# Semi-Automated Safety Analysis for Field Programmable Gate Arrays

Philippa Conmy, Iain Bate
*Software Systems Engineering Initiative,*
*Department of Computer Science, University of York,*
*York, U.K.*
*{philippa, iain.bate}@cs.york.ac.uk*

## Abstract

*Complex Programmable Logic Devices (PLDs) such as Field Programmable Gate Arrays (FPGAs) are becoming increasingly popular for use in High-Integrity Safety Related and Safety Critical Systems. FPGAs offer a number of potential benefits over traditional microprocessor based software systems, such as predictable timing performance, the ability to perform highly parallel calculations, predictable emulation of obsolete components, and (in the case of SRAM based FPGAs) the ability to reconfigure to avoid hardware failures. However these abilities do not come for free and often designers are forced to make pessimistic safety and reliability assumptions leading to conservative overall system designs. In this paper a modular, and hence more scalable approach, to performing FPGA safety analysis is presented.*

## 1. Introduction

Safety critical and safety related systems typically undergo some sort of certification process prior to their deployment. The certification details are dictated by domain specific standards and guidance, however there are a number of common practices and procedures within these. One common item is a safety case which demonstrates how potentially hazardous failures within the system are prevented or mitigated. FPGAs contain thousands of logic gates and are extremely complex once configured. Hence it is extremely difficult to determine the effect that a single low-level failure can have at the system level. As this information it cannot be provided a conservative design approach (e.g. Triple Modular Redundancy (TMR) [6, 14]) has to be taken. This satisfies safety requirements (as the system is robustly protected against failures) but it prevents FPGA benefits, such as the ability to run multiple programs on the same device thus providing power and weight savings, from being fully exploited

This paper contributes the following. First we demonstrate how to apply a semi-automated failure analysis technique to an HDL design synthesized for an FPGA, providing a simple example. We then show how this can help a safety analyst to identify the safety effects of hardware failures within the FPGA, to justify or alter the design and hence provide evidence for a safety case.

The paper is laid out as follows. The next section discusses FPGAs, certification issues and safety analysis in more detail. Section 3 looks at related work. Section 4 describes the generic process we are proposing. Section 5 provides a worked example to demonstrate how it can be applied and Section 6 contains further work and conclusions.

## 2. Certification of FPGAs

### 2.1. Field Programmable Gate Arrays

An FPGA is programmable logic device which has thousands of connected logic cells. The basic structure of these cells is identical but they can be configured to perform different operations. For example, each cell typically contains a Look-Up-Table which can be configured to perform different binary comparisons on input signals such as AND or XOR. These cells are connected together via a series of interconnects to perform higher level processing tasks. Due to its design an FPGA can perform many tasks in parallel e.g. to perform digital signal processing.

FPGAs can have have different hardware implementations, especially for the interconnects. FPGAs with SRAM interconnects can be reconfigured as many times as the user requires. Antifuse interconnects can only be configured once, however they are smaller and offer more routing flexibility. In addition some FPGAs include extra hardware devices such as static memory, multipliers and even traditional sequential microprocessors. This assists with tasks that the FPGA is not well suited for, for example dealing with floating point numbers, but can add further complexity to the certification process as each different device will need to be assessed. The ability to configure or reconfigure an FPGA is not explored within this paper but will be for our further work. Figure 1 shows a typical FPGA design.

FPGAs are configured by synthesizing code written in a Hardware Description Language (HDL) such as HDL or Verilog. The first stage of synthesis converts the code into a netlist which describes the hardware parts and connections which will be used to implement the code. This then undergoes a place and route operation that describes which logic cells will actually be used on the FPGA and how they are connected. The routing will be calculated based on criteria such as acceptable timing performance.

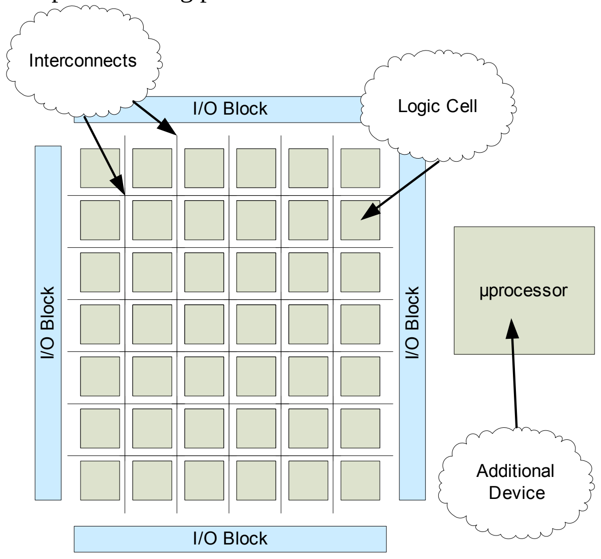

**Figure 1 Simplified representation of FPGA**

FPGAs are susceptible to a number of different hardware failures, but those which are most often cited are grouped as Single Event Effects (SEEs). An SEE causes a bit flip in the device i.e. from 0 to 1 or vice versa. The most commonly discussed is a Single Event Upset (SEU) which in which a bit flip may be stuck until the device or the cell is reset. Other SEEs are transient (resets itself without interventation) or permanent. Several SEEs may be found in a group together, e.g. caused by a radiation burst, rather than a just single failure on its own.

One of the difficulties in analyzing an FPGA is determining the effect (if any) that an SEE has at the system level. In some instances an SEE may never cause a failure, for example if an SEE causes an incorrect input into a multi-input Look Up Table (LUT) the output may still be correct depending on the internal truth tables and other input values. E.g. if two values undergo an AND then the same output will be achieved for 00, 10, and 01. More detailed discussion of SEE effects can be found in [5, 2].

On the other hand the failure may be very significant if a value is flipped which is then used by multiple other cells, effectively becoming a common mode failure for multiple cells, and hence for multiple functions. However, it may be that even though outputs from the FPGA are affected there is actually no safety concern. Also, an SEE which affects multiple cells may be of little concern as it is easier to detect. However an SEE which causes an output value to be subtly incorrect but credible may be of more interest. This reflects the inherent complexity of an FPGA which is not just due to the number of internal cells but also to the number of permutations of interactions between them.

In our experience the inability to determine the effect of an SEE has led to pessimistic assumptions about the reliability of an FPGA to be made and hence conservative (and expensive) design decisions such as multiple lanes or monitors are made ([6, 11]). Alternatively an FPGA may simply not be chosen for use within systems of a high criticality. Our desire is to help determine the safety effect of an SEU and to improve the design in areas of significance, potentially improving both the ability to certify an FPGA based system and also to support less conservative designs.

One feature of our work which differs from others is that we can also consider other hardware failures of concern, in particular errors in the clock which can lead to synchronization, propagation, and timing issues.

## 2.2. Certification Standards and Guidance

Certification is used within this paper as a general term for the process of demonstrating that a high-integrity system performs as intended and/or required to an authority of some kind. For the safety critical systems being discussed here safety analysis will form a large part (or in some cases all) of the certification process. Safety analysis is the term used to describe any method which shows how a failure within a system could lead to a hazardous event (i.e. one which could lead to an accident e.g. engine fire). After analysis it should be possible to show how potentially hazardous failures have been managed. There are various techniques for this, some or all of which are prescribed by standards or guidance within a domain. For brevity this paper only discusses U.K. Defense standard 00-56 issue 4 [9], IEC 61508 [4] and Eurocae/RTCA guidance DO178B [13] and DO254 [12], as these provide a good cross-section of the different approaches taken to certification and help identify issues of concern to analysts and designers of systems containing FPGAs.

One major issue for FPGA developers is that there is some confusion over whether an FPGA should be classified as software or hardware. On one hand it is highly programmable or reprogrammable, but on the other hand the code is run directly in hardware. The pragmatic approach is to assume that development of the HDL code will be performed using processes similar to that for software (e.g. a V lifecycle), *as well as* examining the hardware using established techniques for electronics. We have performed our review based on this dual approach.

### 2.2.1. U.K. Defence Standard 00-56 Issue 4

This standard is used to manage safety in military equipment. The standard is based on the As Low As Reasonably Practicable (ALARP) principal that the effort/cost taken to reduce a given failure should be proportionate with its potential severity and likelihood. The standard requires the production of a safety case to demonstrate that hazards have been identified and risks associated with those hazards appropriately managed. It does not prescribe how this is to be achieved. In practice safety analysis would be used to support the safety case and to identify hazards. As a result there is an opportunity to explore new methods of providing safety evidence, such as those proposed in this paper for FPGAs. This is contingent on those methods being shown to provide useful and reliable information.

### 2.2.2. IEC 61508

This is a generic standard used for the development of electronics and programmable electronics. To meet the standard the developer must first assign a Safety Integrity Level (SIL) to the system based on the risks associated with it malfunctioning. This assignment is assisted by various safety analyses. Then the rigour of processes applied during development is dictated by the assigned SIL and need to demonstrate that risks are managed (e.g. in a high SIL a formal proof of code might be required but not for lower SIL).

Software and hardware are treated separately in the standard. It is likely that some of the recommended software practice would need to be adapted for application to an FPGA e.g. how to ensure code is fully tested and to assess the synthesis tools and their output.

Hardware guidance requires (amongst other things) an estimation of the failure of safety functions due to random hardware failures. This is of significance when certifying an FPGA since it is very hard to identify the effect of an SEU so, even if an estimate of the rate of occurrence is known, it cannot be linked easily to a specific safety function at a system level. This can lead to a very pessimistic failure rate being assumed when an FPGA is used, as suggested by Isaac in [5].

### 2.2.3. DO178B/DO254

DO 254 provides Design Assurance Guidance for Airborne Electronic Hardware and lists PLDs as one type of target device. It specifically does not define firmware and suggests the use of DO178B (Software Guidance for Airborne Systems) to certify functions implemented by software. Based on our knowledge of industrial practice, and again taking a pragmatic approach, we are assuming both documents would be used when certifying a system containing an FPGA.

Both documents require an initial safety assessment which identifies the potential severity of failures within functions. The functions are defined by the developer and may be supported by a system, sub-system or group of systems. In this instance a Development Assurance Level (DAL) is assigned to the supporting item(s). Similarly to IEC 61508, the DAL then defines the types of processes and rigour that the analyst should follow in order to achieve certification and ensure their system is safe. The methods described in both documents are also similar. Again, we would argue here that it is important to be able to understand how a failure in an FPGA could affect one or more functions at a system level otherwise a pessimistic view must be taken, leading to a lack of trust in the FPGAs output and it requiring a monitor, and/or the use of replicated FPGA devices. They may also not be avoided for use for more critical systems.

## 3. Related Work

The previous section established that determining the safety effect of failures within an FPGA would be extremely helpful to support the certification process. This section first examines typical techniques used to determine the effect or significance of SEEs. Then we look at failure analysis techniques and demonstrate why we have opted to examine FPTC as a potential way forward.

### 3.1. SEE detection

In [14] the authors describe a method for determining the effect of SEU in a TMR system, i.e. one in which three identical lanes are used for redundancy in calculations and a monitor determines whether there is (intolerable) disagreement in their output. In their case all three lanes are on the same FPGA. They use a tool to determine if an SEU at a given location can affect more than one lane and if it can they determine it to be significant. The advantage of their approach is that it can highlight the extent of the effect of an SEU. However, it is limited to a certain type of design solution and at present they do not determine whether there is an actual safety effect, rather it is assumed that any effect is undesirable. As discussed in the previous section, one of the drivers for our work is to support the safety analysis of FPGAs. This would include all design approaches.

A common method for detecting the effect of SEUs is to use fault injection to simulate an SEU. One example of this can be found in [2] where the authors use two FPGAs, one loaded with a correct configuration file and one with an configuration file with a simulated SEU inserted. A third FPGA is used to compare the output of the two FPGAs and note when they differ. If they do it is assumed there the SEU is significant. Again this technique suffers from an inability to determine whether the SEU is significant in terms of safety effect although it could perhaps be adapted to do so. We have not chosen to use this type of method as our proposed approach looks at a broader class of failures than simply SEEs. An advantage of the fault injection approach is that test case generation can be fully automated.

A third method to detect permanent SEEs is described in [1] by Emmert et al. who use 'Roving STARs' (Self-Test AReas) within reconfigurable FPGAs. They reconfigure the FPGA cells around any permanent SEEs detected and by moving the STARs around during operation they can cover the entire FPGA. The disadvantages of such an approach (apart from the fact that only permanent SEEs are detected) is that it requires the FPGA to be reconfigured around the test areas during operation which can affect system timings and cause output to be interrupted. In addition the effect of an SEU occurring in a non-test area would be different for each configuration and so safety analysis would need to consider all potential configurations. The issue of certification of a system using reconfiguration is not addressed in this paper, but is part of our ongoing research. A discussion of one possible approach can be found in [11]. It is of note that a system which can reconfigure around faults is potentially more reliable and hence can offer safety benefits compared to a static one.

### 3.2. Failure and Safety Analysis

There are numerous different safety analysis techniques which can be applied at different times during a systems development lifecycle. A good over view of these can be found in [8]. At early stages of development these help provide insight into whether a design needs to be adapted to deal with identified failures and at later stages of development the results of the analysis can be used as evidence to demonstrate failures are adequately managed. Techniques can be top down i.e. starting with a hazardous system level event and working down to individual components to see how they could contribute to its occurrence. A typical top-down technique is Fault Tree Analysis which shows how component failures combine (and their probabilities) to cause a single event, Other techniques are bottom up, i.e. they show how individual component failures can contribute to one or more hazards system events. A typical technique would be Failure Modes and Effects Analysis (FMEA) which looks at individual failures and considers their effects. FMEA can be supported using different guidewords which suggest different failure types, e.g. backwards/too much for chemical processing or timing/omission/value for computer component analysis.

We believe that a bottom up technique would best support our desire to determine the overall effect that a single SEE can have, i.e. how it could fan out to multiple system level events. However, FMEA based techniques are generally manually applied and an FPGA can have thousands of different connections to consider and routes to consider. Hence it is impractical to use manual analysis.

Failure Propagation and Transformation Calculus (FPTC) [15] is a bottom up safety analysis approach in which individual components can be analysed independently to identify their failure characteristics, and then a tool can be used to calculate the actual failures which could be propagated through a network of these components. An analyst can then look at the concatenated results in order to determine overall safety effects. We believe this technique has the potential to assist in the assessment of the effect of failures in FPGAs. FPGAs are constructed of the same types of hardware components, replicated many times. Hence we can analyse the failures of each component type individually and reuse the results. Then we construct a network to represent how they are connected together on the FPGA (based on the synthesis outputs) and assess how an individual SEE or clock failure would propagate through the system outputs. However, some automation to help generate FPTC networks and also to guide the analyst may be required for a device as complex as an FPGA. FPTC has also mainly been applied at a higher level for analysis in previous case studies (e.g. software process level [15]) so it's efficacy for such a low-level analysis needs to be assessed.

## 4. Application of FPTC to FPGA Netlists

Based on the previous two sections our aims in this work are to:

- Provide evidence about the effects of low-level failures on system level events and improve design.
- Enhance confidence in an FPGAs ability to perform in more critical systems, thus supporting more efficient designs

This section describes our proposed process and adaptations to FPTC in order to achieve these aims.

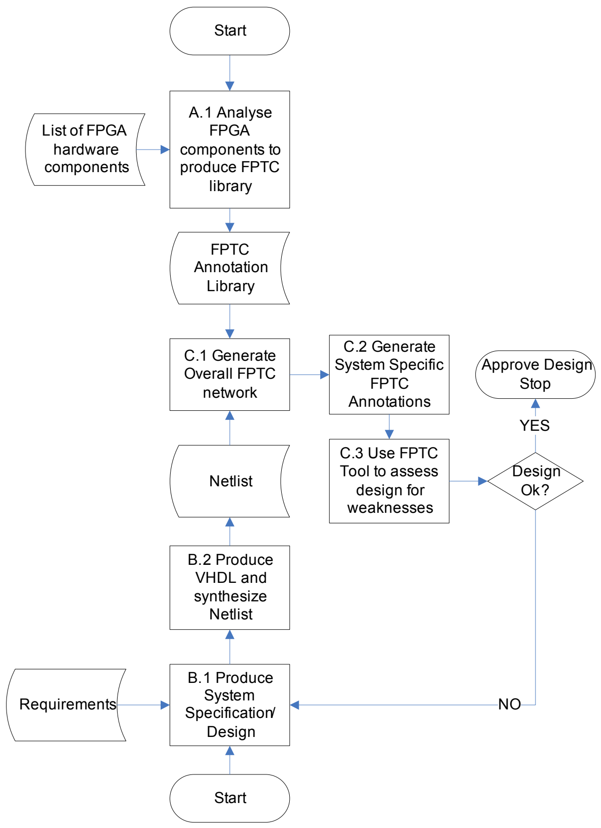
Start
List of FPGA hardware components
A.1 Analyse FPGA components to produce FPTC library
FPTC Annotation Library
C.1 Generate Overall FPTC network
C.2 Generate System Specific FPTC Annotations
Approve Design Stop
YES
C.3 Use FPTC Tool to assess design for weaknesses
Design Ok?
Netlist
B.2 Produce VHDL and synthesize Netlist
Requirements
B.1 Produce System Specification/ Design
NO
Start

**Figure 2 Summary of FPGA analysis process**

### 4.1. Overall Analysis process

A summary of the FPTC process is shown in Figure 2. It comprises of three parts: Generic reusable FPGA component analysis (A.1), generation of FPGA hardware design (B.1-B.2) and system specific safety analysis (C.1-C.3). We now describe these in turn.

### 4.2. Part A: FPTC Component Analysis

This section describes the FPGA component analysis process (A.1) used to generate a reusable library of FPTC annotations. An FPGA is made of numerous different logic cells. These cells typically contain LUTs, flip-flop circuits, multiplexors, inputs, outputs, and clock signals [2]. As we are using the Xilinx ISE we are basing our analysis on their lists of generic design elements as described on their website [16].

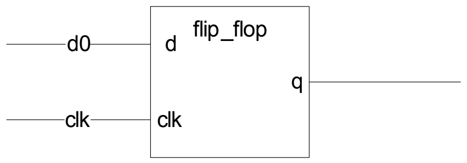

**Figure 3 Flip-Flop Component**

Figure 3 shows a flip-flop device of the type typically found in an FPGA. This device copies the value of *d0* to its output *q* on the low to high clock (*clk*) transition. In order to produce an FPTC failure annotation we need to consider the possible failures which could be provided on input (see Table 1) and also consider any errors which could be introduced by the device. From this we produce a set of output failures. These are summarised in Table 2.

The results are expressed in the FPTC notation as shown in Figure 4 (see [15] for full syntax and semantics). The left hand side of an FPTC annotation indicates the values which are received on input, and the right hand side indicates the corresponding outputs. Here the ordering is that the *d0* is received on the first input and the *clk* signal on the second. Each separate input and output set is contained in curled braces. The notation doesn't require a particular ordering of the inputs (as demonstrated in [15]), but the analyst must ensure they are consistent when the items are connected, i.e. a *clk* component must be connected to the *clk* input in order for the results to be generated correctly. The asterisk "*" symbol is used to indicate that no fault is received on input, and the underscore "_" is used to indicate that any value can be received on a given input. If multiple input failures lead to a single output type of failure (as is true in this case) then these can be concatenated together.

**Table 1 Flip-Flop input failures**

| Input/Output | Failures | Comments |
|---|---|---|
| *clk* | Early/late | The clock may be running correctly or may be early or late |
| *d0* | Value | The value of *d0* may be correct or may be false high/ false low or undetermined. We express this as a single failure of "value" for now. |

**Table 2 Flip Flop output failures**

| Input Failure | Output Failure | Comment |
|---|---|---|
| *clk* Early | stale_value | If the clock triggers the flip-flop to copy the value of *d0* early then *d0* may not contain the most recent value, hence stale_value is produced |
| *clk* Late | stale_value | If the clock triggers a late copy of d0 to q then it is possible the value of q has been read already hence the output failure is again stale_value |
| *d0* Value | Value | If the value of *d0* is incorrect then the flip-flip will simply pass this on |
| *None* | Value | It is possible that the flip-flop may have been affected by an SEE hence it could be the source of a value failure. |

({*}, {fault early, fault late})->({fault stale_value})

()->({fault value})

({fault value},{*})->({fault value})

**Figure 4 FPTC annotations for flip-flop component**

FPTC components are linked together to form a network using the Eclipse based analysis tool described in [10]. The tool can be used to either generate the entire set of potential failures throughout the network or to analyse the path of an individual injected fault anywhere in the network. Essentially the tool attempts to match inputs to a component with the left hand side patterns as provided in the annotations. Every time there is a match the right hand side is then provided as input to next component it is connected to. An example of this is shown in Figure 5. In this example a clock is the source of a *late* fault (shown on the output arrow) and another input (*Input1*) has no fault. These two inputs match with the first annotation on *FLIP_FLOP* but not the second and hence its output is *stale_value* only.

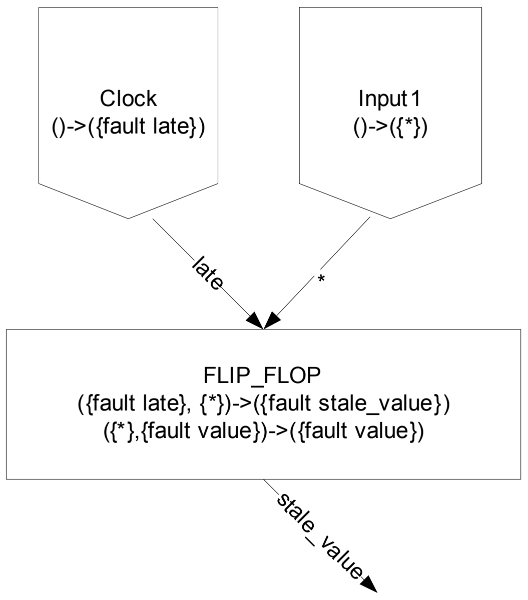
Clock
()->({fault late})
Input1
()->({*})
late
*
FLIP_FLOP
({fault late}, {*})->({fault stale_value})
({*},{fault value})->({fault value})
stale_value

**Figure 5 FPTC matching example**

Note that the fault tokens are entirely generated by the analyst and not dictated by the notation or the tool. Whilst this gives great flexibility it is up to the analyst to ensure that token names are consistently spelt.

Using the method laid out in this section we have analysed a number of common FPGA hardware components. These results are used in the next analysis stage described in section 4.4.

## 4.3. Part B: FPGA Design

This section describes the processes corresponding to B.1 and B.2 in Figure 2. The aim of our work is to investigate the potential safety effects of hardware failures within an FPGA. Therefore we need a hardware representation of our intended design. To this end we are currently producing VHDL descriptions of potential designs based on system requirements and then synthesizing these via Xilinx ISE Project Designer. As we are refining the technique and at proof of concept stage we are at present examining netlist files, rather than configuration files with place and route information. Our further work will investigate the place and route aspects in more detail. Note that we are not at present investigating any fundamental weaknesses of or comparing any particular FPGA chips, rather we are exploring the analysis theory. One other issue which has been raised is that of the synthesis routines and whether different options for optimization of the compiler will affect the potential safety of the output. Again we will look at this during further work.

The different areas of concern in terms of safety effect are:

- Identification of systematic design flaws i.e. flaws in the program logic
- Analysis of the effect of random hardware failures upon the design

Note that in the first case we are concerned with problems introduced during the design process, whereas in the second case we are reacting to issues which may not be apparent until the hardware implementation is considered. In both cases our intent is to ideally improve the design so that a safety weakness is no longer included, but if that is not possible we wish to alter the program design to mitigate against unacceptable failures. Also, it is our intention that results of the analysis (once changes are made) can be used as evidence to support a system safety case.

### 4.4. Part C: FPTC Analysis of FPGA design

This section describes the processes corresponding to C.1-C.3 in Figure 2.

Once we have generated a netlist using the ISE tool we convert it manually to a network of FPTC components instances within our Eclipse tool. Each instance (barring outputs) is annotated with the pre-produced FPTC annotations. These may need to be adapted to ensure that the correct input channel lines up with the correct output types from the connected components. In addition, if the output from a component is used more than once then it is repeated. This corresponds to C.1 in our process.

C.2 in our process involves the production of system specific failure annotations on the output pins. The purpose of this is to link failures to specific system level events which may or may not be hazardous. This allows us to trace the generic internal hardware failures to one or more system level events, thus supporting our aim of identifying how internal failures cause system level hazards.

Part of our future work is to perform as much of process C1 as possible automatically. For this we will use netlists in the Electronic Data Interchange Format (EDIF) [3], the library of FPTC results, and output FPTC network files in the XML format used for the tool. We will still need to examine output pins manually as these have the system specific data. However, automation is essential if this technique is to be scalable to anything beyond simple examples and in order to be of practical use to developers of (complex) safety critical systems.

Once we have the FPTC network in the tool it can be used to generate the full set of potential failure patterns throughout the hardware network (item C.3). Alternatively a fault can be injected on any of the components and the paths it propagates along can be assessed. Again, it would be preferable to aid this process automatically where possible as the hardware network is likely to be large and complex. There are several possibilities for this. One is to tag output pin failures one (or more) at a time and get the tool to highlight possible paths which could lead to those tagged failures. Alternatively the tool could highlight components whose outputs affect multiple other components in the network. Ultimately though, a human analyst will still need to assess whether these paths represent an acceptable level of risk or whether the design needs to be altered (e.g. to prevent a single component or cell being used so much that an SEU or clock error would cause a catastrophic output failure).

## 5. Example Incinerator System

### 5.1. Example Description

This section presents a simple example in order to demonstrate our ideas and the value of the approach. The example used is a monitor/control within an incinerator system (based on that described [7], but with an extra monitor added to mitigate against a potential overheat as described below). This incineration burns medical/toxic waste. One hazard associated with this system is that toxic waste escapes without being incinerated. Therefore a cut off valve has been installed which prevents the addition of waste when no flame is present. This is engaged when two out of three conceptually different sensors indicate the lack of flame or too low a temperature. The sensors detect light, heat and ionized particles respectively. The advantage of this approach is that it is highly unlikely that a common fault will cause more than one of these sensors to fail at once. However, if there is disagreement amongst the sensors (e.g. two say that it is safe to operate and one doesn't) then a warning light is displayed to a human operator.

A second hazard associated with the system is the possibility of overheating, another warning light has been installed which lights if the temperature is over a certain value. The operator observes this and can switch off the entire system if they feel this is necessary.

## 5.2. Converting the example to FPTC

The functionality to control the warning lights and shut-off valve has been written as a VHDL program and converted into a netlist for input into the FPTC tool (processes B1-B2). The resultant hardware network has four input pins, one clock signal reused by multiple different items, four flip flops of the type described in section 3.1, six slightly different flip flops which also contain reset pins, and seven different LUTs, and six outputs (sensor problem warning light, shut-off valve, overheat warning light, and three copies of values for another system). A block diagram version of the hardware representation is shown in Figure 6 to illustrate the network.

The components were then input into the FPTC tool, and the pre-produced failure annotations added for each component type (process C.1). These were adapted in a couple of instances purely to ensure the correct alignment of inputs to outputs. Note that it will be possible to do this adaptation as part of an automated process.

The three output pins of concern were annotated with specific failures (process C.2). It was determined that the *Overheat_light* receiving either a stale or incorrect value had the possibility of not showing the light when the system was overheating, or of falsely reporting an overheat when there wasn't one. Whilst this may not at first seem to be hazardous, there is the possibility that if the overheat light repeatedly shows an incorrect value then the operator will cease to trust it and may ignore a genuine warning. The *Valve_cntrl* has the possibility of not closing when the flame is not present if it has the wrong or stale input value. It may also close when no fault has occurred which is not hazardous but if it happens repeatedly the system may not be reliable enough to be useful. *Warning_light* may also incorrectly show a sensor error when there is none due to a stale or incorrect value problem. Again there may be an issue with an operator ignoring a genuine problem if the warning light is repeatedly wrong.

## 5.3. Results

This section describes some of the findings of the FPTC analysis in order to demonstrate its usefulness. This corresponds to process section C.3. We describe one failure finding in detail to demonstrate the technique and then summarise some other interesting findings.

The first (and perhaps unsurprising) result is that a clock failure can cause problems on all three outputs. Whilst the effect is obvious the set of causes that lead to this behaviour are not, and can be due to a complex chain of events. For example, Figure 7 shows a small section of the FPTC network with a late clock failure. The late clock signal failure passed into *FDR_1* causes a stale value to be input into *LUT3* and *LUT4_2*. Suppose that due to the internal logic in *LUT4_2* no error is actually produced (see section 2.1). This doesn't mean that the error has had no effect. Firstly note that its output will not actually be picked up on this cycle by *FD_4* in any case. Second, note that FD_4 is also affected by the *CLK* error, and produces a stale value to be picked up by the warning light output. However, suppose *LUT3* does produce an incorrect value. This will also be ignored this cycle by *FD_1* as it is affected by the late error but may affect the next cycle if there is another late failure.

To summarise some other results, the analysis also shows that by some other chains of events (caused by an SEE) it is possible for confusing output to be given to the operator, for example indicating that the incinerator is about to overheat whilst also initiating the shutoff valve due to a lack of flame. In this situation we assume that the operator would infer that the system had malfunctioned and so no alteration is felt to be necessary to the design as they would intervene. This is an example of why understanding the effect of an SEE can be useful for the safety case.

One other issue uncovered with the design is that the operator may get a warning light that one sensor is malfunctioning and not get an overheat warning light. However, if this is due to the heat sensor itself malfunctioning then the system could in fact be overheating. Therefore we suggest altering the design to indicate whether it is the heat sensor which is malfunctioning so that the operator can take appropriate action. Note that if the design is altered then a new FPTC network should be produced and analysis should be performed again (steps C.1-C.3). As this can be labourious we intend to introduce some analysis aids of the type discussed in section 4.4 as part of the future work.

## 6. Conclusions

This paper has described how a semi-automated safety analysis process known as FPTC can be adapted for use in determining the system level effects of low-level hardware faults.

In order to develop this technique we need to automate conversion of hardware description files into FPTC networks so that more complex examples can be examined. In addition, we will be adapting the FPTC analysis tool to add the ability to highlight failure pathways through the system of significance so that multiple output effects can be assessed. Another area of interest is to assess if there are reusable techniques to avoid certain failure paths. Finally, we will adapt the technique so that configuration files with place and route information can also be examined.

With these changes in place the technique could be used to assess complex safety critical FPGA designs, potentially supporting the certification of FPGAs in less conservative system architectures and, further, leading to weight and cost savings.

## 7. Acknowledgments

The authors would like to thank the U.K. Ministry of Defence for their support and funding.

## 8. References

[1] Emmert, J. M., C. E. Stroud, B. Skaggs and M. Abramovici, "Dynamic Fault Tolerance in FPGAs via Partial Reconfiguration", *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000.

[2] Graham, P., M. Caffrey, J. Zimmerman and D. E. Johnson, "Consequences and Categories of SRAM FPGA Configuration SEUs", *Military and Aerospace Programmable Logic Devices International Conference*, 2003.

[3] IEC, "Electronic Design Interchange Format (EDIF) (IEC 61690-2)", 2000.

[4] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems (IEC 61508)", 2000.

[5] Isaac, T. A., "Firmware in Safety Critical Subsystems", *International System Safety Conference*, Providence, Rhode Island, USA, 2004.

[6] Kanstensmidt, F. L., L. Sterpone, L. Carro and M. S. Reorda, "On the Optimal Design of Triple Modular Redundancy Logic for SRAM-based FPGAs", *IEEE Design, Automation and Test in Europe DATE,* 2005, pp. 1290 – 1295.

[7] Kuphaldt, T. R., "Lessons In Electric Circuits, Volume IV - Digital: Converting Truth Tables into Boolean Algebra", 2007.

[8] Leveson, N. G., "Safeware", Addison-Wesley, 1995.

[9] Ministry of Defence, "Safety Management Requirements for Defence Systems, Part 1 Requirements (00-56)", in Ministry of Defence, ed., U.K. Ministry of Defence, 2007.

[10] Paige, R. F., L. M. Rose, X. Ge, D. S. Kolovos and P. J. Brooke, "Automated Safety Analysis for Domain-Specific Languages", *Workshop on Non-Functional System Properties in Domain Specific Modeling Languages*, 2008.

[11] Rousseau, B., P. Manet, D. Galerin, D. Merkenbreack, J. D. Legat, F. Dedeken and Y. Gabriel, "Enabling certification for dynamic partial reconfiguration using a minimal flow", *Design, Automation and Test in Europe*, Nice, France, 2007, pp. 983-988.

[12] RTCA/EUROCAE, "Design Assurance Guidance for Airborne Electronic Hardware, DO-254/ED-80", RTCA/EUROCAE, 2000.

[13] RTCA/EUROCAE, "Software Considerations in Airborne Systems and Equipment Certification, DO-178B/ED-12B", RTCA/EUROCAE, 1992.

[14] Sterpone, L. and M. Violante, "A New Analytical Approach to Estimate the Effects of SEUs in TMR Architectures Implemented Through SRAM-Based FPGAs", *IEEE Transactions on Nuclear Science*, 52 (2005), pp. 2217-2223.

[15] Wallace, M., "Modular Architectural Representation and Analysis of Fault Propagation and Transformation", *Proceedings of the Second International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures*, Elsevier, 2005, pp. 53-71.

[16] Xilinx, "Design Elements: http://toolbox.xilinx.com/docsan/xilinx5/data/docs/lib/lib0050_34.html", 2008.

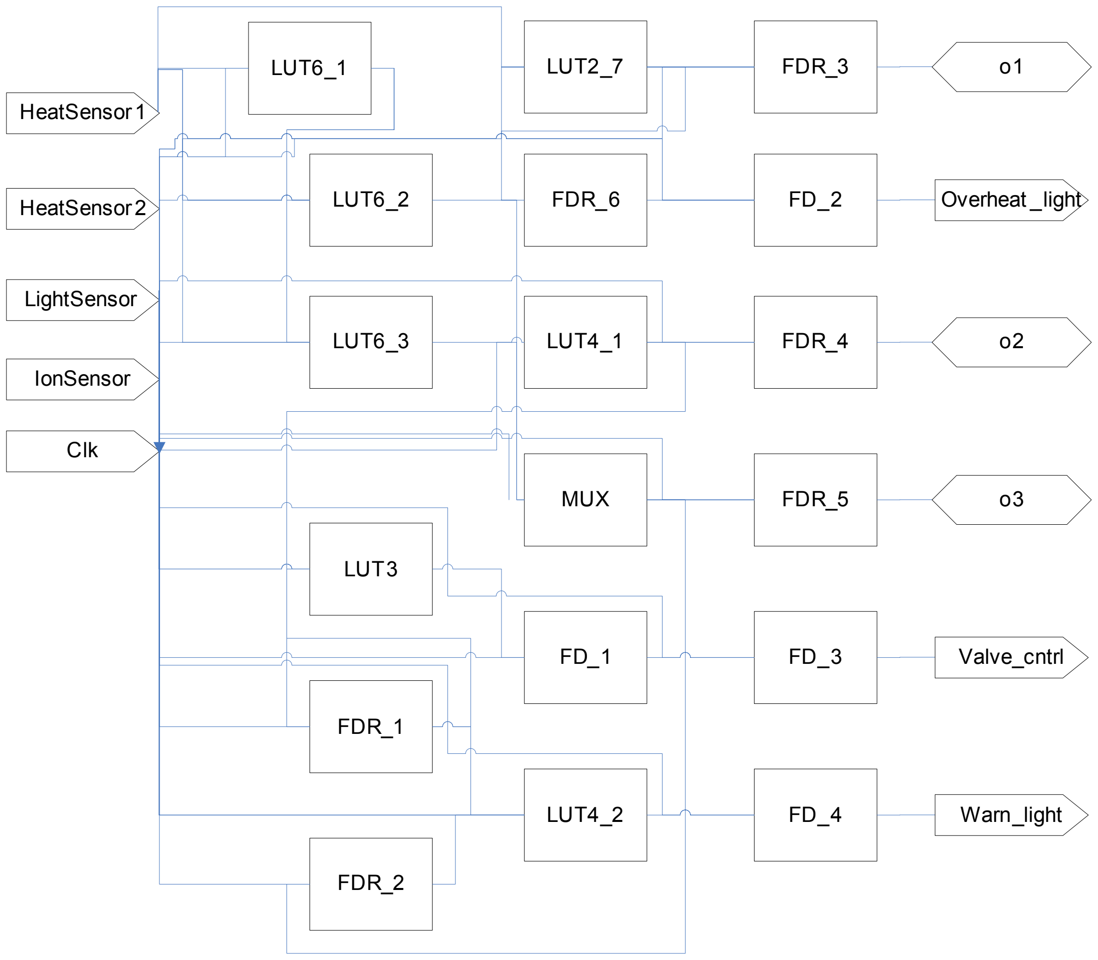

**Figure 6 Block diagram representation of incinerator example**

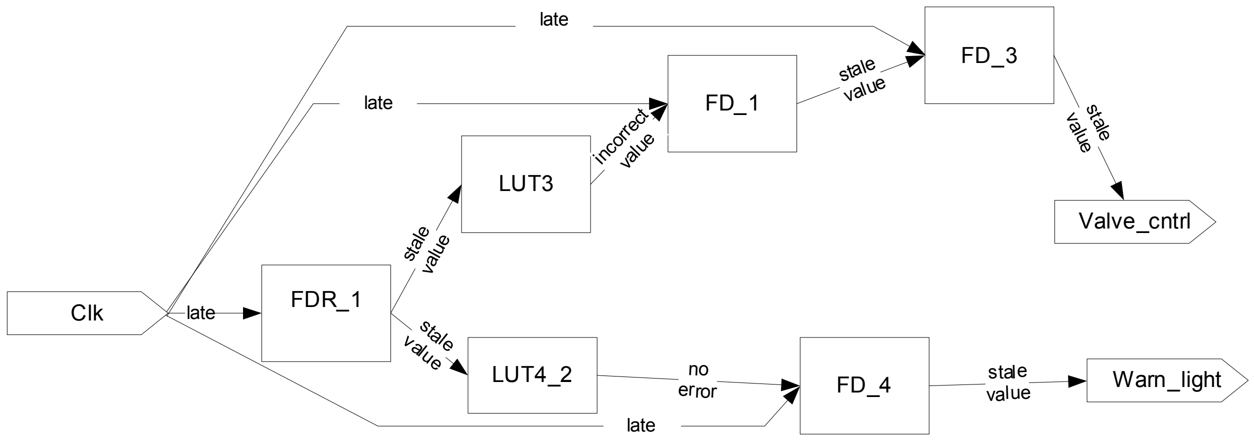

**Figure 7 FPTC clock late example**