Specifying and verifying fault-tolerant hardware

Scott Hazelhurst* Jean Arlat†

March 9, 2002

Abstract

Fault tolerant systems are an important class of system, often used in safety critical or highly available applications. For these systems, as well as verifying the functional and timing properties we must verify that the fault-tolerant mechanisms do protect the system in the ways expected.

This report proposes an integrated framework for specifying and verifying fault-tolerant systems: functional, timing and fault tolerance properties. The specification is given using a temporal logic, TL, as a set of assertions of which describe the behaviour as well as the faults which should be tolerated. The faults themselves are represented as trigger-action pairs: the trigger says when a fault manifests itself, and the action says how the fault manifests itself.

The system being verified is represented as a finite state machine (FSM). The fault descriptions are used to construct observer and saboteur FSMs, which when composed with the original FSM allow a wide range of faults be modelled and fault tolerance properties verified. The verification is done using a model checking algorithm called symbolic trajectory evaluation. This framework has been implemented in the VossProver verification system, and a case study has been carried out, with promising experimental results.

1 Introduction

The importance of building fault-tolerant systems for safety-critical applications has been recognised for well over 30 years. Given their purpose, it is especially important to validate that they do have their desired or claimed fault tolerance properties. Some very successful evaluation schemes have been proposed, typically using schemes of fault-injection coupled with testing (see [11] for a discussion). Although testing-based techniques are successful, there are some limitations to these approaches: fault tolerance properties are often expressed informally; and just as exhaustively testing functional properties of a system is an intractable problem, so is testing fault tolerance properties. Though a high degree of confidence can be obtained using the appropriate testing methods, this is highly computationally intensive, and there must still be uncertainty about the result.

In other domains, formal methods have been proposed as a solution to these problems, and especially with hardware verification a large degree of success has been obtained [13]. Although formal methods have also been used in verifying fault-tolerant designs or specifications (e.g. [3, 14, 15, 18]) formal methods have not had wide-spread use in verifying fault-tolerant systems, especially verifying designs at a relatively low-level of abstraction.

This report explores the use of formal methods in specifying and verifying fault-tolerant hardware systems. The key questions explored are:

- What is a suitable language for specifying the desired fault tolerance properties?
- How can formal verification techniques be used for verifying these properties?

*School of Computer Science, University of the Witwatersrand, Johannesburg, South Africa, scott@cs.wits.ac.za

†1Laboratoire d'Analyse et d'Architecture des Systèmes, Centre National de la Recherche Scientifique, Toulouse, France, arlat@laas.fr

This report proposes a method of expressing fault tolerance (FT) properties of interest, and a complementary method of verifying these properties. Using these methods, the following design methodology is proposed. (1) The system is formally specified — both nominal and FT behaviour. (2) The basic non-fault-tolerant design is verified (where this design can be clearly distinguished from the fault-tolerant one). (3) The third phase is the verification of the fault-tolerant design without the presence of faults (to show that the introduction of fault tolerance has not introduced errors). (4) The final phase is the verification of the circuit in presence of faults to show that the fault tolerance mechanisms work. The main focus of this paper is the specification and verification of FT properties.

Outline: Section 2 presents the basic framework for specification and verification (based on temporal logic and model-checking). Section 3 presents a method of specifying and verifying fault tolerance properties. Section 4 presents a case study to evaluate the approach and identify its strengths and weaknesses. Section 5 concludes and suggests appropriate future research.

2 Framework for specification and verification

The choice of a specification language is difficult because there are many competing requirements of both a technical and human origin. Natural language and first-order logic are both expressive, but neither are ideal. Natural language specifications are imprecise, and first-order logic specifications may quickly become too detailed to be understandable.

This paper explores the use of a temporal logic for expressing FT properties. The major motivation for this is that temporal logics have proved very useful for specifying functional and timing properties of systems, so if a temporal logic can be used for expressing FT properties as well, a uniform framework can be provided for specification. So it is useful to know the strengths and weaknesses of a temporal logic based approach to specifying fault-tolerant systems.

2.1 The logic TL

Only a brief introduction to the logic is given here – for details see [9]. Systems are modelled as finite state machines, i.e., by a set of states $\mathcal{S}$ and by a deterministic next state function $\mathbf{Y} : \mathcal{S} \to \mathcal{S}$. (Note that the state space is modelled as a lattice which allows a certain amount of non-determinism to be expressed implicitly [6, 17].)

The specification is done using the temporal logic TL [10]. The core of TL is a set of predicates which allows the description of the instantaneous state of the system, e.g. whether a node in a circuit has a certain value or whether two nodes are related in a certain way. We denote the set of core *simple* predicates G. Typical predicates might be: $[\mathit{Clk}] = \mathsf{H}$ (is the clock high?); $[\mathit{Reset}] = \mathsf{L}$ (is the reset line low?); and $[\mathit{Score}] < 16$ (is the value of group of lines identified by *Score*, when considered as a bit-vector, less than 16?).

Predicates are combined using logical operators such as conjunction and negation, and temporal operators which allow us to refer to time-dependent behaviour. TL has two temporal operators: *next-time* and *until*. In practice, only the next-time operator is used. Although TL is comparatively inexpressive, it has been used successfully in a range of examples [9], and its simplicity supports a very efficient model-checking algorithm. The syntax of the logic is given by the following BNF —

$$\mathrm{TL} ::= \; G \mid \mathrm{TL} \wedge \mathrm{TL} \mid \neg \mathrm{TL} \mid \mathtt{Next}\,\mathrm{TL} \mid \mathrm{TL}\,\mathtt{Until}\,\mathrm{TL}.$$

While the truth of a predicate is evaluated with respect to a state, the truth of a TL formula is given with respect to a sequence of states, since it can refer to a number of time instants. The informal semantics is that the first state in the sequence refers to time 0 and successive states in the sequence refer to successive instants in time. The formal semantics of a formula is given by the satisfaction relation *Sat* ($\mathit{Sat} : \mathcal{S}^\omega \times \mathrm{TL} \to \mathcal{Q}$). Given a sequence σ and a TL formula g, *Sat* returns the truth of g with respect to the sequence σ. *Notation*: Let $\sigma = s_0 s_1 s_2 \ldots$ be a sequence in $\mathcal{S}$: then $\sigma_i = s_i$, and $\sigma_{\geq i} = s_i s_{i+1} \ldots$.

Definition 2.1. *Semantics of* TL

1. *If* $g \in G$ *then* $\mathrm{Sat}(\sigma, g) = g(s_0)$.
2. $\mathrm{Sat}(\sigma, g \wedge h) = \mathrm{Sat}(\sigma, g) \wedge \mathrm{Sat}(\sigma, h)$
3. $\mathrm{Sat}(\sigma, \neg g) = \neg \mathrm{Sat}(\sigma, g)$
4. $\mathrm{Sat}(\sigma, \mathtt{Next}\, g) = \mathrm{Sat}(\sigma_{\geq 1}, g)$
5. $\mathrm{Sat}(\sigma, g\, \mathtt{Until}\, h) = \vee_{i=0}^{\infty}(\mathrm{Sat}(\sigma_{\geq 0}, g) \wedge \ldots \wedge \mathrm{Sat}(\sigma_{\geq i-1}, g) \wedge \mathrm{Sat}(\sigma_{\geq i} h,))$

Examples and Derived Operators: Disjunction and implication are examples of derived operators. Other derived operators are possible. The most important derived operator is the `During` [] operator defined as: $\mathtt{During}\,[(f_0, t_0), \ldots, (f_n, t_n)]\ g \stackrel{\text{def}}{=} \bigwedge_{j=0}^{n}(\bigwedge_{k=f_j}^{t_j} \mathtt{Next}^k g)$, which asks whether g is true from time f_0 through t_0, f_1 through t_1, ... , and from f_n through t_n. Here are some examples.

- *diff* $([Output], x_1 + x_2) < 2 * delta$. Is the absolute difference between the value on the set of lines denoted by *Output* and the sum of x_1 and x_2 less than $2 \times delta$? (Names of state components are in square brackets, so here *Output* is the name of a state component, while x_1, x_2 and $delta$ are variables and *diff* is a function.)
- $[Clk] = \mathsf{H} \wedge \mathtt{Next}^{10}([Clk] = \mathsf{L} \wedge [Reset] = \mathsf{L})$: at time 0, is the clock high, and at time 10 are the clock and reset lines low?
- $\mathtt{During}\,[(0, 9), (20, 29)]\ ([Clk = \mathsf{L}]) \wedge \mathtt{During}\,[(10, 19), (30, 39)]\ ([Clk = \mathsf{H}]) \wedge$
  $\mathtt{During}\,[(0, 2)]\ ([Reset] = \mathsf{H}) \wedge \mathtt{During}\,[(3, 39)]\ ([Reset] = \mathsf{L})$

  The formula asks if the clock is low for 10ns, then high for 10ns, then low for 10ns, and then high for 10ns; and whether the reset line is high for 3 ns (time 0 through 2 inclusive), and then low from time 3 to time 39.

2.2 Specification of systems

The specification of a system's nominal behaviour is given by a set of assertions. Each assertion consists of a pair of TL formulas, and is written like this: $\langle g \Longrightarrow\!\!\!\!\gg h \rangle$. If a model $\mathcal{M}$ satisfies this assertion, in every run of the system in which g is true, h is true too. g, the *antecedent*, can be thought of as supplying the 'input' or 'stimulus' to the circuit, while h, the *consequent* is the expected reaction to the stimulus. Where necessary we write $\mathcal{M} \models \langle g \Longrightarrow\!\!\!\!\gg h \rangle$ to emphasise that the assertion is about model $\mathcal{M}$.

Both functional and timing properties can be specified this way. For example, the following specification could describe the behaviour of a multiplication circuit, describing the result (including bit-widths), when the inputs must be stable and when the output will be stable:

$$\begin{array}{c} \mathtt{During}\,[(0, 9), (20, 29)]\ ([Clk = \mathsf{L}]) \wedge \mathtt{During}\,[(10, 19), (30, 39)]\ ([Clk = \mathsf{H}]) \wedge \\ \mathtt{During}\,[(0, 39)]\ [A] = x[7-0] \wedge [B] = y[7-0] \wedge \\ \Longrightarrow \qquad \mathtt{During}\,[(35, 39)]\ [C] = (x \times y)[15-0] \end{array}$$

Significant technical detail has been omitted here. For example, the state space is a lattice – which is used for abstraction – and the truth domain is a four-valued logic rather than a boolean one. Partly the omission is for space reasons, but also because the methodology of verifying fault tolerance systems proposed here does not rely on the particular temporal logic used, or the model-checking algorithm used. Interested readers should consult [9, 10].

2.3 Symbolic Trajectory Evaluation and the VossProver Verification system

Symbolic trajectory evaluation (STE) is a model-checking algorithm due Bryant and Seger [17], and extended by Hazelhurst and Seger [9]. It is particularly suited for hardware verification, especially where accurate models of system behaviour, including timing are important. STE has a complementary compositional theory, and has been applied to a range of different circuits [9].

The VossProver verification system is built on top of Seger's Voss system [16]. The Voss system consists of three major components: an efficient implementation of binary decision diagrams [5]; an

event driven symbolic simulator with comprehensive delay and race analysis capabilities; and a general purpose, functional language called FL. STE's compositional theory has been implemented as a simple proof system in the VossProver [8]. Using FL as a script language, a verifier can interact with the proof system to either perform STE on a circuit or to use the compositional theory.

Circuits to be verified are represented internally as finite state machines. These FSMs are constructed automatically from gate-level or switch-level circuit descriptions and a number of standard input formats are supported. Voss also has its own format, called EXE. The FSM models that Voss builds are accurate models of the circuits, including timing.

3 Specification and verification of fault tolerance

3.1 Specifying fault tolerance properties

A fault is modelled with two components: a trigger and a corresponding action. The idea is that the circuit behaves normally, but that whenever the trigger is true, the behaviour of the circuit is modified (as little as possible) so as to make the action true as well. To specify fault tolerance properties, assertions are generalised to contain four pieces of information (the antecedent and consequent as before; a fault trigger; and a fault action) and is denoted thus $g \Longrightarrow\!\!\!\!\gg h$ where θ triggers ϕ (called f-assertions).

Informally, the intended meaning of $g \Longrightarrow\!\!\!\!\gg h$ where θ triggers ϕ is that in every run of the machine $\mathcal{M}$, whenever g is true, so is h *whether or not the fault described by* θ *triggers* ϕ *occurs*. Since the antecedent, consequent, trigger and action may refer to the same circuit components and variables (or different ones), an f-assertion can express a variety of behaviours in the face of faulty and non-faulty-behaviour.

In this framework, the trigger and the action can be *any* TL formulas. However, the exploratory study of Section 4 makes the following restrictions: only the non-temporal fragment of the logic can be used; and the action must non-ambiguously describe the fault for any affected nodes in the circuit.

The primary motivation of this restriction is not so much ease of implementation and efficiency of verification but simplicity of specification. The semantics of what is meant when temporal operators are used in both the trigger and action are tricky and requires some study. Examining the strengths and weaknesses of just using the non-temporal fragment of the logic is a meaningful and useful start, and indicates where extensions are necessary.

3.2 Modelling faults

So far we have modelled the finite state machine as if it were monolithic. In fact, for circuit models a convenient and efficient way to model the circuit is to represent the state of the system as a tuple, with each node (state-holding component) in the circuit making up one component of the tuple. Thus, if a circuit has n components, then $\mathcal{S} = \mathcal{C}^n$ where $\mathcal{C}$ is the set of values that an individual component can take. Similarly, the next-state function is decomposed into n next-state functions, one for each component. So, if $\mathbf{s} = \langle s_1, \ldots, s_n \rangle$, $\mathbf{Y}(\mathbf{s}) = \langle Y_1(\mathbf{s}), \ldots, Y_n(\mathbf{s}) \rangle$, with each Y_i being of type $\mathcal{S} \to \mathcal{C}$. For convenience we name each node by its index in the tuple description of the state space.

Let θ triggers ϕ be a fault. Let FA be the set of nodes that are described in ϕ and for each node $j \in FA$, let v_j be the value required for ϕ to be true. We modify each $\mathbf{Y}_i$ so that it reflects the faulty behaviour if it happens.

Formally, define $\hat{Y}_i(\mathbf{s}) \stackrel{\text{def}}{=} \begin{cases} Y_i(\mathbf{s}) & i \notin FA \\ combine(Y_i(\mathbf{s}), v_j) & i \in FA, \end{cases}$

where $combine$ combines the correct value and the faulty value in the appropriate way. If $\theta(s)$ is false, $combine$ produces the correct result; if $\theta(s)$ is true, $combine$ produces the faulty value. (The actual implementation of $combine$ is more complex than described here to take into account the lattice state-space: readers familiar with STE should note that $combine$ is monotonic.

The global next state function that takes into account the fault is $\widehat{\mathbf{Y}(\mathbf{s})} \stackrel{\text{def}}{=} \langle \hat{Y}_1(\mathbf{s}), \ldots, \hat{Y}_n(\mathbf{s}) \rangle$. Finally, we can define our 'faulty' FSM to be $\widehat{\mathcal{M}} \stackrel{\text{def}}{=} (\mathcal{S}, \hat{\mathbf{Y}})$.

Note that if all temporal logic formulas were allowed in fault descriptions, then the definitions presented here would need to be generalised. This is a topic of further research.

Semantics of an f-assertion: Given a model $\mathcal{M}$, the formal semantics of an f-assertion $g \Longrightarrow\!\!\!\!\gg h$ where ϕ triggers θ is given by

$$\mathcal{M} \models g \Longrightarrow\!\!\!\!\gg h \text{ where } \phi \text{ triggers } \theta \stackrel{\text{def}}{=} \widehat{\mathcal{M}} \models \langle g \Longrightarrow\!\!\!\!\gg h \rangle.$$

3.3 Verifying f-assertions

The verification of f-assertions is accomplished by combining STE and the idea of saboteurs presented in [1]. The basic idea is as follows:

- Suppose we wish to show $\mathcal{M} \models g \Longrightarrow\!\!\!\!\gg h$ where ϕ triggers θ;
- Construct an observer machine $\mathcal{M}_O$ able to observe the state of $\mathcal{M}$. When $\mathcal{M}_O$ detects that ϕ is true of the current state of $\mathcal{M}$, it sets an internal flag to trigger the fault (see also [2, 6] for other work which has used the idea of observers);
- Construct a saboteur machine $\mathcal{M}_S$ that can inject the fault into $\mathcal{M}$. When $\mathcal{M}_O$ triggers the fault, the saboteur 'hijacks' the machine $\mathcal{M}$ and injects the fault described by θ.
- A new machine $\widehat{\mathcal{M}}$, which is the composition of $\mathcal{M}$, $\mathcal{M}_O$ and $\mathcal{M}_S$ is constructed.
- We verify $\widehat{\mathcal{M}} \models \langle g \Longrightarrow\!\!\!\!\gg h \rangle$.

All the constructions described above are done automatically and the only human intervention required is the provision of the circuit description and the f-assertion. The algorithms that perform these constructions and compositions have been implemented in the VossProver system.

4 A case study

This section explores the methodology presented in the previous sections through a case study. The system chosen as a case study is presented in [12] and described in detail in [4]. The overall architecture of the system is presented in Figure 1.

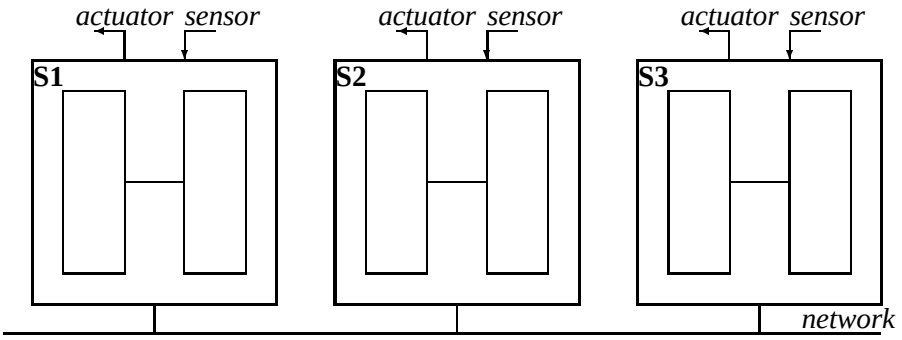

Figure 1: Overall architecture of system

The system has n *channels* (here, 3 channels, labelled *S1*, *S2* and *S3*). Each channel communicates with its environment, taking in data from sensors and then issuing commands to actuators. The channels

communicate with each other on a network. The basic premise of this system is that by implementing the system with n channels, the system is able to tolerate faults, either of sensors or of the channels themselves.

The channels operate the same protocol (described in detail in [4]). In each round of operation the channels all go through 11 phases, and one of the channels acts as master. The protocol works by each channel reading its own sensor data, broadcasting its sensor data to the other channels, followed by a process of agreeing on the data to be used and the result produced. There is also a mechanism for electing a new master.

Fault tolerance is also provided internally. Each channel consists of two *nodes* and an internal connection. One node is the *control node* that actually performs the above steps. The *monitor node* performs exactly the same steps, except that it does not communicate its results outside the channel. However, if the monitor and control produce different results, there is simple circuitry that disconnects the channel from the external network, ensuring fail-safe behaviour. Also, if either the control or monitor do not read inputs quickly enough, the channel will be extracted from the circuit. There is an error state into which a channel goes if such problems are detected; a channel only moves out of the error state if reinitialised.

The implementation examined here was based on the design described and used in [4]. That implementation was given in behavioural VHDL. The design was translated into Voss EXE format (essentially a gate-level description). Though the translation was done by hand, in principle this step could be automated. The major difference between the behavioural VHDL and EXE implementations is the way in which time is dealt with. In behavioural VHDL, it is possible to describe behaviour like 'wait $10\mu s$' directly, which is not possible at the gate-level. At the gate-level a clock has to be introduced to deal with time. For convenience of specification, only one clock is used in this implementation, but it would be straightforward (though the specification would be more cluttered) for each channel to have its own clock. For the version of the system where data and addresses are 32-bit numbers, the circuit had over 100 000 gates and 10 000 state-holding components (the implementation is rather crude!).

4.1 Verification of nominal behaviour

The specification and verification of the nominal behaviour of this system (i.e. the behaviour without faults) is an interesting exercise in its own right. However, as the main point of this paper is the specification and verification of fault-tolerant behaviour only a few points are sketched here. More detail can be found in [7].

A complete specification requires many assertions to be given. In the case study, six sample assertions were verified. The two basic goals were to show that:

- When the system is initialised, the first channel initialised is declared master, and the first round of operation is correct.
- If at the beginning of a round the system is in a consistent state, and one of the channels is the master, then the circuit works correctly and ends the round in a consistent state.

Description of clock: As this circuit is clocked, we need to refer to the clock. The TL formula, *ClockAnt* is defined to do this. The clock goes up and down for 30 clock cycles each of 100ns; for the first half of the cycle the clock is low and the second half it is high. Formally the definition is:

```
During [(0, 49), (100, 149),  ..., (2900, 2949),(3000, 3049)] Clock = F and
During [(50, 99), (150, 199), ..., (2950, 2999),(3050, 3099)] Clock = T
```

Specifying the new master: The informal specification of the circuit is that at the end of each round, that channel which had the sensor that produced the median value of all sensor values should be elected master, i.e., if the channel picked the right value, it should be the master next round. This turned out to be an interesting property to specify. A direct translation of the informal specification turns out to be wrong since more than one channel can pick the same sensor value. Instead this property is specified as:

For each channel, if it is the master then it picked the median value; and exactly one of the channels is the master.

The formal definition is given by

```
During (2700, 2749)
   (if_c S1cont:Pstatus then_c s1_got_med) & (if_c S2cont:Pstatus then_c s2_got_med) &
   (if_c S3cont:Pstatus then_c s3_got_med) & exactly_one_master_active
```

where `Sxcont:Pstatus` is a flag that indicates whether channel *x* is the master. The auxiliary formula `s1_got_med` is defined to be `s1[3-0]= median [s1[3-0],s2[3-0],s3[3-0]]` (and similarly for channels 2 and 3). `exactly_one_master` is just the exclusive or of the status components of each channel (the component has a high voltage if it is the master, a low voltage otherwise).

4.2 Verifying fault tolerance behaviour

As with the nominal behaviour, many aspects of the fault tolerance behaviour can be checked. A primary criterion for a specification language is that it should allow the properties to be expressed in meaningful and concise way. We need experience in specifying fault-tolerant behaviour and this is one of the goals of the paper. The examples given below illustrate the type of fault tolerance properties that can be checked.

Stuck at faults: This is a fault which always occurs, or at some time becomes true always. Here is an example of how such a fault can be modelled: $\mathbf{t}$ triggers $([S1contnwval] = \mathbf{f})$. This says that the network validation signal of channel S1 is always false. We want to show that in this case, the other two channels still work, and agree on the right value (in this implementation, the median of two numbers is the minimum of two).

We define the antecedent *Ant* as

```
ClockAnt & network_signal=F & (During (0,10) S2init=T)&(During (11, 3099) S2init=F)&
(During (0,200)(S1init=T & S3init=T)) & (During (201,3099)  S1init=F & S3init=F) &
(During (1500, 1599) S1input=s1[2-0] & S2input=s2[2-0] & S3input=s3[2-0])
```

This is the first result proved (here we assume that the output is twice the sensor value agreed on by the channels).

```
Ant==>>
  During (2600, 2649)
    S2output=2*min[s2[2-0],s3[2-0]][3-0] and S3output=2*min[s2[2-0],s3[2-0]][3-0]
Fault assumption: stuck at fault    :  S1contnwval = F
```

Expressing relationships In the above example, the consequent is too strong in one way — another implementation might choose the maximum of two values as the median. In another sense it is too weak, since it does not really make explicit the notion of fault tolerance that we want. Here is an alternative result — logically weaker than the previous one, but it gives a more meaningful result. The essence of this result is: provided the sensor values are within a certain range of each other, then the output of the two working channels will be within some acceptable range from the median of all three sensor values.

```
Ant  ==>>
   During [(2600, 2649)] if_c well_behaved_sensors then_c well_behaved_output
Fault assumption: stuck at fault    : S1contnwval = F
```

where `well_behaved_sensors` is defined to be:

```
diff [max [s1[2-0],s2[2-0],s3[2-0]],min [s1[2-0],s2[2-0],s3[2-0]]] < delta[2-0]
```

and `well_behaved_output` is defined as

```
diff [S2output,(2*median [s1[2-0],s2[2-0],s3[2-0]])[3-0]] < 2*delta[2-0]
```

Provided the difference between the maximum and the minimum of the sensor values is δ (for any 3-bit number δ), then the output value of S2 (and analogously S3) will be within 2δ of the median of the right result if S1 fails (i.e., if S1 fails, S2 and S3 will be almost right). δ is symbolic – i.e., we verify the result for all values of δ within a certain range. Also note the care that has to be taken in specifying the bit-widths of the numbers concerned. Again there is some tedium here and it certainly clutters the specification, but it is a detail that needs to be taken care of (since the desired fault-tolerant result is not true if δ is too large, because the system does finite arithmetic). Therefore, the specification precisely describes the fault tolerance limitations.

Triggering faults on input values: The antecedent here is the same as for the stuck at fault. But here, we assert that the fault is only triggered for some values of the input. The consequent then shows that if the fault is triggered we get one result, while if it is not triggered we see the nominal behaviour ('7' is used as the synchronisation signal on the network.)

```
Ant  ==>>
    During [(2600, 2649)]
           S2output = if (s1[2-0] = 7) then (2*min [s2[2-0],s3[2-0]])[3-0]
                      else (2*median [s1[2-0],s2[2-0],s3[2-0]])[3-0]
Fault assumption --- Trigger : s1[2-0] = 7;  Fault    :  S1contnwval = F.
```

Triggering faults on state information: Faults can also be triggered on state information. This is useful when it is difficult to know when (or even if) a fault should be triggered using only input values or time. In this example, we insert an error into S1 when it gets into a state in which it is about to broadcast on the network (it broadcasts 0, rather the right sensor value). Note we force the same error into the control and the monitor node (otherwise it would automatically get extracted from the circuit).

```
Ant  ==>>  During [(2600, 2649)] S2output = (2*median [0,s2[2-0],s3[2-0]])[3-0]
Fault assumption:
Trigger: S1cont:Pstinterchange and S1cont:Pmyid; Fault: S1contdata=0 and S1montdata=0
```

Verifying fault-checking components: At a lower level of abstraction some of the circuitry that implements the fault tolerance can, of course, be checked for its functional behaviour directly. For example, in this circuit the monitor and control nodes check each other. We can show that the circuitry that performs this checking will detect errors. This is straight-forward.

4.3 Computational cost

To assess the practical worth of using formal verification, we need to consider the computational cost, since the computational costs are non-trivial. Of course, one must assess these costs in terms of the costs of not finding errors. And it is often possible to verify smaller versions of design at early stages so that even if the final verification takes many hours to run, during design and implementation the verification algorithm can be used effectively. Nevertheless, computational costs are critical.

Table 1 shows the cost of verifying the nominal and FT behaviour. The largest circuit verified had approximately 10000 state holding components and 150 000 gates. The verification was run on a 500 MHz Pentium II. Six nominal properties and six FT properties were verified, for four different versions of the circuit (varying the datapath bit-width from 4 to 32). For each run, the size of the circuit and the cost of the verifications is shown in the table.

The results show that the cost of verifying the circuit are well within the capacity of the VossProver tool (especially if one considers the fact the figures are inflated by the overheads of loading the system, building the circuit etc, which usually only has to be done once a session). The cost of verification appears to grow more quickly than the number of gates (but still logarithmic in the size of the state space, though a more through analysis is needed here).

Bit-width	4	8	16	32
Number of gates	26000	50000	85000	150000
Cost for nominal properties (s)	63	85	128	308
Cost for FT properties (s)	19	31	69	275

Table 1: Cost of verification of nominal and FT properties in seconds

Human cost: Specification requires significant human insight. The verification of all results shown here is completely automatic (though this is not something that we can expect at this stage in general).

5 Conclusion

This paper has examined the use of formal methods in verifying fault-tolerant systems' designs, presenting an approach to specifying and verifying fault-tolerant systems. The logic used also allows a range of fault conditions to be expressed. These fault conditions are given as trigger-action pairs: the trigger indicates when a fault will occur and the action says what type of fault occurs. Using the ideas of saboteurs [1], the method of symbolic trajectory evaluation can be generalised to be able to verify FT properties.

A case study was performed to evaluate the proposed approach. Overall, the case study shows that the approach is successful. The nominal behaviour of the circuit was verified, and then a range of FT properties were examined including: stuck at faults; faults triggered by input values; and faults triggered by state conditions.

In addition, the expected behaviour could be modelled exactly (e.g., the output is x) or approximately (e.g., the output is within a certain range). It is also possible to verify directly that certain fault-monitoring circuitry performs its task. The experimental results also showed that the computational costs of STE were quite reasonable.

Future research: There are a number of issues for future research:

Language for specifying fault tolerance: The case study explored the use of the non-temporal fragment of TL for specifying the trigger-action pairs. This proved capable of expressing a range of different behaviour. We need to do more case studies to get experience in what type of FT properties need to be proved, in order to find where the limits are. It appears that allowing temporal operators in both triggers and actions would be useful, and the framework of using observers and saboteurs should be able to cope with the extension. One particular anticipated difficulty is the specification of both absolute and relative times in the fault specification.

Use for determining fault tolerance coverage: One interesting possibility is to generalise the specification of the fault and/or antecedent, expecting the verification to fail. The point of this is that the information given by the VossProver explaining why the verification failed could be used to determine fault tolerance coverage.

Compositional theory: A very important technique to overcome the state explosion problem that bedevils symbolic model checking is the use of compositionality. STE has a simple but successful compositional theory that has been implemented in the VossProver [8]. If we wish to use STE to deal with fault tolerance we need to extend the theory for combining assertions to a theory that deals with combining f-assertions.

Improving the performance of algorithms: And, of course, all algorithms can benefit from improvement. Even though in the case study chosen the STE algorithm could easily prove the required assertions and f-assertions, the insatiable demands for memory and CPU cycles needs to be met . . . .

Acknowledgements:

This work was funded in part by the South African National Research Foundation and the Centre National de la Recherche Scientifique.

References

[1] J. Arlat, J. Boué, and Y. Crouzet. Validation-based Development of Dependable Systems. *IEEE Micro*, 19(4):66–79, Jul-Aug 1999.

[2] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In D.I. Dill, editor, *CAV '94: Proceedings of the Sixth International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 182–193, Berlin, June 1994. Springer-Verlag.

[3] C. Bernadeschi, A. Fantechi, S. Gnesi, and A. Santone. Formal validation of fault tolerance mechanisms. In *Digest of FastAbstracts of the 28th International Symposium on Fault-Tolerant Computing*, pages 66–67. IEEE Computer Society Press, 1998. http://www.chillarege.com/ftcs/fastabstracts/389.html.

[4] J. Boué. *Test de la Tolérance aux fautes par injection de fautes dans des modèles du simuation VHDL*. PhD thesis, Institut National Polytechnique de Toulouse, November 1997. LAAS Report 97503.

[5] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] S. Hazelhurst. *Compositional Model Checking of Partially-Ordered State Spaces*. PhD thesis, University of British Columbia, Department of Computer Science, 1996.

[7] S. Hazelhurst and J. Arlat. Specifying and verifying fault-tolerant hardware. LAAS Report 99514, Laboratoire d'Analyse et d'Architecture des Systèmes, Centre National de la Recherche Scientifique, December 1999.

[8] S. Hazelhurst and C.-J.H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDD's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, April 1995.

[9] S. Hazelhurst and C.-J.H. Seger. Symbolic Trajectory Evaluation. In Kropf [13], pages 3–79.

[10] S. Hazelhurst and C.-J.H. Seger. Model checking lattices: Using and reasoning about information orders for abstraction. *Logic Journal of the IGPL*, 7(3):375–411, May 1999.

[11] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[12] H. Kopetz. The time-triggered approach in real-time system design. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, Basic Research, pages 53–66. Springer, 1995.

[13] T. Kropf, editor. *Formal Hardware Verification: Methods and Systems in Comparison*. State of the Art Survey Lecture Notes in Computer Science 1287. Springer-Verlag, Berlin, 1997.

[14] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[15] J. Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, 1999.

[16] C.-J.H. Seger. Voss — A Formal Hardware Verification System User's Guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, November 1993. Available by anonymous ftp as ftp://ftp.cs.ubc.ca/pub/local/techreports/1993/TR-93-45.ps.gz.

[17] C.-J.H. Seger and R.E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in Systems Design*, 6:147–189, March 1995.

[18] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, January 2000.