Paths to property violation: a structural approach for analyzing counter-examples

Thomas Bochot[1,2], Pierre Virelizier[1], Hélène Waeselynck[3,4], Virginie Wiels[2]

[1]Airbus Operations SAS, 316 route de Bayonne, 31060 Toulouse, France
[2]ONERA/DTIM, 2 Av. E. Belin, BP74025, 31055 Toulouse, France
[3]CNRS ; LAAS ; 7 Av. du Colonel Roche, 31077 Toulouse, France
[4]Université de Toulouse ; UPS , INSA , INP, ISAE ; LAAS ; 31077 Toulouse, France

thomas.bochot@gmail.com, Helene.Waeselynck@laas.fr, virginie.wiels@onera.fr

***Abstract*— At Airbus, flight control software is developed using SCADE formal models, from which 90% of the code can be generated. Having a formal design leaves open the possibility of introducing model checking techniques. But, from our analysis of cases extracted from real software, a key issue concerns the exploitation of counterexamples showing property violation. Understanding the causes of the violation is not trivial, and the (unique) counterexample returned by a model checker is not necessarily realistic from an operational viewpoint. To address this issue, we propose an automated structural analysis that identifies paths of the model that are activated by a counterexample over time. This analysis allows us to extract relevant information to explain the observed violation. It may also serve to guide the model checker toward the search for different counterexamples, exhibiting new path activation patterns.**

Keywords: Analysis of counterexamples; SCADE models; structural paths.

I. INTRODUCTION

The Flight Control software is one of the most critical components inside an aircraft. At Airbus, it is developed using SCADE formal models [1], from which 90% of the code can be generated. A number of formal methods are then applied at the code level [2]: Floyd-Hoare methods to verify functional properties on the manually coded parts, abstract interpretation methods to verify non-functional properties of the entire code (such as absence of run-time errors). The SCADE models, however, are not validated using formal methods. Rather, intensive model tests are performed using desktop simulators.

Airbus has conducted a number of R&D studies to investigate the introduction of model-checking techniques at the SCADE level. Our synthesis of these studies can be found in [3]. We concluded that, in the current Airbus process, the most efficient use is to consider a lightweight approach, where SCADE designers use model-checking as a debugging facility. Analysis is focused on few critical functions, and may consider only a subset of their possible behaviors. First experiments in [3] suggest that this approach may indeed be quite effective, even with respect to subtle flaws that are typically found by lab tests (i.e., tests of real equipments in the loop with advanced flight simulation facilities).

A problem is however the time required by the analysis of the counterexamples found during model debugging. Fig. 1 shows a typical debugging process. Each time a counterexample is found, human analysis has to determine the reasons of the observed property violation. The analysis consists in replaying the counterexample, and identifying the activated parts of the model. Once the details of the execution are well understood, the user tries to determine whether the counter-example may correspond to a realistic aircraft scenario. If it may not, there is a need to investigate alternative formulations of the property and the hypotheses put on the model environment. If it may, the SCADE design has to be revised. A number of iterations are performed this way. At each iteration, either the model, or the property and hypotheses, have to be modified before any new insight is got. The model-checker does not allow for several counter-examples, which slows down the debugging cycles.

Our work aims at alleviating human effort in this process. We propose an automated structural analysis that identifies paths of the model that are activated by a counterexample over time. This analysis allows us to extract relevant information to explain the observed violation. It may also serve to guide the model checker toward the search for different counterexamples, exhibiting new path activation patterns. The analysis has been implemented in a tool, STANCE (Structural Analysis of Counter-Examples), and applied to a case study representative of Airbus models.

Section II presents related work. Section III is the core of the paper. It lays the formal foundations for the structural analysis. Formalization choices are discussed in Section IV. Section V gives a quick overview of STANCE and its application to an example.

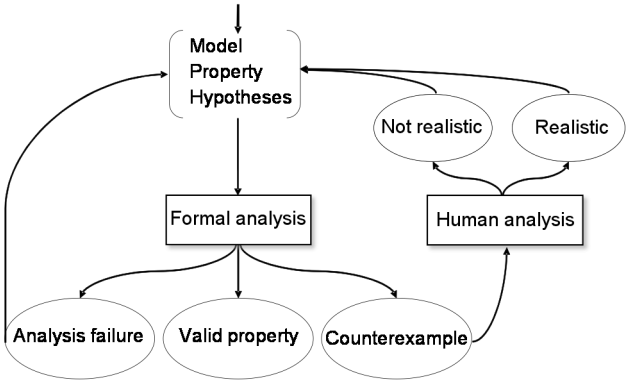


Figure 1. Typical debugging process

II. Related Work

Related work addresses the understanding of counter-examples. A first category of work concerns interactive visualization of counterexamples [4][5][6]. In particular, [6] proposes several connected views. A second category concerns automated analysis of counterexamples. [7] uses domain knowledge (railway domain) to interpret counter-examples in terms that are meaningful to engineers (e.g., in terms of train routes). [8] and [9] consider neighboring correct and incorrect executions to localize the error in the code (for software model checking). [10] and [11] combine visualization and analysis. [10] uses the structure of the property to display information on when the property fails, and with which subconditions. [11] uses the structure of both the model and the property, and annotates counter-examples with proofs explaining the model checking result.

Comparing our work to others, the closest approaches are [8] [9]. Like them, we analyze the execution of counter-examples in terms of structural locations. However, our approach differs in two ways. Firstly, our analysis is performed entirely outside of the model checker. This has the advantage of not requiring any visibility on the tool. Indeed, the proprietary SCADE model-checker is a black-box for its users. The drawback is that we cannot take advantage of the information collected during the building of counter-examples (e.g., sets of correct transitions in [9]). Our analysis has to be kept light, so that it does not significantly add to model checking time. Secondly, our work is dedicated to data flow models while [8] and [9] deal with imperative programs. This induces significant differences in how an execution connects to the structure. In particular, in dataflow models, all operators are executed at each time instant.

Outside the model checking community, structural analysis of data-flow models is performed in testing work. Both hardware and software testing are worth mentioning. A SCADE model is similar to a gate-level model of circuit; as regards software testing, work around the Lustre language (on which SCADE is based) is also quite relevant to us.

In the hardware domain, Automatic Test Pattern Generation (ATPG) algorithms [12] aim at generating test cases in order to reveal specific faults (for example stuck-at faults). Typically, the algorithms compute values for the inputs that imply a given value on a circuit line. This is done by exploration of the structure of the circuit. A difference is that they do not start from a completely instantiated counter-example as we do, but some results on algorithmic complexity can be transferred to our work.

As regards software, Ludic [13] is a debugger for Lustre models. It proposes functionalities for the interactive replay of test cases, with the user guiding exploration of the model structure. We preferred to adopt a non-interactive approach, that does not require any *a priori* knowledge of what the counter-example means, and which internal execution flows are worth being explored. In our case, key information is automatically synthesized from the execution of the counter-example. Finally, [14] defines structural coverage criteria for Lustre models. This led the authors to propose a clean definition of path activation conditions in data-flow programs, which we re-used in our work.

III. Formalization of the Problem

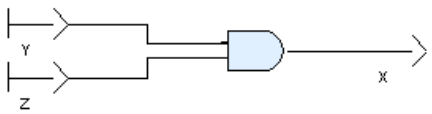
A. Preliminaries

The SCADE language is a formal graphical notation based on Lustre [15]. It is thus (like Lustre) a synchronous data-flow language. Synchronous languages rely on the synchronous hypothesis. In practice, this hypothesis means that the system is able to react to an external event, before any further event occurs.

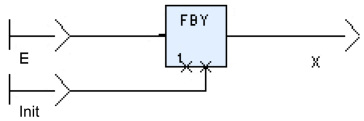
In the SCADE language, any expression X represents a flow. A flow is a sequence $(x_1, x_2, ...)$ of values. A system is intended to have a cyclic behavior, and x_n is the value of X at the n^{th} cycle of the execution. A system is described by a SCADE model with inputs, outputs and local variables. The model defines a set of equations to compute the output flows from the input ones. An equation can be seen as a temporal invariant, e.g. $X = AND\ (Y,Z)$ defines the flow $(x_1, x_2, ...) = (y_1 \wedge z_1, y_2 \wedge z_2, ...)$. Fig. 2.a shows how the previous equation is graphically represented in the SCADE language.

Equations use arithmetic, Boolean and if-then-else operators, as well as a specific temporal operator denoted *followed-by* (FBY). The *followed-by* operator introduces a delay of t cycles between its input and its output; an additional initialization input is used to determine the t first values of the output flow. If E and $Init$ are two expressions of the same type respectively denoting the sequences $(e_1, e_2, e_3,...)$ and $(init_1, init_2, init_3, ...)$, and if a delay $t = 1$ on E is wanted, then $X = FBY(E, 1, Init)$ is an expression denoting the sequence $(init_1, e_1, e_2, ...)$. The graphical notation for this operator is given in Fig. 2.b.

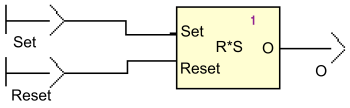
In addition to basic SCADE operators, a library of operators dedicated to the design of flight control software was developed by Airbus. These domain-specific operators are called *symbols*. An example is given in Fig. 2.c. The *R*S* symbol represents a reset-biased latch. It has two inputs *Set*



a. Logical *AND* operator



b. *Followed-by* operator



c. *R*S* symbol

Figure 2. Some SCADE operators and Airbus symbols.

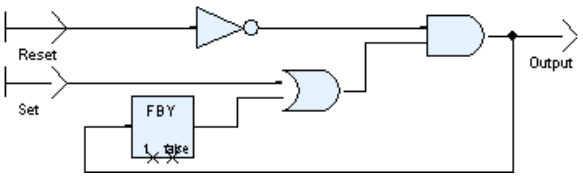


Figure 3. SCADE model of the *R*S* symbol.

and *Reset* and one output *Output*. The latch is initialized to false. *Output* becomes true each time *Set* is true and *Reset* is false, and remains true until *Reset* is true. Each symbol may be described in terms of basic SCADE operators. Fig. 3 shows the SCADE reference model of the *R*S* symbol.

In SCADE, model checking uses the notion of synchronous observers [16]. Informally speaking, an observer monitors the behavior of the system and decides whether it is correct or not. Technically, an observer is a synchronous program that is connected to the target model under verification (MUV), without feedback. Fig. 4 describes a typical verification architecture. It includes two categories of observers: one for the property to be verified on MUV (yielding result *O*), and one for hypotheses about the system environment (*H*). The observers, and their connection to MUV, are specified in SCADE. The resulting global model is then supplied to the model-checker that attempts to falsify *O* while keeping *H* true. A counterexample is returned in case of success. This counterexample is a sequence of input values during *N* cycles that, when applied to the global model, yields a false value of *O* at cycle *N*. It thus demonstrates that MUV violates the property.

Understanding why the counterexample yields a violation is a difficult task. Our objective is to automatically extract the relevant information explaining the false output of the property observer. Not all input values played a role in falsifying *O*, so we would like to point out the relevant data of the counterexample. Moreover, we would like to aid the user to diagnose the problem, whether it comes from an incorrect internal design of MUV or from an incorrect formalization of the property. The proposed solution involves a structural analysis of the global model, that identifies relevant propagation paths connecting the values of *some* input variables at *some* cycles to the false output o_N.

Subsequent subsections B to G gradually formalize the notion of *cause of a counterexample* in terms of such propagation paths. For simplification purposes, the current

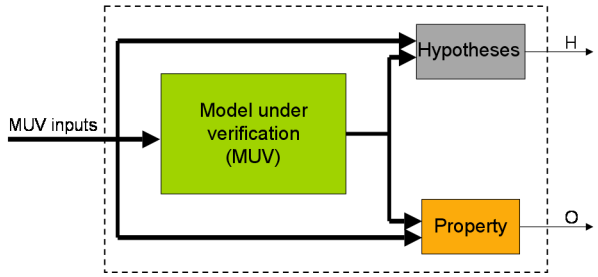


Figure 4. Global model including synchronous observers.

formalization focuses on models with Boolean and temporal operators, for which model checking is well adapted. Our treatment of numerical operators is delayed to future work.

B. *SCADE Models as Graphs*

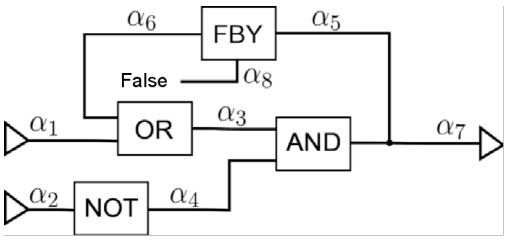
Since our aim is to compute propagation paths, it is convenient to abstract SCADE models by graphs, with nodes representing operators and arcs indicating a propagation channel from the output of one operator to the input of another operator. Fig. 5.a shows the graph associated with the R*S model. Although this model is not a global model in the sense of Fig. 4, we will use it throughout subsections B to G to illustrate the introduced notions.

The graphs we consider involves four types of basic operators:

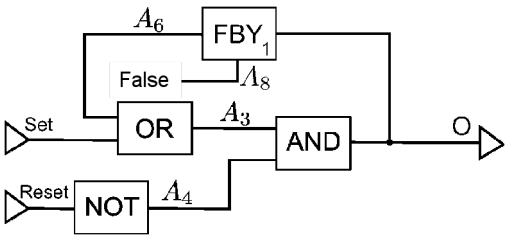
- Interface operators: input operator (noted *IN*), output operator (noted *OUT*), constant input operators *True* and *False*.
- Boolean operators: *NOT(E), AND(E_1,E_2), OR(E_1,E_2)*.
- If-Then-Else operator: *ITE (C,E_1,E_2)*.
- The temporal operator: *FBY(E,1,Init)*.

Without loss of generality, we may consider the generic form *FBY(E,t,Init)* as a cascade of *t* operators introducing a delay of *1*. Similarly, all Airbus symbols can be represented using only basic operators. Our formal definitions can then restrict to models containing basic operators. Of course, the implemented analysis will ultimately have to accommodate the complex operators and symbols. The key point is that their processing can be interpreted in terms of some processing of a flattened model, where each symbol is replaced by its reference description with basic operators.

As seen in Fig. 5.a, a model may contain several instances of a given operator (e.g., here, two instances of *IN*, respectively at the leftmost end of arcs α_1 and α_2). We note



(a) Nodes and arcs



(b) Labeling arcs with variables

Figure 5. R*S graph

Op the set of all operator instances appearing in a model, $Op_{in} \subset Op$ the set of instances of *IN*, *True* and *False* operators, and $Op_{out} \subset Op$ the singleton corresponding to the output of the property observer. In the example of Fig. 5.a, we will assume that the *OUT* instance at the rightmost end of arc α_7 plays the role of this output.

Operators have incoming arcs connected to their input ports, and outcoming arcs connected to their output ports. Let *Arc* be the set of all arcs appearing in the model. We note $arcIn: Op \rightarrow P(Arc)$ and $arcOut: Op \rightarrow P(Arc)$ the functions respectively returning the set of incoming and outcoming arcs of an operator instance. In Fig. 5.a, the *OR* instance has two incoming arcs α_1 and α_6, and one outcoming arc α_3. The *FBY* instance introduces a feedback loop. Its incoming arcs are α_5 (connected to the delayed input port) and α_8 (connected to the initialization port); its outcoming arc is α_6. Several arcs may be connected to an operator output port. For example, the *AND* instance has two outcoming arcs α_5 and α_7, both connected to the same port. However, by construction of SCADE models, it is not possible for several arcs to be connected to the same input port. Also, members of Op_{in} have no incoming arcs, while the (unique) member of Op_{out} has no outcoming arc.

Let *Var* be the set of model variables. A labeling surjection $arc2var: Arc \rightarrow Var$ assigns a variable to each arc. With this labeling, two arcs carry the same variable if and only if they come out of the same output port of an operator instance. In Fig. 5.b, Variable *O* is assigned to both α_5 and α_7. By convention, we always label *O* the arc going to the *OUT* instance of the property observer. Variable *O* is the only model output variable considered by our analysis. The model has a set $Var_{in} \subset Var$ of input variables, associated with arcs coming out of Op_{in} operators. In Fig. 5.b, $Var_{in} = \{Set, Reset, A8\}$.

C. Model Execution, Scenarios and Counterexamples

The definitions of model execution, scenarios and counterexamples all involve the valuation of variables, which we first introduce. Model variables are valuated by Boolean flows, that is, sequences of Boolean values. Since our aim is to analyze finite counterexamples, we need only consider finite (albeit arbitrarily large) flows. Let *Flow* be the universe of all possible finite Boolean flows, and let $size: Flow \rightarrow \mathbb{N}_1$ be the function returning the size of a flow. A valuation σ can be defined as a partial function that associates flows to some of the model variables, $\sigma: Var \nrightarrow Flow$. We note Σ the universe of all possible valuations. Two subsets of Σ are worth being distinguished.

Definition 1 (Σ_{in}, Σ_{all}). We define Σ_{in} as the set of valuations of all input variables, and only of these variables, by flows of a same size. Σ_{all} is the set of valuations of all model variables by flows of a same size.

$$\Sigma_{in} = \{\sigma \in \Sigma \mid dom(\sigma) = Var_{in} \wedge \exists N \in \mathbb{N}_1,\ \forall v \in Var_{in},\ size(\sigma(v)) = N\}$$

$$\Sigma_{all} = \{\sigma \in \Sigma \mid dom(\sigma) = Var \wedge \exists N \in \mathbb{N}_1,\ \forall v \in Var,\ size(\sigma(v)) = N\}$$

We are now equipped to define the notion of scenario, and of the execution of a scenario by the model. A counter-example will then be a scenario, the execution of which valuates the output variable *O* to *false* at the last cycle.

Definition 2 (scenario). A scenario σ is a member of Σ_{in}. The size *N* of the flows assigned to the input variables determines the size of the scenario (number of cycles).

Definition 3 (model execution). An execution of the model is a function $Exec: \Sigma_{in} \rightarrow \Sigma_{all}$ that, for a scenario σ, returns a total valuation σ' such that:

- σ' has the same size as σ.
- $\forall v \in Var_{in},\ \sigma'(v) = \sigma(v)$
- $\forall v \in Var\text{-}Var_{in},\ \sigma'(v)$ is determined in accordance with the semantics of the SCADE operators.

In our work, the semantics is given by a simulation tool, allowing us to supply scenarios to the model (e.g., to replay a counter-example). The SCADE suite® includes such a simulator. Matlab/Simulink® is another example of environment offering simulation facilities for SCADE models, provided a SCADE to Simulink gateway is used. Note that the result σ' of an execution is well defined. The scenarios have a finite size, and the operators we consider (Boolean operators, ITE, FBY) have well-defined outputs whatever their input values: an execution always terminates, and all internal and output variables are valuated.

Before introducing a counterexample as a scenario yielding a false output *O*, we need convenient notations to denote the value of a variable at a given cycle. We first define a projection function $prj: Flow \times \mathbb{N}_1 \nrightarrow Bool$ that, for a Boolean flow *f* and an index *n*, $1 \leq n \leq size(f)$, returns the n^{th} value of *f*.

Notation 1 ($V(n)_\sigma$). We note $V(n)_\sigma$ the value of variable *V* at cycle *n*, during the execution of scenario σ:

$$V(n)_\sigma = prj\ (Exec(\sigma)(V),\ n)$$

Definition 4 (counterexample). A counterexample is a scenario, the execution of which falsifies the property at the last cycle. More formally, a scenario σ of size *N* is a counter-example iff:

- $\forall n \in [1, N-1],\ O(n)_\sigma = true.$
- $O(N)_\sigma = false.$

A counterexample for the R*S model is shown below:

cycle	1	2	3
Set	T	F	F
Reset	F	F	T
A8	F	F	F
A3	T	T	T
A4	T	T	F
A6	F	T	T
O	T	T	F

σ — Total valuation $Exec(\sigma)$

D. Basic Definitions for Paths

A SCADE model can be seen as a graph, the arcs of which are labeled with variables. The execution of a scenario (a counterexample) valuates these variables. We now focus

on propagation paths activated by the execution of a scenario, and playing a role in the observed valuation of model variables. We start with basic definitions for paths.

Definition 5 (path). A path of length $K \in \mathbb{N}_1$ is a finite sequence of model arcs $<\alpha_1, \ldots \alpha_K>$ such that:

$\forall k \in [1, K-1], \exists op \in Op,$

$$\alpha_k \in arcIn(op) \wedge \alpha_{k+1} \in arcOut(op)$$

Let P be the set of paths in the model. For any path $p \in P$, we note $birth(p)$ its first arc α_1, and $death(p)$ its last arc α_K. The path is then *complete* if it connects an input to the output of the property observer.

Definition 6 (complete path). A path $p \in P$ is complete iff:

$$\exists op_1 \in Op_{in},\ birth(p) \in arcOut(op_1) \wedge$$
$$\exists op_2 \in Op_{out},\ death(p) \in arcIn(op_2)$$

The R*S model has an infinite number of complete paths, due to the loop introduced by arcs $\alpha_5, \alpha_6, \alpha_3$.

Definition 7 (order of a path). The order of a path $p \in P$ is the number of FBY operators traversed via their delayed input port.

The order thus measures the delay introduced between the signal carried at $birth(p)$ and the one at $death(p)$. For example, complete path $<\alpha_1, \alpha_3, \alpha_5, \alpha_6, \alpha_3, \alpha_7>$ from Fig. 5.a has order *1*, while $<\alpha_8, \alpha_6, \alpha_3, \alpha_7>$ has order *0*.

Definition 8 (origin of a path). *Origin* is a partial function $P \times \mathbb{N}_1 \nrightarrow Var \times \mathbb{N}_1$ that, for a path p and a cycle number $n > order(p)$, returns a variable v and number n' such that:

- $v = arc2var(birth(p))$.
- $n' = n - order(p)$.

Consider again $p = <\alpha_1, \alpha_3, \alpha_5, \alpha_6, \alpha_3, \alpha_7>$ having order *1*. For $n>1$, $Origin(p,n) = (Set, n-1)$. The function indicates that, via p, the value of O at cycle n may depend on the value of input *Set* at cycle *n-1*. But the dependence is potential. It is effective only for scenarios activating p at cycle n.

E. Activation Condition of a Path

The definition of the activation condition of a path is taken from the work of Lakehal and Parissis [14] who investigated structural testing of Lustre programs. SCADE being based on Lustre, it is straightforward to reuse their definition in our work. Accordingly, the activation condition of a path p, noted $AC(p)$, is a Boolean Lustre expression built recursively by backward traversal of the path, from the last arc to the first. Fig. 6 shows the construction of $AC(p)$. Depending on the traversed operators, local conditions OC are gradually appended to produce $AC(p)$. We refer to [14] for further explanation, and just provide one example below.

The example is path $p = <\alpha_1, \alpha_3, \alpha_5, \alpha_6, \alpha_3, \alpha_7>$ in the R*S model. Its activation condition is recursively built as:

$(false \rightarrow pre(true \wedge OC(\alpha_1, \alpha_3) \wedge OC(\alpha_3, \alpha_5)) \wedge$
$OC(\alpha_6, \alpha_3) \wedge OC(\alpha_3, \alpha_7)$

Which yields:

$(false \rightarrow pre(true \wedge (Set \vee \neg A6) \wedge (\neg A3 \vee A4))) \wedge$
$(A6 \vee \neg Set) \wedge (\neg A3 \vee A4)$

This condition involves two temporal Lustre operators, the initialization (->) and "previous" (*pre*) operators. The first subcondition of form *false -> (...)* forces a false initial value. Indeed, as the path traverses FBY via the delayed input port, it cannot be active at cycle *1*. At cycle $n>1$, the activation condition of p involves the previous values of *Set*, *A6*, *A3* and *A4* at cycle *n-1*, as well as their current values. The overall result depends on the considered scenario.

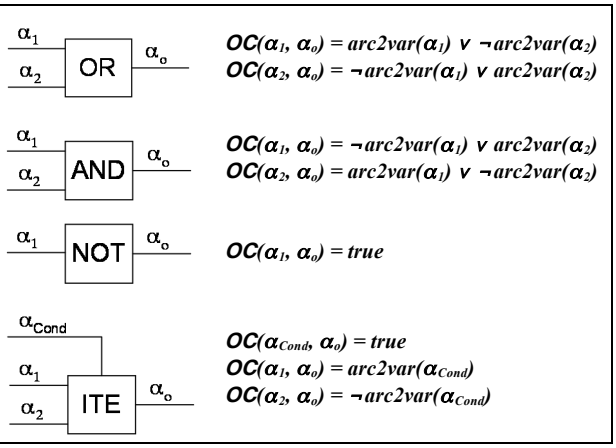
Notation 2 ($A(p, \sigma, n)$). We note $A(p, \sigma, n)$ the value of the activation condition of path p, at the n^{th} cycle of execution of scenario σ.

At cycle n of scenario σ, the active paths are the ones such that $A(p, \sigma, n) = true$. Among all active paths at cycle n, we distinguish the ones having their first arc labeled with an input variable, and call them *propagation paths*. Propagation paths allow scenario inputs to have an impact on the values of model variables. For example, if the first arc of path p is labeled by input variable *I* and its last arc is labeled

Let $p = <\alpha_1, \ldots \alpha_K>$ be a path of length $K \geq 1$. The activation condition of p, noted $AC(p)$, is a Lustre expression defined as follows:

1. If $K=1$ then $AC(p) = true$.
1. else
   Let $p' = <\alpha_1, \ldots \alpha_{K-1}>$ be the prefix of p of length $K-1$,
   Let *op* be the operator such that $\alpha_{K-1} \in arcIn(op)$
   and $\alpha_K \in arcOut(op)$,
   (a) if *op* is a Boolean or ITE operator then
   $AC(p) = AC(p') \wedge OC(\alpha_{K-1}, \alpha_K)$,
   where $OC(\alpha_{K-1}, \alpha_K)$ depends on *op* (see Fig. 6.b).
   (b) if *op* is an operator FBY (E,1,Init) then
   i. if α_{K-1} is connected to the delayed input port *E*,
   $AC(p) = false \rightarrow pre(AC(p'))$
   ii. if α_{K-1} is connected to the initialization port *Init*,
   $AC(p) = AC(p') \rightarrow false$

(a) Main definition



(b) OC conditions

Figure 6. Path activation conditions (adapted from from [14])

V, then $\mathcal{A}(p, \sigma, n) = true$ indicates that the computation of $V(n)_\sigma$ depends on the input data $I(n\text{-}order(p))_\sigma$.

Definition 9 (set of input propagation paths $Pe(\sigma, n)$). The set of input propagation paths at cycle n of scenario σ is: $Pe(\sigma, n) = \{p \in P \mid \mathcal{A}(p, \sigma, n) = true \wedge arc2var(birth(p)) \in V_{in}\}$

Let us go back to the previous example of path in the R*S model. We consider the scenario given in Section III.C and reproduced hereafter.

cycle n	1	2	3
Set	T	F	F
Reset	F	F	T
A8	F	F	F
A3	T	T	T
A4	T	T	F
A6	F	T	T
O	T	T	F

cycle n	1	2	3
¬A3 ∨ A4	T	T	F
A6 ∨ ¬Set	F	T	T
pre(¬A3∨ A4)	–	T	T
pre(Set ∨ ¬A6)	–	T	F
false -> pre(…)	F	T	F
$\mathcal{A}(p, \sigma, n)$	**F**	**T**	**F**

As can be seen, path p belongs to $Pe(\sigma, 2)$. Its prefix paths $< \alpha_1 >$, $< \alpha_1, \alpha_3 >$, … also do. Input $Set(1)_\sigma$ had thus an impact on all the following values: $A3(1)_\sigma$, $O(1)_\sigma$, $A6(2)_\sigma$, $A3(2)_\sigma$, $O(2)_\sigma$.

Path p does not belong to $Pe(\sigma, 3)$. It played no role in the falsification of output O at the last cycle. Intuitively, the falsification was caused by input $Reset(3)_\sigma$, via the propagation path $< \alpha_2, \alpha_4, \alpha_7 >$.

F. Causes for a Counterexample

When analyzing a counterexample σ of size N, we search for the causes of $O(N)_\sigma = false$, that is, of the observed property violation at cycle N. More generally, a cause may explain the value of any model variable at cycle $n \in [1, N]$ of some scenario.

The causes of the value of $V = arc2var(\alpha)$ at cycle n are searched over propagation paths from inputs to arc α. A cause $\mathcal{C}$ will then be a subset of $Pe(\sigma, n)$, containing paths the last arc of which is α. The origins of these paths (see Definition 8) determine some input values $V_i(n_i)_\sigma$ that had an impact on $V(n)_\sigma$. For $\mathcal{C}$ to be a cause, these input values must be sufficient to:

- Ensure that all paths in $\mathcal{C}$ are active at cycle n;
- Ensure that V takes the value observed at cycle n.

These sufficient conditions can be expressed by referring to variants σ' of scenario σ, that keep the inputs $V_i(n_i)_\sigma$ determined by the cause but let other inputs receive arbitrary values.

Definition 10 (cause). For arc α labeled by variable V, a cause of value $V(n)_\sigma$ observed during the execution of σ is a set of path $\mathcal{C}$ such that:

1. $\mathcal{C} \subseteq Pe(\sigma, n)$
2. $\forall p \in \mathcal{C}$, $death(p) = \alpha$
3. Let $Origins = \{(V_i, n_i) \in Var_{in} \times \mathbb{N}_1 \mid \exists p \in \mathcal{C}, (V_i, n_i) = origin(p,n)\}$.
   $\forall \sigma' \in \Sigma_{in}$ such that σ' has the same size as σ, and $V_i(n_i)_{\sigma'} = V_i(n_i)_\sigma$ for all $(V_i, n_i) \in Origins$, we have:
   (a) $\forall p \in \mathcal{C}$, $\mathcal{A}(p, \sigma', n) = true$
   (b) $V(n)_{\sigma'} = V(n)_\sigma$

For a target arc, there may be several causes, each independently explaining the observed value at cycle n.

Notation 3 ($\mathcal{C}(\alpha, \sigma, n)$). We note $\mathcal{C}(\alpha, \sigma, n)$ the set of causes of the value taken by variable $arc2var(\alpha)$ at the n^{th} cycle of execution of scenario σ.

If α is the output arc of the property observer, and σ is a counter-example of size N, we say that $\mathcal{C}(\alpha, \sigma, N)$ is the set of causes *for the counterexample*. Each cause is then a set of complete paths, and $\mathcal{C}(\alpha, \sigma, N)$ is a set of sets of complete paths.

We illustrate Definition 10 with a simple R*S scenario of size *1*, that turns out to involve multiple causes. Fig. 7 visualizes the execution of this scenario, as well as the 3 complete paths it activates (higher order complete paths are inactive):

- $AC(p_1) \Leftrightarrow$ (Set ∨ ¬A6) ∧ (¬A3 ∨ A4)
- $AC(p_2) \Leftrightarrow$ (¬A4 ∨ A3)
- $AC(p_3) \Leftrightarrow$ (true -> false) ∧ (A6 ∨ ¬Set) ∧ (¬A3 ∨ A4)

We show that $\mathcal{C}(\alpha_7, \sigma, 1) = \{ \{p_2\}, \{p_1, p_3\}, \{p_1, p_2, p_3\} \}$.

Let us first show that $\{p_2\}$ is a cause:

1. $\{p_2\} \subseteq Pe(\sigma, 1)$
2. $death(p_2) = \alpha_7$
3. $Origins = \{(Reset, 1)\}$.
   $\forall \sigma' \in \Sigma_{in}$ such that σ' has size 1 and $Reset(1)_{\sigma'} = true$, we have $A4(1)_{\sigma'} = false$. Then:
   (a) $\mathcal{A}(p_2, \sigma', n) = true$
   (b) $O(1)_{\sigma'} = O(1)_\sigma = false$.

Similarly, $\{p_1, p_3\}$ is a cause:

1. $\{p_1, p_3\} \subseteq Pe(\sigma, 1)$
2. $death(p_1) = death(p_3) = \alpha_7$
3. $Origins = \{(Set, 1), (A8, 1)\}$.
   $\forall \sigma' \in \Sigma_{in}$ such that σ' has size 1, $Set(1)_{\sigma'} = false$ and $A8(1)_{\sigma'} = false$, we have $A6(1)_{\sigma'} = false$ and $A3(1)_{\sigma'} = false$. Then:
   (a) $\mathcal{A}(p_1, \sigma', 1) = true$ and $\mathcal{A}(p_3, \sigma', 1) = true$
   (b) $O(1)_{\sigma'} = O(1)_\sigma = false$.

It then easy to deduce that $\{p_1, p_2, p_3\} = \{p_2\} \cup \{p_1, p_3\}$ is also a cause. It could be verified that no other subset of complete paths is a cause, hence $\mathcal{C}(\alpha_7, \sigma, 1) = \{ \{p_2\}, \{p_1, p_3\}, \{p_1, p_2, p_3\} \}$.

Causes $\{p_2\}$ and $\{p_1, p_3\}$ independently explain the falsification of the output at cycle 1. The third cause includes the other ones, and does not bring any new insights. It is not *minimal*.

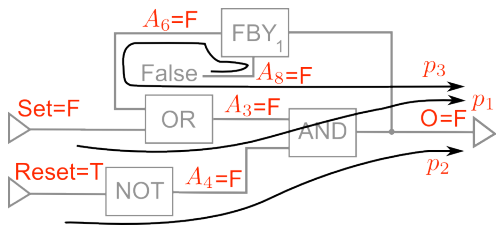


Figure 7. A scenario with multiple causes

G. Minimal Causes

Definition 11 (minimal cause). A cause $c_i \in \mathcal{C}(\alpha, \sigma, n)$ is minimal iff: $\forall\, c_j \in \mathcal{C}(\alpha, \sigma, n), c_j \neq c_i \Rightarrow (c_j \not\subset c_i)$.

Minimality ensures that the cause does not contain unnecessary paths. Any proper subset of paths is no longer a cause. The set of minimal causes provides more concise information than the set of all causes. It can be used to characterize a counterexample.

Notation 4 ($\mathcal{C}_{min}(\alpha, \sigma, n)$). We note $\mathcal{C}_{min}(\alpha, \sigma, n)$ the set of minimal causes of the value taken by variable *arc2var*(α) at the n^{th} cycle of execution of scenario σ.

The set of minimal causes not only provides insights into a counterexample, but also allows the comparison with other counterexamples obtained from the same model.

Table I shows various counter-examples for the R*S model. The first three are of size 1, while σ_4 is of size 3. Input *A8* is not represented, as it corresponds to a false flow in each case.

It can be seen that σ_3 and σ_4 give exactly the same information, that is, a reset of the latch always puts the output to false. Shifting the reset to cycle 3 does not change the observed violation pattern.

Counterexamples σ_1 and σ_2 reveal another violation pattern corresponding to $\{p_1, p_3\}$: if the latch is not set at cycle *1*, the false initialization will yield a false output.

It is interesting to obtain several such violation patterns before attempting to fix a problem. This calls for the generation of several counterexamples, the model checker being guided toward the search for new causes. For example, knowing $\{p_2\}$ from σ_1, we may tell the model checker to search for a violation with p_2 inactive, or with p_3 now active. We thus propose that the search be guided by path activation objectives extracted from the known minimal causes. The case study of Section V will illustrate the concept.

TABLE I. FOUR COUNTER-EXAMPLES

	Set	Reset	O	$\mathcal{C}_{min}$
σ_1	F	F	F	$\{\{p_1, p_3\}\}$
σ_2	F	T	F	$\{\{p_2\}, \{p_1, p_3\}\}$
σ_3	T	T	F	$\{\{p_2\}\}$
σ_4	TFF	FFT	TTF	$\{\{p_2\}\}$

IV. Discussion of Formalization Choices

Our formal background has reformulated the general problem of *assisting in the analysis of counter-examples*, in terms of more precise problems involving the *calculation of the minimal causes of counter-examples*. Minimal causes are defined as subsets of propagation paths that are active at the property violation cycle.

Elaborating Definition 10 was not straightforward. We present here other definitions we considered, and why we did not retain them. Minimality was also found difficult to calculate efficiently, which has consequences for the tool to come in Section V.

A. Alternative Input-Oriented Definition

The causes of the value taken by $V = arc2var(\alpha)$ at cycle n could have been defined as sets of pairs $(V_i, n_i) \in Var_{in} \times \mathbb{N}_1$ such that:

$$\forall\, \sigma' \in \Sigma_{in},\ \wedge_i\, (V_i(n_i)_{\sigma'} = V_i(n_i)_{\sigma}) \Rightarrow V(n)_{\sigma'} = V(n)_{\sigma}$$

In this definition, the cause is searched over the set of input values of the scenario, without explicit consideration for active paths.

The difference with our path-oriented definition is exemplified by the treatment of if-then-else selection. Assume the model merely contains an operator *ITE* (C, I_1, I_2), a counter-example of size *1* is found, and it assigns a false value to both inputs I_i.

In the input-oriented definition, a cause for the counterexample is $\{(I_1,1), (I_2,1)\}$. Indeed, false I_i values yield a false ITE output whatever the value of C. Input C can be ignored. We do not care about whether propagation occurs via I_1 or I_2. The model semantics ensures that the output is falsified in any case.

In contrast to the previous one, our definition attaches importance to identifying the internal propagation paths. Any cause must thus include the path connecting input C to O, because C determines the selected value. Paths via I_1 or I_2 cannot belong to the same cause, they correspond to different scenarios. We retain this definition to match usual debugging activity. To understand incorrect results, the user examines the locations that are traversed while execution proceeds, hence our focus on propagation paths.

B. Causes and Model Hypotheses

Our definition of causes does not account for hypotheses constraining MUV behavior (see Fig. 4). Complete paths from inputs to O do not traverse the hypotheses observer.

Definition 10 could reintroduce the hypotheses, by considering only scenarios that fulfill them. Currently, σ fulfills them by construction if it is a counterexample returned by the model checker. But its variants σ' assign arbitrary values to some inputs (σ' execution is still well-defined).

Let us take the example of a toy MUV corresponding to a single OR operator with inputs I_1, I_2. We assume that the OR output is the global O to falsify. Let p_1 be the path connecting I_1 to O, and p_2 be the other path from I_2 to O. An hypothesis $I_1 = I_2$ is added. A counter-example for this

global model has both inputs to false.

According to Definition 10, $\{p_1\}$ is not a cause: there exists a scenario σ' retaining a false I_1 but letting I_2 be true, that neither activates p_1 nor falsifies the output. If we change Definition 10 to account for the hypothesis, then scenario σ' no longer exists and $\{p_1\}$ is indeed a cause. By hypothesis, a false I_1 yields a false I_2 which then ensures propagation to the OR output.

We decided not to account for such semantic consequences. The calculation of minimal causes should correspond to a light analysis, giving quick feedback on counter-examples. Model checking is highly time consuming, the cost of any additional treatment must be negligible to the user. We thus deemed it preferable to avoid consideration for hypotheses.

C. Minimality and Reconvergent Paths

Reconvergent paths also have semantic consequences that make the calculation of minimal causes complex. Such paths introduce different propagation channels for a given input value. Fig. 8 provides a toy example. There are two different paths p_1 and p_2 connecting $in_1(n)_\sigma$ to $O(n)_\sigma$. On the contrary, the R*S model does not exhibit reconvergence. The different paths from an input to O have different orders, the propagated origins (in the sense of Definition 8) are different. Generally speaking, the reference models we took to describe Airbus symbols in terms of basic operators are all convergence-free.

To show the consequences of reconvergent paths, let us assume a scenario of size *1* for the model in Fig. 8, with both inputs to true. The three complete paths p_1, p_2 and p_3 are all active. From Definition 10, the causes of the value $O(n)_\sigma$ are:

$\{ p_1\}, \{ p_2\}, \{ p_1, p_2\}, \{ p_1, p_3\}, \{ p_2, p_3\}, \{ p_1, p_2, p_3\}$

and the minimal ones are $\{ p_1\}$ and $\{ p_2\}$. Path p_3 does not belong to any minimal cause. Due to the reconvergent paths, the semantics of the model is actually $O = in_1$.

Let us now consider a tree-like view of the model. An input in_1' is introduced at the beginning of p_2. Ignoring $in_1 = in_1'$, the set of causes is now: $\{\{ p_1, p_2\}, \{ p_1, p_3\}, \{ p_1, p_2, p_3\}\}$, with minimal causes $\{ p_1, p_2\}$ and $\{ p_1, p_3\}$. Note that any cause for the tree-like model is also a cause for the original model. But minimality is lost, because we ignored the semantics consequence of reconvergence at the AND operator. This suggests that the calculation of *causes* can be implemented by a simple backward traversal of the model structure (with some local reasoning at each operator), while the calculation of *minimal* causes cannot. Rather, a mix of backward and forward analyses is required to account for reconvergent paths. The problem is well-known from work on testing hardware circuits, and yields an exponential complexity of ATPG algorithms [12].

In our case, it is crucial that the structural analysis has a negligible computational time compared to model checking. We thus decided not to account for reconvergent paths. Our tool considers models as tree structures, and we accept that the causes returned to the user may be non-minimal in some cases.

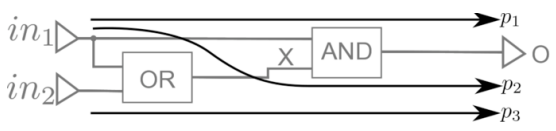


Figure 8. Model with reconvergent paths

V. Implementation and Case Study

A. Algorithm

A recursive algorithm has been defined to compute causes. It starts the analysis of the counter-example from the output arc of the property observer at violation time. Then, it performs a backward analysis both structural (from the output to the inputs) and temporal (from cycle N of the violation to previous instants). For example, the causes of the output of an FBY operator at cycle $n>1$ are built from the causes of the delayed input at cycle $n-1$. If the current operator has several inputs, the joint contribution of their causes is taken into account. For example, all the inputs of an AND operator must be true for the output to be true, while each false input independently explains a false output. Dedicated treatments have been defined for each operator, be it a basic operator or a symbol.

The terminal case of the recursion is when an IN operator is reached. Before this, intermediate operators may be traversed several times, since the analysis may involve different cycle numbers. In the worst case, the N cycles of execution have to be considered. Hence, worst-case complexity is quadratic in N and the size of the model.

We have proved that the algorithm terminates and that it computes causes of the counter-example. The demonstration is made by induction on the execution tree of the algorithm [17]. We use the reference models of symbols in terms of basic operators to establish the correspondence between the proposed treatments and the definitions in Section IV.

The minimality property is not inductive in the general case, but we prove that the algorithm computes the minimal causes if the global model does not exhibit reconvergence.

B. Prototype

A prototype called STANCE (STructural Analysis of Counter-Examples) implements this algorithm. It has been developed in the Matlab/Simulink environment [18] because this environment allows rapid prototyping, provides similar modelling capacities as SCADE and includes a model checker called SLDV (based on the same technology as SCADE design verifier).

The prototype represents about 700 lines of code in the Simulink script language. Its use is illustrated hereafter.

C. Application

Thanks to this prototype, the approach has been evaluated on an Airbus case study. We will not explain in detail the case study, but try to give a feel of the usefulness of the structural analysis.

The model under verification is the computation of a decision depending on two conditions and an authorization. The high-level form of the decision is $(C_1 \vee C_2) \wedge A$, where the conditions are complex. The model and its property

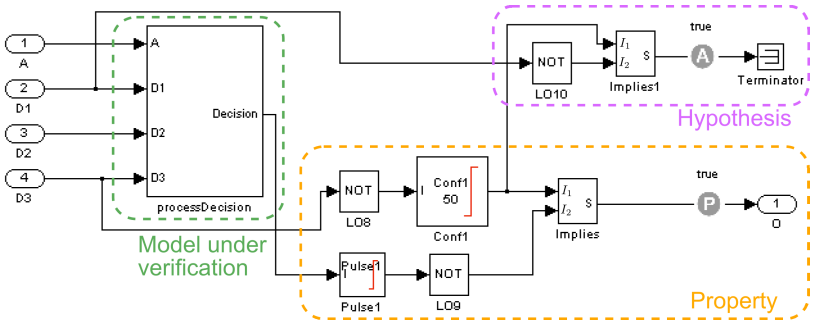


Figure 9. Global model for the case study

observer include temporal symbols such as confirmers, triggers or latches. Fig. 9 shows the verification architecture (in the Simulink syntax).

When launched on this global model, SLDV returns a counter-example of size 53: it provides the value of the four inputs at the 53 cycles. This counter-example can be simulated, but understanding the reasons for the violation requires some effort.

STANCE builds the complete counter-example (values of 24 variables at the 53 cycles) using Simulink simulator. Then it analyzes this counter-example in order to compute the causes. It finds one, we could verify that it is minimal. The computation of the cause takes less than 1s, and STANCE returns two visual results (Fig. 10):

- The model under verification where paths of the cause are colored;
- A list of key events to provide information on the timing aspects of the cause. For example, *A(53): 1* means that variable *A* has value true at cycle 53.

The two views are complementary. The list provides temporal information that is not present in the colored model, which only gives information on structure. Key events include not only the origins of the paths of the cause, but also key values for temporal operators that are useful to understand the counter-example.

Airbus engineers have judged these two visual results very useful. The colored model allows the user to focus on the structural part of the model that is responsible for the violation of the property. Then the key events provide detailed information to explain the violation.

D. Generation of New Counterexamples

New counterexamples aim at revealing new ways to falsify the property. As explained in section III.G, a structural approach can be used to guide the model checker toward the search for new counterexamples. To demonstrate the concept, we have implemented a first path-based strategy in STANCE. The strategy seeks to trigger the activation of new paths. For this, STANCE stores the non-active input arcs for each operator encountered during backward analysis

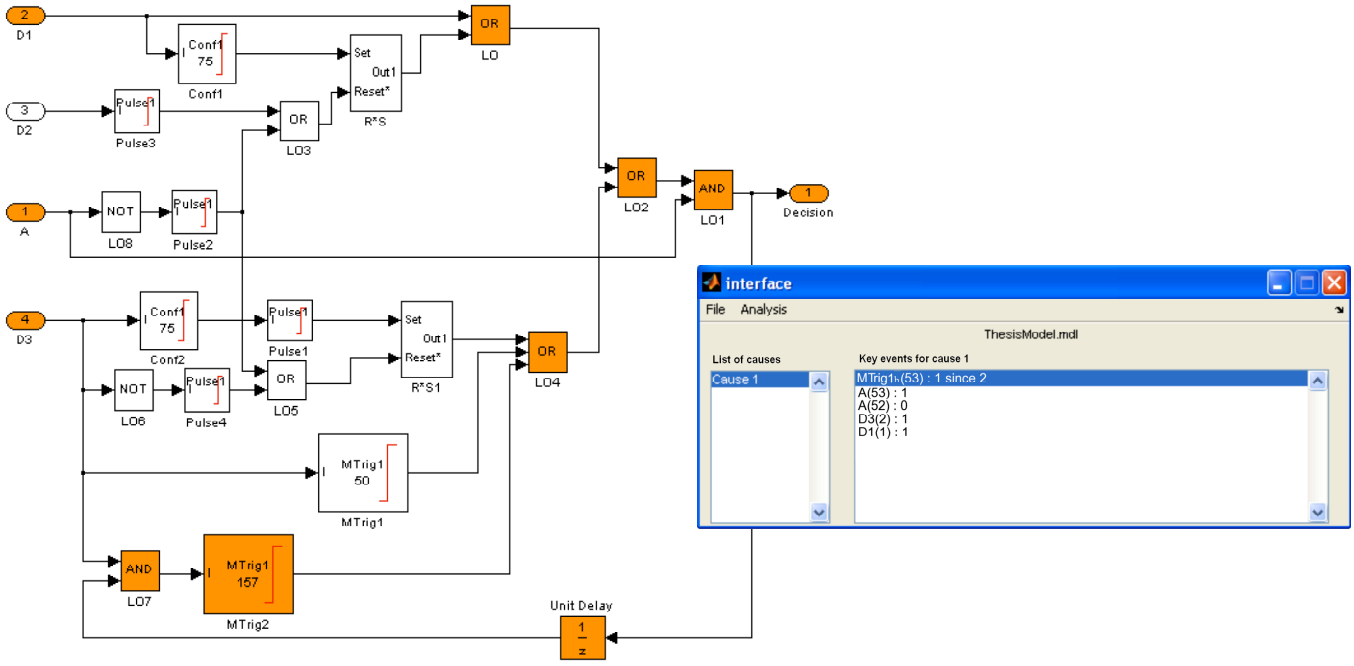


Figure 10. Colored model and key events provided by STANCE

of the causes. Then, for counter-example generation, it automatically modifies property P into $B \Rightarrow P$, where B requires that one of the stored arcs becomes active at the level of the corresponding operator. Currently, STANCE merely selects the first non-active arc of the rightmost operator. More elaborated strategies could be studied, this will be the subject of future work.

The simple strategy has been applied to the case study and a new counterexample of size 77 has been found. Its analysis returns two minimal causes. As visually shown in Fig. 11, the second one activates completely different paths in the model. Once again, STANCE was very useful to understand this counterexample that provided new insights into the model behavior.

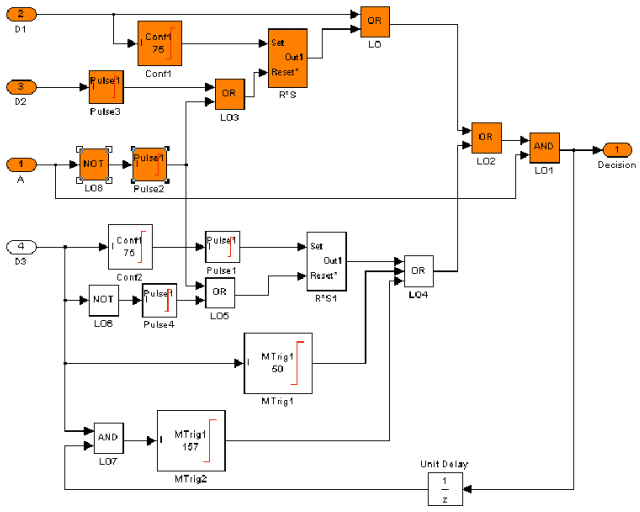


Figure 11. Colored model for the new counter-example

VI. CONCLUSION

This paper has presented an approach for the automated analysis of counterexamples. It is based on the notion of cause defined as a set of paths in the model. The approach has been implemented in the STANCE tool. A first case study has given promising results for the verification of Airbus models. The analysis is currently restricted to Boolean models with a unique clock, a first extension would be to take numerical models and multiple clocks into account.

The notion of cause allows the comparison of counter-examples and thus opens avenues for future work on strategies of generation of new counter-examples. An example of such a strategy has been implemented in STANCE, but it is only a first assessment of the feasibility and potential interest of the idea. More work needs to be done on the definition and comparison of different generation strategies. Work on GATeL [19] and concolic testing [20] [21] will be a source of inspiration.

REFERENCES

[1] http://www.esterel-technologies.com/products/scade-suite

[2] S. Duprat, J. Souyris, and D. Favre-Félix, "Formal verification workbench for Airbus avionics software," Proc. Embedded Real-Time Software (ERTS), 2006.

[3] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels, "Model checking flight control systems: the Airbus experience," In ICSE Companion, 2009, pp. 18-27.

[4] C. Artho, K. Havelund, and S. Honiden, "Visualization of concurrent program executions," Proc. Workshop on Software Architectures and Component Technologies (SACT'07), 2007.

[5] H. Aljazzar, and S. Leue, "Debugging of dependability models using interactive visualization of counterexamples," Proc. 5th Int. Conf. on Quantitative Evaluation of Systems (QEST 2008), Sept. 2008.

[6] Y. Dong, C.R. Ramakrishnan, and S. A. Smolka, "Evidence Explorer: a tool for exploring model-checking proofs," Proc. Computer-Aided Verification (CAV'03), 2003.

[7] L. van den Berg, P. Strooper, and W. Johnston, "An automated approach for the interpretation of counter-examples;" Proc. Workshop on Verification and Debugging 2006, ENTCS vol 174, Elsevier, 2007.

[8] A. Groce, and W. Visser, "What went wrong: explaining counter-examples," Proc. SPIN, 2003.

[9] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," Proc. POPL, 2003.

[10] I. Beer, S. Ben David, H. Chockler, A. Orni, and R.J. Trefler, "Explaining counter-examples using causality," Proc. CAV, 2009, pp. 94-108.

[11] M. Chechik, and A. Gurfinkel, "A framework for counterexample generation and exploration," STTT 9(5-6): 429-445, 2007.

[12] M. Bushnell, and V. Agrawal, Essential of electronic testing for digital memory and mixed-signal VLSI circuits, Kluwer Academic Publishers, 2000.

[13] F. Gaucher, E. Jahier, B. Jeannet and F. Maraninchi, "Automatic state reaching for debugging reactive programs," Proc. Int. Workshop on Automated and Algorithmic Debugging (AADEBUG'03), 2003.

[14] A. Lakehal, and I. Parissis, "Structural coverage criteria for LUSTRE/SCADE programs," Softw. Test., Verif. Reliab., vol. 19, issue 2, 2009, pp. 133–154.

[15] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous dataflow programming language Lustre," In Proceedings of the IEEE, vol. 79, issue 9, 1991.

[16] H. Halbwachs, F. Lagnier, and P. Raymond, "Synchronous observers and the verification of reactive systems," Proc. Third Int. Conf. on Algebraic Methodology and software Technology (AMAST'93), Workshops in computing, Springer Verlag, Twente, June 1993.

[17] T. Bochot, Vérification par model checking des commandes de vol : applicabilité industrielle et analyse de contre-exemples, PhD thesis, Université de Toulouse, ISAE, 2009.

[18] http://www.mathworks.com/products/simulink/

[19] B. Marre, and A. Arnould, "Test sequence generation from Lustre Descriptions: GATEL," Proc. Int. Conf. on Automated Software Engineering (ASE), 2000.

[20] B. Marre, P. Mouy, and N. Williams, "On-the-fly generation of K-path tests for C functions," Proc. Int. Conf. on Automated Software Engineering (ASE), 2004.

[21] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," Proc. ESEC/FSE, 2005.