Designing Safe, Reliable Systems using Scade

Parosh Aziz Abdulla[1], Johann Deneux[1], Gunnar Stålmarck[2], Herman Ågren[2], and Ove Åkerlund[2]

[1] Uppsala University, department of Information Technology
box 337 SE-75105 Uppsala, Sweden
[2] Prover Technology AB, Rosenlundsgatan 54 SE-11863 Stockholm, Sweden

Abstract. As safety critical systems increase in size and complexity, the need for efficient tools to verify their reliability grows. In this paper we present a tool that helps engineers design safe and reliable systems. Systems are reliable if they keep operating safely when components fail. Our tool is at the core of the Scade Design Verifier integrated within Scade, a product developed by Esterel Technologies. Scade includes a graphical interface to build formal models in the synchronous data-flow language Lustre. Our tool automatically extends Lustre models by injecting faults, using libraries of typical failures. It allows to perform *Failure Mode and Effect Analysis*, which consists of verifying whether systems remain safe when selected components fail. The tool can also compute minimal combinations of failures breaking systems' safety, which is similar to *Fault Tree Analysis*. The paper includes successful verifications of examples from the aeronautics industry.

1 Introduction

Embedded controllers are found in an increasing number of systems. Their role consists of continuously processing flows of data coming from sensors to control various devices. The increase in size and complexity of these controllers has followed that of the systems they belong to. Manual verification is no longer an option, and non-exhaustive testing has its limits. They must be complemented with exhaustive methods, if possible in an automated way.

Formal methods such as *model checking* [CES86] are good candidates. They have been improving for several years within the research sector, and have recently started to reach the industry. Model checking consists of automatically verifying that a *model* representing a system meets all of its requirements. In order for the method to work, both the model and the requirements must be described formally. We present our tool, *Prover SL Data Edition* (Prover SL DE), which performs reachability analysis using SAT-based model checking [CBRZ01,SSS00]. It is integrated within several designing tools, including *Scade Suite*, a set of software tools developed by Esterel Technologies. Scade Suite includes the following tools:

– A graphical editor to build formal models and to specify safety properties. Alternatively, it is possible to translate existing models written in other languages.

– The Scade Design Verifier, built on top of Prover SL DE, to automatically verify that models satisfy all safety properties.
– A simulation environment to interactively execute models step by step.
– A C code generator. Since the code is automatically generated from the formal model, it is correct by construction, assuming the formal model is correct.

Designing safe systems is important, but it is also vital to make them reliable (fault-tolerant) i.e. they must remain safe even during failures of components. The use of formal methods to prove reliability is an attractive solution, since it increases the level of confidence in the design. There are two ways to verify that systems are reliable:

– Failure Mode and Effect Analysis (FMEA). In this approach, one tries to find the consequences of failures of components. This is usually achieved by means of simulation.
– Fault Tree Analysis (FTA). This method is the opposite: one wants to find the causes of a specific safety violation. In other words, the goal is to find combinations of components which must fail in order to make the system unsafe.

Both FMEA and FTA are described in details in [VGRH81].

We have recently extended Prover SL DE to support the two methods for reliability analysis described above. This paper describes the formal language used to build models and specify safety properties, as well as the techniques used in this tool. We also describe how to perform both safety and reliability analysis using Scade and Prover SL DE.

Related work There are a number of tools to perform safety and reliability analysis of complex systems using formal methods:

FSAP/NuSMV-SA [BV03] uses SMV's [McM93] input language as the modeling language. It supports automated BDD-based [Bry86] verification using NuSMV [C+02], fault tree generation and order analysis. Requirements are specified using Computation Tree Logic [BPM83], allowing for both safety and liveness properties.

In [BCS02], **Altarica** [GLP+98] is used to model systems. Fault trees are automatically generated and analyzed using Aralia [DR97], an efficient BDD-based fault tree analysis tool. Model checking is performed with Cadence Lab SMV. Safety requirement are specified in Linear Time Logic [Pnu77], a logic capable of expressing both safety and liveness properties.

Scade with Prover SL DE differs from these tools in the following ways:

– It is limited to model-checking of safety properties, which implies that it is not possible to check liveness properties. Since most properties used in practice are actually safety properties, this limitation is acceptable. Further, we use the same formal language both for requirements and for the model, which may make it easier for system designers and safety engineers to adopt our tool.

- The SAT-solver used in our tool supports rational and integer linear arithmetics, in addition to non-linear arithmetics over finite domains.
- The model-checking algorithm does not rely solely on BDDs. Although BDDs can deal with many large formulas efficiently, they are known to perform poorly in some cases. In order to be able to handle as many systems as possible, our tool uses a combination of SAT procedures [DLL62,SS98] as well as BDDs.

Outline This paper is organized as follows: We will first describe the modeling language used in Scade. Then we will show how SAT-based model checking [CBRZ01,SSS00] is used to automatically verify that the design satisfies all requirements. Since we are also interested in designing reliable systems, we will continue with reliability analysis, i.e FTA and FMEA.

2 The modeling language

In order to formally verify systems using model checking, one must be able to build formal models of these systems. We use Lustre [CPHJ87], a synchronous dataflow language. A *dataflow*, or *flow*, is a variable whose value can change over time. All flows are synchronized, meaning that there is a single global clock controlling when flows change. The amount of real time passing between two clock ticks is not necessarily constant. Each flow is typed: it can be Boolean, integer or real. *Nodes* combine flows to generate new flows. Several basic nodes are provided: Logic operators (AND, OR, NOT...), integer and real arithmetic operators (addition, multiplication, division...). The third type consists of timed operators:

- Delays: The `PRE` operator makes it possible to refer to the previous value of a flow. It can, for example, be used to memorize values: `A = PRE A`. The *current* value of `A` is defined to be the *previous* of A.
- Initial value: The `->` operator is used to specify the value of a flow during the first time step. Consider the following example: `A = True ->NOT PRE A`. This defines flow `A` to be initially True. After that, the value is inverted every time step, thus modeling a square clock signal.

A system is modeled as a node, possibly composed of several sub-nodes. Recursion is not allowed, meaning that a node may not include itself as one of its sub-nodes, or in one of its respective sub-nodes. Therefore, it is possible to "flatten" the top node by substituting their contents to sub-nodes.

Scade provides a graphical interface to create, edit and visualize nodes. There are two ways to visualize nodes: The *network view* (Figure 1) and the *state machine view*. A textual equivalent representation of Figure 1 in Lustre can be seen in figure 2. This fictitious example is a controller for the doors of a lift. Requests to open the door are received from other parts of the system. These requests are granted provided that the lift is not in motion, and that it is at

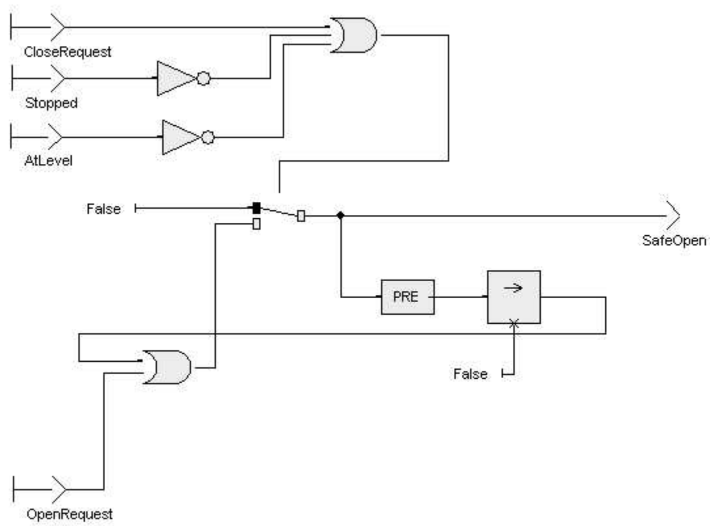

Fig. 1. Graphical representation of a lift door controller

```
node LiftDoor(OpenRequest: bool; CloseRequest: bool;
              Stopped: bool; AtLevel: bool)
returns (SafeOpen: bool) ;
let
  SafeOpen = if (CloseRequest or  not (Stopped) or  not (AtLevel))
    then False
    else (False -> (pre SafeOpen)) or OpenRequest ;
tel ;
```

Fig. 2. Textual representation of the lift door controller

level with the floor. If the open request is granted, the door is kept open until the safety conditions are violated, or a close request is received.

Lustre is also used for expressing safety requirements. The system being in a safe state is denoted by a specific Boolean flow in the model being true. The model checker verifies whether this flow is always true. In other words, it performs safety analysis by proving that the system constantly remains in a safe state. Popular alternatives for specifying requirements are time logics such as Linear Time Logic or Computation Tree Logic. See for example [Hol97] or [McM93]. Our decision to use Lustre has the advantage that users need not learn an additional requirement language. Although this implies that we are limited to verifying safety properties, we consider this to be an acceptable restriction since they constitute the majority of properties used in safety and reliability analysis.

Back to our lift door example, two requirements could be:

```
OpensWhenSafe = (OpenRequest and AtLevel and Stopped) -> SafeOpen;

ClosesWhenUnsafe = (!AtLevel or !Stopped) -> !SafeOpen;
```

The first requirement ensures that users do not get trapped inside the lift, that the door opens when requested if it can be done safely. The second requirement makes sure that the lift cannot harm its passengers by opening while in motion, or when not at level with the floor.

Lustre supports *assertions* to restrict the possible values of input flows. Similarly to requirements, assertions are represented by Boolean expressions which must always be True. They differ from requirements in the sense that they express assumptions about the environment of the system, which is not part of the model. In the lift door example, we may assume that the environment will never request to open and close the door simultaneously:

```
assert not (OpenRequest and CloseRequest);
```

When generating C code from a Lustre model, assertions can be translated into C assert macro calls. Assertions are also used by Scade Design Verifier to speed up the verification. Instead of verifying the model for all possible combinations of inputs, the verification is limited to those inputs satisfying the assertions.

In the next section, we describe some of the techniques used in Prover SL DE, upon which Scade Design Verifier is built.

3 Verifying safety

Prover SL DE verifies safety properties of transition systems. We will first define the terms *transition systems* and *safety properties*, then explain how our tool performs this kind of verification.

3.1 Transition systems

A *transition system* is a tuple (S, S_0, T), where

- S is a set of states,
- $S_0 \subseteq S$ is the set of *initial states*
- $T \subseteq S \times S$ is the transition relation.

A *safety property* P is a set of states denoting the good states.

Let $Reach_T(S)$ be the set of states reachable from S using the transition relation T.

We want to decide if a transition system is safe, i.e. given a transition system $M = (S, S_0, T)$ and a safety property P, is it the case that $Reach_T(S_0) \subseteq P$?

Lustre models are transition systems. The state of a Lustre model is denoted by the current values of all its flows. The set of initial states is specified in the model using initial value (->) operators. The transition relation is specified using delay operators (PRE). The set of states is the set of all assignments to flows in the model. This set is potentially infinite, because of the use of unbounded types (integers and reals). Although Lustre can express complex arithmetic expressions, Prover SL DE is limited to:

- Linear arithmetics over the set $\mathbb{Q}$ of rational numbers, i.e. expressions of the form:
$$a_0 * C_0 + \ldots + a_n * C_n \diamond C$$
where $a_0, \ldots, a_n$ are variables, $C, C_0, \ldots, C_n$ are constants and $\diamond \in \{=, \neq, >, <, \leq, \geq\}$.
- Non-linear arithmetics over finite domains.

3.2 SAT-based model checking

Building an explicit representation of the reachable states space is in general not feasible (in our case it is even impossible). Instead, we represent sets symbolically using predicates. Checking for the non-reachability of a set of bad states can now be done by checking non-satisfiability of Boolean and linear arithmetic formulas. This technique is *SAT-based model checking* extended to arithmetics.

For a set of states S, let $S(s)$ be a predicate such that $s \in S \iff S(s)$.

For a sequence of states $s_0 \ldots s_n$, $path(s_0 \ldots s_n)$ is a predicate denoting that the sequence corresponds to a path through the graph of the transition relation.

$$path(s_0 \ldots s_n) = \forall i \in \{0, \ldots, n-1\} : T(s_i, s_{i+1})$$

The reachability problem for a transition system (S, S_0, T) and a safety property P can be reformulated as follows:

$$\forall n \geq 0 : \forall s_0 \ldots s_n : path(s_0 \ldots s_n) \wedge S_0(s_0) \Rightarrow P(s_n)$$

Two methods to solve this problem are *Bounded Model Checking* [CBRZ01] and *Induction Over Time* [SSS00].

The first method is suitable for debugging, i.e. finding errors in unsafe systems.

$$bmc_n(s_0 \ldots s_n) = path(s_0 \ldots s_n) \wedge S_0(s_0) \Rightarrow P(s_n)$$

It proceeds iteratively by increasing n until $bmc_n(s_0 \ldots s_n)$ is falsifiable, in which case we have found a shortest path to a bad state. However, this method will never terminate for safe systems.

In the second method, we try to prove by induction over k that the system is safe:

- **Base case:** $bmc_n(s_0 \ldots s_n)$ is a tautology.
- **Induction hypothesis:** $ih_n(s_k \ldots s_{k+n}) = path(s_k \ldots s_{k+n}) \wedge \forall 0 \leq i \leq n : P(s_{k+i})$
- **Induction step:** $is_n(s_{k+n}) = \forall s_{k+n+1} : T(s_{k+n}, s_{k+n+1}) \Rightarrow P(s_{k+n+1})$

Concretely, we increase n, starting from 0, until:

$$(\forall s_0 \ldots s_n : bmc_n(s_0 \ldots s_n)) \wedge (\forall s_k \ldots s_{k+n} : ih_n(s_k \ldots s_{k+n}) \Rightarrow is_n(s_{k+n}))$$

If we succeed, we have proved that the system is safe. If the system is not safe, then the Bounded Model Checking step of the base case will detect it. This procedure is still incomplete: Consider the case where an unreachable loop leads to a bad state, shown in figure 3. The induction step will never succeed,

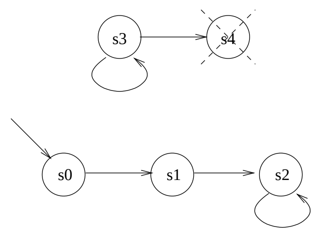

Fig. 3. A good unreachable loop leading to a bad state. The safety property includes s0, s1, s2 and s3, but not s4. s0 is the initial state

even though the system is correct. This is solved [SSS00] by modifying the *path* predicate to loop-free paths:

$$path(s_0 \ldots s_n) = \forall i \in \{0, \ldots, n-1\} : T(s_i, s_{i+1}) \wedge \forall j \in \{0, \ldots, n\} : i \neq j \Rightarrow s_i \neq s_j$$

3.3 Deciding the satisfiability of formulas

We have now described how to transform the problem of deciding whether a transition system is safe into deciding whether a formula is satisfiable or not. The kind of formula we have to deal with are *math-formulas* [ABC$^+$02]. A math-formula combines Boolean propositions and linear arithmetic predicates:

- A constant c in $\mathbb{Q}$ is a *math-term*.
- A variable v over $\mathbb{Q}$ is also a math-term.
- $c.v$ is a math-term.
- If t_1 and t_2 are math-terms, then so are $t_1 + t_2$ and $t_1 - t_2$.
- A Boolean proposition is a math-formula.
- If t_1 and t_2 are math-terms, then $t_1 \diamond t_2$ where $\diamond \in \{=, \neq, >, <, \leq, \geq\}$, is a math-formula.
- If ϕ_1 and ϕ_2 are math-formulas, then $\neg\phi_1$ and $\phi_1 \Box \phi_2$ where $\Box$ is a logical connective are math-formulas.

A naive procedure for deciding the satisfiability of a math-formula, which is a NP-hard problem [ABC$^+$02], is to examine all satisfying assignments to the boolean variables in the formula, and for each of these solve the resulting system of linear constraints. The proof engine implements an efficient solver [ABCH02] for MATH-SAT which combines SAT techniques, such as Stålmarck's saturation method [SS98], Davis-Putnam-Loveland-Logemann [DLL62], Reduced Ordered Binary Decision Diagrams [Bry86], linear programming techniques and constraint propagation. In practice the proof engine can decide a strict superset of math-formulas, mainly due to the constraint propagation. Even if a given satisfiable formula contains non-linear predicates, the proof engine often manages to decide it. In the case of integers restricted to finite ranges, Prover SL DE converts them to bit vectors, and uses binary arithmetics to perform all operations. This method is able to handle non-linear arithmetic over finite ranges.

4 Reliability analysis

In this section we explain shortly FTA and FMEA, and describe how to use Scade to design reliable systems.

4.1 FTA and FMEA

A failure is the inability of a piece of equipment to perform its task. Here we make a distinction between system-level and component-level failures. We restrict the use of the term "failure" to component-level failures. When the system itself fails to meet its expected performance, we say it is "unsafe", or that it violates a "safety requirement". A system is reliable when it can sustain several failures before becoming unsafe. More precisely, it is N-fault-tolerant if it remains safe unless more than N failures happen. Two popular methods to assess reliability are Failure Mode and Effect Analysis (FMEA) and Fault Tree Analysis (FTA) [VGRH81]. The term "failure mode" refers to the way a component fails. For instance, a valve may fail in different ways: It can be stuck in the opened position, in the closed, or in some intermediate position. Each way of failing is called a failure mode.

The first method, FMEA, consists of investigating the effects of failure modes. Designers specify a list of components that fail in addition to the way they fail, then the system is simulated to check if it becomes unsafe. The second method, FTA, can be seen as the opposite approach. It aims at finding the causes of safety violations. A fault tree (Figure 4) is a graph relating failures of components and safety violations. The root of the tree is called *Top Level Event*, and represents an event that should not occur in a safe system. In this example, the top event consists of the opening of the doors of a lift while it is moving or when it is not at the level of the floor. The leaves are called *basic events*. They represent failures of components as well as their failure mode. Here, the left basic event represents the event that the motion detector fails to report movement. The right event denotes the failure of the sensor to detect that the lift is not at the level of the floor. The internal nodes are Boolean connectives. The connective represented in this example is an OR gate. The fault tree is in fact a graphical representation of a Boolean formula satisfied when the system is unsafe. The variables in the

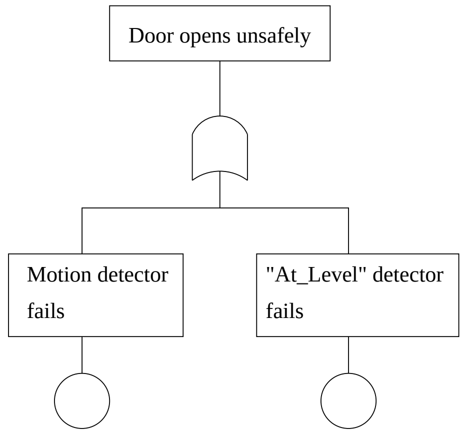

Fig. 4. A simple fault tree

formula denote failures of components. The goal of FTA is to find the minimal combinations of basic events leading to the top event. In other words, one wants to compute the minimal cut sets or prime implicants of the Boolean expression represented by the fault tree.

We have extended Prover SL DE to support these two methods. We will now describe how we perform reliability analysis using Scade and Prover SL DE.

4.2 Fault injection

In order to assess the reliability of a system, its model must include failure modes. The process of adding failure modes into an existing model is called *fault injection*. We have implemented a graphical user interface (Figure 5) allowing designers to select the components susceptible to fail as well as their failure mode. This results in a new model including failure modes, which is then analyzed using the methods described in section 3. Failures of components are modeled by modifying flows representing components outputs. The original flows, called *nominal* flows, are replaced by modified flows, called *extended* flows. The value of an extended flow is decided by the failure mode. All possible failure modes affecting the nominal flow are modeled by a Lustre node called *failure mode node*. A typical failure mode node has two or more inputs and one output. One of the input is the nominal flow, and the output is the extended flow. The remaining inputs are Boolean flows called *failure mode variables* controlling which failure mode is triggered. Figure 6 represents two failure modes affecting a Boolean flow: The value of the nominal flow (`in`) is ignored and the extended flow (`out`) is set to False or True. This failure mode node can be used to model two failure modes of a switch, for instance:

- `FM_OFF`, in which case the switch acts as if it was stuck in the OFF position, or

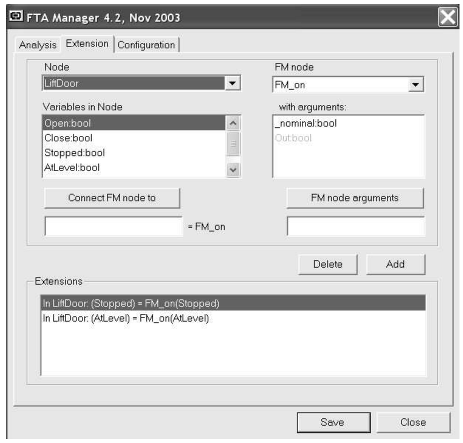

Fig. 5. The fault injection panel

– FM_ON, the switch behaves as if it was stuck in the ON position, possibly because of a short-circuit.

```
node FM_Fails_ON_or_OFF(in: bool; FM_ON: bool; FM_OFF: bool) returns
  (out: bool)
let
   out = if (FM_ON) then True else
         if (FM_OFF) then False else
         in;
   assert not (FM_ON and FM_OFF);
tel;
```

Fig. 6. A failure mode node

The result of the fault injection into the lift door model (Figure 2) is shown in Figure 7. The FTA_ prefix marks extended flows added during fault injection. FM_Fails_ON is a failure mode node where a signal remains constantly True. Flows with names starting with FM_ trigger failure modes.

4.3 FMEA in Scade

Using the same graphical interface shown in Figure 5, designers constrain the occurrence of failures. Typical kinds of constraints include:

– At most N failure modes can occur. This is equivalent to “At most N failure mode variables can switch from False to True”.

```
node FTA_LiftDoor(OpenRequest : bool ; CloseRequest : bool ;
                  Stopped : bool ; AtLevel : bool,
                  FM_Stopped_Fails_ON : bool;
                  FM_AtLevel_Fails_ON)
returns (SafeOpen : bool) ;
var
  FTA_Stopped: bool;
  FTA_AtLevel: bool;
let
  FTA_Stopped = FM_Fails_ON(Stopped, FM_Stopped_Fails_ON);
  FTA_AtLevel = FM_Fails_ON(AtLevel, FM_AtLevel_Fails_ON);

  SafeOpen = if (CloseRequest or  not (FTA_Stopped) or  not (FTA_AtLevel))
    then False
    else (False -> (pre SafeOpen)) or OpenRequest ;
tel ;
```

Fig. 7. The model of a lift door after fault injection

- At most N failure modes can happen simultaneously, which is the same as "At most N failure mode variables can be True at any point in time"
- Once a component fails, it never recovers and continues to fail indefinitely
- A failure mode X cannot happen.
- When failure mode X is triggered, it continues to happen for T time steps.

These constraints are specified in Lustre, in a manner similar to requirements. A constraint node has a single Boolean output flow, and any number of input flows of any type. These input flows can take any value, as long as the constraint node's output remains true. Scade Design Verifier verifies that the safety requirement is always respected, assuming all constraints are met. If this is not the case, a sequence causing the system to become unsafe is returned.

4.4 FTA in Scade

The goal of FTA is to compute the minimal combinations of failures (also called minimal cut set) causing a safety violation. Our tool proceeds by checking whether the system is safe assuming that N failure modes occur, starting with $N = 0$, and then increasing N. At each step, Scade Design Verifier verifies if the system is safe. If it is not, the Design Verifier generated a counter-example containing the values of each flow at each time step until the safety requirement was violated. From this counter-example, the set of flows representing failure modes that were triggered is extracted. These flows constitute a cut set. The operation is repeated until all cut sets smaller than a user-fixed limit have been found.

The first step, when $N = 0$, amounts to verifying that the system is safe. If it is not, then it is obviously not reliable. Otherwise N is increased to 1 and the system's safety is checked again, assuming one failure mode occurs. If the system is not safe, a counter example is generated. Since the verification was restricted to the case where one failure mode occurs, one of the failure mode variables in the counter-example must be True at some point in time. This failure mode variable represents one of the minimal cut sets of size 1. The tool continues by doing another analysis with N unchanged until no more cut sets of size 1 can be found. N is then increased, and the same steps are taken until N reaches a user-fixed limit, usually 4 or 5. The process is summarized below:

ComputeMCS(M: system model, *req*: safety requirement, N_{max}: integer):
Let N be an integer
Let S be a set of cut sets
$N := 0$
$S := \{\}$
Repeat
    Let C_1 be the constraint:
        at most N failure mode variables become True
    Let C_2 be the constraint:
        no combination of failure modes found in S is triggered
    Let c_x be a counter-example
    $c_x :=$ *Verify*$(C_1 \wedge C_2, M, req)$
    If c_x is not empty (i.e. the system is not safe)
        Extract a cut set s from the counter-example c_x
        $S := S \cup \{s\}$
    Else (i.e. the system is safe) $N := N + 1$
Until $N = N_{max}$

Verify$(C_1 \wedge C_2, M, req)$ is a call to the model checker. The verification is constrained to those executions satisfying C_1 and C_2. If the system is not safe, a counter-example is returned and stored in c_x.

5 Applications

In order to evaluate the tool, our industrial partners provided a number of examples. We describe three of them in this section: *air inlet control*, *nose wheel steering* and *hydraulic system*. All models are designed and analyzed on widely available laptops equipped with Intel Pentium3 processors with 512MB of RAM.

5.1 Air inlet control

This system is a controller to automatically manoeuvre opening and closing of doors of an aircraft to regulate the inflow of air to an auxiliary power unit. Since faulty cooling of the auxiliary power unit is a hazardous event the automated manoeuvring is safety critical.

This model consists of a state transition diagram, regulating the doors movement. The system contains 21 Boolean inputs, 12 Boolean outputs and 2 rational inputs. 20 flows among the inputs are affected by fault injection, resulting in 40 new Boolean inputs. Arithmetic expressions found in this model are limited to simple comparisons.

In this case many variables represent input coming from sensors telling if doors are closed or open, or information about motor status.

One safety requirement concerns the movement of doors when landing. Landing is detected by a sensor recognizing if there is any weight on the wheels. The corresponding input flow in the model is named "weight_on_wheel". When this variable changes from False to True, i.e. a landing event was detected, the airflow doors must be open.

The verification, taking less than a minute, concludes that the system is safe, i.e. the requirement is respected when no components fail. It is however not reliable, since 5 different single failures and 3 double failures can make the system unsafe.

5.2 Nose wheel steering

This example is a control system to ensure suitable manoeuvrability for different aircraft operations whilst on the ground. It was originally designed in Mathworks, Matlab/Simulink, then automatically translated using tools from the Scade suite.

The Scade model includes 36 inputs (33 Boolean and 3 rational). The requirements concerns the validity of the value of the steering angle, computed by the controller. It must remain within predefined bounds. All 33 Boolean inputs are affected by failures, thus doubling the number of variables in the system after fault injection.

This requirement is fulfilled when no failures are allowed, i.e. the design is safe. It is not reliable, since 32 minimal cut sets of size 1 were found. The analysis took about 10 minutes.

5.3 Hydraulic system

This system controls the hydraulic power supply to devices ensuring aircraft control in flight, landing gear, braking system, etc. Three independent hydraulic subsystems are shared between consuming devices in order to achieve fault tolerance. The hazardous event we want to investigate in this case is the total loss of hydraulic power.

This system was originally modeled in Altarica, whose semantic is close to Lustre's, making it easily translatable to a Scade model. Since fault injection was performed on the original Altarica model, it was not performed again on the Scade model. Unlike the other examples presented in this section, the original model already takes into account failures of components.

This analysis, which took about 3 minutes, found no single or double cut sets, 11 cut sets of size 3 and 24 cut sets of size 4.

6 Conclusion

In this paper we have presented a methodology to perform FMEA and FTA using Scade Suite and Scade Design Verifier from Esterel Technologies. Scade Design Verifier is based on the proof engine Prover SL Data Edition from Prover Technology, which has also been presented.

Future work. Our users remarked that sequences showing violations of requirements are too complex. They contain too many variables, and it is hard to find which ones are "interesting", i.e. which variables have a key role in the unreliability of a system. This problem and several solutions are discussed in [RS04]. Our implementation of Fault Tree Analysis, which repeatedly calls the model checker, is currently quite naive. We plan to optimize the model checker for this kind of usage, thus possibly reducing the number of calls and hopefully speeding up each verification. Finally, we will also extend the tool to support order analysis [BV03].

References

[ABC+02] Gilles Audemard, Piergiorgio Bertoli, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. A sat based approach for solving formulas over boolean and linear mathematical propositions. In *Proceedings of the 18th International Conference on Automated Deduction*, pages 195–210. Springer-Verlag, 2002.

[ABCH02] Gunnar Andersson, Per Bjesse, Byron Cook, and Ziyad Hanna. A proof engine approach to solving combinational design automation problems. In *Proceedings of the 39th conference on Design automation*, pages 725–730. ACM Press, 2002.

[BCS02] Pierre Bieber, Charles Castel, and Christel Seguin. Combination of fault tree analysis and model-checking for safety assessment of complex system. In *Proceedings of the fourth European Dependable Computing Conference (EDCC-4), Toulouse*. Springer Verlag, October 2002.

[BPM83] M. Ben-Ari, A. Pnueli, and Z. Manna. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8):677–691, Aug. 1986.

[BV03] Marco Bozzano and Adolfo Villafiorita. Improving system reliability via model checking: The fsap/nusmv-sa safety analysis platform. In *Proceedings of the 22nd International Conference on Computer Safety, Reliability and Security [SAFECOMP 2003]*, September 2003.

[C+02] Alessandro Cimatti et al. NuSMV2: an opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer-Verlag, July 27–31 2002.

[CBRZ01] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specification. *ACM Trans. on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CPHJ87] P. Caspi, D. Pilaud, N. Halbwachs, and J.Plaice. Lustre: a declarative language for programming synchronous systems. In *14th ACM Symposium on Principles of Programming Languages, Munchen*, January 1987.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[DR97] Yves Dutuit and Antoine Rauzy. Exact and truncated computations of prime implicants of coherent and non-coherent fault trees within aralia. *Reliability Engineering and System Safety*, 1997.

[GLP+98] Alain Griffault, Sylvain Lajeunesse, Gérald Point, Antoine Rauzy, Jean Pierre Signoret, and Philippe Thomas. The altarica language. In *Proceedings of the International Conference on Safety and Reliability, ESREL'98*. Balkema Publishers, June 20-24 1998.

[Hol97] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, SE-23(5):279–295, May 1997.

[McM93] K.L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc.* 18^{th} *Annual Symp. Foundations of Computer Science*, pages 46–57. IEEE, 31 October–2 November 1977.

[RS04] K. Ravi and F. Somenzi. Minimal assignments for bounded model checking. In *Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference*. Springer-Verlag Heidelberg, April 2004.

[SS98] Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. In G. Gopalakrishnan and P. Windley, editors, *Proceedings 2nd Intl. Conf. on Formal Methods in Computer-Aided Design, FMCAD'98, Palo Alto, CA, USA, 4–6 Nov 1998*, volume 1522, pages 82–99, Berlin, 1998. Springer-Verlag.

[SSS00] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Int. Conf. on Formal Methods in Computer-Aided Design*, volume 1954, 2000.

[VGRH81] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl. *Fault Tree Handbook*. U. S. Nuclear Regulatory Commission, NUREG-0492, Washington DC, 1981.