Abstract—One effective fault injection approach involves instrumenting the RTL in a controlled manner to incorporate fault injection, and evaluating the behaviour of the faulty RTL whilst running some benchmark programs. This approach relies on checking the effects of faults whilst the design is executing a specific binary image, and therefore the true impact of the fault is limited by the shadow of the program image. Another limitation of this approach is the use of extra hardware for fault injection which is not needed during the fault-free running of the design. The aim of this paper is to propose a new approach for transient fault injection based on symbolic simulation and model checking that circumvents the problems experienced due to application dependent fault injection and RTL modification. In this paper we present our approach and analyse the effect of transient faults whilst the design is executing a specific binary image, and evaluating the behaviour of the faulty RTL whilst running some benchmark programs. Our approach can be applied generally to any faulty design, not necessarily a processor.

I. INTRODUCTION

The goal of fault tolerance is to make the hardware work correctly in the presence of faults. This usually entails conducting a study on fault injection in a hardware design (RTL), to learn about the possible effect of faults, and then devise a technique to mask the effect of these faults. A transient fault or a single-event-upset (SEU) poses a formidable challenge due to the multi-dimensional nature of the fault. Two dimensions identified earlier [1] have been spatial, and temporal. We have identified a third one – the data dimension. The spatial dimension means, a transient can hit any component of a design, which means we need to analyse all components and all paths of a design. The second is temporal - a transient can hit at any time, thus one needs to cover all temporal possibilities during the running of a design. The third dimension - data means that a transient can hit a circuit whilst it is processing some data. This is especially true of processors with their own memory and they execute a program when a transient hits it. So one needs to analyse the effect of a transient fault on a design taking into account that the design could be executing any application (data). This is the main focus of this paper.

Conventionally [2–4], researchers have studied transient fault by modifying the design to incorporate fault injection hardware, loading a set of benchmark programs, and evaluate the effect of faults on the design whilst it is being simulated. There are a few problems with this approach. First is that one can study the impact of fault on an RTL in the presence of only a finite number of benchmark programs (data). The problem is in finding out which application programs can help the designer maximise the ability to capture the impact of faults. Even if there is a consensus on which programs can best capture the most diverse set of functionalities of the design, carrying out a simulation run in the presence of faults, at all time instances (temporal) and for all paths (spatial) in a design is at best incomplete and inconclusive. Another problem of this approach is that the original design has to be modified to incorporate fault injection pins, which means the design being checked for faults is not the same design that the designer intended to have in the first place. Figure 1 shows how fault injection has been studied traditionally.

Fig. 1. Existing fault injection approaches. Traditionally fault injection is carried out by comparing the behaviour of two different RTLs, one that has been instrumented to incorporate faults, and the other – golden RTL – which expresses the intention of the designer in a fault-free case. Both these RTLs are simulated whilst running specific binary program images. In the instrumented RTL, faults are injected using a fault library and injection functions. The comparison is usually done by comparing the output values of gates, registers, memory elements and so on.

The key idea in our approach is that we inject faults in the properties, rather than the RTL. A property in our case is derived from informal specifications about the design, and it expresses the behaviour of a specific input-output interface of a design symbolically, using notations within a formal logic.

We articulate two sets of properties, one that describes a fault-free functioning of the circuit and other that captures the intended fault. We then verify both these sets of properties against a given RTL using a model checker. The property that expresses the fault will typically return a counter-example from a model checking run, and these counter-examples (Boolean formulas) are then used to understand what has gone wrong in the circuit due to the fault.

Typically, in symbolic simulation one places symbolic Boolean variables at different circuit nodes\(^1\) during simulation.

\(^1\)The nodes can be either input or output ports or nets.
The symbolic variable itself is an abstraction that encodes the two logic states 0 and 1. The outcome of the simulation is either the Boolean formula True, or a complex Boolean expression (built from variables placed initially during simulation) that captures the inter-dependency amongst circuit nodes. Because the variables themselves can represent any state 0 or 1, a Boolean formula built from them captures in an abstract and generic way, all the possible states consisting of 0s and 1s that would make the formula true. The power of debugging comes from interpreting these symbolic counter-examples.

In this paper, we show how by using the framework of symbolic trajectory evaluation (STE) [5] - a temporal logic based symbolic simulation and model checking method, we can perform symbolic, transient fault injection on a 32-bit multi-cycle RISC processor, in an application independent way, and without changing in any way the RTL in which we inject faults. In principle, one can use other model checking techniques, our approach is not limited to using only STE. We provide a brief note on the preliminaries that are required to understand the details presented in later sections. In this section, we assume familiarity with logical connectives that appear in propositional logic such as \( \neg, \land, \lor, \Rightarrow \). We use \( \forall \) quantifier from first-order logic to express the fact that something holds “for all values”, and use the \( \vdash \) notation to denote a theorem. We also use the notation \( (a \rightarrow b | c) \) to express “if \( a \) then \( b \) else \( c \)”.

II. RELATED WORK

Recently, there has been an emergence of using formal property checking to study fault injection [6–8]. The basic technique employed is to instrument the RTL in a controlled way to incorporate fault injection, and then check the behaviour of the faulty RTL whilst running specific benchmark programs, using a formal property language. The problem with this approach is that due to extra fault injection hardware being inserted in the original RTL, extra variables are introduced during model checking and it becomes considerably difficult due to the increase in the state-space. Moreover, Leveugle et al. [7] noted that the property checking language PSL used by them and also used by Krautz et al. [8] was difficult to use in practice. Even though Hazenhurst et al. [6] used STE for property specification and model checking, they manipulated low-level FSMs which is extremely tedious, compute intensive and less intuitive.

III. STE MODEL CHECKING

We wanted to explore features in our property specification language that would enable us to write properties about complex hardware designs with ease, that could be used for model checking efficiently. STE has been used successfully in large-scale datapath verification [9] of microprocessors. STE gains its strength by combining the ideas of ternary modelling (using 0,1 and X) with symbolic simulation (using symbolic variables) over time. The presence of Xs provides abstraction necessary to handle the complexity of verifying symbolic properties during model checking.

Circuit models in STE are defined on three-valued states, and are constructed on-the-fly during simulation, from the FSM (.exe) representation of the circuit. Specifications in STE, take the form of what are known as symbolic trajectory formulas. Formally, we define the syntax of formulas [5, 9] as follows:

**Definition 1. Syntax of STE formulas**

\[
\begin{align*}
  f &\triangleq n = 0 & \text{- node } n \text{ has value } 0 \\
  | & n = 1 & \text{- node } n \text{ has value } 1 \\
  | & f_1 \land f_2 & \text{- conjunction of formulas} \\
  | & f \text{ when } G & \text{- } f \text{ is asserted only when } G \text{ is true} \\
  | & \neg G & \text{- } G \text{ holds in the next time step}
\end{align*}
\]

where \( f_1 \) and \( f_2 \) range over formulas, \( n \in \text{string} \) ranges over the nodes of the circuit, and \( G \) is a propositional formula over Boolean variables (i.e. a Boolean ‘function’) called a guard.

The advantage of using a Boolean expression is in specifying conveniently many different operating conditions in a compact form. The various guards that occur in a trajectory formula can have variables in common, so this mechanism gives STE the expressive power needed to represent inter-dependencies among node values. For example, we can associate an arbitrary propositional formula \( G \) with a node using the construct ‘\( n \) is \( G \)’ defined by

\[
n \text{ is } G \triangleq ((n \text{ is } 1) \text{ when } G) \text{ and } ((n \text{ is } 0) \text{ when } \neg G)
\]

We also use a convenient form of expressing the temporal formula, by using from and to functions.

\[
f \text{ from } i \text{ to } j \triangleq N^i f \text{ and } N^{i+1} f \text{ and } \ldots \text{ and } N^{j-1} f
\]

where the convention is that \( N^0 f = f \). The interested reader is referred to [5]. For implementation efficiency Boolean expressions are denoted by binary decision diagrams (BDDs), and an STE predicate is represented as a 5-tuple. Thus, the predicate expressing “net23 is \( a \land b \)” from 6 to 10 when \( \neg p \) would be represented in Forte by: \( \neg p, \text{”net23”}, a \land b, 6, 10 \).

Verification takes place by testing the validity of an assertion or property, of the form \( (A \Rightarrow C) \), where both \( A \) and \( C \) are trajectory formulas. Intuitively, the antecedent \( A \) provides the stimuli to the circuit, and the consequent \( C \) expresses what the designer expects to see. In practice, every successful STE run i.e., a run that returns the value True, is a theorem that holds for all the Boolean variables mentioned in the property. However, when the outcome of an STE model checking run is a counter-example, it is an indication of a bug in the hardware, and in the context of STE, it means that if we can come up with a satisfying assignment of Boolean values True (logic 1) and False (logic 0) to the Boolean variables in the counter-example, one can explicitly reveal the trace (consisting of 0s and 1s) that would be responsible for the bug. Usually there is more than one way to satisfy the counter-example, and this means that in one symbolic model checking run, we can succinctly capture all the possible traces.
**IV. PROPOSED APPROACH**

The key idea in our approach is that faults are captured in the properties, and are verified against a fault-free RTL. This is possible once we have obtained a golden RTL — one that does not contain any design bugs, and a set of golden properties that capture the intended specification of the designer assuming the design would be free from any fault. Once we obtain the golden pair of RTL and properties, we modify the set of properties, to capture the transient faults we would like to investigate. The modified set of properties with the fault, is then verified against the golden RTL using STE model checking.

We have seen in the previous section that our properties are expressed in terms of \( A \Rightarrow C \), where the antecedent \( A \) specifies the initial stimuli given to the circuit, and the consequent \( C \) expresses what we intend to observe about certain points in the circuit. When we verify a property-with-a-fault against the golden RTL, the fault itself is expressed by modifying \( A \), whilst the expected behaviour of the circuit expressed in \( C \) remains unchanged. Thus, the outcome of this model checking run is a counter-example, a Boolean constraint that needs to be satisfied in order to make the property-with-a-fault work exactly as the property-without-the fault. This Boolean constraint denoted by a BDD, is an abstraction of all the possible traces involving 0s and 1s, which if fixed would make the circuit work correctly, for the faulty property. Since the counter-example is made from Boolean variables placed on the circuit nodes in the antecedent of the property, it gives insight into the architectural inter-dependency amongst the various circuit nodes in the presence of a fault — a piece of information we believe could be useful to the designers for developing tolerance. Below we summarise our approach, and a high-level view is presented in Figure 2.

**Phase I**

1) Given an RTL, obtain a set of informal specifications from a designer, and if they do not exist, develop them.

2) Formalise the specifications in the previous step into a set of formal properties, in our case they are written using the syntax of STE.

3) Using a model checker, verify if the original RTL satisfies the formal properties generated in the previous step, again in our approach, we use STE model checking. If there are no counter-examples generated at this stage, we have obtained a golden RTL \( \mathcal{M} \), and a golden set of properties \( \mathcal{P} \).

4) If there are bugs in the design, and/or properties, we have to refine the RTL and/or properties, until we obtain the golden pair of RTL and properties.

**Phase II**

1) We know that properties describe a specific input-output interface in an RTL, so corresponding to each property \( p \in \mathcal{P} \), we identify the different transient faults that could potentially affect all the different inputs mentioned in the property, and determine at what times, they could do so. We then use this information to construct a set of faulty properties \( \mathcal{F} \).

2) We then verify each property \( p \in \mathcal{F} \) against the golden RTL, using model checking, the outcome of which is a set of counter-examples for each \( p \). These counter-examples are symbolic, Boolean formulas that record the disturbance in the behaviour of the RTL, using the symbolic variables we had used during the simulation run, thus revealing the architectural inter-dependencies amongst different nodes in the RTL in the presence of a fault. These Boolean formulas can be analysed by designers, to design an appropriate tolerance scheme.

Often the findings made as a result of our approach may seem obvious to a designer, but our goal is to precisely formalise these findings in a way that it becomes obvious to the designer with the level of assurance guaranteed through an efficient, comprehensive and formal technique such as the one we have proposed.

**V. EXPERIMENTAL RESULTS**

Our processor shown in Figure 3, is designed in VHDL, and is a 32 bit multi-cycle unipipelined RISC architecture adapted from [10]. It is then synthesised (see Figure 2) using the Altera Quartus II tool to a BLIF model (Berkeley Logic Interchange Format) which is then used for model checking using STE. The BLIF model generated from Quartus II is converted to a finite-state machine (FSM) using \texttt{exl2exe} that is provided as a part of the Forte distribution by Intel. An FSM is represented in Forte by a file with a .exe extension. The choice of this processor is merely to demonstrate a proof-of-concept. All the experiments we carried out were conducted on Linux running in VMware on a computer running a single-core Intel 1.7 Ghz Centrino processor with 2GB RAM.

What we show in terms of functionality of this processor is common to any processor - presence of a PC, a branch prediction logic and so on. The general strategy\(^2\) is to use STE

\(^2\)This strategy is used for processor verification in the absence of faults as well.
inference rules [9,11], to decompose properties about CPU verification into properties about different functional blocks such as fetch, decode, execute, write-back and control, and check for faults in each block separately. Once properties about each block have been examined independently, the integrated design with all the blocks – the full processor — is re-checked for faults. However, when this is done at the level of the full processor, the symbolic property checking itself does not have to be done again for each and every unit; only the interfaces have to be done. The properties that characterise each faulty block simply become instances of the properties that we had checked earlier for each stand-alone unit. The overall set of properties for the full processor in the presence of faults, are stitched from all the component smaller properties about each functional unit.

In this section, we show how we carry out the fault injection in the fetch unit of the processor. The fetch unit consists of a PC, an adder and an Instruction Memory. The adder computes the address of the next instruction when there is no branch detected, and the output of the ALU is not zero. In case a branch is detected the address comes from the AddResult port of another adder (which is a part of the execute unit), and the multiplexor selects between this value and PC+4.

We first show that the fetch unit works correctly in the absence of any fault, and we do this using just one STE property.

A. Program Counter Works Correctly

We assume that AddRes denotes a seven-bit vector of new variables and state that the higher six bits of the PC take on symbolic values between time 1 and 2. We abbreviate this by the NextPC_is_AddResult\(^1\). Similarly, we state that the node “Branch” takes on the symbolic value Branch, abbreviated by Branch_initialised, and “Zero” has a symbolic value

\(^1\)The reader should note that these denote trajectory formulas, and they are used (as shown later on) to construct an STE property.

Zero, denoted by Zero_initialised, both for 10 clock cycles. The bus “AddResult” takes on the symbolic value AddRes for 10 cycles (AddResult_initialised). We denote symbolic values PC[7 : 2], for the higher six bits of the PC “PC[7 : 2]”, denoted by PC_initial. We don’t need to assert any values to the lower two bits of the PC since we have already clamped them to logic 0 in the RTL (to increment PC by 4). We state the formula (NextPC_is_PC_plus_4) that the next state of the PC is incremented by 4, by using the built-in add function (ADD_int_bvn_fix) on integer vector.

\[
\text{let NextPC_is_PC_plus_4} =
\begin{cases}
\text{ADD_int_bvn_fix 1 PC MSB) and} \\
\text{PC[1], PC[0]} & \text{is [F,F]} \text{from 1 to 2};
\end{cases}
\]

We initialise the clock below.

\[
\text{let clock} = (\text{“clk” is F from 0 to 1) and (“clk” is T from 1 to 2)}\
\vdots
\text{“clk” is T from 9 to 10;}
\]

We then verify the property that either the PC is incremented by 4 or takes on the branch address.

//PC works correctly
\[
\neg \text{clock and PC_initial and reset_is_low and AddResult and Zero_initialised and Branch_initialised} \\
\Rightarrow (\text{NextPC_is_AddResult when (Zero \land Branch)) and} \\
(\text{NextPC_is_PC_plus_4 when (\neg Zero \lor \neg Branch))};
\]

The reader should note that we have not quantified all the free Boolean variables in the theorem for the sake of presentation clarity.

B. Fault Injection in the PC

We only needed two properties to check the following two kinds of SEUs in the PC.

1. multiple bits in the PC
2. single bit in the PC

We have developed a library of functions for injecting faults in different units of the CPU. A sample function that injects the fault in the upper six bits of the PC is shown in the property below, it takes the 5-tuple representation of the PC and inverts the symbolic values placed on the PC.

Injecting a bit-flip fault in the higher six bits of the PC, we get a counter-example BDD — Branch \& Zero.

//Multiple bits in the PC are faulty
\[
\neg (\text{Branch \land Zero}) \Rightarrow \\
(\text{clock and (map(\lambda (a,b,c,d,e),(a,b,-c,d,e))PC_initial}) and reset_is_low and AddResult and Zero_initialised and Branch_initialised) \Rightarrow \\
(\text{NextPC_is_AddResult when (Zero \land Branch)) and} \\
(\text{NextPC_is_PC_plus_4 when (\neg Zero \lor \neg Branch))};
\]

We still expect to see that the next value of PC is either the result of branch or a normal PC increment but not anymore because of the fault. What this means is that if we assert the Branch and the node Zero to be a logic 1, the PC will work correctly in spite of the fault. This seems to suggest that in the presence of a branch, the PC is less susceptible to SEUs in the PC. In the property below, we show that the faulty

Fig. 3. CPU datapath showing functional blocks. We present the analysis of the fetch unit shown in black.

We first show that the fetch unit works correctly in the absence of any fault, and we do this using just one STE property.

A. Program Counter Works Correctly

We assume that AddRes denotes a seven-bit vector of new variables and state that the higher six bits of the PC take on symbolic values between time 1 and 2. We abbreviate this by the NextPC_is_AddResult\(^1\). Similarly, we state that the node “Branch” takes on the symbolic value Branch, abbreviated by Branch_initialised, and “Zero” has a symbolic value

\(^1\)The reader should note that these denote trajectory formulas, and they are used (as shown later on) to construct an STE property.
property would behave as a correct property if we can ensure that both the Branch and Zero are asserted high. This is shown by making Branch ∧ Zero an assumption of the theorem that represents the outcome of the faulty property.

We show the effect of injecting a bit-flip in any single bit of the top six bits of the PC.

// Single bit in the PC is faulty
let PC_fault i = clock and (fault r PC initial i)
and reset r is low and AddResult and Zero asserted
and Branch asserted =>
(NextPC is AddResult when (Zero ∧ Branch)) and
(NextPC is PC plus 4 when ((Zero) ∨ (~Branch)));

The outcome of the fault is similar to the one when we inject the fault in all bits of the PC.

⊢ (Branch ∧ Zero) ⊃ ∀ i. (2 ≤ i ≤ 7) ⊃ (PC_fault i)

An 8-bit PC has 256 possible states that it could be in, whilst a transient hits any bit of the PC, or several bits at once. As a result of the fault, the state of the PC would change and this would alter the program execution flow. Even though the modified state of the PC as a result of the bit-flip is a subset of these 256 states, the amount of evaluation that needs to be done using conventional simulation, is quite significant, and in spite of this it would be very difficult to generalise the effect of the fault on the PC in terms of the architectural features of the processor. We believe learning the impact of faults in terms of the architecture generically, is crucial for the designers in order to devise an effective tolerance scheme. The disadvantage of taking a conventional simulation approach becomes even more significant in the presence of SEUs in memories, caches, and other state-holding components in the processor’s datapath.

Due to lack of space we cannot present the case of an SEU affecting the lower-two bits of the PC, and the case when there is a fault in the branch part of the datapath. But we need only one STE property in each case that describes each fault. The reader should note that the increase in effort whilst using symbolic simulation is linear not exponential as is the case with test based approach, using conventional simulation.

C. Verifying that instruction memory works correctly

Our Instruction Memory is 256 deep and 32 bits wide. Here, we show how we verify that the memory works correctly using one STE property. We assume we have a predicate WriteData saying that the vector of BDD variables denoted by WD is placed onto the Write data (“WriteData[31 : 0]”) port of the memory between time 0 and 1. Similarly we assume we have asserted the predicate ReadAddress and WriteAddress that assert 8-bit symbolic values at the read address and the write address ports respectively. Also let us assume that the memory read (MemRead) and the write (MemWrite) is asserted.

We define 256 distinct constants IMem0...IMem255 that denote 32-bit names for each of the 256 rows of the memory. Further, we assume that we have declared 256 distinct 32-bit symbolic BDD variables to initialise the memory. We will denote this vector of variables as mem0...mem255.

Memory is initialised by assigning symbolic values to the names of the memory elements.

let InitMem =
IMem0 is mem0 from 0 to 1 and
:
IMem255 is mem255 from 0 to 1;

We would like to check if the read-after-write operation in the memory works correctly. For this we define a symbolic function below that says that if the read takes place from the same location where a new data (WD) has been written, it should fetch the new data, else the old state of the memory (memi) is retained.

let Read_after_write_fn =
(RAdd = Zero) → ((we ∧ (WAdd = Zero)) → WD | mem0)
| (RAdd = One) → ((we ∧ (WAdd = One)) → WD | mem1)
|:
| (RAdd = TwoFiftyFive)
→ ((we ∧ (WAdd = TwoFiftyFive)) → WD | mem255)

The following simply assigns the symbolic read-after-write function to the Read port of the memory.

let ReadData =
“ReadData[31 : 0]” is Read_after_write_fn from 3 to 5;
// Read - After - Write property works correctly
⊢ clock and WriteAddress and MemWrite and WriteData and ReadAddress and MemRead and InitMem
⇒ ReadData

It took 10.2 seconds to verify the above property.

D. Fault Injection in Instruction Memory

We have injected several bit-flip faults in the memory to examine the path the fault takes from the source to the output instruction stream coming from the read port of the memory. There are two different kinds of faults we will discuss here. First is the case when we inject a fault in an entire row. We will call these row faults. The other kind is when we arbitrarily choose a bit anywhere in the memory and flip its value, known as a bit fault. We developed one property each for capturing the row fault and the bit fault. These were parameterised in terms of the bit position, and we executed them to check for 8192 single-bit-flip cases, and 32 row faults, and we discovered that the effect of injecting one bit of a row in the memory is the same as infecting all the bits of a given row. Below we show a sample property for fault in all the bits of row 255.

// Checking if read after write property is correct
// when we inject an SEU in row 255

Checking if read after write property is correct

let ReadData and MemRead and WriteData and ReadAddress and MemWrite and InitMem
⇒ ReadData

The reader should note here that in practice the memory verification runs were carried out using symbolic indexing [9], not quite literally the way they have been shown in this paper for presentation clarity.
What we have observed is that for a given row if we inject a bit-flip in any of the columns of the memory, the counter-example remains the same. However, from one row to another the counter-examples vary, and this is for a good reason. Our memory is organised as a grid of 256 rows (0 to 255) and 32 columns (0-31 bits). When we inject the fault in the entire 32 bits of row 255, and check the read-after-write property again in presence of this fault, we can only obtain the correct values of memory during the read-after-write operation, if we do not read from the faulty locations of the memory — which is exactly what the counter-example for this fault states. The counter-example says that as long as we write to the address location indexed by the symbolic values WriteAddress[7 : 0], and all of them take on a logic 1 value, we have written the new data at location 255. If we subsequently read from any other location but location 255, (at least one of the read address bits amongst ReadAddress[7 : 0] take on a logic 0 value), we will be able to assert that the read-after-write works correctly. This is exactly the case with each of the other memory locations when we fault other rows between 0 and 254. Checking the read-after-write for the fault at any particular bit \( j \) in a given row, for each different \( j \) between 0 and 31, the counter-example generated for each \( j \) is identical, and in fact is the same as the case for the fault in entire row \( i \). The reason is exactly the same as explained above for the fault in each row. It took 176.67 minutes (approx 3 hours) to completely check all the 8192 single-bit errors, and 148.76 minutes (approx 2.5 hours) to check for all 32 row faults, on a 1.7 Ghz Intel Centrino, with 2GB RAM, running Linux in a virtual machine. The benefit of using symbolic simulation and model checking is that we get a reduction from an exponential to linear. We did not have to conduct \( 2^{(m \times n)} \) simulations (for \( m \) rows and \( n \) columns) for analysing single-bit errors in the memory; instead we only did \( m \times n \) simulations. Whilst performing a single-bit injection, we could also pick any other arbitrary row to inject a fault, and we only needed to conduct \( m \times n \) simulations rather than \( 2^n \times m \). We have verified the other functional units of the processor in a fault-free case, using 9 properties for the decode unit, 15 for the execute, 2 for write back stage, and 5 for the control. Due to space restrictions we cannot elaborate on the fault analysis of these units in this paper.

VI. Conclusions and Future Work

This paper presented a new approach of fault injection that analyses the impact of transient faults on a microprocessor independent of any specific application or benchmark program, and without modifying the original design. We are able to do so because we use symbolic simulation and model checking.

By harnessing the strengths of symbolic simulation based specification and verification, we address the spatial-temporal-data dimension of the transient fault. Injecting faults systematically in all the inputs mentioned in the properties we are able to check the effect of a fault in all possible places in the circuit. This allows us to address the spatial aspect of the fault. Because our properties can describe behaviour over time, we are able to explore the injection of a fault at different time points, by obtaining symbolic, counter-example traces. This way we are able to analyse the effect of a fault over varying time intervals, thereby allowing us to address the temporal nature of the transient fault. Since a symbolic simulation run is an abstraction of several simulation runs over 0s and 1s, we get an exponential reduction in a number of individual simulations over 0s and 1s, but also an exhaustive cover over all the possible patterns that one needs to simulate using 0s and 1s, by using a single symbolic simulation run. This allows us to address the data dimension. The main challenge in our work was to craft a set of properties, that can capture the correct behaviour of the design, a significant problem that has been recently studied by Funmi et al [12]. At present, we derive our specifications from a textbook description [10] of a RISC processor. Fault injection itself does not pose much demand on resources (time and memory requirements), but the analysis of the counter-examples can be at times, challenging. However, we believe that the designers would benefit from these counter-examples since they are an architectural image of what has gone wrong due to a fault. In the long run it would be good to have an automated program that discovers insights about the tolerance, from these counter-examples. As part of our ongoing work we are looking to investigate a similar problem which is to work out the minimal bit of redundancy that is required to accomplish the maximum fault tolerance for the most critical parts of a processor.

VII. Acknowledgement

This project was funded through an EPSRC grant no EP/D057663/1 for the project on Reliable Low Power Embedded Computing System.

REFERENCES