Technical Report No: 2001/01

Proving Safety Properties of FPGAs

Adrian Hilton
Jon G. Hall

2001

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

http://computing.open.ac.uk

Proving Safety Properties of FPGAs

Adrian Hilton
Teleca
88/89 High Street
Winchester, Hants
England
adi@suslik.org

Jon Hall
The Open University
Walton Hall
Milton Keynes
England
j.g.hall@open.ac.uk

ABSTRACT

FPGAs are increasing in complexity and being used as important components of safety-critical systems. Emerging safety standards require analytic reasoning to demonstrate the safety of FPGAs in such systems. This report describes a method which uses a synchronous process algebra to produce formal proof that an FPGA program satisfies safety properties, and demonstrates its use in the specification of safety functions for a safety-critical system.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering

General Terms

Synchronous receptive process theory

Keywords

SRPT, FPGA, safety critical systems

1. INTRODUCTION

Programmable logic devices (PLDs)are increasing in size and complexity. From their origins as more flexible versions of Progrmmable Logic Arrays (PLAs), they are now able to compute complex and useful calculations at high speeds. Becker *et al*[2] describe an implementation of a CDMA (Code Division Multiple Access) receiver, a key component of 3rd generation mobile phones, on a new design of dynamically reconfigurable coarse-grained FPGA. This illustrates how reconfigurable devices can perform an important computation to take significant processing load off a conventional modern microprocessor.

For some years the development and testing of software in high-integrity systems has been the subject of stringent safety standards. In the past two years new emerging standards have started to require similar rigour in the development of safety-related electronic hardware, such as FPGAs, due to these devices being used to perform safety-critical functions within such systems.

This paper proposes a method for description of an programmed LUT (Look-Up Table) FPGA using a process algebra, and shows how the resulting description can be used to make rigorous proofs about certain properties of the system.

Section 2 of this paper describes the current issues of safety and correctness in high-integrity systems. Section 3 describes the synchronous receptive process algebra SRPT. Section 4 shows how SRPT can be used in specification and proof of properties of simple FPGA elements. Section 5 gives an example of the proof of safety properties of a system safety monitor. Finally, section 6 summarises the work done to date and outlines future directions for the work.

2. SAFETY

2.1 Definitions

In this paper we make use of the following definition:

> a *safety-critical system* is a collection of components acting together where interruption of the normal function of one or more components may cause injury or loss of life.

The interested reader is referred to Leveson [10] pp 156, 181 for the details of this definition. Note that a failure need not lead to injury or loss of life for the system to be safety-critical; it is enough that the possibility exists. It is important to note that the *context* of a system affects its criticality; a system is not safety-critical until it is powered up in its intended environment and is able to cause injury by its failure.

Failure is the nonperformance or inability of the system or component to perform its intended function for a specified time under specified conditions. A *fault* is a higher-order event; all failures are faults, but not all faults are failures. For example, a fault can caused by a component behaving correctly, not failing, but reacting to an erroneous external signal. (Leveson pp 172-173)

2.2 Assessing Integrity

Integrity in the context of safety-critical systems and their components is measured in terms of the probability of total or partial failure. A high-integrity system is expected to fail with lower frequency than a low-integrity system.

The emerging European safety standard IEC 61508 [8] specifies four Safety Integrity Levels – SIL 1, SIL 2, SIL 3 and SIL 4 – which express the expected frequency of a dangerous failure in a system. SIL 4, the highest integrity level, specifies a frequency of less than 1 failure per 10^4 demands for a low-demand mode of operation, or less than 1 failure per 10^8 hours for high-demand or continuous operation.

There is normally a requirement for all components of a system at SIL N to be SIL N themselves. However, it is possible to argue for a reduced SIL for a specific component if it can be shown that the component has a limited effect on the system. A 'white box' safety analysis procedure, such as that described in [14], aims to trace the output data of each component through the system and determine whether it can contribute to any predetermined system hazard. As lower SILs will, in general, be cheaper to produce than higher ones, by reducing the required SIL of the component, the required development effort can be reduced accordingly.

2.3 Relevant Standards

There are many standards relevant to the development of safety-critical systems. Those chosen for a particular project will depend on the requirements of the customer and the legal system of the country or countries in which it is to be used. However, there are few widely-used standards which are specifically intended to cover the use of complex digital logic in safety-critical systems. The two standards presented below are the most important European standards for this area at the moment.

IEC Standard 61508 [8] (abbreviated to as IEC 61508) is intended to apply across multiple industry sectors, setting out a generic safety management approach for systems with electrical, electronic or programmable electronic components. It addresses *functional safety* in systems, i.e. those safety properties that depend on the system functioning correctly.

Part 2 of IEC 61508 contains the safety requirements for any electrical, electronic or programmable components of the system. It specifies those aspects of hardware failure which should be addressed by the developer, for instance that PLDs need to detect such hardware faults as stuck-at failures for registers, bus faults and welded-together contacts. Techniques such as those described in [13] could be used to analyse PLDs for some classes of these failures.

The developer must also consider properties of the 'conventional' software in the system (that software executed on one or more microprocessors), such as correct 'watchdog' operation and information redundancy, with the analysis list determined by the system's SIL. Interestingly, the *programmable* part of the PLDs in such systems is not addressed in detail; there are requirements for aspects of the design to be analysed, but no concrete requirements for implementation language or related aspects as there are for co

UK Defence Standard 00-54 [12] is a new interim standard for the use of safety-related electronic hardware (SREH) in UK defence equipment. It relates to systems developed under the UK Defence Standard 00-56 safety systems document [11] (or an equivalent international standard such as IEC 61508).

Defence Standard 00-54 contains the following recommendations which are of particular interest to developers wishing to incorporate PLDs in safety-critical systems.

> (§12.2.1) A formally defined language which supports mathematically based reasoning and the proof of safety properties shall be used to specify a custom design;
>
> (§13.4.1) Safety requirements shall be incorporated explicitly into the Hardware Specification using a formal representation; and
>
> (§13.4.4) Correspondence between the Hardware Specification and the design implementation shall be demonstrated by analytical means, subject to assumptions about physical properties of the implementation.

where 'custom design' refers to the non-standard components of the electronic component under examination, and in particular to an FPGA's program data. The standard's guidances provide more information about the motivation behind the standard:

> The principal concern which has caused this Interim Standard to be produced is that electronic hardware designs used in critical applications have been getting steadily more complicated [. . .] Therefore the focus of this Interim Standard is on analysis and proof to supplement test.

2.4 Summary

We see then that new safety standards require us to be able to reason analytically about the safety and correctness of programmable logic devices within safety-critical systems. The following section describes a mathematics-based model which allows us to do this.

3. SYNCHRONOUS RECEPTIVE PROCESS THEORY

The process algebra CSP [7] has been used successfully to demonstrate partial correctness of protocols and industrial parallel systems. Supporting tools such as FDR [3] allow semi-automatic analysis of relatively large and complex parallel systems, proving them free from deadlock and livelock. However, CSP falls short of the ideal for describing FPGAs since it is asynchronous (two events may not happen at the same time) and not receptive (CSP processes can refuse events, which does not reflect the reality of digital logic).

Synchronous Receptive Process Theory (SRPT) was developed by Barnes [1] by combining Receptive Process Theory [9] and CSP [7]. It is a process algebra, defining algebraic rules for combining processes. In contrast to CSP it is synchronous and receptive: SRPT processes may not refuse events. In this section we summarise the key features of SRPT, and show how it may be used to describe digital circuits and prove statements about their properties.

3.1 Basic SRPT

An SRPT system description has an alphabet Σ of events which are variables in the system. There are some processes $\{P_k\}$, for which each process $P \in \{P_k\}$ has an input alphabet ιP in Σ and output alphabet oP in Σ. For each P, ιP and oP must be disjoint, finite, and their union must be non-empty. ιP consists of the events to which process P may react, and oP the events which the process controls. There is a set *Var* of process variables, each of which will range over $\{P_k\}$.

Processes are defined using the following grammar:

$$\begin{array}{rcl} P ::= & \bot_{I,O} & \text{chaos} \\ \mid & x & \text{process variable} \\ \mid & P \sqcap P & \text{non-deterministic choice} \\ \mid & [!O\,?X \to P_X] & \text{output prefix} \\ \mid & P \parallel P & \text{parallel composition} \\ \mid & P \setminus O & \text{hiding} \\ \mid & P[S] & \text{renaming} \\ \mid & \mu\, x : I, O \bullet P & \text{recursion} \end{array} \tag{1}$$

In the above definition, O denotes a subset of the output alphabet oP, X denotes a subset of the input alphabet ιP and S is a bijection over Σ. P_X specifies a process in a family with the same alphabets as P, but where the processes are parameterised by subsets $X \in \mathbb{P}(\iota P \cup oP)$.

Since we will use the output prefix form extensively, it is worth providing an informal definition here. The above definition specifies that the process will, during the next timestep t, output all events in O and receive some set of events X from its input alphabet. From timestep $t + 1$ onwards it behaves exactly like process P_X.

Each definition of a process P is in terms of a reaction to input events (a subset of ιP). Unlike CSP, an SRPT process cannot refuse an event which is in ιP; it simply observes such events happening. What it can do is react to those events by signalling events in its output alphabet. The synchronous nature of SRPT means that processes may receive and output any number of events at once.

As a simple example, an AND gate has no control over its two inputs, but exerts control over its output according to the values of the inputs in the previous timestep. It cannot provide an output at time t which relates to inputs received at time t; there is always a delay before the reaction is visible.

Barnes [1] defines a set of axioms and derives laws for algebraic combination of terms from this grammar. For example:

$$\begin{array}{ll} \textbf{a-10}: & [!B\,?X \to P_X] \parallel [!C\,?Y \to Q_Y] \equiv \\ & [!(B \cup C)\,?Z \to P_{(Z\cup C)\cap\iota P} \parallel Q_{(Z\cup B)\cap\iota Q}] \end{array} \tag{2}$$

This states that when combining two output-prefixed processes, we initially see the combined output of both processes, which we would naturally expect. From then on P's behaviour may additionally be affected if its input alphabet includes one or more events from the output alphabet of Q, and vice versa.

We will not use non-deterministic choice in our process specifications; Barnes' axioms show that if it is not in a system to start with then it will never arise (it is never on the right hand side of an axiomatic expression if it is not on the left hand side).

As an example of an SRPT system definition, in the following subsection we will describe a 1-cycle n-bit adder with the SRPT algebra.

3.2 Example – Adder

An n-bit binary adder ADD_n may be defined, for any positive fixed n, within SRPT. We represent each wire coming into and out of the adder with an event in the process alphabets; input wires have events in the input alphabet, and output wires have events in the output alphabet. An event occurring at a specified time denotes a logical high voltage on that wire at that time; the lack of an event denotes a logical low voltage.

time	A	B	Z
0	a_1	b_2	$\emptyset$
1	a_1, a_2	b_1	z_1, z_2
2	a_2	b_2	z_3

Figure 1: Example run of $ADD_2(\emptyset)$

Given the following disjoint sets of events:

$$\begin{array}{l} A = \{a_1, \dots, a_n\}, B = \{b_1, \dots, b_n\} \\ Z = \{z_1, \dots, z_{n+1}\} \end{array} \tag{3}$$

then we define the main process in the system as:

$$\begin{array}{rcl} \iota ADD_n & = & A \cup B \\ oADD_n & = & Z \\ ADD_n(R) & = & [!R\,?X \to \\ & & ADD_n(X \cap A + X \cap B)] \\ ADD_n & = & ADD_n(\emptyset) \end{array} \tag{4}$$

where we choose an encoding of the natural numbers in each of A, B and Z, and define the $+$ operator to map pairs of encoded numbers to the encoding of their sum:

$$+ : \mathbb{P}A \times \mathbb{P}B \to \mathbb{P}Z \tag{5}$$

The above definition of ADD_n states that at each time tick t the ADD_n outputs events from Z taken from its parameter R, then works out its next parameter by adding the numbers represented by the A and B events in that time tick's input.

Note that our process definitions parametrise processes to tell them what to output. This effectively encodes state within the processes, though in this example state at time t never affects the process after time $t+1$. The actual process ADD_n is arbitrarily defined in our example to output 0 on the first clock tick.

An example "run" for $ADD_2(\emptyset)$ could be as shown in Figure 1. Note that the environment controls the subsets of A and B that appear; only the Z events are controlled by the process.

3.3 Composition

SRPT allows us to compose processes to form larger ones, in serial or in parallel. Parallel composition is done with the $\parallel$ operator, but the most useful composition is normally serial since this allows us to break down a calculation into multiple stages. There is no explicit serial composition operator in SRPT; instead, composition in a system is effected by renaming process alphabets so that output events in one process are input events in another process. Composition is a key tool to allow us to build complex systems out of relatively simple processes. We now present an example of composition.

An 8-bit adder $ADD8$ could be formed out of smaller adders as follows. Defining the process events:

$$\begin{array}{rcl} A & = & \{a_1, \dots, a_8\}, B = \{b_1, \dots, b_8\} \\ Z & = & \{z_1, \dots, z_9\} \\ H & = & \{h_1, \dots, h_{10}\} \end{array} \tag{6}$$

we get the alphabets:

$$\begin{array}{rcl} \iota ADD8 & = & A \cup B \\ oADD8 & = & Z \end{array} \tag{7}$$

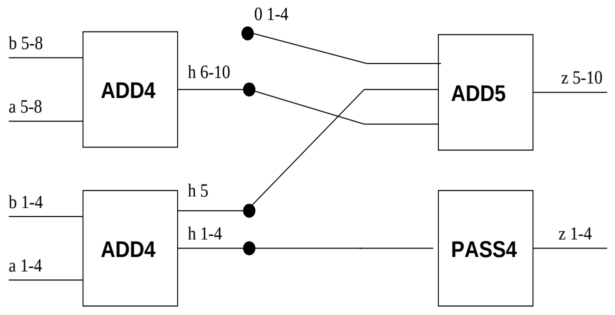

Figure 2: The decomposition of *ADD8*

and the process:

$$\begin{array}{ll} \mathit{ADD8} & = \\ \mathit{ADD4} & [a_{1-4} \rightarrow a_{1-4}, b_{1-4} \rightarrow b_{1-4}] \\ & [z_{1-5} \rightarrow h_{1-5}] \parallel \\ \mathit{ADD4} & [a_{1-4} \rightarrow a_{5-8}, b_{1-4} \rightarrow b_{5-8}] \\ & [z_{1-5} \rightarrow h_{6-10}] \parallel \\ \mathit{PASS4} & [a_{1-4} \rightarrow h_{1-4}] \\ & [z_{1-4} \rightarrow z_{1-4}] \parallel \\ \mathit{ADD5} & [a_{1-5} \rightarrow h_{6-10}, b_{1-5} \rightarrow h_5, 0, 0, 0, 0] \\ & [z_{1-6} \rightarrow z_{5-10}] \end{array} \qquad (8)$$

Figure 2 shows the decomposition of this process in terms of smaller adders and a pass block.

The process event binding notation

$$P[e_{1-M} \rightarrow x_{1-M}][f_{1-N} \rightarrow y_{1-N}] \qquad (9)$$

introduced here represents process P with input alphabet elements e_X renamed to x_X for all X in $1 - M$, and output alphabet changed similarly from f_X to y_X. Where an input event is given as 0, this indicates the *never* event which never occurs; similarly, 1 indicates the event which always occurs. In electrical terms this is the hard-wiring of the input to ground or to positive, respectively. Where an output event is given as 0 this indicates that the output is shorted to ground, effectively being ignored.

The process

$$\mathit{PASS}_n[a_{1-n}][z_{1-n}] \qquad (10)$$

simply passes its n inputs out unchanged after one cycle, acting here as a delaying mechanism.

In this system the events in H are "hidden" and fill an important role, analogous to subprogram variables in an imperative language. They provide storage space for the results of intermediate calculation, here including the low four bits of the result h_{1-4} which we can compute in the first cycle.

This 8-bit adder has a two-cycle delay compared to the one-cycle delay of the 4- and 5-bit adders which is assumed in the general ADD_n definition. The delay of an SRPT system is important, since we will find in composition that it is not easy to get the right data to the right place *at the right time*.

3.4 Denotational Semantics

Barnes defines the meaning of a system in SRPT in terms of process traces. This allows us to make rigorous analytical arguments about what does or not does happen in an SRPT system, to the level required by standards such as Defence Standard 00-54 [12] for the most safety-critical of systems.

In a given system, each process P with input alphabet I and output alphabet O has a semantics defined in terms of its set of traces:

$$RT_{I,O} = (\mathbb{P}(I \cup O))^* \qquad (11)$$

A trace $t \in RT_{I,O}$ consists of a sequence of sets of events: $t : \mathbf{seq}\,\mathbb{P}(I \cup O)$. Each element of the sequence corresponds to an (non-negative integer) time value of the global clock, and gives the events in I and O for that process which happen at that time.

Barnes defines the following closure conditions for a set T of traces for the above process P. In these conditions, s and r are traces, $\langle$ and $\rangle$ delimit a trace element and $\frown$ is the trace concatenator.

$$\begin{array}{ll} \mathbf{I} & \langle\rangle \in T \\ \mathbf{II} & s \frown r \in T \Rightarrow s \in T \\ \mathbf{III} & s \frown \langle X \rangle \in T \wedge Y \subseteq I \Rightarrow \\ & \quad s \frown \langle (X \cap O) \cup Y \rangle \in T \end{array} \qquad (12)$$

Conditions **I** and **II** are general properties of a trace semantics, requiring prefix closure of the set of traces.

Condition **III** requires explanation: it states that the environment can offer *any* subset of input events at a given step, and that output at that step must be independent of the input at that step.

RM, the underlying model for the language, is the set of all triples (I, O, T) where I and O are finite disjoint input and output alphabets of Σ and T is a set of traces which satisfies the above closure conditions. Each element $t \in RM$ then represents a unique process. We can group elements of RM by input and output alphabets: $RM^{I,O}$ denotes the set of all processes with input alphabet I and output alphabet O. RM_T is the set of all traces in RM.

The full derivation of semantic functions is given in [1] §5.4; again, we summarise.

The binding function BIND_R maps from a set *Var* of process variables to RM. This is what the user is effectively defining when he or she writes the process definitions and decides on the names of the process variables.

The semantic function $\mathcal{M}_\mathcal{R}$ maps each process term to an element of RM. The associated function $\mathcal{T}_\mathcal{R}$ maps process terms to RM_T, and ι, o map process terms to their input and output alphabets $\subseteq \mathbb{P}\Sigma$. Hence where σ represents an element of BIND_R we have:

$$\mathcal{M}_\mathcal{R}[\![P]\!]\sigma = (\iota[\![P]\!]\sigma, o[\![P]\!]\sigma, \mathcal{T}_\mathcal{R}[\![P]\!]\sigma) \qquad (13)$$

This can be read as "Given the user definition σ of P and associated SRPT processes, $\mathcal{M}_\mathcal{R}$ maps P onto its input alphabet, output alphabet, and the set of all traces valid for it."

All that then remains is to define ι, o and $\mathcal{T}_\mathcal{R}$ for each of the process terms. The interested reader is referred to Barnes for details of these definitions.

4. SRPT SPECIFICATION AND PROOF

To provide specifications for the actions of a process, we make statements about its traces. We take as a generic example the process for an n-bit FPGA cell $\mathit{CELL}_{n,f}$ pointwise computing a function $f : \mathbb{P}A \times \mathbb{P}B \rightarrow \mathbb{P}Z$, defined as follows:

$$\begin{aligned}
\iota CELL_{n,f} &= A \cup B \\
o CELL_{n,f} &= Z \\
CELL_{n,f}(R) &= [!R\,?X \rightarrow \\
&\quad CELL_{n,f}(f(X \cap A, X \cap B))] \\
CELL_{n,f} &= CELL_{n,f}(\emptyset) \qquad (14)
\end{aligned}$$

We assume that A, B and Z are disjoint subsets of Σ. The specification for $CELL_{n,f}$ would be as follows:

$$\begin{aligned}
&\forall\, t \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}]\!]\sigma : \\
&\forall\, i \in \mathbb{N} : \\
&\forall\, C \in \mathbb{P}A : \\
&\forall\, D \in \mathbb{P}B : \\
&(t[i] \cap A = C \quad \wedge \\
&t[i] \cap B = D \quad \wedge \\
&\#t > i) \quad \Rightarrow \\
&t[i+1] \cap Z \;=\; f(C, D) \qquad (15)
\end{aligned}$$

where $t[i]$ refers to the ith element of the trace t. This can be read as "if the trace at step i has input events C from set A and D from set B then the output events in the trace at step $i+1$ must represent the result of $f(C, D)$".

For correctness, we must prove:

Lemma 1.

$$\begin{aligned}
&\forall\, E \subseteq Z : \\
&\forall\, t \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!]\sigma : \\
&\#t > 0 \;\Rightarrow\; t[0] \cap Z = E \qquad (16)
\end{aligned}$$

i.e. the output set E passed as a parameter to $CELL_{n,f}$ will always appear as the first output.

This arises from Barnes' definition for output prefix:

$$\begin{aligned}
&\mathcal{T}_{\mathcal{R}}[\![[!B\,?X \rightarrow P_X]]\!]\sigma = \{\langle\rangle\} \cup \\
&\{\langle B \cup Y\rangle \frown s \mid Y \subseteq I \wedge s \in \mathcal{T}_{\mathcal{R}}[\![P_Y]\!]\sigma\} \qquad (17)
\end{aligned}$$

where $I = \iota[\![[!B\,?X \rightarrow P_X]]\!]\sigma$, i.e. the user's representation of the input alphabet of this process.

To prove Lemma 1, substitute

$$\begin{aligned}
B &= E \\
P_X &= CELL_{n,f}(f(X \cap A, X \cap B)) \qquad (18)
\end{aligned}$$

into Equation 17, giving:

$$\begin{aligned}
&\mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!]\sigma = \{\langle\rangle\} \cup \\
&\{\langle E \cup Y\rangle \frown s \mid Y \subseteq I \;\wedge \\
&s \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(f(Y \cap A, Y \cap B))]\!]\sigma\} \qquad (19)
\end{aligned}$$

and note that $Y \subseteq I$ cannot contain any elements of Z since Z is the output alphabet O and SRPT bans intersection of I and O. Therefore the output events in the first element of any non-null trace must be exactly E. □

We have shown that, as long as the right E is computed, $CELL_{n,f}(E)$ will give the correct outputs on the next step. This is an apparently simple result, but important so that the foundation for higher-level analytical work is solid.

Lemma 2.

$$\begin{aligned}
&\forall\, E \subseteq Z : \\
&\forall\, i \in \mathbb{N} : \\
&\forall\, t \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!]\sigma \quad \textit{s.t.} \\
&(t = s \frown r) \wedge (\#s = i) : \\
&\exists\, F \subseteq Z \quad \textit{s.t.} \\
&r \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(F)]\!]\sigma \qquad (20)
\end{aligned}$$

i.e. after any number i of steps of the trace t of $CELL_{n,f}(E)$, the trace r formed by steps $i+1$ onward of t is the trace of $CELL_{n,f}(F)$ for some F.

The proof is by induction on i. For $i = 1$ we must show:

$$\begin{aligned}
&\forall\, E \subseteq Z : \\
&\forall\, t \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!]\sigma \quad \text{s.t.} \\
&(t = \langle U\rangle \frown r) \quad \text{for} \quad \text{some } U, r \\
&\exists\, F \subseteq Z \quad \text{s.t.} \\
&r \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(F)]\!]\sigma \qquad (21)
\end{aligned}$$

Substituting

$$\begin{aligned}
F &= f(Y \cap A, Y \cap B) \\
U &= E \cup Y \\
r &= s \qquad (22)
\end{aligned}$$

into Equation 19 gives us:

$$\begin{aligned}
&\mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!]\sigma = \{\langle\rangle\} \cup \\
&\{\langle U\rangle \frown r \mid r \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(F)]\!]\sigma\} \qquad (23)
\end{aligned}$$

and we have the desired result that traces satisfying Equation 23 match the specification in Equation 21, proving the base case. □

The induction step is to show that if Equation 20 is valid $\forall\, i \in 1..N$ then it is valid for $i = N+1$. We want to show that:

$$\begin{aligned}
&\forall\, E \subseteq Z : \\
&\forall\, t \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!]\sigma \quad \text{s.t.} \\
&(t = h \frown g) \wedge (\#h = N+1) : \\
&\exists\, F \subseteq Z \quad \text{s.t.} \\
&g \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(F)]\!]\sigma \qquad (24)
\end{aligned}$$

i.e. if after N steps the process is still behaving like the process $CELL_{n,f}(R)$ for some R then after the next step it is behaving like $CELL_{n,f}(F)$ for some F.

We substitute $h = s \frown \langle X \cup Y\rangle$ and $r = \langle X \cup Y\rangle \frown g$, where $X \subseteq I$ and $Y \subseteq O$, into Equation 24 to expand the hypothesis to:

$$\begin{aligned}
&\forall\, E \subseteq Z : \\
&\forall\, t \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!]\sigma \quad \text{s.t.} \\
&(t = s \frown r) \wedge (\#s = N) \quad \wedge \\
&(s[N] = \langle (X \subseteq I) \cup (Y \subseteq O)\rangle \quad \Rightarrow \\
&\exists\, F \subseteq Z \quad \text{s.t.} \\
&r \in \mathcal{T}_{\mathcal{R}}[\![[!Y\,?X \rightarrow CELL_{n,f}(F)]]\!]\sigma \qquad (25)
\end{aligned}$$

We note that the process described in the last line is in fact $CELL_{n,f}(Y)$, and use Equation 19 to get:

$$\begin{aligned}
&\forall\, E \in\subseteq Z :\\
&\forall\, t \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!]\sigma \quad \text{s.t.}\\
&(t = s \frown r) \wedge (\#s = N) \quad \wedge\\
&(s[N] = \langle (X \subseteq I) \cup (Y \subseteq O)\rangle \quad \Rightarrow\\
&r \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(Y)]\!]\sigma
\end{aligned} \tag{26}$$

The induction hypothesis gives us:

$$\begin{aligned}
&\forall\, E \subseteq Z :\\
&\forall\, t \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(E)]\!]\sigma \quad \text{s.t.}\\
&(t = s \frown r) \wedge (\#s = N) :\\
&\exists\, F \subseteq Z \quad \text{s.t.}\\
&r \in \mathcal{T}_{\mathcal{R}}[\![CELL_{n,f}(F)]\!]\sigma
\end{aligned} \tag{27}$$

which is equivalent, so we have the desired result. □

Combining Lemmas 1 and 2 gives us the proof that all the traces of $CELL_{n,f}$ satisfy the specification in Equation 15

The result is applicable to all stateless one-cycle cells (i.e. those cells where output at time $t + 1$ is solely dependent on input at time t), and is a useful foundation for proof at a higher level of abstraction. We give an example of this in the following section.

5. SAFETY MONITOR EXAMPLE

5.1 System Definition

For a substantial example, we take a safety-critical subsystem of a hypothetical military aircraft stores management system (SMS). The SMS is designed to control the arming and release of ordnance from hardpoints on the aircraft. The main relevant hazards of the SMS are the uncommanded arming or release of ordnance, which could result in destruction of the host aircraft or the dropping of ordnance on friendly territory. Such a system is often rated at SIL-3 or SIL-4, requiring the most rigorous demonstrations of safety.

The control of a typical SMS is mostly implemented in software on a conventional microprocessor, with digital and analogue devices providing support where required, and a bus such as the MIL STD 1553 allowing communication with other parts of the system such as the ordnance release units and the pilot's release controls. We assume that the safety-critical subsystem under examination consists of the microprocessor with coupled memory, bus interface and programmable logic devices, since this is what has final control over release and arming of all ordnance.

Programmable logic is useful in such systems because the systems typically have hard real-time constraints. The microprocessors used in these systems tend to be several years behind the leading edge due to issues of reliability and the long development cycle of a safety-critical system. It is sometimes difficult to get everything done within the required time frame. Programmable logic can take simple but CPU-intensive tasks away from the main processor and make it easier to meet the system constraints. The normal alternative to programmable logic is the use of an ASIC, but the small production run of most safety-critical systems and the frequency of requirements changes may mean that using an ASIC would be significantly more expensive than using programmable logic.

5.2 System Safety

The safety features of this system will include:

- a watchdog timer which must be reset every 25ms or it will shut down the system (to stop or restart a hung system);
- the use of *keywords* to command dangerous actions; and
- voting for readings from safety-critical sensors.

One convention for software is to have the watchdog resetting code as the last event in the main program loop, each iteration of which is designed to complete in just less than the watchdog time. This means that if the stack gets corrupted or the program goes off into an unanticipated path then the watchdog should be able to shut down the system before too much harm is done.

A *keyword* is a unique data value which is written to a specific mapped memory location (and hence to the controller of an actuator in the system) to enable a dangerous action. Any other value written to that location will have no effect. Some systems command a shutdown if bad keywords are persistently written to a location.

We aim to encapsulate as many safety functions as possible in programmable logic devices and reason about their correctness there rather than attempt to prove safety properties of software. This is not to say that we can then ignore the safety of the software – far from it, but by moving functionality to an FPGA we reduce the size of the program and it is (generally) easier to reason about the behaviour of a small program than a large one.

We shall now describe and specify several safety-related processes.

5.3 Watchdog Timer

The Watchdog Timer has a single input, which is toggled by software to reset the timer, and a single output which is typically used to raise a high-priority interrupt and trigger a system shutdown.

The SRPT description of the watchdog $WATCH_k$ which shuts down after $k + 1$ steps without an input toggle is:

$$\begin{aligned}
\iota WATCH_k &= \{w\}\\
o WATCH_k &= \{d\}\\
WATCH_k &= W0_k(k)
\end{aligned} \tag{28}$$

where event w represents a high voltage on the watchdog input, and d represents a high voltage on the system-disabling watchdog output.

The $W0_k(x)$ process describes a watchdog where the last input toggle was to 0 and there are x steps left until shutdown trigger. $W1_k$ is the same except that the last input toggle was to 1.

$$\begin{aligned}
W0_k(0) &= [!\{d\}\,?X \to W0_k(0)]\\
W1_k(0) &= [!\{d\}\,?X \to W1_k(0)]\\
\forall\, x \geq 1 &:\\
W0_k(x) &= [!\emptyset\,?X \to\\
&\qquad W1_k(k) \lhd (w \in X) \rhd W0_k(x-1)]\\
W1_k(x) &= [!\emptyset\,?X \to\\
&\qquad W0_k(k) \lhd (w \notin X) \rhd W1_k(x-1)]
\end{aligned} \tag{29}$$

Time	Process	Input	Output
1	$W0_k(2)$	-	-
2	$W0_k(1)$	w	-
3	$W1_k(3)$	w	-
4	$W1_k(2)$	w	-
5	$W1_k(1)$	w	-
6	$W1_k(0)$	-	d
7	$W1_k(0)$	w	d

Figure 3: w "stuck-on" input for $k = 3$

where $A \lhd B \rhd C$ is read "if B is true then do A, otherwise do C".

The trace specification on WATCH_k for each trace t is as follows. First, the detection of "w stuck-on" input:

$$\begin{aligned} \forall\, i \in \mathbb{N} \quad \text{s.t.} \quad & w \notin t[i] \wedge w \in t[i+1] : \\ & (\forall\, j \in 1..(k+1) : w \in t[i+j]) \Rightarrow \\ & (\forall\, m \geq (k+2) : d \in t[i+m]) \end{aligned} \tag{30}$$

An example of w being "stuck-on" is in Figure 3.

Next, the detection of stuck-off input:

$$\begin{aligned} \forall\, i \in \mathbb{N} \quad \text{s.t.} \quad & w \in t[i] \wedge w \notin t[i+1] : \\ & (\forall\, j \in 1..(k+1) : w \notin t[i+j]) \Rightarrow \\ & (\forall\, m \geq (k+2) : d \in t[i+m]) \end{aligned} \tag{31}$$

Note that these specifications as written won't detect the case that the process always or never receives w as an input. This is to reduce visual complexity of the specifications. They also do not require that the watchdog *only* output d when a failure has occurred.

The above is a partial specification for the watchdog timer. It is not compact, mostly because we do not use abbreviating functions in the descriptions for clarity. If we now define

$$\begin{aligned} & \textbf{breaks} : \mathcal{T}_{\mathcal{R}} \times \mathbb{P}\Sigma \rightarrow \textbf{seq}\, \mathbb{N} \\ & \textbf{breaks}\,(t, x) = \{ i \mid i = 1 \vee \\ & \quad (\{x\} \cap t[i] \neq \{x\} \cap t[i+1]) \} \end{aligned} \tag{32}$$

so that $\textbf{breaks}\,(t, x)$ gives a list of the points in the trace t where event x starts or ends appearing, then we could rewrite the specification of WATCH_k as:

$$(\forall\, i \leq b : d \notin t[i]) \wedge (\forall\, i \geq (b+1) : d \in t[i]) \tag{33}$$

where:

$$\begin{aligned} b &= B[a] + (k+1) \\ B &= \textbf{breaks}\,(t, w) \\ a &= \textbf{min}\, m : B[m+1] - B[m] \geq (k+1) \end{aligned} \tag{34}$$

This is a complete specification for the watchdog, and has also patched the hole of not catching all-w or never-w sequences.

5.4 Keyword Checker

A keyword checker is a process which takes as input a w-bit keyword along with a lines which denote the actuator to activate. No more than one of the actuator lines may be raised at any one time.

When an actuator line is raised, the checker validates the given keyword against an internal table: the result is one of ON, OFF or BAD. If ON then the checker raises the appropriate actuator output line. If OFF or BAD then it lowers the line, and if BAD or more than one input actuator line is raised then it sets a "failure" output.

The process description is as follows:

$$\begin{aligned} P &= \{p_1 \ldots p_a\} \\ Q &= \{q_1 \ldots q_w\} \\ R &= \{r_1 \ldots r_a\} \\ I = \iota \mathit{KEYW}_{w,a} &= P \cup Q \\ O = o\mathit{KEYW}_{w,a} &= R \cup \{f\} \\ \mathit{KEYW}_{w,a} &= \mathit{KEYW}_{w,a}(\mathbf{0}, \emptyset) \end{aligned} \tag{35}$$

P events are actuator selection, Q events form keywords, and R events are actuator controls.

The process $\mathit{KEYW}_{w,a}(f, X)$ will output events in X, identify which actuator is being requested, check the supplied keyword, adjust its table f mapping actuator to ON or OFF accordingly, then transition to a $\mathit{KEYW}_{w,a}$process passing the updated table and, if any failure has occurred, the f failure event. We will not provide a full SRPT description here for the sake of brevity. Instead we will turn to the specification.

The keyword evaluation is given by **words**:

$$\textbf{words} : \mathbb{P}Q \times \mathbb{N} \rightarrow \{\text{ON}, \text{OFF}, \text{BAD}\} \tag{36}$$

The specification of each trace $t \in \mathcal{T}_{\mathcal{R}}$ is as follows:

$$\begin{aligned} \forall\, i \in \mathbb{N} &\quad : \\ \#(P \cap t[i]) \geq 2 &\quad \Rightarrow \quad t[i+1] \cap O = \{f\} \\ \#(P \cap t[i]) \leq 1 &\quad \Rightarrow \\ \forall\, j \in 1 \ldots a &\quad : \\ \textbf{last}\,(i, j) = \text{ON} &\quad \Leftrightarrow \quad p_j \in t[i+1] \\ (\,\textbf{words}\,(Q \cap t[i], j) = \text{BAD} &\quad \wedge \\ P \cap t[i] = \{p_j\}) &\quad \Rightarrow \quad f \in t[i+1] \end{aligned} \tag{37}$$

where:

$$\begin{aligned} \textbf{last}\,(i, j) &= \textbf{words}\,(Q \cap t[n], j) \text{ s.t.} \\ n &= \textbf{max}\, m \text{ s.t. } (P \cap t[m]) = \{p_j\} \wedge \\ & \quad\; m \leq i \\ \text{or } \textbf{last}\,(i, j) &= \text{OFF if no such } n \end{aligned} \tag{38}$$

returns the last legitimate action for actuator j at or before trace index i.

5.5 Sensor Voter

A number of sensors require a voter to check their readings and forward them to the main processor along with a "failed" bit. We now address specifying this voter.

We assume that each sensor has three binary indicators, and that the correct output of the sensor, output by the voter, is taken by majority voting from the indicators. We also require that a sensor be regarded as failed if one channel has been wrong for k consecutive tests, and that failure is permanent once it occurs. Hence the voter will give a voted sensor output v and a failure output f.

$$\begin{aligned} \iota \mathit{VOTER}_k &= \{c_1, c_2, c_3\} \\ o\mathit{VOTER}_k &= \{v, f\} \\ \mathit{VOTER}_k &= \mathit{VOTER}_k(\mathbf{0}, \emptyset) \end{aligned} \tag{39}$$

Again, we do not provide a full description of the process $\mathit{VOTER}_k(g, X)$. Suffice it to say that X provides the next output events, as before, and g counts the number of consecutive divergences from majority vote of each sensor input line.

The specification must first define the majority voting:

$$\begin{aligned}\mathbf{maj} &: \mathbb{P}\Sigma \rightarrow \mathbb{B}\\ \mathbf{maj}\,(X) &= \#(\{c_1, c_2, c_3\} \cap X) \geq 2 \qquad (40)\end{aligned}$$

and so we can now write the specification for each trace $t \in \mathcal{T}_{\mathcal{R}}$:

$$\begin{aligned}\forall\, i \in \mathbb{N} &: v \in t[i+1] \Leftrightarrow \mathbf{maj}\,(t[i])\\ \forall\, i < \mathbf{min}\,(e_1, e_2, e_3) &: f \notin t[i+1]\\ \forall\, i \geq \mathbf{min}\,(e_1, e_2, e_3) &: f \in t[i+1] \qquad (41)\end{aligned}$$

where each e_j defines the earliest index in the trace t when the sensor j has been regarded as failed:

$$e_j = \mathbf{min}\, m \text{ s.t. } \forall\, n \in 0..(k-1) : \mathbf{div}\,(j, m+n) \qquad (42)$$

where:

$$\mathbf{div}\,(j, m) = (c_j \in t[m]) \Leftrightarrow \neg\, \mathbf{maj}\,(t[m]) \qquad (43)$$

identifies whether input j diverges from the majority vote at point m in the trace.

5.6 Commentary

We have taken three typical components of a safety-critical system which could conceivably be implemented using PLDs, and have provided SRPT-based specifications for them. For the watchdog timer, the simplest of the three, we have also provided a full SRPT description.

We have seen that the SRPT trace-based specifications can, with carefully-chosen syntactic abbreviations, be expressed in a few lines and yet make rigorous and useful statements about the required properties of a process. The previous section has shown how it is possible to prove that a SRPT process description satisfies a specification, though clearly there is some way to go until this proof mechanism is easy enough to use effectively in a commercial project.

Note that there is a clear gap between the SRPT description of a process and its final implementation as a set of programmed cells in a LUT-based FPGA. It should however be relatively simple to map such SRPT descriptions as given here into equivalent VHDL, Pebble or netlist formats. We have used SRPT as a compromise between the high-level specification languages, such as Z, and the low-level implementation languages such as netlists and VHDL. The tradeoff we make is in ease of specification against simplicity of compiling to our target format.

6. CONCLUSIONS

6.1 Work to date

In this article we have shown how SRPT can be used to model non-trivial FPGA programs and prove certain safety properties in a rigorous way. We have also seen that it provides a precise way of specifying the requirements for an FPGA program, which makes it easier to define correctness tests.

In [6] we outlined a complete development process from a high-level specification to an FPGA implementation, using the SPARK Ada [4] high-integrity programming language as an intermediate step between a Z specification and an SRPT description. SPARK Ada has a formal definition, and compiling a program from SPARK Ada to a set of SRPT processes while maintaining provable correctness requires the ability to reason about the correctness of SRPT processes. This report has contributed towards that goal.

6.2 Future work

Decomposition of an FPGA program into logically coherent blocks must currently be done manually. The authors are investigating the use of high level languages such as Handel-C and SPARK Ada for compilation directly to SRPT processes.

Reasoning about CSP processes is made easier by modelling tools such as FDR. It may be possible to produce a similar modelling tool for SRPT, providing simple analysis on possible traces of a process using analytic means. Clearly, for SRPT to be used in real system development programs there needs to be appropriate and effective tools support.

7. ACKNOWLEDGEMENTS

This work was undertaken as part of study for a Ph.D. with the Computer Science department of The Open University. Financial support from Teleca is gratefully acknowledged.

8. REFERENCES

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

[2] J. Becker, T. Pionteck, and M. Glesner. DReAM: A dynamically reconfigurable architecture for future mobile communication applications. In Hartenstein and Grünbacher [5], pages 312–321.

[3] Formal Systems (Europe) Ltd. *FDR User Manual*, May 1997.

[4] J. Garnsworthy and B. Carré. SPARK - an annotated Ada subset for safety-critical systems. *Proceedings of Baltimore Tri-Ada Conference*, 1990.

[5] R. W. Hartenstein and H. Grünbacher, editors. *Proceedings of the 10th International Conference on Field Programmable Logic and Applications (FPL'00)*, volume 1896 of *Lecture Notes In Computer Science*. Springer-Verlag, August 2000.

[6] A. Hilton and J. Hall. On applying software development best practice to FPGAs in safety-critical systems. In Hartenstein and Grünbacher [5], pages 793–796.

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

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

[9] M. Josephs. Receptive process theory. *Acta Informatica*, 29:17–31, 1992.

[10] N. Leveson. *Safeware: System Safety and Computers*. Addison-Wesley Publishing Company, 1995.

[11] Defence Standard 00-56 issue 2, 1997. Safety Management Requirements for Defence Systems.

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

[13] M. Renovell. A specific test methodology for symmetric SRAM-based FPGAs. In Hartenstein and Grünbacher [5], pages 300–311.

[14] A. Simpson and M. Ainsworth. White box safety. In *Proceedings: Avionics Conference and Exhibition*. ERA Technology Ltd., 1999. ERA Report 99-0815.