# ACL2 for the Verification of Fault-Tolerance Properties: First Results [*]

Laurence Pierre, Renaud Clavel, Régis Leveugle
TIMA (CNRS-Grenoble INP-UJF)
46 Av. Félix Viallet
38031 Grenoble cedex - France
{Laurence.Pierre, Renaud.Clavel, Regis.Leveugle}@imag.fr

## ABSTRACT

We target the development of new methodologies for analyzing the robustness of circuits described at the Register Transfer (RT) level, with respect to errors caused by transient faults. Analyzing the potential consequences of errors usually involves fault-injection techniques, using simulation or emulation-based solutions. Our goal is to take advantage of the logical power of theorem proving tools to get alternative solutions that would allow to reason purely symbolically on errors. In this paper we present our preliminary results with the ACL2 theorem prover, in the context of devices that have auto-correction features. First we give a logical definition of the error model as a conjunction of characteristic properties, from which robustness analysis can be performed. Then we improve the methodology to deal with hierarchical systems.

## Categories and Subject Descriptors

B.8.1 [**Reliability, Testing, and Fault-Tolerance**]; F.4.1 [**Mathematical Logic**]: Mechanical theorem proving

## General Terms

Hardware reliability

## Keywords

Robustness analysis, Formal methods

## 1. INTRODUCTION - CONTEXT

Designing dependable circuits requires in particular evaluating, at each step in the design flow, the achieved level of robustness against various types of faults or errors [1]. Critical systems traditionally include systems designed for space missions, aeronautics and other human transports, nuclear plant control, etc. In such systems, an erroneous piece of information may lead to dramatic consequences in terms of human lives. In those cases, *errors* are generally the consequence of natural phenomena such as particle impacts, electromagnetic perturbations, electrical noise or degradations due to aging. The causes of errors, called *faults*, were usually modeled in digital systems as single bit-flips or signals stuck either at 1 or at 0. Faults may be *permanent* or *transient* depending on their physical origin. With the evolution of technologies, circuits are increasingly sensitive to transient faults that have therefore become the main concern for designers. Also, faults increasingly lead to multiple-bit errors that are more difficult to detect or tolerate in the system.

In the last decade, the problem of fault consequences has also become one of the main concerns in another context, i.e. the design of secure circuits, including in particular cryptographic devices. This is the case of circuits for smart cards, but not limited to this particular domain. Such devices generally manipulate secret information, e.g. a secret key to encrypt or decrypt data. Several types of attacks are known, aiming at discovering the secret stored in the circuit. One type of attack is based on disturbing the circuit by voluntarily creating errors during the application execution, using for example a laser. The erroneous results can then be exploited to perform some cryptanalysis (DFA : Differential Fault Attack) [2].

The two contexts deal with different dependability attributes: safety, availability or reliability in the first case, security (and more precisely confidentiality) in the second case. However, the basic concern is the same in both cases from the designer point of view since the goal is to ensure a given level of robustness with respect to faults. The goal is either to guarantee that no error (in a specified set of potential errors derived from the selected fault model) can lead to the feared events, identified as critical from the application point of view, or to limit the probability of such events to an acceptable value. In order to achieve this goal, designers must analyse at design time the potential consequences of errors, and when necessary add additional protections in the circuit. Such an analysis is usually based on so-called *fault injection techniques*, classically using either simulation or emulation [8]. Unfortunately, such techniques require very long experiment durations, that are often not acceptable in particular in the case of complex circuits and multiple-bit errors. In consequence, current practice is based on partial

[*]This work is supported by the French National Research Agency ANR in the framework of the "FME3" project (ANR-07-SESU-006).

analyses, injecting only a subset of all possible errors. Furthermore, the number of injected errors is often limited to a very small percentage. Such an approach can be sufficient in some cases to be reasonably confident in the efficiency of some protection mechanisms. However, this cannot be considered as a guarantee that a given dependability property holds for all possible errors in the specified set. Also, it is not possible with such an approach to precisely quantify the probability of a given event; only estimations can be obtained unless exhaustive fault injections are performed.

Our goal is therefore to develop and evaluate new methodologies helping the designer in better ensuring that the achieved level of robustness is actually sufficient with respect to the application constraints. The focus is on synchronous digital circuits subject to transient faults resulting in single or multiple erroneous bits. Through the use of formal techniques, we mainly target small or medium size circuits due to computation complexity. We however expect a more thorough (and potentially faster) dependability characterization compared with exhaustive fault injections using classical approaches. The main goal is the ability to *formally prove that some dependability properties always hold* for a given set of potential errors due to transient faults. The interest in using formal methods for dependability analysis is growing but, to our knowledge, the application of theorem proving techniques has not yet been considered.

We report here some preliminary results about adapting the features of ACL2 to deal with fault-tolerance properties in the case of auto-correcting circuits. We first recall some existing results, which mainly focus on model-checking oriented techniques rather than on theorem proving methods. Then we describe the framework in which we use ACL2 to reason about transient errors in VHDL descriptions, and the main ideas for characterizing these errors. Finally, we present a solution for improving the efficiency of the approach by decomposing the problem using the hierarchical structure of the descriptions.

## 2. RELATED WORK

Following the seminal proposal of [7], some recent papers document preliminary approaches to applying formal verification to dependability evaluation. They consider various problems, and most of them make use of model checking or symbolic simulation techniques.

The approach of [5] focuses on *measuring the quality of fault-tolerant designs*, and works by comparing fault-injected models with a golden model. Injection points are latches, and a HDL model is used to mimic fault injection. The BDD's that correspond to the fault-injected and golden models are built by symbolic simulation for a given number of cycles. Properties that characterize correction capabilities are checked on these models.

The model checker SMV is used in [10] to *identify latches that must be protected* in an arbitrary circuit. The approach considers soft errors in latches, using the SEU error model. Formal models for all the fault-injected circuits are built (fault injection is performed in one latch for each one of them) and SMV checks whether the formal specification of the original circuit still holds in each case, thus indicating whether the corresponding latch must be protected or not. Results on a Verilog implementation of the SpaceWire communication protocol are given.

The goal of [3] is to *analyze the effects of transient faults*, using both symbolic simulation and model checking. Injected faults are pictured by modifying the premises of the properties that should be satisfied without faults. Counter examples generated by the model checker are used to interpret the effects of the injected faults. Simple examples of fault injection in the program counter or in the instruction memory of an unpipelined RISC processor are reported.

The purpose of [6] is the validation of *mechanisms implemented in software for handling transient hardware faults*. This work focuses on transient hardware faults, more precisely bit-flips in data memory locations. Such faults are emulated by manipulating the variables in programs which undergo symbolic execution. Fault injection is characterized by specific symbolic execution rules that are used for instance to determine the consequences of faults in terms of strongest postconditions (within the verification tool KeY for Java programs).

A definition of the robustness of a circuit in terms of its input/output behaviour is given in [4]. Different fault models are considered and an algorithm to *compute a measure of the robustness* is given: it builds a fault-injected model, “unroll” the circuit and its fault-injected counterpart, and estimates a measure of robustness by SAT-solving equivalence properties. The interest in using induction to improve the method is mentioned.

All these approaches are interesting but they share the same drawback: soft errors are enumerated and processed separately by applying the same procedure to all of them. Our aim is to provide solutions that avoid duplicating similar verifications for each error individually. The solution we propose in this paper is a first step towards this goal. Using the *defspec* and/or *encapsulate* constructs of ACL2, a kind of meta-characterization of soft errors can be implemented. We show that it allows to prove theorems that express robustness properties.

Theorem provers in general, and ACL2 in particular, have already been used in the context of fault-tolerant systems e.g., for verifying fault-tolerant protocols in distributed systems where faulty processors may send conflicting information (e.g. Byzantine agreement protocols [11]). To our knowledge, this is the first time results are reported regarding the use of ACL2 in the framework of hardware dependability analysis.

## 3. OVERALL FRAMEWORK

We consider synchronous circuits described in VHDL at the RT (Register Transfer) level and we target the verification of dependability properties in presence of transient faults.

Our environment to deal with VHDL descriptions is sketched in Figure 1. Using a specialized tool called VSYML [9], we parse the VHDL code, perform symbolic execution, and we get an XML representation of the transition and output functions (for a Mealy machine):

$\delta : I \times S \to S$

$\lambda : I \times S \to O$

where $I$, $O$ and $S$ refer to the sets of input values, output values, and state values (memory elements).

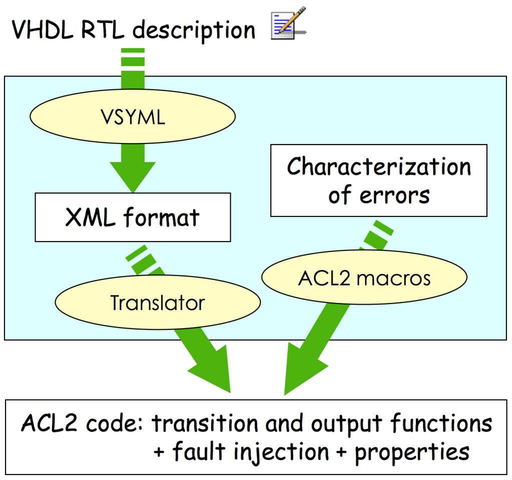
VHDL RTL description
VSYML
XML format
Characterization of errors
Translator
ACL2 macros
ACL2 code: transition and output functions + fault injection + properties

**Figure 1: Fault injection - From VHDL to ACL2**

Our characterization of faults described in section 4 is general and applicable to any RTL description. ACL2 macros have been defined to help instantiate this model mechanically. Hence we get an ACL2 source code that includes the Lisp implementation of $\delta$ and $\lambda$, the appropriate characterization of fault injection, and the properties to be verified (to be supplied manually).

## 4. CHARACTERIZATION OF ERRORS

### 4.1 Errors in ACL2

As a first attempt, we define and then formalize in ACL2 the fault model that corresponds to the *presence of a single or multiple-bit error in a single register* of the circuit. Future work will take into account other realistic models such as single bit flips or multiple-register errors for instance.

Instead of defining explicitly the *fault-injection function* $f$, we characterize it logically as a function that satisfies the following conjunction of properties:

- it takes as parameter a state $s \in S$ and returns a state $f(s) \in S$
- $f(s)$ is different from $s$ (injection is actual)
- only one memorizing element (n-bit register) differs from $s$ to $f(s)$

Such a characterization can easily be implemented in ACL2 using an *encapsulate* (or *defspec*) construct: $f$ is only specified by its signature (more precisely, its number of arguments) and is associated with three "constraints" i.e., three theorems that correspond to the three properties above. To guarantee the consistency of the constraints, a "witness" function has to be defined locally to the *encapsulate* construct, but it is unknown outside this construct. The three theorems are exported, thus $f$ is only represented by these constraints. Here is such an *encapsulate* construct for the representation of the fault-injection function in a register of type "natp".

```
(encapsulate
   (((STD-natp-error *) => *))
;  - Witness
  (local (defun STD-natp-error (x)
           (if (natp x) (1+ x) ''error'')))
; - Properties
  (defthm STD-natp-type1    ; returns a natp
    (implies (natp x)
             (natp (STD-natp-error x))))

  (defthm STD-natp-type2    ; takes a natp
    (implies (not (natp x))
             (equal (STD-natp-error x)
                    ''error'')))

  (defthm STD-natp-def      ; fault-injection is actual
    (implies (natp x)
             (not (equal (STD-natp-error x) x))))

  ; the third property (only one memorizing element
  ; differs) is expressed in each non elementary
  ; component
)
```

This implementation is a simplification of the previous characterization, since the state space is reduced here to only one element. We will see in section 5 that our current implementation of the property "only one memorizing element differs from $s$ to $f(s)$" is expressed in each composite component by a theorem of the form

$$\bigvee_i (f(s) = inject_i(s))$$

where each $inject_i$ translates an injection in the $i^{th}$ memorizing element.

Given such a representation of $f$ and considering for instance the case of a device that has a property of auto-correction in one clock cycle in the presence of single faults, we can verify related theorems e.g., the following one:

*For any initial error-free state $s_0$, if a fault is injected after $n$ clock cycles ($n > 0$) then it will be corrected one clock cycle later* i.e., the resulting state will be equivalent to the resulting state without fault injection:

$$\delta(i, f(\delta^n(\iota, s_0))) \Leftrightarrow \delta(i, \delta^n(\iota, s_0))$$

where $\iota$ is an input sequence, and $i$ is the current input.

### 4.2 Example

Let us consider the example of Figure 2. This is a counter equipped with a TMR (Triple Modular Redundancy) system that ensures fault-tolerance: the memory element (register) is triplicated and a voting system produces a single output. Registers contain integers, and the boolean input *inc* conditions the incrementation. A simple (non compositional) associated VHDL description is as follows:

```
entity Counter is
    port (clock: in bit; inc: in bit;
          reset: in bit; count_out: out integer);
end Counter;

architecture Struct of Counter is
   signal count1, count2, count3 : integer := 0;
begin
  count1_out_process: process(clock, reset)
  begin
    if reset = '0' then count1 <= 0;
    elsif clock'event and clock = '1' then
      if (count1 = count2) or (count1 = count3) then
        if inc = '1' then count1 <= 1+count1;
        end if;
      elsif (count2 = count3) then
        if inc = '1' then count1 <= 1+count2;
        else count1 <= count2;
        end if;
      end if;
    end if;
  end process count1_out_process;
  count2_out_process: process(clock, reset)
  begin
    if reset = '0' then count2 <= 0;
    elsif clock'event and clock = '1' then
      if (count2 = count3) or (count2 = count1) then
        if inc = '1' then count2 <= 1+count2;
        end if;
      elsif (count3 = count1) then
        if inc = '1' then count2 <= 1+count3;
        else count2 <= count3;
        end if;
      end if;
    end if;
  end process count2_out_process;
  count3_out_process: process(clock, reset)
  begin
    if reset = '0' then count3 <= 0;
    elsif clock'event and clock = '1' then
      if (count3 = count1) or (count3 = count2) then
        if inc = '1' then count3 <= 1+count3;
        end if;
      elsif (count1 = count2) then
        if inc = '1' then count3 <= 1+count1;
        else count3 <= count1;
        end if;
      end if;
    end if;
  end process count3_out_process;
  count_out_process: process(count1, count2, count3)
  begin
    if count1 = count2 then count_out <= count1;
    elsif count2 = count3 then count_out <= count2;
    elsif count3 = count1 then count_out <= count3;
    else count_out <= 0;
    end if;
  end process count_out_process;
end Struct;
```

We can verify that this system is able to come back to a correct state after one clock cycle, provided that there is only one erroneous value in the registers $R_1$, $R_2$ or $R_3$. Let $\delta$ and $\lambda$ be the transition and output functions of this system, we use ACL2 to prove the theorems:

1. if the initial state $s_0$ is $(0, 0, 0)$, and if a fault is injected after $n$ clock cycles then it will be corrected one clock cycle later:
   $s_0 = (0, 0, 0) \Rightarrow \delta(i, f(\delta^n(\iota, s_0))) \Leftrightarrow \delta(i, \delta^n(\iota, s_0))$
2. more generally, if the initial state $s_0$ is $(X, X, X)$ where $X$ is any integer:
   $s_0 = (X, X, X) \Rightarrow \delta(i, f(\delta^n(\iota, s_0))) \Leftrightarrow \delta(i, \delta^n(\iota, s_0))$
3. more simply, if fault-injection occurs in any valid state $s_n$ (a state is valid if the values of the three registers are identical), then it will be corrected one clock cycle later:
   $s_n = (X, X, X) \Rightarrow \delta(i, f(s_n)) \Leftrightarrow \delta(i, s_n)$

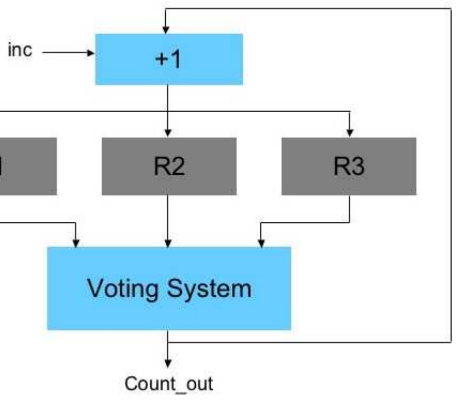

**Figure 2: Counter with TMR**

Similarly we prove that, under the same hypothesis, the primary output *Count_out* always remains valid. CPU times for the ACL2 proofs of the theorems above are[1]:

- Theorem 1: 5.47 seconds
- Theorem 2: 6.35 seconds
- Theorem 3: 0.01 seconds. There is no term with $\delta^n$ in this theorem, thus there is no induction and the proof is therefore immediate.

However, as soon as the size of the circuit becomes more representative, CPU times grow quickly (in particular if there are many control inputs, that give rise to many case splittings). A solution to this problem is to improve the model in order to perform hierarchical proofs.

## 5. HIERARCHICAL MODEL

The ACL2 books for the examples of this section can be found at `http://tima.imag.fr/vds/FME3/`.

### 5.1 Principles

Complex hardware systems are typically described hierarchically as the interconnection of simpler components: elementary gates or arithmetic units, IP's (Intellectual Property),... We can take advantage of this hierarchical construction to perform hierarchical verifications, thus considerably improving the efficiency of the method.

To that goal, we reason as follows with a component $\mathcal{C}_1$ enclosed in a component $\mathcal{C}_2$:

[1]on an Intel Core2 Duo (3.0 GHz) under Linux

From the VHDL description of component $\mathcal{C}_1$, we know

- the sets $I_1$, $O_1$, $S_1$ (deduced from the input/output ports and local signals declarations),
- the transition and output functions $\delta_1 : I_1 \times S_1 \rightarrow S_1$ and $\lambda_1 : I_1 \times S_1 \rightarrow O_1$.

For this component, we also have an error model specified by a function $f_1$.

Using all these characteristics, and locally to component $\mathcal{C}_1$, we determine:

- a predicate $Sp_1$ which is the state recognizer for $\mathcal{C}_1$ i.e., $Sp_1(s) = true \Leftrightarrow s \in S_1$
- a predicate $Sreach_1$ which is the recognizer for the reachable (error-free) states of $\mathcal{C}_1$. For instance, in the example of section 4, reachable states are such that the contents of the three registers are identical
- a set $\mathcal{P}_1$ of fault-tolerance properties (theorems) for $\mathcal{C}_1$.

The definitions of the functions are local to $\mathcal{C}_1$. The outside world, in particular component $\mathcal{C}_2$, only knows the existence of $\delta_1$, $\lambda_1$, $Sp_1$, $Sreach_1$ and $f_1$ (but is unaware of their definitions), and can use theorems $\mathcal{P}_1$ to infer other properties.

Component $\mathcal{C}_2$ is characterized by similar constituents, and its properties $\mathcal{P}_2$ are deduced from $\mathcal{P}_1$ (and possibly from the properties of all other components contained in $\mathcal{C}_2$).

## 5.2 Example

Let us illustrate these principles with a simple version of a cash withdrawal system. The interface between the ATM (automatic teller machine) and its controller is pictured in Figure 3. The controller is given in Figure 4, the corresponding VHDL entity is:

```
entity ATM is
  generic (max_try : natural := 3);
  port (clock : in bit;
       reset : in bit;
       inc : in bit;        -- card insertion
       cc : in natural;     -- card code
       codin : in natural; -- proposed code
       val : in bit;        -- validate
       done_op : in bit;  -- ATM operation completed
       take : in bit;       -- card withdrawn
       outc : out bit;      -- eject
       keep : out bit;      -- keep the card
       start_op : out bit; -- start ATM ooperation
       e_detect : out bit); -- error detected
end ATM;
```

The inputs *inc* and *take* are true respectively when the card is inserted and withdrawn, and *cc* is the card code. The input *codin* is the code that is entered through the keyboard, *val* is used to validate, and *reset* to cancel the operations. The outputs *outc* and *keep* indicate respectively that the card can be withdrawn or that the machine keeps it.

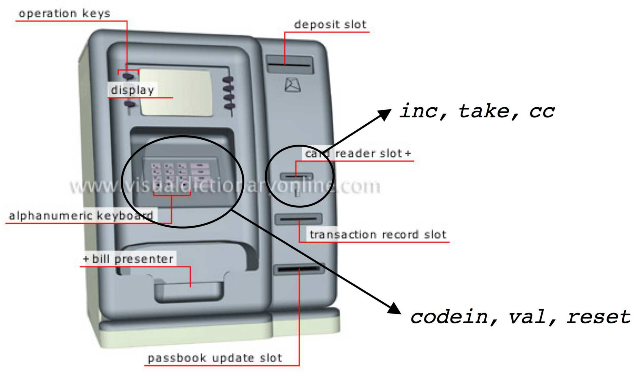

**Figure 3: Interface of the ATM**

If the customer gives the right code before a number of attempts given by the generic parameter *max_try*, the FSM reaches state *code_ok* and banking operations are allowed (*start_op* indicates that the corresponding mode is entered, *done_op* becomes true when the operations are done). Otherwise the FSM reaches state *code_error* and the card is kept.

The local signal *failed* becomes true if an internal error has been detected. In that case, the behaviour of the machine is different from its behaviour in nominal cases, in particular the access to banking operations is forbidden (it will be verified in section 5.2.2).

This system contains 3 registers:

- *n* that stores the current number of attempts,
- *ok* that contains the valid code,
- and *code* where the value of *codein* is stored, to be compared to the right code.

Hence, the component *ATM* includes 3 instances of registers, declared as follows:

```
component REG generic (default_value : natural);
   port (clock : in bit;
         in_value : in natural;  -- loaded value
         ld_flag : in bit;       -- load flag
         out_value : out natural;
         e_detect : out bit);    -- error detection
end component;
```

The output *out_value* gives the value stored in the register, and the output *e_detect* can indicate the detection of an error. We consider three different VHDL architectures for this component *REG*:

- $A_1$: a classical register, that has no fault-tolerance properties (its output *e_detect* is useless and stuck at false, and its only chance to restore a correct value is when the *ld_flag* bit is set)
- $A_2$: a register that is able to detect a single error (the register is in fact duplicated and the values are compared: *e_detect* is set to true if they are different)
- $A_3$: a TMR register that has the property of autocorrection in case of a single error (see section 4).

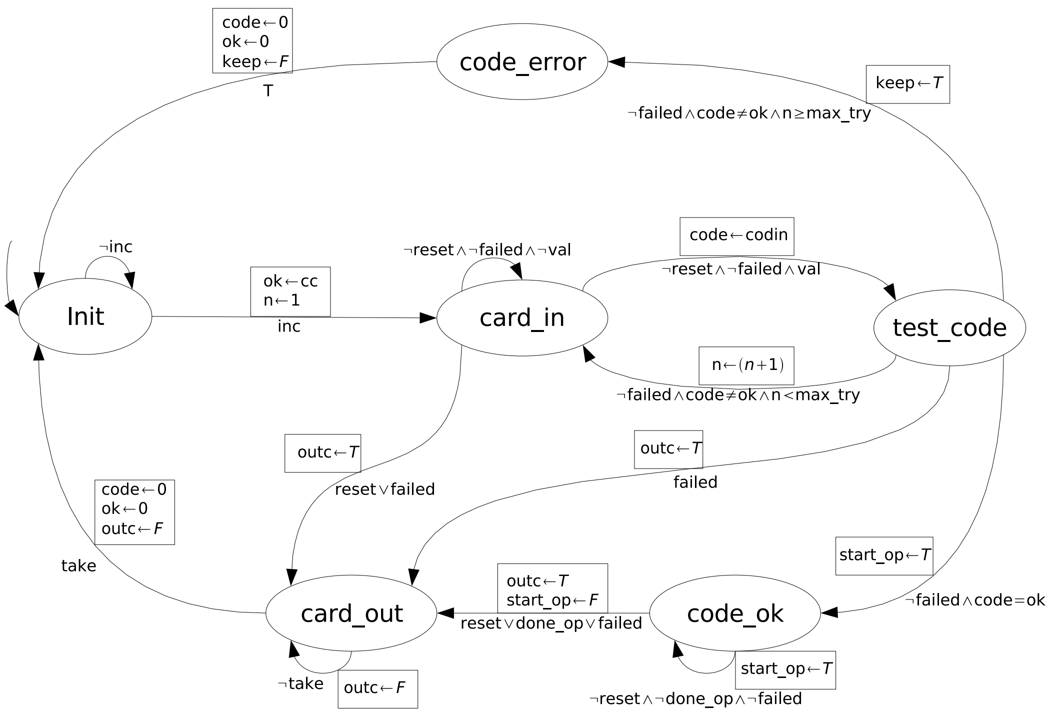

**Figure 4: FSM of the cash withdrawal system**

### *5.2.1 Register sub-component*

Let us give some more details about $A_2$ and $A_3$ ($A_1$ has an analogous ACL2 encoding but is simpler). For the register $A_2$ equipped with an error detection mechanism, we have the following functions (see the ACL2 code below):

- a transition function *REG-det-next* and output functions *REG-det-out_value* and *REG-det-e_detect*
- a state recognizer *REG-det-Sp*, and a recognizer for the reachable (error-free) states *REG-det-reach_state*
- an error-injection function *REG-det-error*

and we prove the following properties $\mathcal{P}_{REGdet}$:

- *REG-det-error*: $S \to S$ (written using *REG-det-Sp*, see theorem *REG-det-error-type*)
- $\textit{REG-det-error}(s) \neq s$
  (see theorem *REG-det-error-spec1*)
- A soft error has no effect if a new value is just being loaded in the register:
  $\textit{REG-det-reach\_state}(s) \wedge ld\_flag \Rightarrow$
  $\textit{REG-det-next}(i, \textit{REG-det-error}(s)) \Leftrightarrow \textit{REG-det-next}(i, s)$
  (see theorem *REG-det-thm-hardened-1*)
- Every single error is detected:
  $\textit{REG-det-reach\_state}(s) \Rightarrow$
  $\quad \textit{REG-det-e\_detect}(\textit{REG-det-error}(s))$
  (see theorem *REG-det-thm-hardened-2*)

These proofs make use of a characterization of the error function *REG-det-error* as explained at the beginning of section 4, which is hidden in a local *encapsulate*.

Here are the essentials of the ACL2 implementation:

```
(defspec REG-det
  (((REG-det-Sp *) => *)          ; state recognizer
   ((REG-det-next * *) => *)      ; transition function
   ((REG-det-out_value * *) => *) ; output functions
   ((REG-det-e_detect * *) => *)
   ((REG-det-reach_state *) => *) ; reachable states
   ((REG-det-error *) => *)       ; error
  )
  ...
  (local (encapsulate     ; error model
     (((REG-det-error *) => *))
       ...
     (defthm REG-det-error-1
        (equal (REG-det-Sp (REG-det-error x))
               (REG-det-Sp x)))
     (defthm REG-det-error-2
        (implies (REG-det-Sp x)
                 (not (equal (REG-det-error x) x))))
     ; the error can be located in the first or in
     ; the second register:
     (defthm REG-det-error-3
        (or (equal (REG-det-error x)
                   (REG-det-inject1 x))
            (equal (REG-det-error x)
                   (REG-det-inject2 x)))))
  )

  (defthm REG-det-error-type
     (equal (REG-det-Sp (REG-det-error S))
            (REG-det-Sp S)))

  (defthm REG-det-error-spec1
     (implies (REG-det-Sp S)
              (not (equal (REG-det-error S) S))))

  (defthm REG-det-thm-hardened-1
      (implies (and (REG-det-Sp S)
                    (REG-det-reach_state S)
                    (true-listp I) (equal (len I) 2)
                    (natp
                        (nth *REG-det/in_value* I))
                    (booleanp
                        (nth *REG-det/ld_flag* I))
                    ; loading:
                    (nth *REG-det/ld_flag* I))
               (equal (REG-det-next I
                                    (REG-det-error S))
                      (REG-det-next I S))))

  (defthm REG-det-thm-hardened-2
      (implies (and (REG-det-Sp S)
                    (REG-det-reach_state S))
               (REG-det-e_detect nil
                                 (REG-det-error S))))
  ...
)
```

As for the TMR register (architecture $A_3$), we have the following functions (see the ACL2 code below):

- a transition function *TMR-next* and output functions *TMR-out_value* and *TMR-e_detect*
- a state recognizer *TMR-Sp*, and a recognizer for the reachable (error-free) states *TMR-reach_state*
- an error-injection function *TMR-error*

and we prove the following properties $\mathcal{P}_{TMR}$:

- *TMR-error*: $S \rightarrow S$ (written using *TMR-Sp*, see theorem *TMR-error-type*)
- *TMR-error*$(s) \neq s$ (see theorem *TMR-error-spec1*)
- *TMR-reach_state*$(s) \Rightarrow$ *TMR-next*$(i,$ *TMR-error*$(s)) \Leftrightarrow$ *TMR-next*$(i, s)$ (see theorem *TMR-thm-hardened-1*)
- *TMR-reach_state*$(s) \Rightarrow$ *TMR-out_value*$($*TMR-error*$(s)) \Leftrightarrow$ *TMR-out_value*$(s)$ (see theorem *TMR-thm-hardened-2*)

```
(defspec TMR
  (((TMR-Sp *) => *)          ; state recognizer
   ((TMR-next * *) => *)      ; transition function
   ((TMR-out_value * *) => *) ; output functions
   ((TMR-e_detect * *) => *)
   ((TMR-reach_state *) => *) ; reachable states
   ((TMR-error *) => *)       ; error
  )
  ...
  (local (encapsulate     ; error model
     (((TMR-error *) => *))
       ...
     (defthm TMR-error-1
        (equal (TMR-Sp (TMR-error x)) (TMR-Sp x)))
     (defthm TMR-error-2
        (implies (TMR-Sp x)
                 (not (equal (TMR-error x) x))))
     ; the error can be located in any of the three
     ; registers:
     (defthm TMR-error-3
        (or (equal (TMR-error x) (TMR-inject1 x))
            (equal (TMR-error x) (TMR-inject2 x))
            (equal (TMR-error x) (TMR-inject3 x)))))
  )
  (defthm TMR-error-type
     (equal (TMR-Sp (TMR-error S)) (TMR-Sp S)))

  (defthm TMR-error-spec1
     (implies (TMR-Sp S)
              (not (equal (TMR-error S) S))))

  (defthm TMR-thm-hardened-1
      (implies (and (TMR-Sp S)
                    (TMR-reach_state S)
                    (true-listp I) (equal (len I) 2)
                    (natp (nth *TMR/in_value* I))
                    (booleanp
                       (nth *TMR/ld_flag* I)))
               (equal (TMR-next I (TMR-error S))
                      (TMR-next I S))))

  (defthm TMR-thm-hardened-2
      (implies (and (TMR-Sp S)
                    (TMR-reach_state S))
               (equal (TMR-out_value nil
                                     (TMR-error S))
                      (TMR-out_value nil S))))
  ...
)
```

### 5.2.2 ATM system

For the three registers $n$, $ok$ and $code$ of the ATM system, we can choose any register architecture $A_1$, $A_2$ or $A_3$ (through different VHDL configurations). Depending on the selected architectures, we get different fault-tolerance properties for *ATM*. In the following, we consider that error injection can occur in any of these registers but cannot occur in the FSM symbolic state.

Let us consider the case where we choose to *instantiate each register with architecture $A_3$ (TMR)*. At that point, we have to express in "defspec ATM" below that $n$, $ok$ and $code$ are three instances of the component characterized by the set of properties $\mathcal{P}_{TMR}$ specified in "defspec TMR". In other words, we would like to mimic a kind of quantification
$\forall\, n, ok, code \;/\; \mathcal{P}_{TMR}(n) \wedge \mathcal{P}_{TMR}(ok) \wedge \mathcal{P}_{TMR}(code)$...
However such a construct, that would be useful for translating component instantiation, does not exist in *encapsulate* or in *defspec*. Hence in "defspec ATM", we include three local "encapsulate" which arise from the three registers being instantiated with TMR components (it is worth mentioning that such repetitive lines of code can easily be obtained by Lisp macros).

For the *ATM* system, we have the following functions:

- a transition function *ATM-next* and output functions *ATM-outc*, *ATM-keep*,*ATM-start_op* and *ATM-e_detect*
- a state recognizer *ATM-Sp*, and a recognizer for the reachable (error-free) states *ATM-reach_state*
- an error-injection function *ATM-error*

and we prove the following properties $\mathcal{P}_{ATM}$:

- *ATM-error*: $S \rightarrow S$ (written using *ATM-Sp*, see theorem *ATM-error-type*)
- *ATM-error*$(s) \neq s$ (see theorem *ATM-error-spec1*)

- *ATM-reach_state*$(s) \Rightarrow$
  *ATM-next*$(i,$*ATM-error*$(s)) \Leftrightarrow$ *ATM-next*$(i, s)$
  (see theorem *ATM-thm-hardened-1*)

- *ATM-reach_state*$(s) \Rightarrow$
  *ATM-start_op*$(i,$*ATM-error*$(s)) \Leftrightarrow$ *ATM-start_op*$(i, s)$
  (see theorem *ATM-thm-hardened-2*), and similarly for the other outputs.

The proofs of these theorems $\mathcal{P}_{ATM}$ make use of the previously proven properties $\mathcal{P}_{TMR}$.

```
(defspec ATM
  (((ATM-Sp *) => *)           ; state recognizer
   ((ATM-next * *) => *)       ; transition function
   ((ATM-outc * *) => *)       ; output functions
   ((ATM-keep * *) => *)
   ((ATM-start_op * *) => *)
   ((ATM-e_detect * *) => *)
   ((ATM-reach_state *) => *) ; reachable states
   ((ATM-error *) => *))       ; error

  ...
  (local (encapsulate     ; for register N
     (((ATM-n_reg-error *) => *))
     (defun ATM-n_reg-Sp (S) (TMR-Sp S))
     (defun ATM-n_reg-next (I S) (TMR-next I S))
     (defun ATM-n_reg-out_value (I S)
         (TMR-out_value I S))
     (defun ATM-n_reg-e_detect (I S)
         (TMR-e_detect I S))
     (defun ATM-n_reg-reach_state (S)
         (TMR-reach_state S))
     (local (defun ATM-n_reg-error (S)
         (TMR-error S)))
     (definstance TMR n_register
        :functional-substitution
            ((TMR-error ATM-n_reg-error))
        :rule-classes :rewrite)))

  (local (encapsulate     ; for register OK
    (((ATM-ok_reg-error *) => *))
    (defun ATM-ok_reg-Sp (S) (TMR-Sp S))
    (defun ATM-ok_reg-next (I S) (TMR-next I S))
    (defun ATM-ok_reg-out_value (I S)
        (TMR-out_value I S))
    (defun ATM-ok_reg-e_detect (I S)
        (TMR-e_detect I S))
    (defun ATM-ok_reg-reach_state (S)
        (TMR-reach_state S))
    (local (defun ATM-ok_reg-error (S)
        (TMR-error S)))
    (definstance TMR ok_register
       :functional-substitution
           ((TMR-error ATM-ok_reg-error))
       :rule-classes :rewrite)))

   (local (encapsulate    ; for register CODE
      (((ATM-code_reg-error *) => *))
      (defun ATM-code_reg-Sp (S) (TMR-Sp S))
      (defun ATM-code_reg-next (I S) (TMR-next I S))
      (defun ATM-code_reg-out_value (I S)
          (TMR-out_value I S))
      (defun ATM-code_reg-e_detect (I S)
          (TMR-e_detect I S))
      (defun ATM-code_reg-reach_state (S)
          (TMR-reach_state S))
      (local (defun ATM-code_reg-error (S)
          (TMR-error S)))
      (definstance TMR code_register
         :functional-substitution
             ((TMR-error ATM-code_reg-error))
         :rule-classes :rewrite)))
  ...
  (local (encapsulate      ; error model
     (((ATM-error *) => *))
     (local (defun ATM-error (x) (ATM-inject1 x)))
     (defthm ATM-error-1
         (equal (ATM-Sp (ATM-error x))
              (ATM-Sp x)))
     (defthm ATM-error-2
         (implies (not (ATM-Sp x))
                  (equal (ATM-error x) ‘‘error’’)))
     (defthm ATM-error-3
         (implies (ATM-Sp x)
                  (not (equal (ATM-error x) x))))
     ; the error can be located in any of the three
     ; registers:
     (defthm ATM-error-4
        (or (equal (ATM-error x)
                   (ATM-inject1 x))
            (equal (ATM-error x)
                   (ATM-inject2 x))
            (equal (ATM-error x)
                   (ATM-inject3 x))))))

  (defthm ATM-error-type
     (equal (ATM-Sp (ATM-error S))
            (ATM-Sp S)))

  (defthm ATM-error-spec1
     (implies (ATM-Sp S)
              (not (equal (ATM-error S) S))))

  (defthm ATM-thm-hardened-1
     (implies (and (ATM-Sp S)
                   (ATM-reach_state S)
                   (true-listp I) (equal (len I) 7)
                   (booleanp (nth *ATM/reset* I))
                   (booleanp (nth *ATM/inc* I))
                   (natp (nth *ATM/cc* I))
                   (natp (nth *ATM/codin* I))
                   (booleanp (nth *ATM/val* I))
                   (booleanp (nth *ATM/done_op* I))
                   (booleanp (nth *ATM/take* I)))
              (equal (ATM-next I (ATM-error S))
                     (ATM-next I S))))

  (defthm ATM-thm-hardened-2
     (implies (and (ATM-Sp S)
                   (true-listp I) (equal (len I) 2)
                   (booleanp (nth *ATM/reset* I))
                   (booleanp (nth *ATM/done_op* I))
                   (ATM-reach_state S))
              (equal (ATM-start_op I (ATM-error S))
                     (ATM-start_op I S))))

  (defthm ATM-thm-hardened-3
     (implies (and (ATM-Sp S)
                   (ATM-reach_state S))
              (equal (ATM-keep nil (ATM-error S))
                     (ATM-keep nil S))))

  (defthm ATM-thm-hardened-4
     (implies (and (ATM-Sp S)
                   (true-listp I) (equal (len I) 2)
                   (booleanp (nth *ATM/reset* I))
                   (booleanp (nth *ATM/done_op* I))
                   (ATM-reach_state S))
             (equal (ATM-outc I (ATM-error S))
                    (ATM-outc I S))))
  ...
)
```

In the case where *the ATM registers are all instantiated us-*

| Proof | Number of local events | Total number of exported theorems | Number of error-related theorems | Total CPU time |
|---|---|---|---|---|
| Register | | | | |
| *Simple* | 7 | 16 (12 for typing) | 4 | 0.20 s |
| *With error detection* | 8 | 16 (12 for typing) | 4 | 0.24 s |
| *TMR register* | 9 | 16 (12 for typing) | 4 | 0.88 s |
| ATM | | | | |
| *With error-detecting registers* | 14 | 21 (16 for typing) | 5 | 2.34 s |
| *With TMR registers* | 14 | 21 (16 for typing) | 5 | 10.58 s |
| *With TMR registers (flat)* | | | | 2559.14 s |

**Table 1: Proofs - Size and CPU time**

*ing architecture* $A_2$, errors can be detected but not corrected and we have the following properties:

- *ATM-error*: $S \rightarrow S$
- *ATM-error*$(s) \neq s$
- Banking operations cannot start if an error occurs:
  *ATM-reach_state*$(s) \Rightarrow$
  $\neg$ *ATM-start_op*$(i$,*ATM-error*$(s))$
- Errors are detected:
  *ATM-reach_state*$(s) \Rightarrow$
  *ATM-e_detect*(*ATM-error*$(s))$

The proofs of these theorems make use of the previously proven properties $\mathcal{P}_{REGdet}$.

Table 1 summarizes CPU times for all the proofs related to this ATM example. Exported theorems include simple lemmas such as typing properties or various simplifying rules, as well as the fault-tolerance properties given above.

We see for instance that the complete certification of the book that corresponds to the ATM with TMR registers (last row) is performed in 10.58 seconds. The proof of property *ATM-thm-hardened-1*:
*ATM-reach_state*$(s) \Rightarrow$
*ATM-next*$(i$,*ATM-error*$(s)) \Leftrightarrow$ *ATM-next*$(i, s)$
takes 6.91 seconds.

It is worth noticing that, having this theorem proven, verifying that *starting from an error-free state, if a fault is injected after n clock cycles (n any positive integer) then it will be corrected one clock cycle later*:
*ATM-reach_state*$(s) \Rightarrow$
*ATM-next*$(i$,*ATM-error*(*ATM-next*$^n(I, s))) \Leftrightarrow$
*ATM-next*$(i$,*ATM-next*$^n(I, s))$
is performed in 0.12 seconds (with function *ATM-next* disabled).

Moreover the benefits of the hierarchical decomposition are substantial. For instance, while the certification of the book for the ATM with TMR registers takes 10.58 seconds with the hierarchical approach, it takes 2559.14 seconds for a flattened description (last line of Table 1).

## 6. CONCLUSION

We have presented our first results about the application of ACL2 to the verification of fault-tolerance properties in digital designs. The approach still has to be improved. For the ATM example of section 5 for instance, we get satisfying CPU times for verifying auto-correction in one clock cycle (proof of *ATM-thm-hardened-1* in 6.91 seconds) but many circuits need more than one clock cycle to restore a correct state and we should be able to consider them. To get an idea of the feasibility of such proofs with our approach, we verified auto-correction of the ATM example after several clock cycles (deliberately ignoring theorem *ATM-thm-hardened-1*) and we observed that CPU times grow very quickly (for instance, auto-correction after only 2 clock cycles is verified in 325.85 seconds).

One of the main aspects to be enhanced is that our current implementation of the property "only one memorizing element differs from $s$ to $f(s)$" in the error model is expressed by a theorem of the form

$$\bigvee_i (f(s) = inject_i(s))$$

where each $inject_i$ translates an injection in the $i^{th}$ memorizing element. We are working on providing a solution that could avoid the use of these $inject_i$ i.e., based on a theorem of the form $f(s_i) \neq s_i \Rightarrow \forall j \neq i, f(s_j) = s_j$. CPU times should be significantly improved using such a representation.

We are also currently enriching the methodology to deal with other kinds of fault models/properties. In particular, we are working on formalizing fault-injection functions on bit-vector registers, hence allowing to characterize SEU (Single Event Upset) and MEU (Multiple Event Upset) error models. We target case studies such as a FIR (finite impulse response) filter that computes

$$S_t = \sum_{k=0}^{n} I_{t-k} * C_k$$

where the $C_k$ coefficients are stored in a ROM and the successive inputs $I_{t-k}$ are put into a delay line. Considering errors in this device, especially in this large register, will be of great interest.

## Acknowledgments

The authors are grateful to the anonymous reviewers for their fruitful comments.

## 7. REFERENCES

[1] L. Anghel, R. Leveugle, and P. Vanhauwaert. Evaluation of SET and SEU effects at multiple abstraction levels. In *Proc. 11th IEEE International*

*On-Line Testing Symposium*, July 2005.

[2] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2), Feb. 2006.

[3] A. Darbari, B. Al-Hashimi, P. Harrod, and D. Bradley. A New Approach for Transient Fault Injection using Symbolic Simulation. In *Proc. IEEE International On-Line Testing Symposium*, 2008.

[4] G. Fey and R. Drechsler. A Basis for Formal Robustness Checking. In *Proc. IEEE International Symposium on Quality Electronic Design*, 2008.

[5] U. Krautz, M. Pflanz, C. Jacobi, H. Tast, K. Weber, and H. Vierhaus. Evaluating Coverage of Error Detection Logic for Soft Errors using Formal Methods. In *Proc. DATE'06*, March 2006.

[6] D. Larsson and R. Hähnle. Symbolic Fault Injection. In *Proc. 4th International Verification Workshop*, July 2007.

[7] R. Leveugle. A new approach for early dependability evaluation based on formal property checking and controlled mutation. In *Proc. 11th IEEE International On-Line Testing Symposium*, July 2005.

[8] R. Leveugle and K. Hadjiat. Multi-level fault injections in VHDL descriptions: alternative approaches and experiments. *Journal of Electronic Testing: Theory and Applications*, 19(5), Oct. 2003.

[9] F. Ouchet, D. Borrione, K. Morin-Allory, and L. Pierre. High-level symbolic simulation for automatic model extraction. Currently under submission.

[10] S. Seshia, W. Li, and S. Mitra. Verification-guided soft error resilience. In *Proc. DATE'07*, April 2007.

[11] William D. Young. Comparing verification systems: Interactive Consistency in ACL2. In *Proc. 11th Annual Conference on Computer Assurance*, pages 17–21, June 1996.