Communicating Process Architectures 2006
Peter Welch, Jon Kerridge, and Fred Barnes (Eds.)
IOS Press, 2006

A Circus Development and Verification of an Internet Packet Filter

Alistair A. McEWAN

Department of Computing, University of Surrey,
Guildford, Surrey, GU2 7XH, UK.

a.mcewan@surrey.ac.uk

Abstract. In this paper, we present the results of a significant and large case study in *Circus*. Development is top-down—from a sequential abstract specification about which safety properties can be verified, to a highly concurrent implementation on a Field Programmable Gate Array. Development steps involve applying laws of *Circus* allowing for the refinement of specifications; confidence in the correctness of the development is achieved through the applicability of the laws applied; proof obligations are discharged using the model-checker for CSP, FDR, and the theorem prover for Z, Z/Eves. An interesting feature of this case study is that the design of the implementation is guided by domain knowledge of the application—the application of this domain knowledge is supported by, rather than constrained by the calculus. The design is not what would have been expected had the calculus been applied without this domain knowledge. Verification highlights a curious error made in early versions of the implementation that were not detected by testing.

Keywords. Circus, Development, Verification, Reconfigurable hardware, Handel-C

Introduction

In this paper a case study where a security device is specified, designed, and implemented, is investigated. The interesting aspect of this case study is that the device is to be implemented in hardware, and some of the design requirements are dependent on the hardware on which it deployed. The development of the device is guided by the laws of *Circus*, and proof obligations for development steps centre around proving the correctness of the resulting refinement on each major development phase. The case study in question is an Internet packet filter [3,6], a device which sits on a network, monitoring traffic passing through it, and watching for illegal traffic on the network. Typically, these devices can be employed to monitor, route, or prevent traffic on networks: in all of these cases, but particularly in the case of prevention, confidence in the correctness of the implementation is necessary if network security is to be assured. The major contributions of this paper can be summarised by the following points:

1. The presentation of a top-down design strategy from a set of requirements and a verification strategy for such a development using Z/Eves and FDR.
2. A demonstration of the calculation of concurrency from a sequential specification using laws of *Circus* presented in [5], including the generalised chaining operator.
3. The presentation of design patterns for refining *Circus* processes into *Handel-C* models, incorporating a model of synchronous clock timing.
4. Evidence that laws of *Circus* allow the exploration of refinements, guided by engineering intuition where requirements may not have been explicit in the specification.

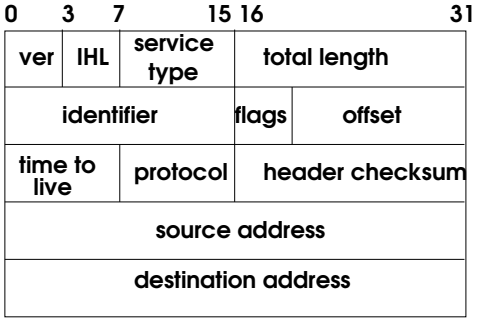

Figure 1. An IP v4 packet header

Section 1 presents the problem domain, and some relevant background material. This is followed by a formalisation of requirements and an abstract specification in section 2. In section 3 through to section 6, different components of the implementation are developed and verified. Section 7 presents the composition of this development as a final *Handel-C* model. Finally, in section 8, some conclusions are drawn.

1. Background

1.1. Packet Filters

An Internet packet filter is an application that monitors network traffic, and performs actions based on criteria concerning that traffic. In this case study, the packet filter monitors traffic on a local section of Ethernet, flagging observations of predetermined source/destination address pairs. An important property of a monitoring device such as this one is it must not interfere with traffic of no concern to it: essentially, its presence should be effectively unobservable unless it is required to take action.

The packet filter assumes traffic is transmitted using the Internet protocol (IP), version 4 [2,9]. In IP v4, a packet consists of a *header*, and a *payload*. The header contains accounting information, whilst the payload contains the information itself. For instance, if a user were accessing a web page, the header would contain information such as their machine address, the address of the web server, and the size of the payload; while the payload would contain (parts of) the web page itself. The structure of an IP v4 packet header is given in figure 1.

Traffic is assumed to be transmitted as a byte-stream. The application should passively observe this byte-stream, identify when sections correspond to a packet 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 the stream will be payload data. The device must be able to identity a packet header, including performing necessary checksum calculations, extract source and destination address from the header, compare it to a dictionary of known addresses, and return the result of this comparison before the full header has passed through the stream; and this must be done with the minimum amount of interference to the stream.

1.2. Identifying Packets

Packet headers are identified in the stream by performing a checksum calculation, which should equal 0 in ones complement, checking the IP version number, which should equal 4, checking the least significant bit, which should always be 0, and checking the protocol number, which in this case should be 6, representing TCP/IP. When IP packets are identified, they are further examined to identify whether or not the packet requires action by the filter. In this case study the filter does not attempt to prevent passage—it simply observes and acknowledges source/destination address pairs predetermined to be of interest.

1.3. Motivation and Related Works

Previous works exist describing how a filter such as this can be implemented on a standard FPGA in a network environment [3,6]. These works explain the architecture of the packet filter and propose techniques, implementable in hardware, which allow the identification of packets and the lookup of addresses in real network time—a time that is no longer than it takes for the network to communicate the traffic. Experimental measurements are presented justifying the design and implementation on a small commodity FPGA in 100M-Bit Ethernet.

Attempts to verify the correctness of the implementation exist in [12,4]. These works use the model of the Unifying Theory to model the existing program. The model uses both Z and CSP constructs, and whilst their combination is justified in the Unifying Theory, they are not combined in a syntactic way to give a structured model in the way that *Circus* does. Nevertheless, these works show that such a model is feasible in modelling the implementation if certain problems can be overcome—for instance, infinite trace generation arising from modelling clock ticks.

These works all contribute to modelling existing programs. This paper goes further in showing that with an understanding of *Handel-C* and the FPGA, an abstract specification can be developed into a concurrent implementation. The structure of the *Circus* specification combined with laws for calculating concurrency allow for the structured verified development of the implementation.

1.4. Content Addressable Memories

A Content Addressable Memory (CAM), also known as an *associative memory*, is a device in which search operations are performed based on content rather than on address. Retrieval of data in a CAM 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 [11,8].

1.5. Building a CAM on an FPGA

A CAM needs the following:

- Storage for data
- Circuitry to compare the search term with the terms in memory
- A mechanism to deliver the results of comparisons
- A mechanism to add and delete the data in memory if the dictionary is not fixed

Conventional CAMs require circuitry to 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. [6] shows how the packet filter application benefits from content based lookups, as fast, constant time lookups are required over an arbitrary data set in the form of a pipeline data stream. The design of CAM adopted is called a *Rotated ROM CAM* [6,3], shown in figure 2. Each dictionary word stored has an associated comparator, and this comparator iterates along the word, comparing the relative positions in the dictionary with the search term word. Its simplicity makes it an ideal CAM architecture to implement on an FPGA because the reconfigurable nature of the FPGA means that the ROM can be designed to be as wide as the number of words in the CAM dictionary and as deep as the width of the words in the dictionary. This design is chosen as it exploits the architecture of the FPGA, allowing a trade-off between hardware costs and the speed of lookups. The trade-off is in the way that the words in memory are not searched in a fully parallel manner,

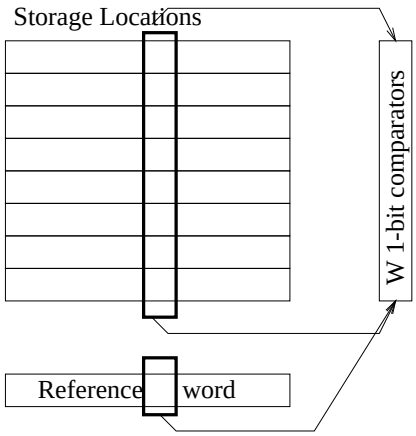

Figure 2. A word parallel, bit serial CAM

thus reducing the amount of hardware required to search the dictionary, bringing with it an increase in the time it takes to search the dictionary.

1.6. Implementation Architecture

The application consists of a number of discrete systems. These are:

- A feeder process taking bytes from the stream and passing them to the packet detector
- The packet detector
- The search engine
- A process to output results

The packet detector is the process responsible for monitoring the data passed to the application, and signalling when a subset of this data constitutes an IP packet. When a packet is detected, the search engine decides if it is a packet of interest by comparing the source and destination addresses with those stored in the CAM.

1.7. The Approach to Verification

In this paper, many laws of *Circus* are used—for instance to manipulate actions into a form where they may be split. Each law typically has a proof obligation associated with it. Discharging of all these proof obligations by hand in a large development is infeasible: there may be many of them. The technique adopted in this case study is much more pragmatic, and is intended to work more as a realistic industrial development. Laws of *Circus* are repeatedly applied in order to manipulate the actions into the required form. Instead of proving the application of each law, the proof is to show that the final result is a valid refinement of the starting point. This proof obligation is met by model-checking. The CSP_M used in verification for each stage may be easily reconstructed.

In order to alleviate the problem of state space explosion in FDR, some simplifications have been employed in the CSP_M. Firstly, only three addresses exist. Two of these are in the CAM—meaning that concurrent lookups are required. The third address is not in the CAM, meaning that there is an address not of interest. Secondly, the function $addr$ that returns the address (pairs) in a packet reads the value of many pipeline cells[1], but in the CSP_M it only looks at one. Thirdly, each cell may only have a value in the range $0..3$. Fourthly, the checksum calculation is simplified: instead of being a predicate over a number of cells,

[1]The pipeline is the component of the application that stores data read in from the network, and consists of a series of individual cells. As an analogy, the pipeline may be thought of as an array, and a cell is analogous to an element in the array.

it only considers the first cell in the pipeline. A data refinement could be proved between this simplification and the real data types implemented; however this is not necessary as the CSP_M verification is really concerned with the action structure and not the data values. Other investigations may require this.

The CSP_M model grows very large when the pipeline is greater than five elements. This has an impact on the CAM lookup, which should take 16 cycles. The dictionary therefore has been carefully chosen such that the words are all distinguishable in the first few bits, so correct results are known long before 16 comparisons have been made. In this way, the requirement that the CAM can output a correct result before a packet has left a (shorter) pipeline may still be met.

Some laws require manipulation of user state. This user state, and operations on it, are encapsulated by Z schemas. Where there are proof obligations associated with schema and abstract data type manipulation, these are discharged using Z/Eves. The source input for Z/Eves is in-lined LaTeX in this paper. In some cases, this LaTeX has been manipulated into a more readable form for presentation purposes—for instance in-line schemas— that requires to be re-written back as full schemas before it can be re-run in Z/Eves. For Z theorems, the tactics needed to instruct Z/Eves to complete the proof are listed in the associated proofs.

The complete development, including all of steps and the tool applied to proof obligations at each step, is given graphically in figure 3, figure 4 and figure 9. The first diagram represents the decomposition of the abstract specification into the three major components. The second diagram represents the decomposition into a chained pipeline and a CAM with sequentialised lookups. The third represents the refinement of this model into a clocked *Handel-C* implementation.

2. An Abstract Specification

In this section an abstract sequential specification of the packet filter is presented. The packet filter reads in a set of bytes from the network; looking to spot addresses. At this stage of development it is not necessary know detail about the representation of these, so they are specified as given sets.

Definition 1 *Given sets and axiomatic definitions :*

$$[Byte, Addr, BitPair]$$
$$RESULT ::= yes \mid no$$

$$\begin{array}{|l} dictsize : \mathbb{N} \\ pipesize : \mathbb{N} \\ addr : (\mathbb{N} \pfun Byte) \pfun (Addr \times Addr) \\ chk : (\mathbb{N} \pfun Byte) \fun \mathbb{B} \\ last : (\mathbb{N} \pfun Byte) \pfun Byte \\ \hline dictsize \geq 1 \\ pipesize = 20 \end{array}$$

□

Addresses of interest are to be stored in a dictionary, while the bytes currently being examined will be stored in a pipeline. At this stage of development it is not yet known how many addresses will be stored in the dictionary when it is finally deployed, so this constant is left loose. An IP header is 20 bytes long: this constant is the length of the pipeline. [2]

[2] The development could have started with a more abstract description of the problem, with a data refinement between it and the concrete. However in this paper the concern is developing the concrete data model into a concurrent implementation.

The partial function $addr$ takes a sequence of bytes and returns a pair of addresses, while the function chk takes a sequence of bytes and returns a boolean result. The former will later be used to extract addresses from a packet, and the latter to identify a packet. The function $last$ takes a sequence of bytes and returns the element at the end of the sequence. The precise definitions are omitted for space.

Definition 2 *The abstract packet filter :*

$$\begin{array}{l}
\mathbf{process}\, Filter \mathrel{\widehat=} \mathbf{begin} \\
\quad USt \mathrel{\widehat=} [\ pipe : \mathbb{N} \pfun Byte;\ dict : \mathbb{N} \pfun (Addr \times Addr) \mid \\
\qquad\qquad \operatorname{dom} pipe = 1..pipesize \land \operatorname{dom} dict = 1..dictsize\ ]
\end{array}$$

UpdateAll

$$\begin{array}{l}
\Delta USt \\
b? : Byte \\
\hline
pipe' = \{x : 1..pipesize - 1;\ y : Byte \mid \\
\qquad x \mapsto y \in pipe \land x + 1 \in \operatorname{dom} pipe \bullet x + 1 \mapsto y\} \cup \{1 \mapsto b?\} \\
dict' = dict
\end{array}$$

$$\begin{array}{l}
Run \mathrel{\widehat=} \mathbf{var}\ c : \mathbb{N};\ h : (Addr \times Addr);\ \mu X \bullet \\
\quad (in?b \rightarrow SKIP \interleave out!last(pipe) \rightarrow SKIP); \\
\qquad \mathbf{if} \neg\ chk(pipe)\ \mathbf{then}\ UpdateAll;\ X \\
\qquad \mathbf{else}\,(\ h := addr(pipe);\ c := 1;\ \mu Y \bullet UpdateAll; \\
\qquad\quad \mathbf{if}\ c < pipesize - 1\ \mathbf{then}\,(in?b \rightarrow \interleave out!last(pipe) \rightarrow SKIP);\ c := c + 1;\ Y \\
\qquad\quad \mathbf{else}\,(in?b \rightarrow SKIP \interleave out!last(pipe) \rightarrow SKIP \interleave match!(h \in dict)) \rightarrow SKIP);\ X\ ) \\
\bullet\ Run \\
\mathbf{end}
\end{array}$$

□

The local state of the abstract system, given in definition 2, has two components: a pipeline and a dictionary. As the pipeline will be built in hardware, the state invariant does not allow its size to ever change. The size of the dictionary is also constant. Unusually, an initialisation operation has not been specified for these. In the case of the pipeline, this is because initially the values are meaningless: when hardware is powered up, registers have an arbitrary value. In the case of the dictionary, this is because the specification is purposefully being left loose with regards to the addresses of interest. When the final implementation is deployed, these would be given.[3]

An operation to update the local state exists. This operation takes the state of the pipeline, and a new byte as input. It adds the new byte to the head of the pipeline, and drops the last byte in the pipeline. The dictionary remains unchanged. There are no outputs.

The main action of the process reads an input into the local variable b and outputs the last element in the pipeline. Then the predicate chk returns $true$ or $false$ indicating whether or not it believes the pipeline to correspond to a packet header. If not, the new data is stored and shifted. If it does then the address pair in the pipeline is recorded in the local variable h before the shift. Another $pipesize - 2$ shift cycles are permitted before a result is output on the channel $match$ indicating whether or not the addresses were known to the dictionary. The condition on $pipesize$ ensures that the result is know to the environment before the data has fully left the pipeline.

[3]Where, and when these were given would depend upon the target implementation environment. For instance, for a purpose built hardware CAM, they could be included in the state invariant; whilst for a software implementation such as a hash table, it may be done by adding a new operation to initialise the dictionary variable.

3. Refining the Abstraction into a Pipeline and a Checksum

3.1. Process Splitting

A technique for introducing concurrency into a *Circus* specification is *process splitting*, discussed in [10] and built on in [5]. If the process paragraphs are disjoint—i.e, the actions of the process each access different components of the state—then the process may be split in two with respect to those disjoint actions and state. Generally processes must be manipulated into an appropriate form.

Let pd stand for the process declaration below, where $Q.pps$ and $R.pps$ stand for the process paragraphs of of the processes P and Q; and F for an arbitrary context (a function on processes). The operator $\uparrow$ takes a set of process paragraphs P and schema expressions Q, and joins the process paragraphs of P with ΞQ. For the expression to be well-formed, the paragraphs in P must not change any state in Q. This is the general form of processes to which process splitting laws apply.

$$
\begin{array}{l}
\mathbf{process}\, P \mathrel{\widehat=} \\
\mathbf{begin} \\
\quad State \mathrel{\widehat=} Q.st \land R.st \\
\quad Q.pps \uparrow R.st \\
\quad R.pps \uparrow Q.st \\
\quad \bullet\ F(Q.act, R.act) \\
\mathbf{end}
\end{array}
$$

The state of P is defined as the conjunction of two other state schemas $Q.st$ and $R.st$. The actions of P are $Q.pps \uparrow R.st$ and $R.pps \uparrow Q.st$. They must handle the partitions of the state separately. In $Q.pps \uparrow R.st$ each schema expression in $Q.pps$ is conjoined with $\Xi R.st$. Similar comments apply to $R.pps \uparrow Q.st$.

Law 1 *Process splitting*

$$pd = (\mathbf{process} P \mathrel{\widehat=} F(Q.act, R.act))$$

provided Q.pps and R.pps are disjoint sets of paragraphs with respect to R.st and Q.st. □

Two sets of process paragraphs pps and pps' are said to be disjoint with respect to states s and s' if and only if $pps = pps' \uparrow s'$ and $pps' \uparrow s$, and no action expression in pps refers to components of s' or to paragraph names in pps'; further, no action in pps' refers to components of s or to paragraph names in pps.

The development is aimed at producing an implementation using the Rotated ROM CAM, thus meeting area and speed requirements on the FPGA. The abstract specification must be split into three components: a pipeline, a CAM, and a checksum calculation. Each of these can then further be refined into their respective *Handel-C* implementations.

Inspecting definition 2 suggests a strategy. The main action may be split into two: one which acts on the dictionary, one on the pipeline; the requirement is that dictionary and pipeline state must also be split. To split Run, it must be manipulated into a suitable form.

3.2. Splitting the Main Action

The implementation is to contain a pipeline of cells, storing data from the network. As these are to be implemented as concurrent registers, user state in each cell must be disjoint from the next—the requirement for *Process splitting* (law 1). Counter-intuitively, therefore, the first development step is to separate the *checksum calculation* from the rest the application.

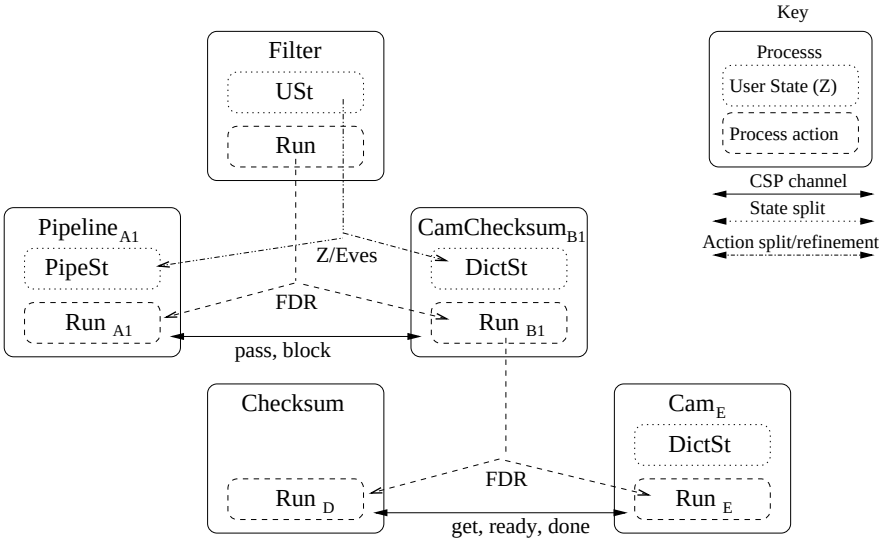

Figure 3. The development and proof strategy for the first system refinements

This is non-obvious as a first step, but is vitally important. The checksum requires to inspect the value of a number of pipeline cells, and given that the states must be disjoint it *cannot* be implemented as a global predicate over these cells. The specification must be manipulated such that the checksum inspects a local copy of the pipeline.

The goal of this development phase is to split the component of the action that maintains the pipeline from that which calculates the checksum and performs the lookup. This is done by replicating the behaviour in two actions, and then removing the unrequired behaviours in each. In doing so, there are several synchronous properties of the specification that care must be taken to preserve. Firstly, in and out occur pairwise. Secondly, when a match occurs, it must interleave the correct pair of in and out events. Thirdly, the address to be looked up must be stored and made available on the correct in-out cycle. To achieve this, two new events are introduced. The event $block$ is used to de-limit in-out cycles after each $UpdateAll$ operation. On each iteration a second new event $pass$ is introduced that communicates the values held in the pipeline to those who may desire read access. In this, and following definitions the actions that pass local state across are factored out in definition 3 for presentation.

Definition 3 *Passing pipeline state :*

$$
\begin{array}{l}
Pass \mathrel{\widehat=} in?b \rightarrow SKIP \interleave out!last(pipe) \rightarrow SKIP \interleave (\interleave\, i : \mathrm{dom}\, pipe \bullet pass.i!pipe(i) \rightarrow SKIP) \\
Pass' \mathrel{\widehat=} in?b \rightarrow SKIP \interleave out!last(pipe) \rightarrow SKIP \interleave (\interleave\, i : \mathrm{dom}\, pipe \bullet pass.i?copy(i) \rightarrow SKIP) \\
Match \mathrel{\widehat=} Pass \interleave match!(h \in dict) \rightarrow SKIP \\
Match' \mathrel{\widehat=} Pass' \interleave match!(h \in dict) \rightarrow SKIP
\end{array}
$$

□

Definition 4 *Introducing internal events $block$ and $pass$, and a new concurrent action :*

$$
\begin{array}{l}
Run_A \mathrel{\widehat=} \mathbf{var}\ c : \mathbb{N};\ h : (Addr \times Addr);\ \mu\, X \bullet \\
\qquad Pass;\ \mathbf{if} \neg\, chk(pipe)\, \mathbf{then}\ UpdateAll;\ block \rightarrow X \\
\qquad\quad \mathbf{else}\, h := addr(pipe);\ c := 1;\ \mu\, Y \bullet \\
\qquad\qquad\quad UpdateAll;\ block \rightarrow SKIP; \\
\qquad\qquad\qquad \mathbf{if}\ c < pipesize - 1\, \mathbf{then} Pass;\ c := c + 1;\ block \rightarrow Y \\
\qquad\qquad\qquad \mathbf{else}\, Match;\ block \rightarrow X
\end{array}
$$

$$
\begin{array}{l}
Run_B \mathrel{\widehat=} \mathbf{var}\ copy : \mathbb{N} \pfun Byte \mid \mathrm{dom}\ copy = 1..pipesize;\ c : \mathbb{N};\ h : (Addr \times Addr);\ \mu\, X \bullet \\
\qquad Pass';\ \mathbf{if} \neg\ chk(pipe)\ \mathbf{then}\ block \rightarrow X \\
\qquad\qquad \mathbf{else}\ h := addr(copy);\ c := 1;\ \mu\, Y \bullet \\
\qquad\qquad\qquad block \rightarrow SKIP; \\
\qquad\qquad\qquad\qquad \mathbf{if}\ c < pipesize - 1\ \mathbf{then}\ Pass';\ c := c + 1;\ block \rightarrow Y \\
\qquad\qquad\qquad\qquad \mathbf{else}\ Match';\ block \rightarrow X
\end{array}
$$

$$
Run \mathrel{\widehat=} (Run_A \llbracket \{\!| in, out, match, pass, block |\!\} \rrbracket Run_B)
$$

□

The checksum behaviour must also be factored out. This step relies on the property that when $det(P)$, $P = P \parallel P$. Run is deterministic, so may be placed in parallel with a second copy of itself. However, although the two actions synchronise on their events—preserving the deterministic property—the operation schemas and state variable assignments do not. If this were disregarded the action would, for instance, execute $UpdateAll$ twice every time it were intended to execute once. The second copy of this action therefore has its own copies of local variables c and h; and does not write to global state. In fact, this development step goes one stage further: it introduces a new variable $copy$ to the new action that has the same type as the pipeline, and each $pass$ cycle updates the local copy. This step further relies on laws for local variable introduction, and for introducing a direction in the $pass$ communication. The replicated action is given in definition 4.

Definition 5 *Removing replicated behaviours in the pipeline :*

$$
Run_{A1} \mathrel{\widehat=} \mu\, X \bullet Pass;\ UpdateAll;\ block \rightarrow X
$$

□

The next step is to separate concerns between the two actions—this relies on the properties of synchronisation and distributed co-termination of the in-out-$pass$ sequence. Run_A is to form the pipeline, therefore Run_B should not engage in in or out. Given that $block$ was introduced to de-limit the in-out sequences, then in and out can safely be dropped from Run_B and removed from the synchronisation. In fact, the same argument also holds for $match$: the pipeline should not be aware of matches, therefore it can be dropped from Run_A. This allows a further simplification: the variable h no longer plays a role in Run_A, so its scope may be restricted to Run_B. Moreover, the role played by c was to implement a loop that caused a number of shifts before a $match$—this is no longer necessary in Run_A. Each of the two components may be individually labelled. This is given in definition 5 and definition 6, where Run_{A1} reads data in and out whilst passing it across to the Run_{B1}, which records this data and indicates the result of a lookup when appropriate.

Definition 6 *Removing replicated behaviours in the CAM and checksum :*

$$
\begin{array}{l}
Run_{B1} \mathrel{\widehat=} \mathbf{var}\ copy : \mathbb{N} \pfun Byte \mid \mathrm{dom}\ copy = 1..pipesize;\ c : \mathbb{N};\ h : (Addr \times Addr);\ \mu\, X \bullet \\
\quad (|||\ i : \mathrm{dom}\ copy \bullet pass.i?copy(i) \rightarrow SKIP); \\
\qquad \mathbf{if} \neg\ chk(copy)\ \mathbf{then}\ block \rightarrow X \\
\qquad \mathbf{else}\ c := 1;\ h := addr(copy);\ \mu\, Y \bullet \\
\qquad\quad block \rightarrow SKIP; \\
\qquad\qquad \mathbf{if}\ c < \#\,\mathrm{dom}\ copy - 1\ \mathbf{then} \\
\qquad\qquad\quad (|||\ i : \mathrm{dom}\ copy \bullet pass.i?x \rightarrow SKIP);\ c := c + 1;\ block \rightarrow Y \\
\qquad\qquad \mathbf{else} \\
\qquad\qquad\quad (|||\ i : \mathrm{dom}\ copy \bullet pass.i?x \rightarrow SKIP\ \ |||\ match!(h \in dict) \rightarrow SKIP\ ); \\
\qquad\qquad\qquad block \rightarrow X
\end{array}
$$

□

Run_{A1} and Run_{B1} can be seen to be disjoint with respect to user state. There is no proof obligation associated with this— but if it were not true, further development would fail. However there is an obligation to show that the new actions have been correctly derived— theorem 1, which states that the parallel combination of the new actions is a refinement of the specification. This may be proved by asserting the equivalent refinement relation using FDR.

Theorem 1 *The calculated actions are a refinement*
$Run \sqsubseteq_A (Run_{A1} \llbracket \{\!|pass, block|\!\} \rrbracket Run_{B1}) \setminus \{\!|pass, block|\!\}$

Proof

assert
$Run \sqsubseteq_{FD} (Run_{A1} \llbracket \{\!|pass, block|\!\} \rrbracket Run_{B1}) \setminus \{\!|pass, block|\!\}$ ✓

□

3.3. Partitioning the Global User State

Although the actions have now been split, they both still reference the single global user state. Run_{A1} accesses the pipeline state, while Run_{B1} accesses the dictionary. In order to show that this process meets the form applicable to process splitting, these states, and the operations upon them, must be disjoint. In this section, the disjoint states are calculated and verified using Z/Eves. By relying on the properties of schema conjunction, the original user state can be split in two.

Definition 7 *Partitioned user state :*

$$PipeSt \mathrel{\widehat=} [\ pipe : \mathbb{N} \pfun Byte \mid \mathrm{dom}\, pipe = 1..pipesize\ ]$$
$$DictSt \mathrel{\widehat=} [\ dict : \mathbb{N} \pfun (Addr \times Addr) \mid \mathrm{dom}\, dict = 1..dictsize\ ]$$

□

Theorem 2 *The partitioned states are correct :* $Ust \Leftrightarrow (PipeSt \land DictSt)$

Proof prove by reduce; □

The update operation only acts on the pipeline, and leaves the dictionary unchanged.

Definition 8 *Partitioning the* $Update$ *operation :*

$$UpdateAll' \mathrel{\widehat=}$$
$$[\ \Delta PipeSt;\ b? : byte \mid$$
$$pipe' = \{x : 1..\#\,\mathrm{dom}\, pipe;\ y : Byte \mid$$
$$x \mapsto y \in pipe \land x + 1 \in \mathrm{dom}\, pipe \bullet x + 1 \mapsto y\} \cup \{1 \mapsto b?\}\ ]$$

□

Theorem 3 *The partitioned Update is correct :* $UpdateAll \Leftrightarrow (UpdateAll' \land \Xi DictSt)$

Proof prove by reduce; □

3.4. A First Application of Process Splitting

The actions and the state are now of the correct form for *Process splitting* to be applied. The abstract pipeline is a very simple process. It contains an abstract data type that holds all of the data present in the pipeline. On each iteration, it reads in a new value, outputs the oldest value, informs the environment of the current state of the pipeline, and then shifts all the data.

Definition 9 *The abstract pipeline process :*

$$
\begin{array}{l}
\textbf{process}\ Pipeline_{A1} \mathrel{\widehat=} \textbf{begin} \\
\quad PipeSt \mathrel{\widehat=} [pipe : \mathbb{N} \pfun Byte \mid \operatorname{dom} pipe = 1..pipesize] \\
\quad UpdateAll' \mathrel{\widehat=} definition\ 8 \\
\quad Run_{A1} \mathrel{\widehat=} definition\ 5 \\
\bullet\ Run_{A1} \\
\textbf{end}
\end{array}
$$

□

The abstract checksum and lookup process of definition 10 contains the dictionary. The main action of this process performs the checksum calculation on the local copy of the state that is passed to it on each pipeline shift, and outputs the result of the lookup accordingly.

Definition 10 *The abstract CAM/checksum process :*

$$
\begin{array}{l}
\textbf{process}\ CamChecksum_{B1} \mathrel{\widehat=} \textbf{begin} \\
\quad DictSt \mathrel{\widehat=} [dict : \mathbb{N} \pfun (Addr \times Addr)] \\
\quad Run_{B1} \mathrel{\widehat=} definition\ 6 \\
\bullet\ Run_{B1} \\
\textbf{end}
\end{array}
$$

□

The complete packet filter is the pipeline in parallel with the checksum synchronising on the channel used to pass pipeline state across. Both also synchronise on the channel $block$, thus maintaining the synchronous nature of the behaviour between the two components.

Definition 11 *The split packet filter :*

$$Filter' \mathrel{\widehat=} (CamChecksum_{B1} \lbrack\!\lbrack \{\!| pass, block |\!\} \rbrack\!\rbrack Pipeline_{A1}) \setminus \{\!| pass, block |\!\}$$

□

4. Implementing the Pipeline as Concurrent Cells

The next stage of development is to implement the pipeline process as an array of concurrent cells, using the generalised chaining operator of [5].

4.1. Implementing the Update Operation

[10] shows a Z-style promotion distributes through a *Circus* process. By factoring out a promotion from the abstract specification of the pipeline, a single cell is exposed. However, the $UpdateAll'$ operation describes a shift of the entire pipeline: the first task, therefore, is to rewrite this in terms of a single element. The schema $Update$ defines an update on a single element of the pipeline, identified by the input variable i.

Definition 12 *A single update :*

$$Update \mathrel{\widehat=} [\ \Delta PipeSt;\ b? : Byte;\ i? : \mathbb{N} \mid pipe' = pipe \oplus \{i? \mapsto b?\}\ ]$$

□

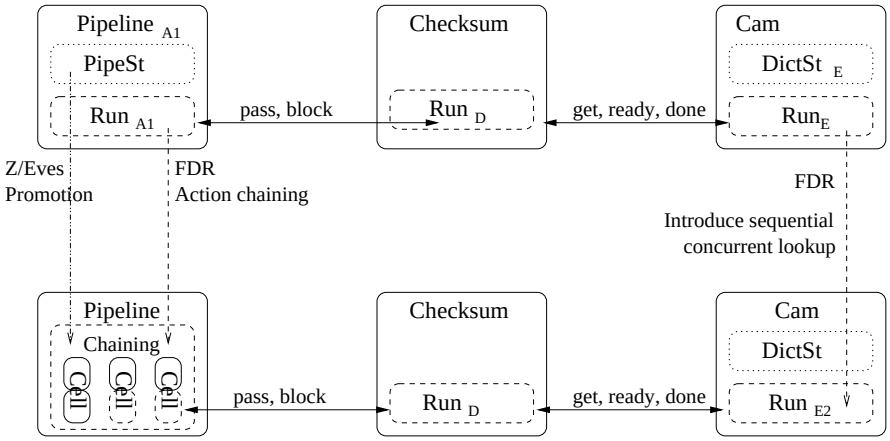

Figure 4. The development and proof strategy for the second system refinements

For this operation to implement a complete pipeline shift, it must act upon all elements of the pipeline. This may be achieved by replicating and interleaving $pipeline$ copies of the operation. It is necessary to store the initial state of the pipeline in a local variable to ensure that each interleaved $Update$ acts on the correct initial value of its predecessor element.

Definition 13 *Interleaved updates :*

$$IUpdate \mathrel{\widehat=} \\ \quad \mathbf{var}\, copy : \mathbb{N} \pfun Byte \mid copy = pipe \bullet \lll i : 1..pipesize \bullet (i \neq 1 \land b = copy(i-1);\ Update)$$

□

Now, the local state may be described in terms of a promotion—a local data type with a schema describing its relation to the global state. A local element is simply a $Byte$, while the global state of the system is a function from natural numbers (the index of each element in the pipeline) to the elements. The update operation changes an individual element to the value of the input variable.

Definition 14 *Local and global views :*

$$PipeCell \mathrel{\widehat=} [\, elem : Byte \,] \\ GlobalPipe \mathrel{\widehat=} [\, pipe : \mathbb{N} \pfun PipeCell \,] \\ UpdateCell \mathrel{\widehat=} [\, \Delta PipeCell;\ b? : Byte \mid elem' = b? \,]$$

□

The global view of the system, in terms of the local elements is given by the schema $Promote$.

Definition 15 *The promotion schema :*

$Promote$

$\Delta GlobalPipe$
$\Delta PipeCell$
$i? : \mathbb{N}$

$i? \in \operatorname{dom} pipe$
$\theta PipeCell = pipe(i?)$
$pipe' = pipe \oplus \{i? \mapsto \theta PipeCell\}$

□

For this promotion to be factored out of the system, it is necessary that it is *free*. That is to say there is no global constraint that cannot be expressed locally. If this were not the case, then each local element of state would need to be aware of other local elements of state: something that is not permitted if processes are to be concurrent; theorem 4 captures this. Theorem 5 states that the promoted update operation is equivalent to the single update operation above.

Theorem 4 *The promotion is free :*
$\exists\, PipeCell' \bullet \exists\, GlobalPipe' \bullet Promote \Rightarrow \forall\, PipeCell' \bullet \exists\, GlobalPipe' \bullet Promote$

Proof prove by reduce; prove □

Theorem 5 *Promoted update is correct :*
$SingleUpdate \Leftrightarrow \exists \Delta PipeCell \bullet PipeCell \wedge Promote$

Proof prove by reduce; □

4.2. A Local Process Implementing a Cell

A single pipeline cell is a process that encapsulates the local data, with an unpromoted input and output action.

$$
\begin{array}{l}
\textbf{process}\ Cell \mathrel{\widehat=} \textbf{begin} \\
\quad PipeCell \mathrel{\widehat=} [\, elem : Byte \,] \\
\quad UpdateCell \mathrel{\widehat=} [\, \Delta PipeCell;\ b? : Byte \mid elem' = b? \,] \\
\\
\quad Run_C \mathrel{\widehat=} \mu\, X \bullet \\
\qquad (in?b \rightarrow SKIP \mid\mid\mid out!elem \rightarrow SKIP \mid\mid\mid pass.i!elem \rightarrow SKIP);\ UpdateCell;\ block \rightarrow X \\
\bullet\ Run_C \\
\textbf{end}
\end{array}
$$

Now it seems possible to concurrently compose a number of these cells to form a pipeline using a law such as process indexing of [10]. However, this will be insufficient—this law requires that there is no interference between local processes. This is precisely not the case here, where each process requires the local value of its numeric predecessor in the pipeline—this was the role played by the local variable $copy$ in the definition of $Update$ earlier in this section. Furthermore, only the first input in and the last input out are externally visible.

4.3. Composing Local Processes

This is exactly the scenario that is achieved by generalised chaining. The promotion of each cell is the function from local to global state, and the states and local operations are shown to observe the requirement that they are disjoint as the promotion is free. Promoting (and renaming) the in, last out, $tick$ and $tock$ events gives the global action with all the internal communications hidden. The index of each process i is taken from the promotion schema.

The operator composes $pipesize$ copies of the process $Cell$ concurrently. A parameter which is given to the operator is a pair of events that are to chain—to synchronise—events in numerically adjacent processes, these pairs of events are uniquely renamed and placed in the synchronisation sets of adjacent processes accordingly. This synchronisation can be used to communicate the initial value of one cell to its neighbouring cell. Internal events are hidden. The *ripple effect* of nearest neighbour communication ensures that the inputs and outputs are ordered correctly. This construction is exampled for a pipeline of three cells in figure 5.

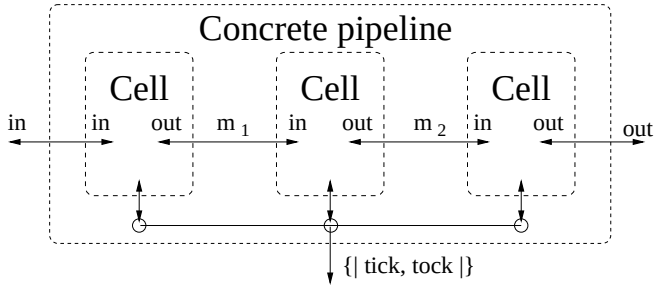

Figure 5. The process $[in \leftrightarrow out, \{\!|tick, tock|\!\}]\, 3 \bullet Cell$

Definition 16 *The pipeline implementation :*

$$Pipeline \mathrel{\widehat=} [in \leftrightarrow out, \{\!|block|\!\}]pipesize \bullet Cell$$

□

Theorem 6 *The implementation of the pipeline is correct*
$Pipeline_{A1} \sqsubseteq_P Pipeline$

Proof

assert
$Pipeline_{A1} \sqsubseteq_{FD} Pipeline$ ✓

□

4.4. Modelling the Local Processes as Handel-C Variables

The description of a single cell is very close to *Handel-C*. The remaining task is to include the model of the clock. Assignments take place on a rising clock edge: this can be modelled using an event $tick$. All assignments in all processes must happen before a clock cycle completes, and this can be modelled using an event $tock$. In adapting the description of a cell to behave as a *Handel-C* process this clock model must be included. It is trivial in this case: each process performs a single assignment after its communications, therefore each iteration in Run_{C1} is a single clock cycle. The event $block$ ensured the synchronous behaviour of pipeline cells on each iteration: this function is now achieved by the clock, so $block$ may be dropped.[4] This description of a clocked cell is now only a syntactic step away from the implementation of each cell as a simple variable.

Definition 17 *A Handel-C model of Cell implementation :*

$$\begin{array}{l}
\textbf{process}\ ClockedCell \mathrel{\widehat=} \textbf{begin} \\
\quad PipeCell \mathrel{\widehat=} [\, elem : Byte \,] \\
\quad UpdateCell \mathrel{\widehat=} [\, \Delta PipeCell;\ b? : Byte \mid elem' = b? \,] \\
\\
\quad Run_{C1} \mathrel{\widehat=} \mu\, X \bullet \\
\qquad (in?b \rightarrow SKIP \mid\mid\mid out!elem \rightarrow SKIP \mid\mid\mid pass.i!elem \rightarrow SKIP); \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad tick \rightarrow UpdateCell;\ tock \rightarrow X \\
\bullet\ Run_{C1} \\
\textbf{end}
\end{array}$$

□

[4] Alternatively, this step could be regarded as renaming $block$ to $tock$ and including the $tick$ event to globally synchronise and separate communications from state updates.

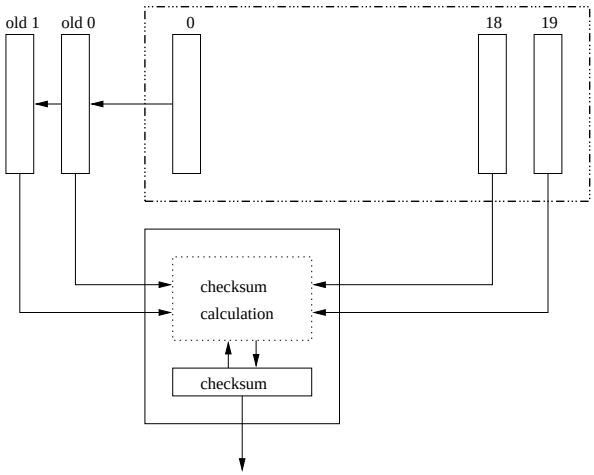

Figure 6. The checksum calculation

Adapting the chained processes is trivial: all instances share the same clock, and this is achieved by placing the events in the global (shared) synchronisation set.

Definition 18 *The clocked Handel-C pipeline implementation :*

$$ClockedPipeline \mathrel{\widehat=} [in \leftrightarrow out, \{\!|tick, tock|\!\}]\ pipesize \bullet ClockedCell$$

□

In the above definition, the clock may be hidden if no further processes are to be introduced to the *Handel-C* implementation, or if they use a separate clock; however it is left visible in this implementation as the CAM will share the same global clock. Theorem 6 states that the clocked implementation is correct.

Theorem 7 *The implementation of the clocked pipeline is correct*

$$Pipeline \setminus \{\!|block|\!\} \sqsubseteq_P ClockedPipeline \setminus \{\!|tick, tock|\!\}$$

Proof

assert

$$Pipeline \setminus \{\!|block|\!\} \sqsubseteq_{FD} ClockedPipeline \setminus \{\!|tick, tock|\!\} \quad \checkmark$$

□

5. Separating the Checksum and the CAM

The checksum is a ones complement sum of 16 bit segments of the packet header, and is a standard checksum used in IP packet identification. If the checksum calculation returns the same result as that contained within the header, and several other sanity checks also hold then the state of the pipeline represents an IP packet header. The source and destination addresses in the pipeline are stored for subsequent inspection. The next stage of development is to separate out the checksum from the process that performs this inspection on the addresses.

Definition 19 *Adding the partner action :*

$$\begin{aligned} Run_{B2} \mathrel{\widehat=} \mathbf{var}\ & h : (Addr \times Addr) ... \bullet \\ & \mu\, X \bullet ...;\ \mathbf{if}\ c < pipesize - 2\ \mathbf{then}\ ...;\ X \\ & \qquad \mathbf{else}\,(... \ |||\ ready \rightarrow match!(h \in dict) \rightarrow done \rightarrow SKIP\,);\ X \\ & [\![\{\!|ready, match, done|\!\}]\!] \\ & \mu\, Y \bullet ready \rightarrow match!(h \in dict) \rightarrow done \rightarrow Y \end{aligned}$$

□

The first steps in separating the checksum and the CAM follow a similar pattern to before. A $match$ is prefixed with $ready$ and followed with $done$. A new second action agrees to this $ready$-$match$-$done$ cycle. As before, $match$ may now be dropped from the original. This is shown in definition 19; some parts of the definition not relevant to this design step have been abbreviated.

If Run_{B2} is to be split, it must not share the global variable h. The main tool for removing such dependencies is the introduction of a new event to communicate state: get is introduced for this purpose. Now restricting the scope of h (and of $copy$ and c to the first action) is trivial, and Run_{B2} may be rewritten as two separate actions Run_D and Run_E.

Definition 20 *The checksum action :*

$$
\begin{array}{l}
Run_D \mathrel{\widehat=} \mathbf{var}\ copy : \mathbb{N} \pfun Byte \mid \mathrm{dom}\ copy = 1..pipesize;\ c : \mathbb{N};\ \mu\, X \bullet \\
\quad (\|\| i : \mathrm{dom}\ copy \bullet pass.i?copy(i) \then SKIP); \\
\qquad \mathbf{if} \neg chk(copy)\ \mathbf{then}\ block \then Check \\
\qquad \mathbf{else}\ c := 1;\ get!addr(copy) \then SKIP \bullet \mu\, Y \bullet block \then SKIP; \\
\qquad\quad \mathbf{if}\ c < \mathrm{dom}\ copy - 1\ \mathbf{then} \\
\qquad\qquad (\|\| i : \mathrm{dom}\ copy \bullet pass.i?copy(i) \then SKIP);\ block \then c = c + 1;\ Y \\
\qquad\quad \mathbf{else}\ ready \then (\|\| i : \mathrm{dom}\ copy \bullet pass.i?copy(i) \then SKIP);\ done \then block \then X
\end{array}
$$

□

Definition 21 *The CAM action :*

$$Run_E \mathrel{\widehat=} \mu\, X \bullet get?term \then ready \then match!(term \in dict) \then done \then X$$

□

5.1. Splitting the Checksum and CAM

The two actions are now of a form that allows the process to be split. The correctness of this development step can be verified by proving the refinement relation holds between the main action of the abstract CAM and checksum process of definition 10 and the newly split actions.

Theorem 8 *The split actions are a refinement*

$$Run_{B1} \sqsubseteq_A (Run_D \,[\![\, \{\!|get, ready, done|\!\} \,]\!]\, Run_E) \setminus \{\!|get, ready, done|\!\}$$

Proof

assert
$Run_{B1} \sqsubseteq_{FD} (Run_D \,[\![\, \{\!|get, ready, done|\!\} \,]\!]\, Run_E) \setminus \{\!|get, ready, done|\!\}$ ✓

□

Definition 22 *The abstract CAM process :*

$$
\begin{array}{l}
\mathbf{process}\ Cam_E \mathrel{\widehat=} \mathbf{begin} \\
\quad DictSt \mathrel{\widehat=} [dict : \mathbb{N} \pfun (Addr \times Addr) \mid \mathrm{dom}\ dict = 1..dictsize] \\
\quad Run_E \mathrel{\widehat=} definition\ 21 \\
\bullet\ Run \\
\mathbf{end}
\end{array}
$$

□

In the checksum, local variables may be encapsulated as the user state of the process.

Definition 23 *The checksum implementation :*

$$
\begin{array}{l}
\textbf{process}\ Checker \mathrel{\widehat=} \textbf{begin} \\
\quad USt \mathrel{\widehat=} [copy : \mathbb{N} \pfun Byte;\ c : \mathbb{N} \mid \mathrm{dom}\ copy = 1..pipesize] \\
\quad Run_D \mathrel{\widehat=} definition\ 20 \\
\bullet\ Run_D \\
\textbf{end}
\end{array}
$$

□

The process monitoring the checksum calculation is now ready for implementation, and may have the program clock introduced. The interleaved $pass$ events are actually emulating read access to the pipeline process, so they must appear before a $tick$—there is no value to be latched in. Other assignments, such as to the local variable c must be latched between the $tick$ and the $tock$. The clock now additionally performs the role of making sure that each $pass$ cycle is synchronous with respect to pipeline shifts (as the clock is shared with the pipeline), so $block$ may be dropped.

Another less obvious role of the clock comes from the fact that the events $ready$ and $done$ occur exactly $\#\,\mathrm{dom}\ copy - 2$ clock cycles after a get is issued (counting starts at 1). These events were introduced to ensure that the CAM would output the result at the correct time. As the CAM is to be implemented on the same FPGA, with the same global clock, the assumption that the $match$ output will happen $\#\,\mathrm{dom}\ copy - 2$ clock cycles after it receives a get can be made. As long as the development of the CAM respects this assumption, $ready$ and $done$ no longer play a significant role in the behaviour of the checksum and the CAM, and may be dropped. This step has not been made as a result of a direct application of a law or of laws: more is said about this in section 6.

Definition 24 *The clocked checksum action :*

$$
\begin{array}{l}
Run_{D1} \mathrel{\widehat=} \mu\, X \bullet \\
\quad (|||\ i : \mathrm{dom}\ copy \bullet pass.i?copy(i) \then SKIP); \\
\qquad \textbf{if}\ chk(copy)\ \textbf{then}\ tick \then tock \then Check \\
\qquad \textbf{else}\ get!addr(copy) \then tick \then c := 1;\ tock \then SKIP \bullet \mu\, Y \bullet \\
\qquad\quad \textbf{if}\ c < \#\,\mathrm{dom}\ copy - 1\ \textbf{then} \\
\qquad\qquad (|||\ i : \mathrm{dom}\ copy \bullet pass.i?copy(i) \then SKIP);\ tick \then c = c + 1;\ tock \then Y \\
\qquad\quad \textbf{else}\ (|||\ i : \mathrm{dom}\ copy \bullet pass.i?copy(i) \then SKIP);\ tick \then tock \then X
\end{array}
$$

□

Definition 25 *A Handel-C model of the checksum implementation :*

$$
\begin{array}{l}
\textbf{process}\ ClockedChecker \mathrel{\widehat=} \textbf{begin} \\
\quad USt \mathrel{\widehat=} [copy : \mathbb{N} \pfun Byte;\ c : \mathbb{N} \mid \mathrm{dom}\ copy = 1..pipesize] \\
\quad Run_{D1} \mathrel{\widehat=} definition\ 24 \\
\bullet\ Run_{D1} \\
\textbf{end}
\end{array}
$$

□

Theorem 9 states that the clocked implementation is correct.

Theorem 9 *The clocked checksum is correct*

$$Checker \setminus \{|ready, done, block|\} \sqsubseteq_P ClockedChecker \setminus \{|tick, tock|\}$$

Proof

assert
$Checker \setminus \{\!|ready, done, block|\!\} \sqsubseteq_{FD} ClockedChecker \setminus \{\!|tick, tock|\!\}$ ✓

□

6. Implementing the Content Addressable Memory

The size of an IP packet header, combined with the clock cycle requirements of the components implemented so far is a useful piece of information in designing the CAM implementation. It allows splitting combinatorial logic over multiple clock cycles (thereby decreasing wall clock time requirements) and a more serial implementation of a CAM, which may re-use comparand registers and reduce area requirements.

The *Rotated ROM* design of [6,3] consists of ROMs of depth 16 bits, and width 2 bits, giving 32 bit words: each one of which corresponds to an 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.

Figure 8 shows the area costs of this design on a sample FPGA, and the clock speeds attainable for increasing CAM sizes in figure 7: these experimental results confirm that a Rotated ROM CAM of the sizes under consideration will permit the FPGA to be clocked at a sufficiently fast speed to allow the rest of the application to monitor a network in real time.

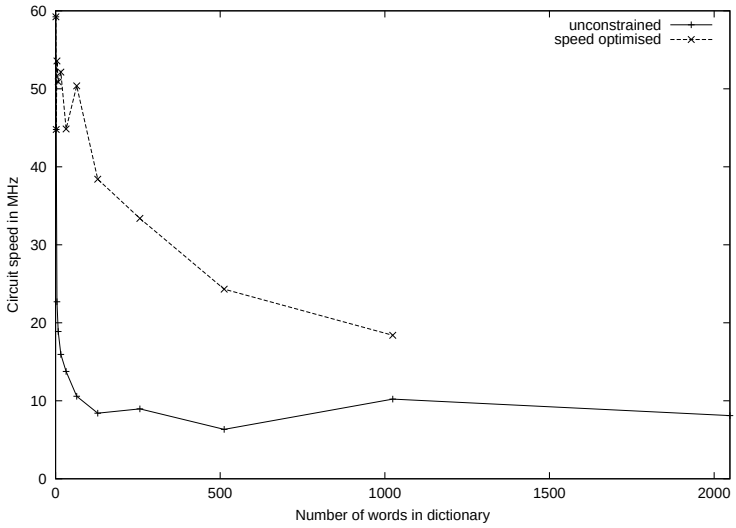

Figure 7. Clock speeds of the Rotated ROM CAM on a Xilinx 40150 FPGA

There are 20 clock cycles available after a packet arrives in the pipeline before the match result must be output. This can be exploited: the comparison in the dictionary can be designed to use many—even all—of these clock cycles, thus reducing combinatorial logic costs and expensive comparators—and the Rotated ROM CAM is an architecture that exploits this. In this section, the abstract CAM of definition 22 is refined into this implementation.

6.1. Making the Dictionary Tight

Initially, the definition of the dictionary state was left loose. This section begins by making it tight—given in the replacement definition of $DictSt$ below, with values taken from table

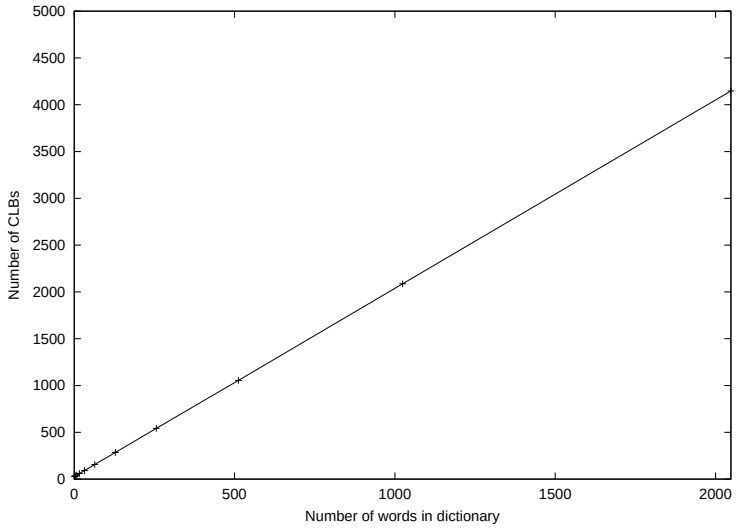

Figure 8. Area costs of the Rotated ROM CAM on a Xilinx 40150 FPGA

Direction	**IP address**	**2x16 bitwise representation**
Source_0	163.1.27.192	10,10,00,11,00,00,00,01,00,01,10,11,11,00,00,00
Destination_0	163.1.27.18	10,10,00,11,00,00,00,01,00,01,10,11,00,01,00,10
Source_1	163.1.27.152	10,10,00,11,00,00,00,01,00,01,10,11,10,01,10,00
Destination_1	163.1.27.162	10,10,00,11,00,00,00,01,00,01,10,11,10,10,00,10

Table 1. Example address pairs

1. The state in the schema $Dict$ is the source/destination addresses given as 2×16 $BitPair$ representations. As the dictionary is being implemented in hardware, the values are invariant (statically determined)—there is no need for an initialisation operation, as they never change.

User state now contains a dictionary consisting of four concrete addresses, implemented as four $BitPair$ sequences of length 16.[5] In implementing the lookup it is tempting to specify this by defining simple equality tests over elements; however, the design goal is directed by area and speed concerns—optimally, a single, comparator for each word in the dictionary.

Definition 26 *The state of the dictionary :*

$$
\begin{array}{|l}
\hline
DictSt \\
dict : \mathbb{N} \pfun \operatorname{seq} BitPair \\
\hline
\operatorname{dom} dict = 1..4 \\
dict.1 = \langle (1,0),(1,0),(0,0),(1,1),(0,0),(0,0),(0,0),(0,1), \\
\qquad\qquad (0,0),(0,1),(1,0),(1,1),(1,1),(0,0),(0,0),(0,0)\rangle \\
dict.2 = \langle (1,0),(1,0),(0,0),(1,1),(0,0),(0,0),(0,0),(0,1), \\
\qquad\qquad (0,0),(0,1),(1,0),(1,1),(0,0),(0,1),(0,0),(1,0)\rangle \\
dict.3 = \langle (1,0),(1,0),(0,0),(1,1),(0,0),(0,0),(0,0),(0,1), \\
\qquad\qquad (0,0),(0,1),(1,0),(1,1),(1,0),(0,1),(1,0),(0,0)\rangle \\
dict.4 = \langle (1,0),(1,0),(0,0),(1,1),(0,0),(0,0),(0,0),(0,1), \\
\qquad\qquad (0,0),(0,1),(1,0),(1,1),(1,0),(1,0),(0,0),(1,0)\rangle \\
\hline
\end{array}
$$

□

In the Rotated ROM CAM, local variables are used to maintain the results of a lookup and index iterations. The definition of the main CAM action below reflects this. The local array

[5] The replacement definition of $BitPair$ is omitted as it is clear from its usage here.

$result$ stores the boolean result of comparing each dictionary entry with the search term. The $match$ line outputs a result indicating whether or not any of the comparisons returned true—this is implemented as the disjunction over the elements in the result array. Rather than a simple comparison over each word, it is implemented as an iteration over comparing each $BitPair$ with the relevant corresponding place in the search term, meaning that there are 16 comparisons performed for each word, with the n_{th} comparison for each word being performed in parallel. When a comparison fails, the fact that that dictionary word does not match the search term is recorded. Implementation of a 2-bit comparator is cheap: in this way, the area costs have been reduced, the combinatorial costs have been drastically reduced. The events $ready$ and $done$ ensure that the implementation still meets the constraints that the lookup should be complete before the word has left the pipeline.

Definition 27 *Sequencing the comparators :*

$$
\begin{array}{l}
Run_{E1} \mathrel{\widehat=} \mathbf{var}\ result : \mathrm{dom}\ dict \to \mathbb{B};\ \mu\, X \bullet \\
\quad get?term \to c := 0;\ \mathrm{ran}\ result := true;\ \mu\, Y \bullet \\
\quad\quad \mathbf{if}\ c = 16\ \mathbf{then}\ ready \to match!(true \in \mathrm{ran}\ result) \to done \to X \\
\quad\quad \mathbf{else} \forall\, i : \mathrm{dom}\ result \bullet result(i) = result(i) \land dict.i(c) = head(term); \\
\quad\quad\quad term := tail(term);\ c := c + 1;\ Y
\end{array}
$$

□

The universal quantifier in definition 27 may be expanded—resulting in the conjunction of the set of assignments indexed by i. As these assignments are all disjoint with respect to the state that they evaluate and assign to they may be implemented in an arbitrary order.

Definition 28 *Concurrent words :*

$$
\begin{array}{l}
Run_{E2} \mathrel{\widehat=} result : \mathrm{dom}\ dict \to \mathbb{B};\ \mu\, X \bullet \\
\quad get?term \to c := 0;\ \mathrm{ran}\ result := true;\ \mu\, Y \bullet \\
\quad\quad \mathbf{if}\ c = 16\ \mathbf{then}\ ready \to match!(true \in \mathrm{ran}\ result) \to done \to X \\
\quad\quad \mathbf{else} \parallel\!\!\parallel\ i : \mathrm{dom}\ result \bullet result(i) = result(i) \land dict.i(c) = head(term); \\
\quad\quad\quad term := tail(term);\ c := c + 1;\ Y
\end{array}
$$

□

Unlike the other components in this case study, concurrency is not introduced using process splitting—it has been introduced with concurrent assignments to user state. To attempt process splitting is not useful: although each entry in the dictionary and the comparisons are disjoint, the result array is not. To split these processes—and therefore the result array—and collate results using further communications would mean that either the clock cycle constraint is not met, or that the implementation is less serial.

Definition 29 *The CAM implementation :*

$$
\begin{array}{l}
\mathbf{process}\ Cam \mathrel{\widehat=} \mathbf{begin} \\
\quad DictSt \mathrel{\widehat=} definition\ 26 \\
\quad Run_{E2} \mathrel{\widehat=} definition\ 28 \\
\bullet\ Run_{E2} \\
\mathbf{end}
\end{array}
$$

□

As the implementation is to be in *Handel-C* on the same FPGA as the other components the global clock may now be introduced, in definition 30. The first clock cycle reads in and assigns the value of the search term. The last clock cycle for any given lookup is occupied by outputting the result. In between, there are 16 clock cycles available for the lookup—this is the assumption that was made in the final development step for the clocked checksum process

in definition 25 that allowed the events $ready$ and $done$ to be dropped. The lookup itself actually consists of a sequence of assignments to the result array, where each element in the array is assigned concurrently. As an assignment in *Handel-C* is latched in during a clock cycle, each assignment in the sequence is a single clock cycle. As there are 16 assignments, one for each $BitPair$ in an address, there are 16 clock cycles. Although this stage of development can be seen to follow from the properties of hiding, it has not been calculated from a direct application of a law as the hiding does not cleanly distribute through concurrency.

It is important that the clock does not block when the process is waiting for input—a communication on get will not happen on every clock cycle. If the extra choice containing $tick$ and $tock$ were not included here, this would be a modelling error that is not apparent in the implementation—in reality, a *Handel-C* process cannot block the clock from ticking whilst waiting for a communication. However, this modelling error would prevent accurate verification of the implementation.

Definition 30 *The clocked CAM action :*

$$
\begin{array}{l}
Run_{E3} \mathrel{\widehat=} result : \operatorname{dom} dict \to \mathbb{B};\ \mu\, X \bullet \\
\quad tick \to tock \to X \\
\quad \Box \\
\quad get?term \to tick \to c := 0;\ \operatorname{ran} result := true;\ tock \to SKIP;\ \mu\, Y \bullet \\
\qquad \textbf{if}\ c = 16\ \textbf{then}\ match!(true \in \operatorname{ran} result) \to tick \to tock \to X \\
\qquad \textbf{else} \parallel i : \operatorname{dom} result \bullet [\![\{\!| tick, tock |\!\}]\!]\ result(i) = result(i) \land dict.i(c) = head(term); \\
\qquad\qquad tick \to term := tail(term);\ c := c + 1;\ tock \to Y
\end{array}
$$

□

Definition 31 *The clocked CAM :*

$$
\begin{array}{l}
\textbf{process}\ ClockedCam \mathrel{\widehat=} \textbf{begin} \\
\quad DictSt \mathrel{\widehat=} definition\ 26 \\
\quad Run_{E3} \mathrel{\widehat=} definition\ 30 \\
\bullet\ Run_{E3} \\
\textbf{end}
\end{array}
$$

□

Verification of the clocked implementation proves to be interesting. If the CSP_M assertion corresponding to theorem 10 is checked, it is found to fail. In this assertion, as with all the other checks of clocked processes, the clock must be hidden to ensure the alphabets of the processes match. Consequently a divergence exists in $ClockedCam$: where it is waiting for an input it may perform an infinite series of clock ticks and a get never occurs.

Verification therefore needs more care: it must ignore this divergence. A simplistic way of achieving this would be to disallow this possibility and re-check; however this may not always be possible for more general examples. Instead, the technique is to show that the divergence did not result from the *real* activities of the process: it is shown to be divergence free when all events other than $tick$ and $tock$ are hidden. The divergence, therefore, must have come from the clock. If the process can then be shown to be a failures refinement then the implementation is correct.

Theorem 10 *The CAM implementation is valid*

$Cam \setminus \{\!| ready, done |\!\} \sqsubseteq_P ClockedCam \setminus \{\!| tick, tock |\!\}$

Definition 32 *The equivalent CSP_M assertions :*

assert
$ClockedCam \setminus \{\!| get, match |\!\} : divergence\ free$ ✓
$Cam \setminus \{\!| ready, done |\!\} \sqsubseteq_{FD} ClockedCam \setminus \{\!| tick, tock |\!\}$ ✓

□

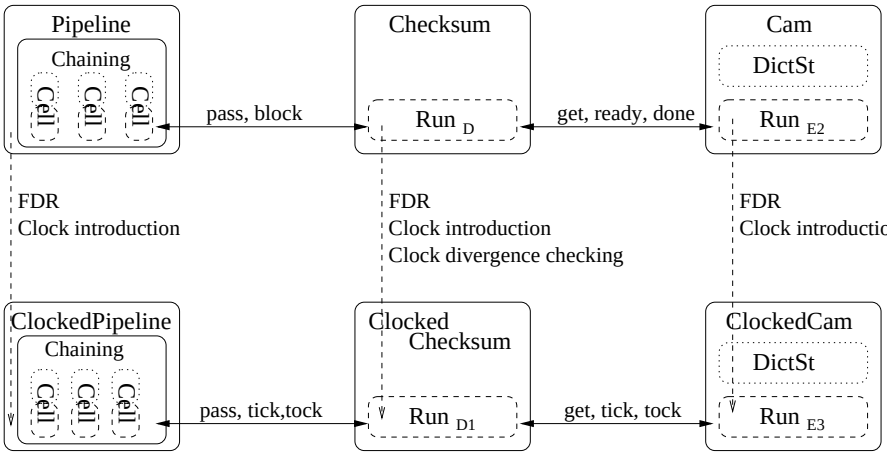

Figure 9. The development and proof strategy for the final clocked system refinements

7. The Final Implementation

The final implementation is the parallel combination of each of the processes developed. This is given in definition 33.

Definition 33 *The final clocked implementation :*

$$\begin{aligned}&\mathbf{process}\ ClockedFilter \mathrel{\widehat=} \\ &\quad (\,(\,ClockedPipeline \,[\![\, \{\!|tick, tock, pass|\!\} \,]\!]\, ClockedChecker\,) \setminus \{\!|pass|\!\} \\ &\quad\ [\![\{\!|get, done, tick, tock|\!\}]\!] \\ &\quad\ ClockedCam\,) \setminus \{\!|get, done, tick, tock|\!\}\end{aligned}$$

□

Theorem 11 *The Handel-C implementation is correct*

$$Filter \sqsubseteq_P ClockedFilter \setminus \{\!|tick, tock|\!\}$$

Proof From the correctness of each stage of development and monotonicity of refinement.□

From the development strategy of figure 3, figure 4, figure 9, and monotonicity of refinement, confidence in the correctness of the implementation is assured. Generally industrial developments are too large to model-check, and monotonicity must be relied upon.

7.1. A Final Twist

In definition 34, the action $BadRun_D$ is presented. This action is included because it highlights an interesting error that may be made in the assumptions.

Definition 34 *A bad checksum process :*

$$\begin{aligned}&BadRun_D \mathrel{\widehat=} \mu\, X \bullet \\ &\quad (|||\ i : \mathrm{dom}\ copy \bullet pass.i?copy(i) \rightarrow SKIP); \\ &\qquad \mathbf{if}\ chk(copy)\ \mathbf{then}\ get!addr(copy) \rightarrow tick \rightarrow tock \rightarrow X\ \mathbf{else}\ tick \rightarrow tock \rightarrow X\end{aligned}$$

□

In the correct checksum process when the calculation returns true the pipeline is not tested again until that packet has left. The above definition evaluates the checksum on each iteration.

When it returns true, it passes the address to the CAM. If the CAM is in the middle of a lookup it refuses to accept an address. Therefore, if the definition above is used the system will deadlock if the checksum returns true before the CAM has finished the last instructed lookup. This version is not a valid refinement: checking the packet filter using this process results in a counter-example evidencing the deadlock. The error was present in early versions of the application. In hardware, leaving a test condition such as this one *permanently running* often appears to be a natural, and inconsequential, assumption. The code was extensively tested in simulation and on real networks and the deadlock was not discovered.

The assumption about the checksum is that it will not produce a false positive: if it does, it may result in *apparent* headers overlapping. In reality, the chance of it doing so is extremely remote: otherwise network routers would encounter the problem regularly. In fact routers typically protect against this problem by disregarding rogue packets. This is why testing did not highlight the issue, as the sample data set had been verified by a router.

Not only does this highlight the value of the formal development in *Circus*, but provides an interesting starting point for requirements checking and investigating the real security offered by the device. While real networks may not present the possibility for false positives, formal development has shown that the device does not function if they did actually happen—and this may form the starting point for a malicious attack on the device.

8. Summary

The case study began with a high level abstract specification of a network packet filter. Through a series of design steps—each one of which was guided by domain knowledge rather than *Circus*—an implementation that corresponds to a *Handel-C* program was calculated. The correctness of each major design step was verified using Z/Eves and FDR. By manipulating the processes into forms applicable to the process splitting law, calculating concurrency in the specification proved to be relatively straightforward; however some of the manipulations—specifically those where assumptions were made about a global clock—relied heavily in places on post-mortem verification. The structure, and rigour, of the development is the most advanced recorded in the the literature for *Handel-C* and *Circus*.

The intention was to capture the level of rigour and applicability of domain expertise that may be adhered to in an industrial development, and show that this level of rigour is both feasible and sufficient for large projects. This was achieved: in fact, an erroneous assumption in the original design was uncovered that testing alone had not exposed.

Due to its simplicity, the implementation has a natural mapping onto *Handel-C*; although as a formal semantics for *Handel-C* has not yet been approved[6], this final step is not as formal as that of [7] or [1]. Due to the nature of refinement in *Circus*, some of the traditional problems in *Handel-C* were naturally avoided: for instance, it should not normally be possible to derive a program where two processes attempt to assign to a variable concurrently. This leads to an interesting artifact in the model: although the *Handel-C* code may share access to variables—in particular read access—the *Circus* model may not. An idiom involving regular updates of local state was appealed to in order to emulate this read access. However, in the final code, there is no need to copy the state of the pipeline in the checksum process—it is *Circus* that requires this. This is clearly an important consideration for hardware area constraints; and is a problem in need of further attention.

Decisions about the design of the device, and where and how concurrency was introduced and exploited was governed by domain knowledge and empirical evidence, rather than solely by laws of *Circus*. The necessity of supporting application of domain knowledge is im-

[6] An item of work we are currently engaged in is a timed model of CSP that matches the timing semantics of *Handel-C*.

portant. Significant gains in the end product were made by targeting design steps at features of *Handel-C* and the FPGA. A different correct implementation could have been developed without this knowledge; but it may not have met the speed and area requirements which only become apparent after hardware has been built and tested. Early experiences, gained from empirical experiments, guided these judgements. A method of including wall clock speed, and hardware area, parameters explicitly into the design process may well make for a development method which becomes very cumbersome, and detracts from the elegance of the natural refinement laws. More work is needed to fully consider this.

The most significant achievement of this case study is that requirements have been met by drawing on expert domain knowledge; and that the correctness of applying this knowledge has been verified at every stage by drawing on formal techniques. This is a significant demonstration in the applicability of formal techniques to a typical engineering process.

The application was compiled and run on a Xilinx 40150 series FPGA which clocked at 20MHz; operating on traffic running at 160M-Bit/s—sufficiently fast to operate as a real time device on standard fast Ethernet. Sample dictionaries of several hundred IP addresses were used on genuine network traffic. The application found and identified the same packets in the stream as standard network monitoring utilities such as *snoop*.

Acknowledgements

The author would like to thank Steve Schneider, Jim Woodcock, Ana Cavalcanti, and Wilson Ifill for their technical guidance, assistance and suggestions with this work.

References

[1] A. L. C. Cavalcanti. *A Refinement Calculus for Z.* DPhil thesis, The University of Oxford, 1997.
[2] S. Kent and R. Atkinson. IP Authentication Header. Technical Report RFC-2401, The Internet Society, November 1998.
[3] Alistair McEwan, Jonathan Saul, and Andrew Bailey. A high speed reconfigurable firewall based on parameterizable FPGA based Content Addressable Memories. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 2, pages 1138–1144. CSREA Press, June 1999.
[4] Alistair A. McEwan. The design and verification of *Handel-C* programs. Technical report, Oxford University Computing Laboratory, 2001. Invited talk, DARPA 2001.
[5] Alistair A. McEwan. *Concurrent program development.* DPhil thesis, The University of Oxford, To appear.
[6] 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.
[7] Carroll Morgan. *Programming from Specifications*. International Series in Computer Science. Prentice-Hall, 1990.
[8] Behrooz Parhami. Architectural tradeoffs in the design of VLSI-based associative memories. *Journal of Microprocessing and Microprogramming*, 38:27–41, 1993.
[9] J. Postel. Internet Protocol. Technical Report RFC-791, The Internet Society, September 1981.
[10] Augusto Sampaio, Jim Woodcock, and Ana Cavalcanti. Refinement in *Circus*. In Lars-Henrick Eriksson and Peter Alexander Lindsay, editors, *FME 2002: Formal Methods—Getting IT Right*, pages 451–470. Springer-Verlag, 2002.
[11] Kenneth J. Schultz and P. Glenn Gulak. Architectures for large capacity CAMs. *INTEGRATION, the VLSI Journal*, 18:151–171, 1995.
[12] J. C. P Woodcock and Alistair A. McEwan. An overview of the verification of a *Handel-C* program. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*. CSREA Press, 2000.