for an IP header to pass through the pipeline, and the area and speed requirements of implementing a Rotated ROM CAM on the FPGA alongside the pipeline. The results of these experiments—figure 2 and figure 3—show that for the sizes of dictionary required, the Rotated ROM is feasible both in terms area costs, and timing relative to the pipeline and the latency required of a 100MBit Ethernet.

III. A CSP SPECIFICATION OF THE CAM

Definition 3.1: An abstract specification of a CAM

$$\begin{aligned}&\mathit{AbstractCam}(\mathit{dict}) \mathrel{\widehat{=}}\\ &\qquad \mathit{input}?\mathit{word} \rightarrow \mathit{output}!\mathit{word} \in \mathit{dict} \rightarrow \mathit{AbstractCam}(\mathit{dict})\end{aligned}$$

□

Definition 3.1 is the most abstract description of a CAM. Initially, the CAM is loaded with a dictionary. When a lookup is requested, the process outputs the value *true* if the word were in the dictionary, and *false* otherwise. Inherent in this specification is the first design decision specific to this application: the dictionary is fixed. Once the process *AbstractCam*

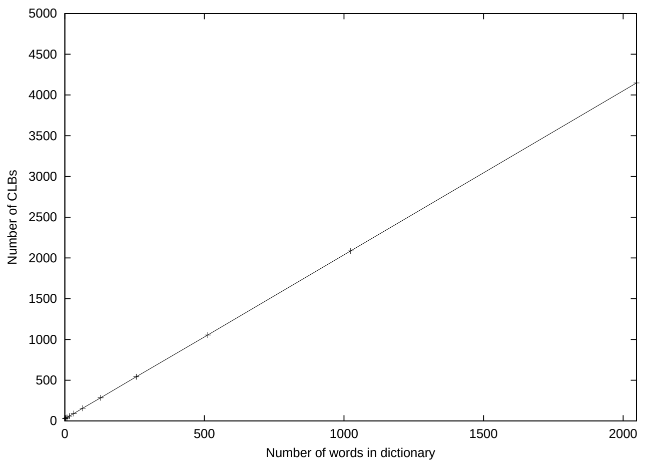

Fig. 2. Area costs of the Rotated ROM CAM on a Xilinx 40150 FPGA

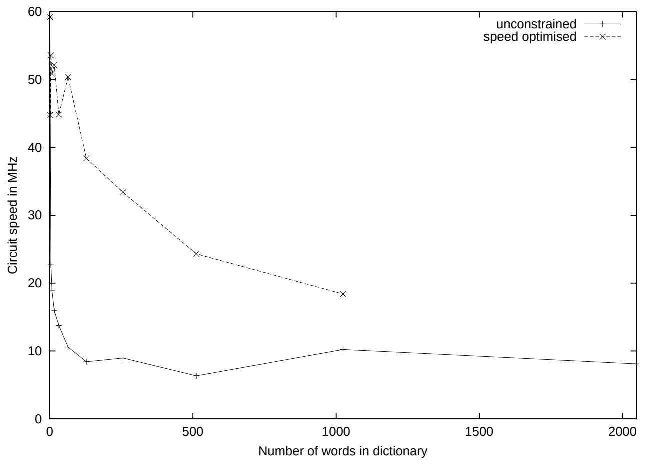

Fig. 3. Clock speeds of the Rotated ROM CAM on a Xilinx 40150 FPGA

has been started, there is no facility to either add or remove elements from the dictionary.

The set *Addr* contains all the IP addresses known: restricted to a small sample for model-checking. IP addresses are 32 bit words, represented as sequences of 16 2-bit pairs. This corresponds to how they will be laid out on the FPGA. Definitions are omitted from the paper to conserve space.

Definition 3.2 is the CSP for a CAM that reads in a pair of addresses and outputs a result. The difference between this and definition 3.1 is that the *Handel-C* clock is exposed.

This example contains two source/destination pairs represented as s_0, d_0 and s_1, d_1. The notation $s_X.i$ returns the i'th element of the sequence s_X. In the initial state in process *CamImp* the CAM can be loaded with search terms. This load instruction corresponds to a channel communication—which, by definition takes one clock cycle—so after loading, the clock *tock*s and control is passed to the process *Lookup*. Alternatively, some time may pass before the instruction is loaded.

Definition 3.2: An abstract CAM exploiting 20 clock ticks

$$
\begin{aligned}
&\mathit{CamImp} \mathrel{\widehat{=}} \\
&\quad in?src?dst \rightarrow tock \rightarrow \mathit{Lookup}(0, src, dst, true, true) \\
&\quad \Box \\
&\quad tock \rightarrow \mathit{CamImp} \\[1em]
&\mathit{Lookup}(i, src, dst, mm0, mm1) \mathrel{\widehat{=}} \\
&\quad i = 16 \;\&\; out!(mm0 \vee mm1) \rightarrow tock \rightarrow \mathit{CamImp} \\
&\quad \Box \\
&\quad 0 \leq i < 16 \;\&\; tock \rightarrow \\
&\qquad \mathit{Lookup}(i+1, tl\,src, tl\,dst, \\
&\qquad\qquad hd\,src = s_0.i \wedge hd\,dst = d_0.i \wedge mm0, \\
&\qquad\qquad hd\,src = s_1.i \wedge hd\,dst = d_1.i \wedge mm1\ )
\end{aligned}
$$

□

Once loaded, the CAM begins the *Lookup* operation. Here the next design decision appears—that the comparison is spread over multiple clock cycles. This decision is not guided by CSP||B or refinement, but by requirements and experiments, and must later be justified. On the first clock cycle of the lookup, the first two bits of the search words are compared against the first two bits of each word in the dictionary; if they are found not to match then this result is recorded. The recording of the result is an assignment and this consumes one clock cycle. Once all of the two-bit pairs have been compared the next clock cycle is used to output a result.

This specification is pure CSP—there is no mention of B, or of CSP||B. In the next section safety and liveness tests are presented that can be automatically checked using FDR, proving this specification offers the required functionality.

IV. Proving properties of the abstract specification

Definition 4.1: Functional correctness

$$
\begin{aligned}
&\mathit{InOutTest} \mathrel{\widehat{=}} \\
&\quad \Box\, src, dst : Addr \bullet \\
&\qquad in.src.dst \rightarrow (\Box\, m : \mathbb{B} \bullet out.m \rightarrow \mathit{InOutTest})
\end{aligned}
$$

$$\mid\ \textbf{assert}\ \mathit{InOutTest} \sqsubseteq_T \mathit{CamImp} \setminus \{\!|tock|\!\}$$

□

Definition 4.1 gives the functional correctness criteria of the system as the process *InOutTest*. This process states that if an address pair is loaded, the CAM eventually produces a result, and that only one result for every lookup is produced. Model-checking this assertion using FDR confirms that the refinement relation holds: therefore every possible behaviour of the implementation *CamImp* respects this.

Definition 4.2 enumerates the clock cycles permitted between loading a search term and outputting the result. In the interests of space the definition of *Tock*16 is omitted—it insists on repeating *tock* 16 times. 16 is significant as it is known that this is the number of cycles that the Rotated ROM design will take to look up a word. This, added to a clock cycle to load the CAM and a clock cycle to deliver a result, giving a total of 18 clock cycles is less than the 20 clock cycles taken by the header to pass through the pipeline. Model-checking with

FDR confirms that this assertion holds; and relative timing requirements are therefore met.

Definition 4.2: Functional correctness with timing

$$
\begin{array}{l}
InOutTockTest \mathrel{\widehat=} \\
\quad \Box\, src, dst : Addr \bullet \\
\qquad in.src.dst \rightarrow tock \rightarrow Tock16 \\
\qquad \Box \\
\qquad tock \rightarrow InOutTockTest
\end{array}
$$

| **assert** $InOutTockTest \sqsubseteq_T CamImp$

□

In deriving a suitable implementation, two considerations must be made. Firstly the CSP specification contains a lot of user state and conditions upon user state—it is preferable to place this into a B machine. Secondly, there is no notion of concurrency in the CSP specification—and the Rotated ROM design requires concurrency. In the next section a CSP||B model is developed that meets both these considerations.

V. Producing a CSP||B model

The first stage of producing an implementation is to turn the abstract specification into one where control is embedded in CSP, and state is embedded in B.

A. The CSP controller and B machine

Definition 5.1: The controller CSPBCamImp0

$$
\begin{array}{l}
CSPBCamImp0 \mathrel{\widehat=} \\
\quad setup!0!s_0!d_0 \rightarrow setup!1!s_1!d_1 \rightarrow tock \rightarrow GetInput \\
\\
GetInput \mathrel{\widehat=} \\
\quad in?src?dst \rightarrow tock \rightarrow RunCam(0, src, dst) \\
\quad \Box \\
\quad tock \rightarrow GetInput \\
\\
RunCam1(i, src, dst) \mathrel{\widehat=} \\
\quad i = 16 \;\&\; gt?mm \rightarrow out!mm \rightarrow tock \rightarrow GetInput \\
\quad \Box \\
\quad 0 \leq i < 16 \;\&\; comp!i!hd\,src!hd\,dst \rightarrow \\
\qquad\qquad\qquad tock \rightarrow RunCam(i+1, tl\,src, tl\,dst)
\end{array}
$$

□

Definition 5.1 gives the CSP control flow. Firstly the CAM is loaded with the dictionary before the clock has *tock*ed for the first time as it is hard-configured into the FPGA. As the dictionary is to be stored in the B machine, this is done via the event *setup*, which invokes the corresponding B operation with the relevant parameters. Once the clock has *tock*'ed the CAM is in a state where it is ready to perform searches. The loading of search terms is as in definition 3.2.

The process *RunCam* takes the search terms and an index variable *i*, initially 0. On the first 16 iterations of the loop it calls the B operation *comp* with the index and the heads of the two search terms. This communication takes a clock cycle, after which the loop continues with an increased index and the tail of the two sequences. On the final iteration, the control loop extracts the result from the B machine via the operation *gt*, which returns whether or not a match with the dictionary was recorded. This result is output to the environment on the channel *out*, consuming a clock cycle, and the CAM returns to its initial state. In this model, the events *comp* and *gt*, used to drive B operations, are hidden from external view.

Definition 5.2: The machine BCamImp0

```
MACHINE           BCamImp0
OPERATIONS

  setup(word,ss,dd) =
    PRE ...
    THEN
      src(word)   := ss    ||
      dst(word)   := dd    ||
      match(word) := TRUE
  END;

  comp(ii,src_pair,dst_pair) =
    PRE ...
    THEN
      IF( (not(src_pair=src(0)(ii)) or
           not(dst_pair=dst(0)(ii)) )
         & (not(src_pair=src(1)(ii)) or
            not(dst_pair=dst(1)(ii))) ) THEN
        match := [ FALSE,FALSE ]
      ELSE IF( not(src_pair=src(0)(ii)) or
               not(dst_pair=dst(0)(ii)) ) THEN
        match := [ FALSE,match(1) ]
      ELSE IF( not(src_pair=src(1)(ii)) or
               not(dst_pair=dst(1)(ii)) ) THEN
        match := [ match(0),FALSE ]
      ELSE skip
      END END END
  END;

  mm <-- gt =
  BEGIN
    match := [ TRUE,TRUE ] ||
    IF TRUE : ran(match) THEN
       mm := TRUE
    ELSE
       mm := FALSE END
  END

END
```

□

The corresponding B machine is given in definition 5.2. In this definition, the paragraphs `SEES`, `VARIABLES`, `INVARIANT` and `INITIALISATION` have all been omitted for space reasons: the reader may assume that they declare local variables, expected state invariants of the machine, and sensible initialisations. In addition to this, preconditions (`PRE`) relevant to each operation have also been omitted.

The machine contains three variables: an array of source addresses, an array of destination addresses, and an array of match lines. Elements with the same indices in each array form source/destination pairs with their corresponding match indicators. Definition 5.2 restricts this to two entries to match the CSP controller. An example invariant is that address pairs are not repeated: this makes sense in a hardware setting as the cost of building two identical ROMs is unnecessary.

The *comp* operation takes three parameters—an index and two pairs of bits. When called with valid parameters, (i.e, within its precondition) it records if the pairs of bits given match the pairs of bits stored at the indexed position in the dictionary. Initially, it is assumed they do, and when a match fails it is recored that they do not. Once a match has failed this is remembered until the match lines are reset. This task is done by the *gt* operation, which returns the disjunction of all the match lines in the CAM—meaning that if any one of the dictionary pairs were found to be a match when the comparisons were complete then the result is true. At the same time, it resets the match lines to *true*, ready for the next lookup operation to be performed.

This CSP controller and related B machine has gone some way to achieving the aims: all the state is embedded in the B. However, it has not captured all of the control that is desired. The decision had been taken that all lookups (the *comp* operation) would be performed concurrently in order to meet timing requirements in a Rotated ROM style, and this is not yet explicitly expressed. The next task therefore is to introduce this concurrency.

B. Separating dictionary words

The first step to implementing concurrent lookups is to rewrite the *comp* operation in the B machine as a new operation, *compare*, that operates on an individual word in the dictionary. The CSP controller may then also be rewritten in order to invoke the operation individually on each word.

The controller in definition 5.3 achieves this. An important detail is that the *compare* operations may be invoked in either order: either on the first word in the dictionary then the second, or the second then the first. Furthermore, even though it is invoked twice, both invocations are on the same clock cycle. Similarly, the *get* operations are invoked individually on each match line, and the controller calculates and outputs the result.

Definition 5.3: The controller CSPBCamImp1

```
$CSPBCamImp1 \mathrel{\widehat=} \ldots as\ before \ldots$
$GetInput \mathrel{\widehat=} \ldots as\ before \ldots$

$RunCam(i, src, dst) \mathrel{\widehat=}$
    $i = 16\ \&\ get.0?mm0 \rightarrow$
                $get.1?mm1 \rightarrow$
                    $out!(mm0 \vee mm1) \rightarrow$
                        $tock \rightarrow GetInput$
    $\Box$
    $i \geq 0 \wedge i < 16\ \&$
        $compare.0.i!hd\ src!hd\ dst \rightarrow$
            $compare.1.i!hd\ src!hd\ dst \rightarrow$
                $tock \rightarrow RunCam(i+1, tl\ src, tl\ dst)$
        $\Box$
        $compare.1.i!hd\ src!hd\ dst) \rightarrow$
            $compare.0.i!hd\ src!hd\ dst \rightarrow$
                $tock \rightarrow RunCam(i+1, tl\ src, tl\ dst)$
```

$\Box$

New operations required in the B machine are given in in definition 5.4. The *gt* and *comp* operations are rewritten as *get* and *compare*; and have an extra parameter identifying the particular word that is to be operated on.

Definition 5.4: New operations in the B machine

```
compare(word,index,src_pair,dst_pair) =
  PRE ...
  THEN
    IF( not(src_pair = src(word)(index)) or
        not(dst_pair = dst(word)(index)) )
   THEN
     match(word) := FALSE
   END
END;

mm <-- get(word) =
PRE word : 0..1
THEN
  mm          := match(word) ||
  match(word) := FALSE
END
```

$\Box$

C. Introducing concurrency

The final observation in the development is to notice that the *compare* and *get* operations are independent, and the order in which they are called is unimportant. That is, an invocation of $compare(0, \ldots)$ does not in any way affect an invocation of $compare(1, \ldots)$, and similarly for $get(0)$ and $get(1)$. A consequence of interleaving and choice laws of CSP, as in example 5.1 leads to the conclusion that they may be executed concurrently.

Example 5.1: Interleaving and choice

$$\begin{aligned} & a \rightarrow SKIP \mathbin{|||} b \rightarrow SKIP = \\ & \qquad a \rightarrow b \rightarrow SKIP \Box b \rightarrow a \rightarrow SKIP \end{aligned}$$

$\Box$

Definition 5.5: The process CSPBCamImp2

```
$CSPBCamImp2 \mathrel{\widehat=} \ldots as\ before \ldots$
$GetInput \mathrel{\widehat=} \ldots as\ before \ldots$

$RunCam(i, src, dst) \mathrel{\widehat=}$
    $i = 16\ \&\ get.0?mm0 \rightarrow$
                $get.1?mm1 \rightarrow$
                    $out!(mm0 \vee mm1) \rightarrow$
                        $tock \rightarrow GetInput$
    $\Box$
    $i \geq 0 \wedge i < 16\ \&$
        $(\ compare.0!i!hd\ src!hd\ dst \rightarrow tock \rightarrow SKIP$
           $[|\{|tock|\}|]$
           $compare.1!i!hd\ src!hd\ dst \rightarrow tock \rightarrow SKIP$
        $);\ RunCam(i+1, tl\ src, tl\ dst)$
```

$\Box$

The rewritten *RunCam* action is given in definition 5.5. A controller employing this version drives two B machines—indexed 0 and 1. This is reflected in the *setup*, *compare*, and *get* operations, where the first parameter now identifies the particular machine it is to operate on.

Particular attention is played to the role of the clock: both operations execute on the same clock cycle and the model must reflect this. Therefore after performing a *compare*, each concurrent branch must perform a *tock* and they must both synchronise on this *tock* as the clock is global.

The corresponding B machine is given in definition 5.6. The difference between this and earlier definitions is that the machine is only concerned with one word.

Definition 5.6: The machine *BCamImp*2

```
MACHINE          BCamImp2
OPERATIONS

  setup(ss,dd) =
    PRE ...
    THEN
      src:=ss || dst:=dd || match:=TRUE
  END;

  compare(index,src_pair,dst_pair) =
    PRE...
    THEN
     IF( not( src_pair = src(index) ) or
         not( dst_pair = dst(index) ) )
     THEN
       match := FALSE
     END
  END;

  mm <-- get = BEGIN mm := match END

END
```

□

This final model is now ready to be implemented in *Handel-C.* However the development steps have not yet been formally justified. In the next section, a proof technique for CSP||B is used to give this justification.

VI. Justifying the development

In [17], [18] a proof method for reasoning about the consistency of a CSP controller and a B machine is given. The term *consistent* means that a CSP invocation of an operation in the B machine respects the precondition of the operation. If this were not the case, the invocation would lead to a divergence in the system.

Example 6.1: A divergent assertion

$$Controller \mathrel{\widehat{=}} op \rightarrow Controller'$$

$$ControllerPlusAssertion \mathrel{\widehat{=}} \\ \quad \textbf{if}\ \text{pre}\ \ op\ \textbf{then}\ op \rightarrow Controller'\ \textbf{else}\ div$$

□

In order to prove this, a predicate called a *Control Loop Invariant* (CLI) is constructed. A CLI is a statement capturing information about all the possible states that the combination of the controller and the B machine can be in at recursive calls: generally, this statement excludes states that cannot be entered or do not exist. If this invariant is shown to be true of the combination, then the CSP controller may be augmented with a set of assertions capturing the pre and post conditions of the B machine operations derived from this invariant, and with enough state information from the B machine necessary to reason about those assertions. If, in a given state, the assertion holds, then behaviour is defined; otherwise it is divergent. The result is a pure CSP description of the combination, augmented with assertions that are known to be true. Example 6.1 is one such process with a divergent assertion injected.

Definition 6.1: The CLI for definition 5.1

$$CLI_{GetInput} \mathrel{\widehat{=}} \forall j : match(j) = true$$

$$\begin{aligned} &CLI_{RunCam} \mathrel{\widehat{=}} \\ &\quad \forall (j) : match(j) \Leftrightarrow \\ &\qquad 0 \le i \le 16 \\ &\qquad \wedge\ \ (\ \wedge_{k=0}^{i-1}\ \ s(k) = src(j)(k)\ ) \\ &\qquad \wedge\ \ (\ \wedge_{k=0}^{i-1}\ \ d(k) = dst(j)(k)\ ) \end{aligned}$$

□

The CLI for definition 5.1 is given in definition 6.1. The invariant has been spilt into two predicates: one for the process *GetInput* and one for the process *RunCam*. The complete invariant is the conjunction of the two.

$CLI_{GetInput}$ must be true when control enters the process *GetInput*: the requirement is that the match lines have been reset to *true*.

In CLI_{RunCam}, the variable *match* refers to the *match* in the machine *BCamImp*0, *i* refers to *i* in the CSP controller, *s* and *d* refer to the source and destination search terms in the controller and the indices extract specific bits, and *src* and *dst* refer to the source and destination addresses in the B machine. The invariant states several things: firstly, that the variable *i* ranges from 0 through to 16. The second and third clauses are concerned with building up the result of the lookup. They state that the value of the *match* variable for any given *j* is the conjunction of the truth of every test conducted to date. The role of this invariant therefore is to state that the CSP will always read a correct *match* result from the B machine.

Of course, if assertions based on this invariant are to be injected into the CSP, it must be shown to be true of the B machine. The general form of this proof obligation is given in definition 6.2, and state that the B machine invariant *Inv* in conjunction with the CLI allow the assumption of the truth of the CLI after the actions of the CSP controller.

Definition 6.2: Proving the CLI

$$Inv \wedge CLI \Rightarrow [\,CSP\ controller\,]\ CLI$$

□

The obligations relevant to definition 5.1 are given in definition 6.3. The first of these states that the process *GetInput* establishes one of two truths. It may establish the state necessary to execute *GetInput*—corresponding to the recursive call. Alternatively it establishes the state necessary to execute *RunCam*—corresponding to reading in an input and setting the required state variables.

Definition 6.3: The proof obligations for CSPB0

$$Inv \wedge CLI_{GetInput} \Rightarrow [\{GetInput1\}]\, CLI_{GetInput}$$
$$Inv \wedge CLI_{GetInput} \Rightarrow [\{GetInput2\}]\, CLI_{RunCam}$$
$$Inv \wedge CLI_{RunCam} \Rightarrow [\{RunCam1\}]\, CLI_{GetInput}$$
$$Inv \wedge CLI_{RunCam} \Rightarrow [\{RunCam2\}]\, CLI_{RunCam}$$

□

A proof obligation such as this is known as a *weakest precondition (wp)* proof. The notation $[S]P$ is a predicate P describing a set of states that may be reached after the execution of S—i.e, P is a postcondition of S. Therefore $[S]$ is a predicate which is true of any initial state where S achieves P—it is the weakest precondition.

A description of how a *wp* proof is carried out is given in [20]. In order to carry out this proof, the CSP controller must be translated into the control expected in a B machine. This process is introduced and justified in [17], [18]. The result of this translation for the process *RunCam* definition 5.1 is given in definition 6.4. The details of the *wp* proof for the invariants of definition 6.4 are omitted from this paper as it is routine using a theorem prover for B, such as B-Toolkit.

Definition 6.4: The B description of *CSPB*0*RunCam*

```
{RunCam1} =
  SELECT i=16 THEN
    IF member(true,ran(match)) THEN
      mm:= true
    ELSE
      mm:= false
    END
  END

{RunCam2} =
  SELECT 0<= i < 16 THEN
    PRE src,dst: BitPair, 0<=ii<=16
    IF (not(src_pair=src(0)(ii)) or
        not(dst_pair=dst(0)(ii)) )
         & (not(src_pair=src(1)(ii)) or
            not(dst_pair=dst(1)(ii)))
    THEN match := [FALSE,FALSE]
    ELSE
    IF not(src_pair=src(0)(ii)) or
       not(dst_pair=dst(0)(ii))
    THEN match := [match(0), FALSE]
    ELSE
    IF not(src_pair=src(1)(ii)) or
       not(dst_pair=dst(1)(ii))
    THEN match := [FALSE, match(1)]
    ELSE
      skip
    END END END
    i:= i+1 || s:= tail(s) || d:= tail(d)
  END
```

□

The CLI presented in this section is also relevant to the second level of development. Some changes are necessary for the third level: invariants are needed for each B machine: although the same invariant is sufficient for both. For space reasons, we omit their definitions from this paper, although they may be easily reconstructed from definition 6.3.

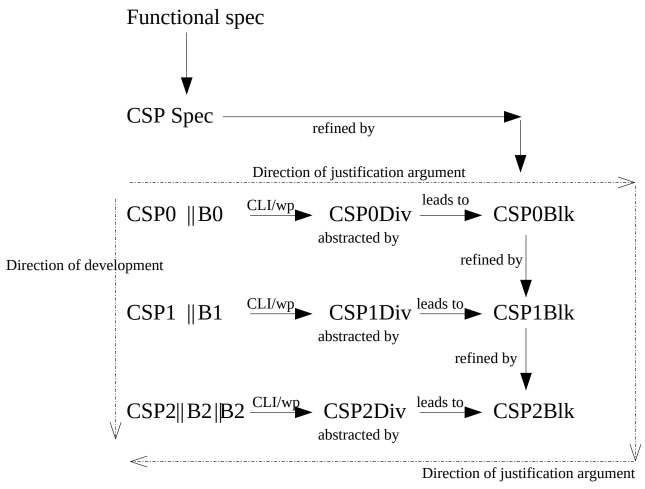

Fig. 4. Justifying the development

A. Automated analysis and refinement checking using blocking assertions

Finally, once the invariants of the B machine have been shown to be true, the assertions in the CSP controller may be rewritten as *blocking assertions*. When a blocking assertion does not hold, behaviour is the deadlock process *STOP*.

Example 6.2: A blocking assertion

$$ControllerBlk(state) \mathrel{\widehat{=}}$$
$$\quad \textbf{if}\ \text{pre}\ op\ \textbf{then}\ op \rightarrow Controller'\ \textbf{else}\ STOP$$

□

As it has been proven that the assertion can be relied upon to be true given the CLI, then the only behaviours possible of the process containing the blocking assertions are those when the assertion is true: the deadlocks introduced can not occur.

Therefore, it can be assumed that the observable behaviours of the process containing the blocking assertion will be (at least) those that can be observed by the real controller in parallel with the B machine. An important consequence of this, upon which our development process relies, is that this blocking process can now be used for analysis purposes. For instance, if the blocking version of the process can be proven to only exhibit the safe behaviours described in section IV, then it is certainly the case that the CSP||B combination will also. Such a verification can be performed automatically on the blocking process using FDR.

This is the principle upon which the development of the CAM relies. The complete procedure is depicted in figure 4. In this diagram, the arrows labelled `CLI/wp` depict the CLI proof obligations necessary to permit usage of the CSP process containing the diverging assertions. The arrows labelled `leads to` show where the versions of the processes containing blocking assertions may be constructed from these processes. The arrows labelled `refined by` show refinements that can be model-checked using FDR, linking (vertically) each phase of development (horizontal line) with the next. The labels `abstracted by` show there is a refinement relation on each horizontal level of the diagram: the blocking process on the right of any given line is an abstraction of the original controller/machine combination on the left. Overall, the diagram shows how each process and proof obligation fit into the lattice in building high assurance arguments in allowing development to go from the top left to the bottom left. The desired direction of development, and the direction of proof needed to justify it, are depicted with labelled arrows.

The final task is to show that this combination is a valid refinement of the original CSP specification of definition 3.1. Of course, it is the case that one cannot demonstrate a refinement between *just* the process $CSP0$ and the CSP specification, as there are behaviours possible of $CSP0$ that are not possible of the specification due to the presence of the B machine and the behaviours it restricts.

The CLI consistency checking technique does provide assurances that the *combination* of the controller and the B machine is a refinement of the blocking assertion version of the process, as in example 6.3. Transitivity of refinement therefore promises that as it can be shown that the blocking version of the process respects the functional requirements, then any refinement of it—including the CSP$\|$B combination—is a valid implementation.

Example 6.3: Refinement: controllers and assertions

$$\mathit{CSPBlk} \sqsubseteq_{FD} (\mathit{CSP}0 \mid\mid \mathit{BMachine}0)$$

□

A similar policy holds for the introduction of the next stage of development. There is no refinement relation that exists between the B machine $B1$ and $B0$. However, the state and operations in the B are internal to the system and therefore unobservable themselves—in the blocking version of the process there is no mention of these. Using the validity of the refinement of the combination, if it can be shown that the blocking version of $\mathit{CSP}1\mathit{Blk}$ is a valid refinement of $\mathit{CSP}0\mathit{Blk}$, then any implementation of $\mathit{CSP}1\mathit{Blk}$ is also acceptable. The assertion $\mathit{CSPBlk}0 \sqsubseteq_{fd} \mathit{CSPBlk}1$ captures this. Proving that this assertion holds demonstrates that the combination of the controller $\mathit{CSP}1$ and the B machine $B1$ is also an acceptable implementation—even though no direct refinement relation exists between it and the controller and B machine of the previous development step.

This principle extends to showing the validity of the third development phase where concurrency is introduced. Once all proof obligations concerning the Control Loop Invariants have been discharged, it is safe to conclude that this is an acceptable implementation.

Figure 4 depicts an interesting point here. Development, and the decisions made during the development process, were all taken based on knowledge of processes on the left hand side of the diagram—even thought there was no hope of *directly* demonstrating the correctness of the design decision. Instead, these decisions were justified by proving the consistency of each combination individually, and relying upon refinement of the CSP blocking processes. This gives rise to the tree-like refinement structure shown in figure 5.

$$\begin{tikzcd}
 & \text{Functional properties} \arrow[d] \\
 & \text{Abstract CSP spec} \arrow[d] \\
\text{CSP0 || B0} & \text{CSP0Blk} \arrow[l] \arrow[d] \\
\text{CSP1 || B1} & \text{CSP1Blk} \arrow[l] \arrow[d] \\
\text{CSP2 || B2} & \text{CSP2Blk} \arrow[l]
\end{tikzcd}$$

Fig. 5. The refinements as a tree

VII. Conclusions and discussion

This paper has demonstrated the use of CSP$\|$B for reconfigurable hardware development; as well as proof techniques that justify the development and give high levels of assurance about the behaviour of the implementation.

An important point in this case study was that the target implementation was known in advance, and the aim was to justify this implementation. This will often, but not always, be the case in reality. Engineers may well know particular aspects of the application domain that they wish to exploit, and a design methodology should support this. However, it may be the case that they do not, and in that eventuality the design methodology should not restrict freedom of expression and experimentation. Most importantly, the design methodology must give engineers ability to argue the correctness of an implementation with respect to the specification in either case.

Section III and section IV showed how an abstract specification (that may consist only of CSP) in conjunction with FDR can be used to verify behavioural properties of a design. This is not a new, or even surprising, result—however it is one that is reasonable to expect of a formal design methodology based on CSP. The aims of this paper stated that it was a desirable property of CSP$\|$B to allow results individual to either CSP or B, and this result demonstrates this.

The development process consisted of three distinct steps: moving state into a B machine, separating that state, and introducing concurrency. Each step was based on design intuition gleaned from empirical experimentation results. The final step in this development is a small syntactic step away from *Handel-C* code and consequently implementation on an FPGA. This step has not been formally described in this paper although related works producing *Handel-C* from CSP in [12],

[11] show a promising approach to this. Further work may be needed to complete the formalism of this step.

Another useful item of further work is to look at whether or not the division between CSP and B may help in the partitioning between hardware and software. For instance, the CSP may form a *Handel-C* hardware network of processes, while the B may be implemented as software running in small custom processors. The natural separation of control flow from abstract data types is a promising approach in thinking about how designs may be split into hardware and software: concurrency issues are natural in hardware whilst complex operations on abstract data types may be more natural in software. Further consideration of this issue may be a fruitful area for further research.

Justification of the development proved interesting: it was not directly possible to prove refinements between each stage of development: that is, between each CSP controller and B machine combination. Furthermore, it was not possible to assert *directly* that the most abstract CSP||B model respected the functional specification.

However, using the technique of verifying the consistency of the combination with Control Loop Invariants, CSP process capturing the behaviours of the CSP||B models were produced. The CLI technique guarantees that the blocking process is an abstraction of the combination; and as it is purely CSP, FDR can be used for analysis. The result was that the functional correctness of the abstract CSP||B model, and each level of development, could be asserted by using the model-checker FDR, and relying on transitive properties of refinement.

A further issue is apparent from the second level of development. The B machine was split into two, with each machine operating on a single dictionary entry. Unlike Z, B does not have a schema calculus for justifying this division of an abstract data type. The problem, therefore, is that there is no formal support for a design decision such as this—even though it was possible to prove the correctness of the overall design step. Additional facilities in the form of a Z-like schema calculus, such as those in [28] may well be helpful—or indeed necessary—for developments such as this one where the aim is to introduce concurrency into a controller. This is another item that we leave for future work.

The central conclusion from this paper is that CSP||B forms the basis of a valid technology for the exploration and development of high assurance hardware and software systems. Further research is to investigate co-design, to understand how a design calculus may be incorporated, and how automatic tool support may be provided in the discharging of proofs.

Acknowledgement

The authors would like to thank AWE for their financial support in this work.

References

[1] J. R. Abrial. *The B–Book, Assigning programs to meanings*. Cambridge University Press, 1996.

[2] Michael J. Butler. *CSP2B: A practical approach to combining CSP and B*. Formal Aspects of Computing, 12:182–198, 2000.

[3] Celoxica. *Handel-C and DK3*. www.celoxica.com.

[4] B-Core (UK) Ltd. *The B-Toolkit*. www.b-core.com/btoolkit.html.

[5] J. C. P Woodcock and Alistair A. McEwan. *An overview of the verification of a Handel-C program*. Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, CSREA Press, 2000.

[6] N. Evans and H. Treharne. *Investigating a file transfer protocol using CSP and B*. Technical report, Department of Computer Science, Royal Holloway, Univeristy of London, 2005.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. International Series in Computer Science. Prentice-Hall, 1985.

[8] Michael Leuschel and Michael J. Butler. *ProB: a model-checker for B*. Proceedings of FME03, Lecture Notes in Computer Science, pages 855–874, Springer-Verlag, 2003.

[9] Formal Systems (Europe) Ltd. *FDR*. www.fsel.com.

[10] INMOS Ltd. *occam Programming manual*. International Series In Computer Science, Prentice-Hall, 1984.

[11] Alistair A. McEwan. A calculated implementation of a control system. Communicating Process Architectures 2004, pages 265–280, IOS Press 2004.

[12] Alistair A. McEwan. *Concurrent Program Development*. DPhil thesis, The University of Oxford, Sumbitted Michaelmas term 2006.

[13] Alistair A. McEwan and Jonathan Saul. *A high speed reconfigurable firewall based on parameterizable FPGA-based content addressable memories*. The Journal of Supercomputing, 19(1):93–105, May 2001.

[14] Carroll Morgan. *Programming from Specifications*. International Series in Computer Science, Prentice-Hall, 1990.

[15] Behrooz Parhami. *Architectural tradeoffs in the design of VLSI-based associative memories*. Journal of Microprocessing and Microprogramming, 38:27–41, 1993.

[16] A. W. Roscoe. *The theory and practice of concurrency*. Prentice Hall Series in Computer Science. Prentice Hall, 1998.

[17] H. E. Treharne and S. A. Schneider. *Communicating B Machines*. ZB2002: International Conference of Z and B Users, Lecture Notes in Computer Science, page 2272, Springer-Verlag, 2002.

[18] S. A. Schneider and H. E. Treharne. *Verifying controlled components*. Proceedings of IFM 04, Lecture Notes in Computer Science, page 2999, Springer-Verlag, 2004.

[19] Steve Schneider. *Concurrent and Real-time Systems The CSP Approach*. Worldwide Series in Computer Science, Wiley, 2000.

[20] Steve Schneider. *the b-method*. Cornerstones of Computing, Palgrave, 2001.

[21] Steve Schneider and Helen Treharne. *Communicating B Machines*. ZB2002: Formal Specification and Development in Z and B, Lecture Notes in Computer Science, pages 416–435, Springer-Verlag.

[22] Kenneth J. Schultz and P. Glenn Gulak. *Architectures for large capacity CAMs*. INTEGRATION, the VLSI Journal, 18:151–171, 1995.

[23] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 2nd edition, 1992.

[24] Djamshid Tavangarian. *Flag-oriented parallel associative architectures and applications*. IEEE Proceedings, volume 136:5, pages 357–365, 1989.

[25] H. E. Treharne and S. A. Schneider. *Using a process algebra to control B operations*. Proceedings of IFM 1999, Lecture Notes in Computer Science, Springer-Verlag.

[26] H. E. Treharne, S. A. Schneider, and S. A. Bramble. *Combining specification with composition*. ZB2003: International Conference of Z and B Users, Lecture Notes in Computer Science, Springer-Verlag, 2003.

[27] Helen Treharne and Steve Schneider. *How to drive a B machine*. ZB2000: Formal Specification and Development in Z and B, Lecture Notes in Computer Science, pages 188–208, Springer-Verlag, 2000.

[28] J. C. P. Woodcock and A. L. C. Cavalcanti. *A concurrent language for refinement*. 5th Irish Workshop on Formal Methods, 2001.

[29] Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof*. International Series in Computer Science, Prentice-Hall, 1996.