Technical Report No: 06/02

Refining Specifications to Programmable Logic

Adrian Hilton
Jon G. Hall

31 October 2003

Department of Computing
Faculty of Mathematics and Computing
The Open University
Walton Hall,
Milton Keynes
MK7 6AA
United Kingdom

http://computing.open.ac.uk

Electronic Notes in Theoretical Computer Science 70 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume70.html 13 pages

Refining Specifications to Programmable Logic

Adrian Hilton [1,3]

Praxis Critical Systems Ltd
20 Manvers St., Bath BA1 1PX

Jon G. Hall [2]

The Open University, Walton Hall
Milton Keynes MK7 6AA, UK

Abstract

Combined hardware/software systems are increasingly being used for safety-critical systems, with hardware taking processing load off the software. To attain the necessary safety integrity levels, new safety standards require that the correctness arguments for safety-critical hardware and software are developed together with the same rigour as for software alone.

In this paper we describe work in progress on the continuing development of such a notation and proof system. Based on process description using Synchronous Receptive Proof Theory, we propose refinement rules for developing a specification into an SRPT implementation.

As illustration, we demonstrate the full formal refinement of a 2^k bit carry look-ahead adder into a Pebble implementation, and test the implementation.

1 Introduction

Programmable logic devices (PLDs) are increasingly important components of complex and safety-critical systems. Standards such as the emerging UK Defence Standard 00-54 [9] and IEC 61508 [5] now require developers to reason about the safety and correctness of programmable logic devices in such systems. In particular, UK Defence Standard 00-54 requires the ability to reason analytically about the safety and correctness of programs loaded into the PLDs using a formal notation and rigorous proof system. In addition, programming such devices is becoming more like programming conventional

[1] Email: adi@suslik.org
[2] Email: j.g.hall@open.ac.uk
[3] Support from Teleca Ltd. and Praxis Critical Systems Ltd. is gratefully acknowledged.

microprocessors in terms of program size, complexity, and the need to clarify a program's purpose and structure.

To address this, in [3] we described a process for developing software for programmable logic in safety-critical systems with the same rigour as for conventional software for these systems. A key component of the process was the ability to reason analytically about the correctness of the programmable logic software.

Much research has established formal refinement as of appropriate rigour for safety critical systems development ([11] is a pre-eminent example, distinguished by its completeness). Our current work focusses on establishing appropriate rigour for the process; in this paper, we report on a formal refinement calculus for high-integrity software running on a system with both a conventional CPU and programmable logic devices. Through the refinement calculus, we will be able to address the concerns of rigour.

1.1 Paper Structure

In Section 2 we describe SRPT, show how it can be used to describe logic blocks and how it maps onto the Pebble hardware description language. In Section 3 we show how to specify SRPT processes and refine them to an implementation. We demonstrate this in Section 4 with the specification and refinement of a 2^k-bit carry look-ahead adder. Finally we draw conclusions and outline further work in the field.

2 Synchronous Receptive Process Theory

Synchronous Receptive Process Theory (SRPT) was developed by Barnes and is described in [2]. It is a process algebra in the CSP family. It differs from CSP by being *synchronous* — more than one event may happen at once — and *receptive* — processes cannot refuse to accept input events.

An SRPT system consists of a set of events Σ and a set of processes $\{P_i\}$. Each process P has an input alphabet $\iota P \subseteq \Sigma$ and output alphabet $oP \subseteq \Sigma$ which must be disjoint; their union non-null. To make the link with PLDs, the occurrence of events in Σ will represent a high on an appropriate input or output line.

Each process P has associated with it a minimum-delay length $k > 1$. At each time step t, process P observes which events in ιP have occurred. At time $t + k$ it outputs a subset of oP which may depend on any previous ιP events at or before time t.

A process has full control over events in its output alphabet but no control over events in its input alphabet. Processes communicate by sharing events. An event in oP and ιQ allows P to send information to Q, but not vice versa. No event $s \in \Sigma$ may appear in the output alphabet of more than one process. This corresponds well to our input line/output line interpretation.

2.1 Notation

Processes are defined using the following grammar. It has been restricted from [2] for this paper; we require only those components relevant to the modelling of synchronous logic blocks.

$$
\begin{array}{rll}
P ::= & x \in \mathit{Var} & \text{process variable} \\
| & [!O?X \rightarrow P_X] & \text{output prefix} \\
| & P \parallel P & \text{parallel composition} \\
| & P \setminus O & \text{hiding} \\
| & P[S] & \text{renaming}
\end{array}
$$

We do not define the semantics of the operators formally; however, the following informal description will suffice for this paper: $x \in \mathit{Var}$ is a process which is made explicit when we bind variable names to the space of processes. $P \setminus O$ behaves the same as P, but all the events in $oP \cap O$ are hidden. $P[S]$ renames the input and output events of P with the bijective renaming function $S : \Sigma \rightarrow \Sigma$. The parallel operator allows a number of processes to proceed in lockstep. They may share events to communicate, as noted above. The output prefix form can be read as 'output events O, receive input events X and then behave like P_X.'

When we later use expressions for choice of process in the output form, this corresponds to binding process variable names to elements in the process space.

2.2 Semantics

Barnes defines process semantics in terms of their possible *traces* i.e. the set of sequences of sets of events which they produce. As usual in such approaches, traces are prefix-closed; however, there is an additional constraint that the environment can offer *any* subset of input events at a given step, and the output at that step must be independent of the input at that step. Of course, in general, subsequent behaviour may depend on that input.

The traces of process P are denoted $\mathcal{T}_{\mathcal{R}}[\![P]\!]$. The space of processes is denoted RM. SRPT defines a semantic function $\mathcal{M}_{\mathcal{R}} : \mathit{SRPT} \rightarrow RM$, where each process P with variable binding σ is mapped to its input and output alphabets and traces:

$$\mathcal{M}_{\mathcal{R}}[\![P]\!]\sigma = (\iota[\![P]\!]\sigma, o[\![P]\!]\sigma, \mathcal{T}_{\mathcal{R}}[\![P]\!]\sigma)$$

Barnes defines the functions ι, o and $\mathcal{T}_{\mathcal{R}}$ over the syntax of the non-recursive terms of SRPT, and then extends $\mathcal{M}_{\mathcal{R}}$ to cover the recursive terms.

The details of the formal semantic model of SRPT are omitted for the sake of brevity; the interested reader is referred to [2, §5.3-5.4]. The semantics are relatively simple since SRPT need not handle the concept of refusing events.

2.3 Equivalence

Two processes P, Q with identical trace sets are considered equivalent.

P and Q are *observationally congruent*, according to Milner, if $F(P)$ is observationally equivalent to $F(Q)$ for any environment F. In SRPT, this environment corresponds to a sequence of subsets of process input events.

If P and Q are equivalent then they must have the same input alphabet, since by the trace well-formedness rules any event in the input alphabet may be offered at any step.

Since we have banned non-determinism, an environment $F = \langle F_1, F_2, \ldots \rangle$, where $F_i \subseteq \iota P$, determines exactly the subset of traces $S_P \subseteq \mathcal{T}_{\mathcal{R}}[\![P]\!]$ which may be observed from P in that environment. S_P is then defined by:

$$s \in S_P \Rightarrow \forall\, i \geq 0 \cdot (s[i] \cap \iota P) = F_i$$

Since $\iota P = \iota Q$ and $\mathcal{T}_{\mathcal{R}}[\![P]\!] = \mathcal{T}_{\mathcal{R}}[\![Q]\!]$, $S_Q = S_P$ and hence P and Q are observationally congruent.

2.4 Transformation to Implementation

One reason for choosing SRPT as a description language was its closeness to the Pebble hardware description language [7]. Pebble is a simple structural language used to program reprogrammable logic devices. The basic correspondence is between the SRPT process and the Pebble `block`. Events in the alphabets of process P correspond to the `WIRE` parameters of the block. Hidden events correspond to block internal wires.

Parallel processes correspond to a block instantiating a number of other blocks, connecting them by relating input and output wires in the same way that the processes share events. A renaming corresponds to a single block instantiation with the instantiation parameters defining the renaming.

The output prefix process form may be parametrised by a function of the previous inputs. An example would be a latch, where the output would depend on the last input which set the latch's value. This is translated into Pebble by a state-generating `block`, feeding into a lookup table `block` to generate the outputs and back into itself to change state.

3 Specification and Proof

Specification of digital logic circuits has been made by a wide range of formalisms, for instance CSP [4] and its timed and synchronous variants. Clocked circuits may be modelled algebraically by models such as CIRCAL [8].

There are many approaches to refinement; for instance, see Back [1] and Morgan [10]. Of particular relevance to our approach in being based on reactive action systems is the refinement of Back [1]. There, refinement is defined in terms of traces. We follow a broadly similar form in our semantics, although

the deterministic nature of our SRPT subset means that we avoid some of the complications encountered by Back.

3.1 Specification frame

Refinement takes an abstract specification to a syntactic form which may be implemented directly. The syntactic form of our abstract specification is similar to that of Morgan:

$$\iota X : o Y : [[\,\mathbf{pre}\,]_t, [\,\mathbf{post}\,]_{t+k}]$$

representing the specification "for the process with input alphabet containing X and output alphabet containing Y, (at all times t) if **pre** is true at time t then at time $t+k$ **post** is true." [4] t and k are necessary because an SRPT process computes in a 'pipelined' (systolic or overlapping) manner; t marks a point where a computation starts and k expresses the length of the pipeline which produces the result. A specification clearly defines a set of traces and so can be considered an SRPT process (if an abstract one!). Our job is to synthesize a concrete SRPT process that has the same traces.

As an example, a 1-cycle AND gate with input events $X = \{x_1, x_2\}$ and output events $Y = \{y\}$ would have precondition **true** and postcondition

$$[x_1 \wedge x_2]_t = [y]_{t+1}$$

The possible traces of this process include:

$$\langle \{x_1\}, \{x_1, x_2\}, \{x_2, y\} \rangle, \langle \{x_1, x_2\}, \{x_1, x_2, y\}, \{y\} \rangle, \langle \rangle$$

An example of an incorrect trace is $\langle \{x_1\}, \{x_2, y\} \rangle$.

The following rules apply to the frame:

(i) **pre** may only relate variables in X.

(ii) the postcondition **post** may relate variables in both X and Y.

(iii) the highest time index of a variable x in **pre** must be less than the lowest time index of an output variable y in **post**, where the time indices of x and y share a common quantified time variable t.

(iv) where variables $[x \in X]_{t+i}$ and $[y \in Y]_{t+j}$ are related in **post**, $i < j$.

3.2 Refinement Relation

Following Back [1], for processes P and Q, the refinement relation $P \sqsubseteq Q$ is defined over the process trace sets as:

$$\mathcal{T}_{\mathcal{R}}[\![Q]\!] \subseteq \mathcal{T}_{\mathcal{R}}[\![P]\!]$$

[4] Indeed, we define the notation $[A]_t$ to mean A evaluated at time t

i.e. any valid trace of Q is a valid trace of P.

Given a specification $S = \iota X : oY : [[\,\mathbf{pre}\,]_t, [\,\mathbf{post}\,]_{t+k}]$, we define its traces $\mathcal{T}_{\mathcal{R}}[\![S]\!]$ as:

$$f \in \mathcal{T}_{\mathcal{R}}[\![S]\!] \Leftrightarrow \forall\, 0 \leq t < (\#f - k) \cdot [\,\mathbf{pre}\,(f)]_t \Rightarrow [\,\mathbf{post}\,(f)]_{t+k} \tag{1}$$

If we are to refine S into processes then we need to show that $F = \mathcal{T}_{\mathcal{R}}[\![S]\!]$ satisfies the SRPT closure conditions. We also need to show that at any step the process represented by F can accept any input, and the input cannot affect the output at that step.

The empty trace $\langle\rangle$ is trivially in F. F is prefix-closed since if $f \in F$ and $f = s \frown \langle Z \rangle$ then

$$\forall\, 0 \leq t < ((\#f - k) - 1) \cdot [\,\mathbf{pre}\,(s)]_t \Rightarrow [\,\mathbf{post}\,(s)]_{t+k}$$

and $\#s - k = (\#f - k) - 1$.

To demonstrate that the process represented by F can accept any input at any step without affecting that step, let $f = s \frown \langle Z \rangle$. Since $f \in \mathcal{T}_{\mathcal{R}}[\![F]\!]$, the RHS of Equation 1 must hold. Then let $r = s \frown \langle\, V \cup U \rangle$ where $V = (Z \cap Y)$ is the set of output events at that timestep and $U \subseteq X$ is any set of input events. We need to show that $r \in \mathcal{T}_{\mathcal{R}}[\![F]\!]$.

Since s prefixes f, we know that $s \in F$. We need then only show that:

$$[\,\mathbf{pre}\,(r)]_{\#r-(k+1)} \Rightarrow [\,\mathbf{post}\,(r)]_{\#r-1}$$

The rules on pre- and post-condition time indices restrict **post** from specifying outputs at t, or from $t + k$ onwards, and similarly restrict **pre** from specifying inputs from $t+k-1$ onwards. Hence any events in U (at time index $\#r - 1$) cannot affect the precondition. The output events V do not change from f to r, hence the postcondition is similarly unaffected, and therefore the third closure condition is met.

This allows us to treat process specifications as actual processes in the following refinement rules.

3.3 Refinement Rules

The refinement calculus we define is of a form similar to Morgan [10]. It is based on the following rules, which allow a more abstract description to be replaced by one closer to SRPT processes. These rules form the basis of the formal transformation from specification to code, and so are crucial to the definition of the high-integrity transformation. Later in the paper, they are applied to a case study which shows precisely how a PLD can be refined from a high-level specification.

A number of other rules (e.g. weaken precondition, strengthen postcondition, expand frame) have been defined, but are omitted here since they are not used in our example.

Refinement 1 *Stateless 1-bit function*

If process $CELL_f$ defines a stateless 1-bit cell executing logic function f:

$$\begin{aligned} \iota CELL_f &= A, oCELL_f = \{y\} \\ CELL_f(R) &= [!R?X \rightarrow CELL_f(f(X \cap A))] \\ CELL_f &= CELL_f(\emptyset) \end{aligned}$$

then:

$$\iota A : o\{y\} : [\,\mathbf{true}\,, [y]_{t+1} = f([A]_t)]$$

$$\sqsubseteq CELL_f$$

Variants of *CELL* are the basic constructors of combinatorial logic as they are a representation of primitive blocks in Pebble. This law is then key to turning a specification into a SRPT process.

Refinement 2 *Parallelism*

If there are two parts of the output of a process which are independent then the process can be split into two, each computing one of the parts. More formally: if $\mathbf{post}_1$ and $\mathbf{post}_2$ share no input or output variables, then

$$\begin{aligned} & \iota X : o(Y \cup Z) : [\,\mathbf{pre}\,, \mathbf{post}_1 \wedge \mathbf{post}_2] \\ \sqsubseteq\, & \iota X : oY : [\,\mathbf{pre}\,, \mathbf{post}_1] \\ & \parallel \iota X : oZ : [\,\mathbf{pre}\,, \mathbf{post}_2] \end{aligned}$$

Refinement 3 *Contract frame*

If some of the input variables are irrelevant to the outputs, we can remove them. More formally: if all of the input variables of $\mathbf{post}$ are contained in B then

$$\begin{aligned} & \iota X : oY : [\,\mathbf{pre}\,, \mathbf{post}\,] \\ \sqsubseteq\, & \iota X \cap B : oY : [\,\mathbf{pre}\,, \mathbf{post}\,] \end{aligned} \tag{2}$$

Refinement 4 *Introduce intermediate*

We may split a process for which an 'intermediate calculation' exists. More formally:

$$g([Y]_{t+2}, [X]_t) \equiv k([Y]_{t+2}, [Z]_{t+1}) \wedge j([Z]_{t+1}, [X]_t)$$

implies

$$\begin{aligned} \iota X : oY : [\,\mathbf{pre}\,, g([Y]_{t+2}, [X]_t)] \equiv\, & (\iota Z : oY : [\,\mathbf{true}\,, k([Y]_{t+2}, [Z]_{t+1})] \\ & \parallel \iota X : oZ : [\,\mathbf{pre}\,, j([Z]_{t+1}, [X]_t)]) \\ & \setminus Z \end{aligned}$$

where $\equiv$ means "refines in both directions". This is an example of where two processes are the refinement of each other.

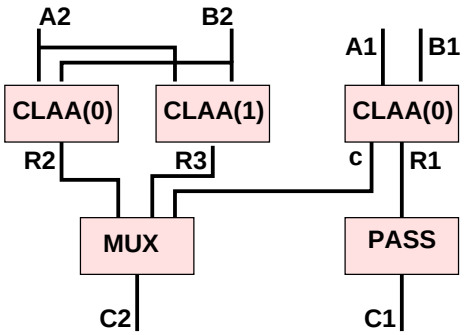

Fig. 1. Carry look-ahead adder structure

3.4 Refinement Process

The refinement process starts with a specification, but its end is less well defined. We need a concept of what is an implementation. This will depend on the basic components of our target device.

For a conventional programmable logic device, a basic component is likely to be a cell which computes a logic function of a small number of inputs and outputs the result at the next time step. More complex PLDs may also incorporate latches and small blocks of ROM or RAM.

At the start of the refinement we must list the processes which we regards as "terminal" i.e. directly corresponding to a PLD component. The refinement is complete when every part of the specification has been refined to a terminal process.

At the end of the refinement we will transform each terminal process into a PLD component, connect them appropriately according to the shared events, and assert that this PLD program satisfies our original specification.

4 Carry Look-ahead Adder

As a demonstration, we will specify and refine a 2^k-bit carry look-ahead adder where our primitive blocks are stateless 1-output cells with 2 or 3 inputs.

4.1 Definition

A *carry look-ahead adder* is an adder which is optimised for execution time rather than area. It works by splitting an addition into two halves (high and low bits), and carrying out two calculations for the high bit – one for if a carry is received, one for if it isn't. A multiplexer then selects the correct high bits calculation.

Figure 1 shows a possible structure of one of these devices. This is the structure that we shall aim to develop in the following refinement. In the figure, each $CLAA(x)$ denotes a half-size carry look-ahead adder; the structure is recursive.

4.2 Specification

For an $n = 2^k$ bit adder, $CLAA_k$, the specification is:

$$\iota(A \cup B) : oC : [\,\mathbf{true}\,, [\mathbb{N}(C)]_{t+1+k} = [\mathbb{N}(A) + \mathbb{N}(B)]_t]$$

where $\mathbb{N}(X)$ maps the subsets of X onto the natural number given by the binary representation of the events. A and B must contain n events, C must contain $n + 1$.

We will in fact find it useful to specify and refine the processes $CLAA_k(x)$ for $x \in \{0, 1\}$, where x is added onto the end result.

Note that the specification requires that the computation complete in $1 + k$ time steps. A simple ripple-carry adder could not in general satisfy this specification since it takes time linear in 2^k to complete.

4.3 Basic Gates

If we set k to 0, and hence n to 1, we get a half adder:

$$HADD = \iota\{a, b\} : o\{c, s\} : [\,\mathbf{true}\,, [2c + s]_{t+1} = [a + b]_t]$$

which may be implemented with a pair of stateless 1-bit functions. We note that this takes two of our 2-input, 1-output cells. We assume that the only cells available for construction are 2-input, 1-output and 3-input, 1-output. This will restrict what we regard as "final code" in our refinement.

We will also want a pass gate (for delays) and a 1-bit choice gate. These have the following specifications:

$$PASS = \iota\{x\} : o\{y\} : [\,\mathbf{true}\,, [y]_{t+1} = [x]_t]$$

$$MUX = \iota\{a, b, c\} : o\{y\} : [\,\mathbf{true}\,, [y]_{t+1} = [(b \wedge c) \vee (a \wedge \neg c)]_t]$$

4.4 Refinement

We proceed by induction on k. We assume that we have complete implementations for all processes $CLAA_k(y)$ for $y \in \{0, 1\}$. Let $n = 2^k$. Now we start with specifying $CLAA_{k+1}(x)$:

$$\iota(A \cup B) : oC : [\,\mathbf{true}\,, [\mathbb{N}(C)]_{t+2+k} = [\mathbb{N}(A) + \mathbb{N}(B) + x]_t]$$

Let $A = A_1 \cup A_2$ where $A_1 = \{a_1, \ldots, a_n\}$ and $A_2 = \{a_{n+1}, \ldots, a_{2n}\}$. Define B_1, B_2, C_1 similarly and $C_2 = \{c_{n+1}, \ldots, c_{2n+1}\}$. From now on we abbreviate the arithmetic by referring to direct addition of event sets. The above equation then rewrites to:

$\iota(A_1 \cup A_2 \cup B_1 \cup B_2) : o(C_1 \cup C_2) :$

$[\ \mathbf{true}\,,$

$[C_1]_{t+2+k} = ([A_1 + B_1 + x]_t)\,\mathbf{mod}\,2 \qquad \wedge$

$[C_2]_{t+2+k} = ([A_1 + B_1 + x]_t)\,\mathbf{div}\,2 + [A_2 + B_2]_t \quad]$

Applying Refinement Law 4 we introduce the intermediate event set $(R_1 \cup R_2 \cup R_3 \cup \{c\})$, the components of which have respective sizes n, $n+1$, $n+1$ and 1. The above then rewrites to:

$(\ \iota(A_{1,2} \cup B_{1,2}) : o(R_{1,2,3} \cup \{c\}) :$

$[\ \mathbf{true}\,,$

$[R_1]_{t+1+k} = ([A_1 + B_1 + x]_t)\,\mathbf{mod}\,2 \qquad \wedge$

$[R_2]_{t+1+k} = [A_2 + B_2]_t \qquad \wedge$

$[R_3]_{t+1+k} = 1 + [A_2 + B_2]_t \qquad \wedge$

$[c]_{t+1+k} = ([A_1 + B_1 + x]_t)\,\mathbf{div}\,2 \qquad]\ (1)$

$\|\ \iota(R_{1,2,3} \cup \{c\}) : oC_{1,2} :$

$[\ \mathbf{true}\,,$

$[C_1]_{t+1} = [R_1]_t \qquad \wedge$

$[C_2]_{t+1} = [(R_3 \wedge c) \vee (R_2 \wedge \neg c)]_t \qquad]\ (2)$

$)\ \setminus\ (R_{1,2,3} \cup \{c\})$

To show that this refinement is correct we need to show that the values for C_1 and C_2 are equivalent before and after the refinement.

$[C_1]_{t+1} = [R_1]_t$

$[R_1]_{t+1+k} = ([A_1 + B_1 + x]_t)\,\mathbf{mod}\,2$

$\Rightarrow [C_1]_{t+2+k} = ([A_1 + B_1 + x]_t)\,\mathbf{mod}\,2$

$[C_2]_{t+1} = [(R_3 \wedge c) \vee (R_2 \wedge \neg c)]_t$

$[R_2]_{t+1+k} = [A_2 + B_2]_t$

$[R_3]_{t+1+k} = 1 + [A_2 + B_2]_t$

$[c]_{t+1+k} = ([A_1 + B_1 + x]_t)\,\mathbf{div}\,2$

$\Rightarrow [C_2]_{t+1} = [A_2 + B_2]_t + ([A_1 + B_1 + x]_t)\,\mathbf{div}\,2$

which is as required.

We take each of the refined processes in turn for further refinement.

$(1) \sqsubseteq$ (via Refinement Law 2) :

$$\begin{array}{l} \iota(A \cup B) : o(R_1 \cup \{c\}) : \\ [\ \mathbf{true}, \\ \quad [R_1]_{t+1+k} = ([A_1 + B_1 + x]_t)\,\mathbf{mod}\,2 \wedge \\ \quad c = ([A_1 + B_1 + x]_t)\,\mathbf{div}\,2 \qquad\qquad]\ (3) \\ \| \ \iota(A \cup B) : oR_2 : \\ [\ \mathbf{true}, [R_2]_{t+1+k} = [A_2 + B_2]_t \qquad]\ (4) \\ \| \ \iota(A \cup B) : oR_3 : \\ [\ \mathbf{true}, [R_3]_{t+1+k} = 1 + [A_2 + B_2]_t \quad]\ (5) \end{array}$$

Here (3), (4) and (5) are equivalent to processes $CLAA_k(x)$, $CLAA_k(0)$ and $CLAA_k(1)$ respectively. We can contract the frames to remove A_1, B_1 from (4), (5) and A_2, B_2 from (3).

$(2) \sqsubseteq$ (via Refinement Law 2) :

$$\begin{array}{l} \iota(R_{1,2,3} \cup \{c\}) : oC_{1,2} : \\ [\,\mathbf{true}, [C_1]_{t+1} = [R_1]_t \qquad\qquad]\ (6) \\ \| \ \iota(R_{1,2,3} \cup \{c\}) : oC_2 : \\ [\,\mathbf{true}, [C_2]_{t+1} = [(R_3 \wedge c) \vee (R_2 \wedge \neg c)]_t\ ]\ (7) \end{array}$$

(6) is equivalent to n parallel $PASS$ cells between R_1 and C_1. (7) is equivalent to $n+1$ parallel MUX cells, choosing from R_2 and R_3 using c, sending to C_2. Again we can contract frames to remove $R_{2,3}$ from (6) and R_1 from (7).

We can now collate the refinement components to produce:

$$\begin{array}{lll} \iota(A \cup B) : oC : & [\mathbf{true}, [C]_{t+2+k} = [A + B + x]_t] & \\ \sqsubseteq & & \\ \quad (& CLAA_k(x)[A_1, B_1][R_1, c] & (3) \\ \quad \| & CLAA_k(0)[A_2, B_2][R_2] & (4) \\ \quad \| & CLAA_k(1)[A_2, B_2][R_3] & (5) \\ \quad \|_{i=1}^{n} & PASS[r_i][c_i] & (6) \\ \quad \|_{i=1}^{n+1} & MUX[r_{n+i}, r_{2n+i}, c][c_{n+i}] & (7) \\ \quad) & \setminus\ (R_{1,2,3} \cup \{c\}) & \end{array}$$

5 Discussion, Conclusions and Further Work

We have combined the process model of Barnes's Synchronous Receptive Process Theory with a modified form of Morgan's refinement calculus syntax to produce a refinement calculus of our own. This enables time-based specification of process behaviour and refinement to an implementation in simple processes. There is a clear map from these processes to implementations in a language such as Pebble.

With a relatively short formal derivation we have produced a full implementation for a family of arithmetic functions, parametrised by size, and demonstrated that the calculations complete in the specified time. This has been done using a predefined set of simple gates *HADD*, *PASS*, and *MUX*.

5.1 Proof means no testing?

Bearing in mind Knuth's famous quote "Beware of bugs in the above code; I have only proved it correct, not tried it" [6] we implemented the above structure in a simple Pebble simulator written in Perl and tested it with random input data.

Knuth was proven prudent. In the original refinement, (3) had mistakenly been asserted equivalent to $CLAA_k(0)$ rather than $CLAA_k(x)$. The tests detected this, it was corrected, and the tests rerun. No errors were found in the corrected version for values of k from 0 to 5.

This is more a comment on the methodology that we used to arrive at our starting point rather than the subsequent refinement. In essence, no matter how good a refinement, it can only be as good as the starting specification from which it was derived. To validate a system against its specification requires testing, as no internal 'fitness for purpose' test can be sufficient.

5.2 Further Work

The small set of refinement rules given above is suitable for simple processes, but more complex refinements will benefit from more powerful and expressive rules. We are developing the design of a substantial PLD construct and will be using this experience to gain insight into the forms of expression which we need to specify and refine.

References

[1] Ralph-Johan Back and Joakim von Wright. Trace refinement of action systems. In *International Conference on Concurrency Theory*, pages 367–384, 1994.

[2] Janet E. Barnes. A mathematical theory of synchronous communication. Technical report, Oxford University Computing Laboratory, 1993.

[3] Adrian Hilton and Jon Hall. On applying software development best practice to FPGAs in safety-critical systems. In Reiner W. Hartenstein and Herbert Grünbacher, editors, *Proceedings of the 10th International Conference on Field Programmable Logic and Applications (FPL'00)*, volume 1896 of *Lecture Notes In Computer Science*, pages 793–796. Springer-Verlag, August 2000.

[4] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[5] IEC Standard 61508, March 2000. Functional Safety of Electrical / Electronic / Programmable Electronic Safety-Related Systems.

[6] Donald E. Knuth. Notes on the van Emde Boas construction of priority deques: An instructive use of recursion. Memo to Peter van Emde Boas, March 1977.

[7] Wayne Luk and Steve McKeever. Pebble — a language for parametrised and reconfigurable hardware. In R. Hartenstein and A Keevallik, editors, *Proceedings of the 8th International Workshop on Field Programmable Logic (FPL'98)*, volume 1482 of *Lecture Notes In Computer Science*, pages 9–18. Springer-Verlag, September 1998.

[8] G. McCaskill and G. Milne. Hardware description and verification using the CIRCAL system. Technical Report HDV-24-92, Department of Computer Science, University of Strathclyde, June 1992.

[9] Interim Defence Standard 00-54 issue 1, March 1999. Requirements for Safety Related Electronic Hardware in Defence Equipment.

[10] Carroll Morgan. *Programming From Specifications.* Prentice-Hall, second edition, 1994.

[11] E.-R. Olderog, Anders P. Ravn, and Jens Ulrik Skakkebæk. Refining system requirements to program specifications. In *Formal Methods for Real-Time Computing*, pages 107–134. Wiley, 1996.