Show Me New Counterexamples: A Path-Based Approach

Kalou Cabrera Castillos, Hélène Waeselynck, Virginie Wiels

▶ To cite this version:

Kalou Cabrera Castillos, Hélène Waeselynck, Virginie Wiels. Show Me New Counterexamples: A Path-Based Approach. 8th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2015), Apr 2015, Graz, Austria. 10.1109/ICST.2015.7102606 . hal-01953299

HAL Id: hal-01953299
https://hal.science/hal-01953299v1

Submitted on 12 Dec 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Show Me New Counterexamples: A Path-Based Approach

Kalou Cabrera Castillos, Hélène Waeselynck
LAAS-CNRS and Univ. Toulouse
7 Av. Colonel Roche
31400, Toulouse, France
Email: kalou.cabrera.castillos@laas.fr / helene.waeselynck@laas.fr

Virginie Wiels
ONERA/DTIM
2 AV. E. Belin, BP74025
31055 Toulouse, France
Email: virginie.wiels@onera.fr

***Abstract*—We consider lightweight usage of model-checking for the debugging of Simulink models. A problem is that model-checkers typically return only one counterexample, which may slow down the debugging process. We propose an approach and a tool to produce several counterexamples, exemplifying different property violation patterns for a given version of the design. The approach uses data collected during the replay of the counterexamples to synthesize queries for the model-checker, so that it finds counterexamples that activate new paths. The approach is applied to an academic example and an industrial model from the automotive domain.**

Keywords*—*synchronous data-flow models, model-checking, structural analysis

I. Introduction

The development of reactive systems commonly uses a model-based approach. The focus is then on designing models from which a prototype or even the real implementation code can be automatically derived. Among the available modeling formalisms, synchronous data-flow languages like Scade [1] or Simulink [2] are well-suited for capturing a reactive behavior. Such a behavior is abstracted in terms of periodically repeated execution cycles in which the system reads inputs from the environment and reacts by emitting outputs. Model simulation facilities allow a thorough testing of the design.

In addition to simulation, Scade and Simulink toolsets also support model-checking, opening the possibility for a formal analysis of models. This paper considers a lightweight approach, in which designers use model-checking as a debugging facility. Analysis is focused on a few critical functions, typically functions synthesizing alarms or elaborating critical Boolean control parameters. The counterexamples returned by the model-checker are used to revise the model under design until it matches the specified property. Our previous work in the avionics domain suggests that this lightweight approach may indeed be quite effective, even with respect to subtle design flaws that are found by lab tests (*i.e.*, tests of the real equipment in the loop with a flight simulator) [3].

The implementation of the debugging approach may however face several practical problems. One is how to proceed when a lengthy counterexample is returned and the reasons for property violation must be extracted from it. We developed a tool, STANCE (Structural Analysis of counterexamples), to alleviate the effort of engineers [4]. STANCE is interfaced with the Simulink toolset. It automatically extracts the subset of time-stamped input values that are sufficient to reproduce the violation, and visually shows the structural parts of the models that are activated by them and play a role in the observed violation. The tool thus filters out irrelevant data from the counterexample, while allowing engineers to focus on the interesting details of its execution.

A second practical problem is that model-checkers typically return only one counterexample, and no further feedback can be obtained until the model (or the property) is modified to eliminate this counterexample. But there can be other unrevealed flaws in the current version of the model, or the revealed flaw may induce alternative ways to violate the property, which are not shown by the first counterexample found. To speed up the debugging process, it would be more efficient to give the engineer as much feedback as possible before she reworks the design. It calls for forcing the model-checker to search for several counterexamples, that exemplify different property violation patterns for a given version of the design. This is the issue addressed in this paper.

Our search for new counterexamples is based on the structural analysis approach implemented in STANCE. It is inspired by the principle of concolic testing [5] [6]. Concolic testing uses concrete data, collected during the execution of test cases, to guide the constraint-based generation of test cases activating new paths. In a similar way, our approach uses data collected during the replay of the counterexamples to synthesize queries for the model-checker, so that it finds counterexamples that activate new paths. This paper explains how the candidate new paths are extracted from the replay data, and how some automatically inserted instrumentations force the model-checker to consider these paths.

Section II provides background about STANCE and the structural analysis of counterexamples. It also introduces a small case study that we use as a running example to illustrate the concepts throughout the paper. Section III presents the search for new counterexamples, and Section IV illustrates its application to the running example. Section V reports on a case study of larger size than the running example, coming from the automotive industry. Section VI presents related work and Section VII concludes the paper.

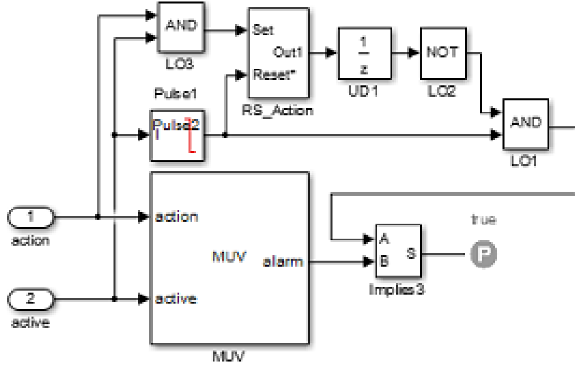


Fig. 1: Example of Simulink model

II. BACKGROUND AND RUNNING EXAMPLE

Figure 1 shows an exemplary verification architecture in Simulink. The model under verification (MUV) is monitored by an observer of the property to check. Both the property and the model are expressed in the same language, *i.e.* the Simulink data-flow language. This language offers basic operators, like Boolean operators (AND, NOT, ...) and the unit delay 1/z. Besides these basic operators, it is usually convenient to define a library of reusable compound operators, the semantics of which can be given in terms of the basic ones. The expression of the property in Figure 1 uses one compound logical operator (*Implies*). It also uses two compound temporal operators: a *pulse*, detecting a falling edge when its Boolean input goes from 1 to 0, and an *RS-latch* used to store Boolean information. The latch is controlled via its Set and Reset inputs (the star annotation indicates that Reset has priority over Set). *Source* and *sink* operators are used for the production of inputs (*e.g.*, the production of *action* and *active*) and for the consumption of the final output (*e.g.*, the final value P of the property observer after the logical implication). This output should always have the value 1. The model-checker is then challenged to falsify it. If it succeeds, it returns a counterexample under the form of a sequence of n time-stamped inputs *action(1) active(1)*, ..., *action(n) active(n)* that, when replayed on the model and its observer, yields a final output sequence *P(1)=1*, ..., *P(n-1)=1, P(n)=0* falsified at cycle n. The counterexample demonstrates that MUV violates the property.

Our STANCE tool performs a structural analysis of the paths activated by the replay of the counterexample. Before presenting the results of this analysis, we need to clarify what it means for a path to be activated. Indeed, the notion of path activation in a synchronous data-flow model departs from the usual one in control-flow graphs.

A. Path activation in data-flow models

Fundamentally, a synchronous data-flow model is a set of equations defining how input flows are transformed to output flows via some functional operators. At each clock tick, all operators are simultaneously executed to produce values. A digraph graphically represents the equations. The vertices are operators and directed arcs carry data from a producer operator to a consumer one. Given an arc α, we note α(n) the value that this arc carries at the nth clock tick, *i.e.* at the nth execution cycle of the model.

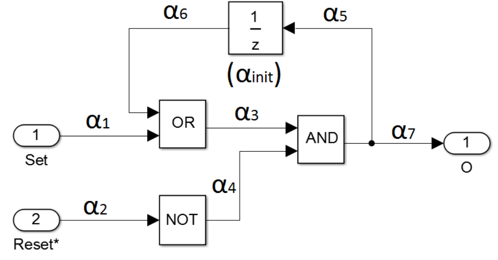


Fig. 2: R*S latch detail with labeled arcs

A path in the digraph is a non empty sequence of arcs such as, for any pair of consecutive arcs (α_i, α_{i+1}), the operator entered by α_i is the one exited by α_{i+1}. The path is a potential data propagation channel: the value of the first arc at some cycle n may affect the value of the last arc at some later cycle $n+j$ ($j \geq 0$). The time shift depends on the number of delays along the path. To explain it, let us zoom on the RS-latch of the property observer. Figure 2 gives its reference model in terms of basic operators. Path $< \alpha_1, \alpha_3, \alpha_7 >$ does not introduce any delay: it connects α_1*(n)* to α_7*(n)*, *i.e.*, the output of the latch may depend on the Set input at the current cycle. Paths of the form $< \alpha_1, \alpha_3 >< \alpha_5, \alpha_6, \alpha_3 >^k< \alpha_7 >$, involving $k > 1$ traversals of a unit delay, indicate that the output may also depend on the Set input k cycles ago. Note that the number of paths in this model is infinite, due to the temporal loop.

Intuitively, in this data-flow context, the activation of a path means that the data propagation is effective. For example, path $p_1 =< \alpha_1, \alpha_3, \alpha_7 >$ is inactive if both the Set and Reset inputs are 1 at the current cycle: the AND operator “stops” the propagation of the Set input to give priority to the Reset. Figure 3 formalizes the path activation conditions for basic operators. It is adapted from [7] (the conditions were originally defined as Boolean Lustre expressions). A condition is built recursively by backward traversal of the path.

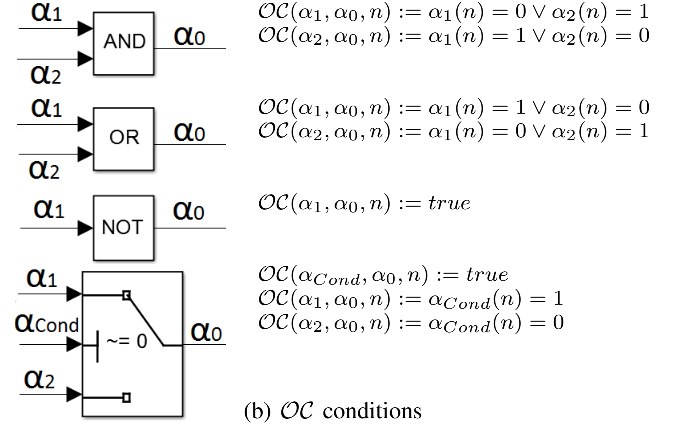
According to this definition, a path consisting of a single arc is trivially active at any cycle. For longer paths, the activation condition depends on the traversed logical and temporal operators. A value of 0 is always propagated by an AND operator, while a 1 is propagated only if the other input is also 1. The Switch operator selects its top or bottom input depending on a condition (the middle input): the active paths are via the condition and the selected input. The Unit Delay propagates the value of its input at cycle $n-1$, to affect calculation at cycle n. If there is no previous cycle ($n = 1$), an initialization value is provided. This value is a (constant) input of the operator, and paths originating from it are considered. In Figure 2, those paths start by the (implicit) arc α_{init}. The activation condition of $< \alpha_{init}, \alpha_6 >$ holds at the first cycle only, while the activation condition of $< \alpha_5, \alpha_6 >$ holds at any cycle but the first one. Generally speaking, if a path introduces k delays, its activation condition does not hold for the first k cycles. So, even if the number of paths is infinite, the number of paths activated by a finite number of execution cycles is actually finite.

Figure 4 exemplifies a concrete execution of the latch

Let $p =< \alpha_1, \ldots, \alpha_k >$ be a path of length $k \geq 1$. The activation condition of p at cycle $n \geq 1$, noted $\mathcal{AC}(p, n)$, is a predicate defined as follows:

- if $k = 1$ $\mathcal{AC}(p, n) := true.$
- else Let $p =< \alpha_1, \ldots, \alpha_{k-1} >$ be the prefix of p of length $k-1$, and let *op* be the operator entered by α_{k-1} and exited by α_k
  - if *op* is a Boolean or a Switch operator then
    - $\mathcal{AC}(p, n) := \mathcal{AC}(p', n) \wedge \mathcal{OC}(\alpha_{k-1}, \alpha_k, n)$, where $\mathcal{OC}(\alpha_{k-1}, \alpha_k, n)$ depends on *op* (see Fig. 3b)
  - if *op* is the unit delay operator 1/z then
    - if α_{k-1} enters the delayed input port, $\mathcal{AC}(p, n) := (n = 1 \Rightarrow false) \wedge (n \neq 1 \Rightarrow \mathcal{AC}(p', n-1))$
    - if α_{k-1} is the constant initialization input, $\mathcal{AC}(p, n) := (n = 1 \Rightarrow true) \wedge (n \neq 1 \Rightarrow false)$
  - if *op* is an arithmetic $(+, -, *, /)$ or relational $(=, \neq, <, \leq, >, \geq)$ operator then $\mathcal{AC}(p, n) := \mathcal{AC}(p', n)$

(a) Main definition



(b) $\mathcal{OC}$ conditions

Fig. 3: Path activation condition (adapted from [7])

model during 2 cycles. The concrete input flows for the Set, the Reset and the initialization of the unit delay, determine the values of the other flows. The right side of the figure shows the complete paths activated at each cycle, *i.e.*, the paths connecting an input to the output. No path of the form $< \alpha_{init}, \ldots >$ is activated: for this concrete execution, the initialization value has no effect on the output of the latch. If we want to understand the output at cycle 1 or 2, we may safely ignore this initialization value and focus our analysis on data propagation along the active paths. Now, we may wonder whether all of the active paths are necessary to cause the observed output. For example, there are 4 active paths at cycle 2, but the subset p_1, p_2 is sufficient to explain the output at this cycle: intuitively, if there is a Set and no Reset at the current cycle, the latch must output a 1. Another interesting subset is p'_1, p'_2, p_2: if we had previously a Set, and no Reset since that time, we must also have output 1. Hence, different subsets of paths may be extracted to independently explain a final output value.

This is exactly what STANCE does. The tool identifies subsets of paths active at the last cycle n of the counterexample, and connecting current and past inputs to the falsified sink property *P(n)*. A subset $\mathcal{C}$ is a cause of the violation if the input values at the origin of the paths are sufficient to:

- Ensure that all paths in $\mathcal{C}$ are active,
- Ensure that *P(n) = 0*.

Our previous work formalized these sufficient conditions [4]. The formalization refers to variants of the counterexample, that keep the input values determined by the cause but let other inputs receive arbitrary values. Whatever the arbitrary values, the values kept in the cause allow us to control the violation of the property via the identified propagation paths.

B. Analysis of the running example by STANCE

Let us now illustrate the approach on the running example of Figure 1. The considered MUV is a case study described in [8]. Its *active* input indicates a temporal interval in which an *action* is expected, *e.g.* if *active* takes the successive values 0110 then *action* is expected at cycles 2 or 3. If the action never occurred during these cycles, MUV shall issue an *alarm*. We model this property by having a latch store the occurrence of the action: it is set by *action AND active*, it is reset by the falling edge of *active* when the deadline for *action* has passed. The implication operator expresses that the *alarm* is required from MUV whenever the end of an *active* interval is detected, and no corresponding *action* occurrence was stored before this end event.

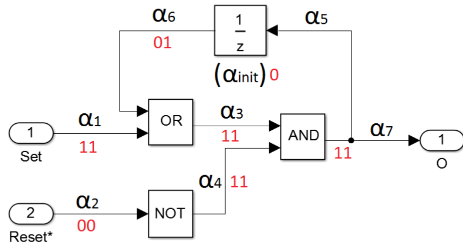
The internal design of MUV is shown in Figure 5b (the coloring is added by our STANCE tool, and will be explained shortly). From [8], we know that this design is flawed. Indeed, when MUV is model-checked, a counterexample is returned. STANCE replays the counterexample and stores the intermediate values of the inputs and outputs of all operators at all cycles. It then performs a structural analysis of the paths activated at the last cycle of the counterexample. It finds two causes. As they are only slightly different, we just present one of them, the one visualized in Figure 5.

STANCE provides two complementary visual results:

- A list of key events of the cause, including the retained input values and some intermediate values that are important to understand the behavior of the traversed operators;
- A colored view of the paths of the cause.

The list of key events (Figure 5a) has separate entries for property and MUV events. Similarly, there are separate colored views of the paths via the property observer (not shown in Figure 5) and via MUV's internal structure (Figure 5b). The counterexample of Figure 5 has only two cycles. In the list of the property events, the retained inputs are *active(1)=1, active(2)=0, action(1)=0*: we have an *active* interval of one cycle (at cycle 1) with no *action*. Still, MUV does not raise an alarm at cycle 2. Other intermediate property events concern the behavior of the latch (is not set at cycle 1), the pulse (detects the end of the active interval at cycle 2), and the implication (falsified at cycle 2).

In order to understand why MUV does not raise an alarm, we have to look at MUV events and use the visual support of the colored paths inside MUV. It is striking that the value of the alarm at cycle 2 does not depend on *action*: the alarm is 0 whether or not there has been an action! The alarm also



Active paths at cycle 1: p_1, p_2.
Active paths at cycle 2: p_1, p'_1, p_2, p'_2.
Where: $p_1 =< \alpha_1, \alpha_3, \alpha_7 >$ from *Set(n)* to *O(n)*
$p'_1 =< \alpha_1, \alpha_3, \alpha_5, \alpha_6, \alpha_3, \alpha_7 >$ from *Set(n-1)* to *O(n)*
$p_2 =< \alpha_2, \alpha_4, \alpha_7 >$ from *Reset(n)* to *O(n)*
$p'_2 =< \alpha_2, \alpha_4, \alpha_5, \alpha_6, \alpha_3, \alpha_7 >$ from *Reset(n-1)* to *O(n)*

Fig. 4: Active input/output paths for a given scenario

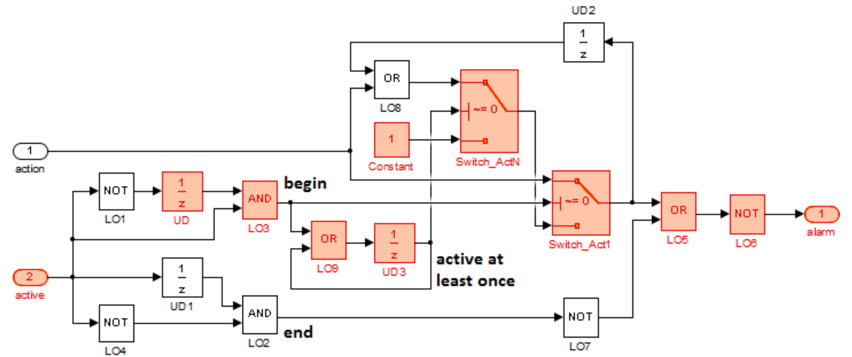
Property Key Events

Bloc Names	Time	Value	Comment
Implies3	2	0	B went false when A was true
MUV/alarm	2	0	MUV output on port alarm
Pulse1	2	1	Falling edge
RS_Action	1	0	Initial value 0 and no Set
action	1	0	Input: 0
active	1	1	Input: 1
active	2	0	Input: 0

MUV Key Events

Bloc Names	Time	Value	Comment
MUV/alarm	2	0	MUV output on port alarm
MUV/Constant	2	1	Constant value: 1
MUV/Switch_Act1	2	1	Condition 0 and selected input was 1
MUV/Switch_ActN	2	1	Condition 0 and selected input was 1
MUV/UD	1	0	Init value: 0
MUV/UD3	1	0	Init value: 0
MUV/active	2	0	Input: 0

(a) Key events



(b) Colored paths

Fig. 5: Analysis of a counterexample by STANCE: visualization of a cause

does not depend on *active(1)*. So, whatever happens at the first cycle, if *active* is 0 at cycle 2, the alarm is 0 as well. Clearly, there is an initialization problem at cycle 1. The alarm depends on two initial values, the ones of *UD* and *UD3*. In MUV and the property observer, all temporal operators have the initial value 0. These values 0 of *UD* and *UD3* jointly determine the condition of *Switch_ActN* at cycle 2, so that the constant bottom input is selected.

In [8], the diagnosis is that the initial value of *UD* is incorrect. The arc labeled “begin” should carry the value 1 at the beginning of each interval, but the detection of this event does not work at the first cycle. The proposed fix is an initialization of *UD* to 1. Note that an alternative fix could keep the initial value 0 but swap operators *LO1* and *UD*. Any of these simple fixes would remove the counterexample. But the detailed analysis of the paths may also suggest a more radical change in the design. The initialization structure (wrongly) activated by the counterexample seems unnecessarily convoluted. The constant input 1 entering *Switch_ActN* is useful only for the initial cycles before a first “begin” is detected. Once a “begin” occurs, the arc labeled “active at least once” is *true* forever (due to the feedback loop arriving at *LO9*), which permanently deactivates the path from the constant input. One may wonder whether it is necessary to process the initial inactive cycles differently from other inactive cycles. A simplified design could align their processing, removing the need for the extra constant input and for its selection via a specific switch condition controlled by a feedback loop. We will later present the simplified design we came to.

This small case study illustrates how STANCE facilitates the understanding of a counterexample. But the insight is limited to this counterexample. Should there be any other initialization flaws, or any other types of flaw after the initialization transients[1], we would not see them. Calling the model-checker with the current version of the model would only return the same counterexample again and again. The only possibility for seeing other ways to violate the property is to eliminate the first one. This may not be the quickest debugging approach, especially if a major revision of the design is considered. Engineers should have the possibility to get more feedback before the revision.

We propose an automated facility for exploring additional counterexamples, exhibiting new causes of violation. It is based on the structural analysis performed for the causes. Intuitively, its principle is to select structural elements that are not activated by the known causes (*e.g.*, the top input of *Switch_ActN*, or the arc from *LO7* to *LO5*) and to challenge the model-checker to violate the property via paths that traverse these elements.

III. SEARCHING FOR NEW COUNTEREXAMPLES

We present the algorithms we have implemented to search for new counterexamples. The key idea is to synthesize in-

[1]For our running example, this is not purely hypothetical. We know from [8] that such a flaw exists in the design.

```
1   Inputs     : model → considered Simulink model
2   Outputs    : causes → array of causes
3   Variables  : instrumentations → already instrumented arcs
4                newInstrus → new intrumentations
5                newCauses → causes from a counterexample
6                counterexample → returned counterexample
7                iModel → instrumented model
8   Start
9    causes = ∅
10   instrumentations = []
11   counterexample = modelchecking(model)
12   if exists(counterexample)
13    N = length(counterexample)
14    modelExec = execute(model, counterexample)
15    causes= findCauses(modelExec)
16    instrumentations = findInstrumentations(modelExec, N, N
          , ∅)
17    for each a in instrumentations do
18     iModel = performInstrumentation(model, a)
19     counterexample = modelchecking(iModel)
20     if exists(counterexample)
21      modelExec = execute(model, counterexample)
22      newCauses = findCauses(modelExec)
23      newCauses = causesFiltering(newCauses, causes)
24      if newCauses≠ ∅
25       N = length(counterexample)
26       newInstrus = findInstrumentations(modelExec, N, N,
              ∅)
27       newInstrus = instruFiltering(newInstrus,
              instrumentations)
28       instrumentations = append(instrumentations,
              newInstrumentations)
29       causes = causes ∪ newCauses
30      endif
31     endif
32    endfor
33   endif
34  End
```

Fig. 6: Top-level algorithm

strumentations that force the model-checker to consider new paths.

A. Top-Level Algorithm

The top-level algorithm is shown in Figure 6. Its input is the data structure *model*, encoding the graph structure of the global model composed of MUV and its property observer (like the global model in Figure 1). Its output is a set of causes extracted from all counterexamples found by the search.

The initial steps of the search are described in Lines 9-16 of the algorithm. The global model is model-checked, possibly returning a first counterexample data structure. Replaying this counterexample allows us to produce a *modelExec* data structure, that supplements the model graph by attaching data to its arcs, *i.e.*, it allows us to store the values carried by the arcs at each execution cycle. The analysis of *modelExec* returns causes, as explained in Section II. Then, an additional analysis, which is the target of this paper, produces a list of candidate *instrumentations* to force the model-checker to consider new paths to violations. The instrumentations constrain the values carried by some arcs at some cycles.

Once the search process has been initiated by the processing of the first counterexample, the top-level algorithm iterates over the elements of *instrumentations* (Lines 17-32). An iteration creates an instrumented version of the model: *iModel*. If *iModel* allows a counterexample to be found, the counterexample is replayed on the original un-instrumented model (Line 21), to extract its causes (Line 22) and produce further candidate instrumentations (Line 26). Filtering functions are used to eliminate causes already found in other counterexamples (Line 23), as well as instrumentations targeting arcs already covered by previous instrumentations (Line 27). The latter filtering function is critical for ensuring termination of the search. For models including temporal loops, there is potentially an infinite number of paths to violation (iterating *n* times the temporal loop, *n+1* times, etc.). It would be possible to keep producing new causes forever, via counterexamples of increasing length. To avoid endless iteration, we arbitrarily stop the search when the analysis of counterexamples does not allow the instrumentation of new arcs. We now explain the two functions that are at the core of our instrumentation-based approach: *findInstrumentations()* and *performInstrumentation()*.

B. Finding instrumentations

Function *findInstrumentations()* uses concrete execution data in *modelExec* to derive a list of instrumentations forcing the activation of new paths. An instrumentation consists of two fields:

- *primary*: a set of local constraints for the values carried by the arcs targeted by the instrumentation;
- *secondary*: a set of additional constraints to ensure the propagation of these values and the falsification of the property.

The primary and secondary constraints are encoded by means of three constructors:

- *basic(arcId, value, delay)*: the arc denoted by *arcId* must carry the given value at the cycle determined by *delay*; *delay* is a null or negative number relative to the cycle of the violation, *e.g.*, delay = -1 means one cycle before the violation.
- *atLeastOnce(arcId, value, delay)*: at the cycle determined by *delay*, the arc denoted by *arcId* must have carried the given value at least once.
- *always(arcId, value, delay)*: the arc denoted by *arcId* has always carried the given value until the cycle determined by delay.

Note the relative notion of time introduced by the delay in each case. We do not prescribe a precise length for the new counterexample. The model-checker can build any counterexample, provided its number of cycles N is such that $N>delay$ and the defined constraint holds at cycle $N-delay$.

To produce the instrumentations, function *findInstrumentations()* performs a backward traversal of *modelExec*, both structural (from the sink property to the inputs) and temporal (from the cycle of the violation to previous execution cycles). Each step considers the time-stamped value of an arc that is part of an active violation path. So, the first step considers the arc entering the sink property operator and its value 0 at the last cycle N of the counterexample. All paths of the causes end with this valued arc, which expresses the violation of the property. Function *findInstrumentations()* identifies the operator at the origin of the arc, *i.e.*, the operator that outputs the value. The operator introduces specific activation conditions for paths that traverse it. As a result, at the level of the operator, some local inputs are currently propagated to produce the output, while

```
1   Inputs     : switch → current switch node
2                N → number of cycle of the counterexample
3                k → current time−stamp
4                secondary → secondary instrumentations
5   Outputs    : instrumentations → found instrumentations

7   Start
8    if switch.condition.value == 1
9     sel_data = switch.dataUp
10    unsel_data = switch.dataDown
11   else
12    sel_data = switch.dataDown
13    unsel_data = switch.dataUp
14   endif
15   instru.primary = basic(switch.condition, not(switch.
          condition.value), −N+k) ∪ basic(unsel_data,
          sel_data.value, −N+k)
16   instru.secondary = secondary
17   condInstru = findInstrumentations(switch.condition, N, k
          , secondary ∪ basic(sel_data, sel_data.value, −N+k)
          )
18   dataInstru = findInstrumentations(sel_data, N, k,
          secondary ∪ basic(switch.condition, switch.
          condition.value, −N+k))
19   instrumentations = append(instru, condInstru, dataInstru
          )
20  End
```

Fig. 7: Specialized function for the Switch operator

some are not. The function searches for alternative ways to obtain the target output via the currently ignored inputs. If it finds some, the corresponding instrumentation constraints are created. The function then selects the currently propagated inputs to launch their processing at the next steps. We thus have recursive calls following the active paths of the causes.

To illustrate the approach, let us assume that at some point, we have a recursive call of the form *findInstrumentations(arcId, N, k, secondary) i.e.*, we are currently processing the value carried by the arc *arcId* at cycle *k*. The total number of cycles of the counterexample is *N*. Parameter *secondary* contains a set of constraints accumulated during the traversal from the root property arc to this arc. It represents a sufficient condition for the value of the arc at cycle *k* to control the falsification of the property at cycle *N* (the building of such a sufficient condition is at the core of our analysis for cause extraction). Function *findInstrumentations()* first identifies the operator at the origin of the arc, say, a *Switch* operator. It then delegates the processing to a specialized function shown in Figure 7. Let us assume that the Switch condition is 1 at cycle *k*. The active paths are thus via the condition and the top input, while the bottom input is ignored. An instrumentation is produced, which forces the output to come from the bottom input. The *primary* constraints are that the condition is 0 and the bottom input has the value V currently carried by the top input. The *secondary* constraints are left unchanged. They ensure that the V output of the switch yields the falsification of the property. So, the conjunction of the primary and secondary constraints instructs the model-checker how to violate the property via new paths.

Finally, we also search for new paths via the top input and the condition of the Switch. Recursive calls to *findInstrumentations()* allow us to find alternative ways to produce the same values V and 1 as in the current counterexample. The recursive calls use an updated version of *secondary*. For example, the call exploring the top input has a constraint on its selection by the switch condition (Figure 7, Line 18).

```
1   Inputs     : latch → current latch node
2                N → number of cycle of the counterexample
3                k → current time−stamp
4                secondary → secondary instrumentations
5   Outputs    : instrumentations → found instrumentations

7   Start
8    instrumentations = []
9    if output(time) == 0
10    if exists(latch.reset == 1 at time t)
11     if initValue == 0
12      instru.primary = always(latch.reset, 0, −N+k)
13      instru.secondary = secondary ∪ basic(latch.output, 0,
             −N+k)
14      secondary = secondary ∪ basic(latch.output, 0, −N+k)
             ∪ ⋃_{t≤i≤k} basic(set,0,−N+i))
15      instrumentations = append(instru,
             findInstrumentations(latch.reset, N, t, secondary
             )
16     endif
17    else
18     instru.primary = atLeastOnce(latch.reset, 1, −N+k)
19     instru.secondary = secondary ∪ basic(latch.output, 0,
           −N+k)
20     instrumentations = instru
21    end
22   else
23    if exists(latch.set == 1 at time t)
24     if initValue == 1
25      instru.primary = always(latch.set, 0, −N+k)
26      instru.secondary = secondary ∪ basic(latch.output, 1,
             −N+k)
27      secondary = secondary ∪ basic(latch.output, 1, −N+k)
             ∪ ⋃_{t+1≤i≤k} basic(latch.reset,0,−N+i)
28      instrumentations = append(instru,
             findInstrumentations(latch.set, N, t, secondary))
29     endif
30    else
31     instru.primary = atLeastOnce(latch.set, 1, −N+k)
32     instru.secondary = secondary ∪ basic(latch.output, 1,
           −N+k)
33     instrumentations = instru
34    endif
35   endif
36  End
```

Fig. 8: Specialized function for the RS-latch operator

The specialized functions for Boolean operators like *Switch* produce only basic instrumentations. When it comes to complex temporal operators, more elaborated constraints need to be expressed. For example, Figure 8 presents the algorithm specialized for an RS-latch, producing constraints of the form *always* and *atLeastOnce*. In Line 12, the *always* constraint is produced in the case where the latch output is 0 due to a reset at some previous cycle, and we want to explore whether it is possible to obtain the same result without any reset. So, the reset input is forced to be always 0, in order to activate new paths via the initialization value of the latch. Line 18 uses *atLeastOnce* in the dual case: the output is currently due to the initialization value, and we want to force an active reset.

Note that the design of the specialized functions is a matter of compromise, in order to keep a reasonable number of instrumentations. In the RS-latch example, if the current output is 0 due to a reset at cycle *t*, we do not explore alternative paths via a reset at another cycle $t' \leq k$, nor do we explore the many ways the *Set* input can be maintained to 0 between cycles *t+1*

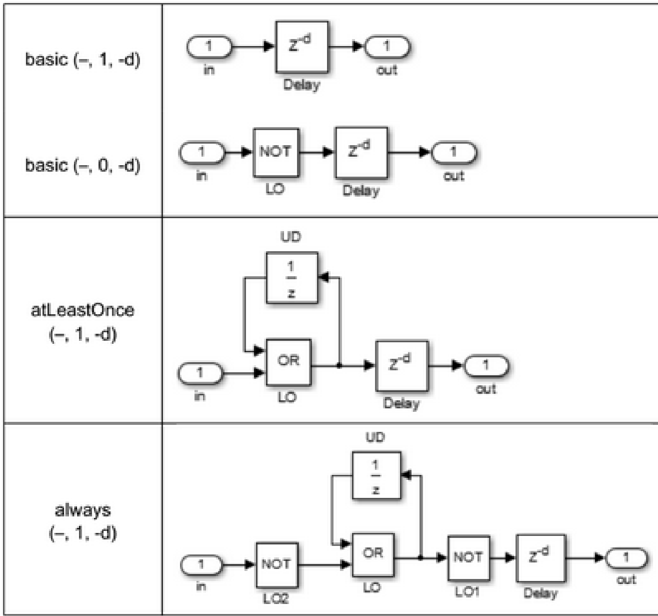


Fig. 9: Simulink blocks implementing the constraints

and k. Anyway, the number of potential causes is infinite, so we are left with the choice of coverage criteria at the level of operators.

C. Instrumenting the model

Once candidate instrumentations have been found, we have to effectively perform these instrumentations on the model. This is the role of function *performInstrumentation()* mentioned in the top-level algorithm. Given a candidate instrumentation I, the function automatically modifies property P into $C \Rightarrow P$, where C is the conjunction of all individual constraints coming from *I.primary* and *I.secondary*. In this way, the model-checker is forced to search for counterexamples that satisfy the constraints while falsifying P.

The individual constraints are inserted by means of dedicated Simulink blocks, shown in Figure 9. For example, a basic constraint is realized by sending the target data to the input port of the basic block. All delays are initialized to 0. In this way, the output of a block is 0 during the first d cycles, yielding the conjunction C of the constraints to be 0 as well. No counterexample of less than $d+1$ cycles can be returned.

IV. Application to the running example

Applied to the running example, the automated search gives the following results. The first counterexample found is the one commented in Section II, having two causes. It produces 6 instrumentations summarized in Figure 10. Some of them come from the exploration of the property observer, while others come from paths traversing MUV. For example, the first instrumentation comes from the property observer. It was decided when analyzing the output of its RS_latch at cycle 1 (i.e., at relative time -1, since the violation is at cycle 2). The counterexample had this output due to the initial value of the latch. The instrumentation forces the output via the activation of the Reset. While iterating over the instrumentations:

```
1. (Property) RS_Action: has output 0 at relative time -1
   due to at least one Reset
2. (Property) LO3: input active is 0 at relative time -1
3. (MUV) LO5: input coming from LO7 is 1 at time 0
4. (MUV) Switch_Act1: condition is 1 and selected top
   input (action) is 1 at relative time 0
5. (MUV) Switch_ActN: condition is 1 and selected top
   input is 1 at relative time 0
6. (MUV) LO3: input active is 0 at relative time -1
```

Fig. 10: Instrumentations produced by analyzing the first counterexample

- Instrumentation 1 finds a new counterexample, having a new cause. The instrumentations produced from it are eliminated by the filtering function (their primary constraints do not cover new arcs).
- Instrumentations 2-4 return no counterexample.
- Instrumentation 5 finds a counterexample but it is eliminated by the filtering of causes (it has the same cause as the new counterexample from Instrumentation 1).
- Instrumentation 6 returns no counterexample.

So, the search terminates with three causes of violation, two from the initial counterexample, and one from the new one.

The newly found problem is the second flaw reported in [8]. It is quite different from the initialization problem described in Section II. The alarm is not raised if the action comes exactly one cycle too late, i.e., when the activity interval has just ended. In [8], the proposed fix is to introduce a delay between *Switch_Act1* and the subsequent OR operator.

We reworked the model based on the insights gained from the causes of the two counterexamples. The result, shown in Figure 11, is simpler than the original design. Model checking proves that it fulfills the property modeled in Figure 1. It fixes the detection of the beginning of an activity interval at the first cycle. It homogenizes the processing of initial and subsequent inactivity intervals. It takes care that an action occurring at the first inactive cycle is too late for the previous activity interval. The major revision of the design was greatly facilitated by our ability to gather information on different flaws, as well as by the visualization of the key violation paths and events in each case.

V. Case Study: a Flasher Manager

A. Considered Model

This model, taken from [9], represents a *flasher manager* manufactured by Geensoft/Dasssault Systems and is responsible for driving the behavior of the flashing lights of a car. The Simulink model of this device is pictured in Figure 12. Its features are the following.

a) Direction change: The driver can indicate a direction change through two Boolean inputs L (for left) and R (for right). The corresponding light (respectively *outL* and *outR*) oscillates between on and off states with a 6 time-units period, thus generating a loop of the sequence [111000] as long as the input is true. When the input falls back to false, the lights stop flashing immediately.

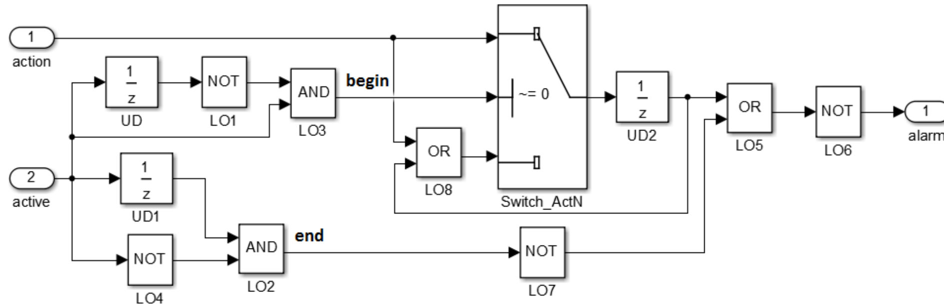


Fig. 11: Revised design of the running example

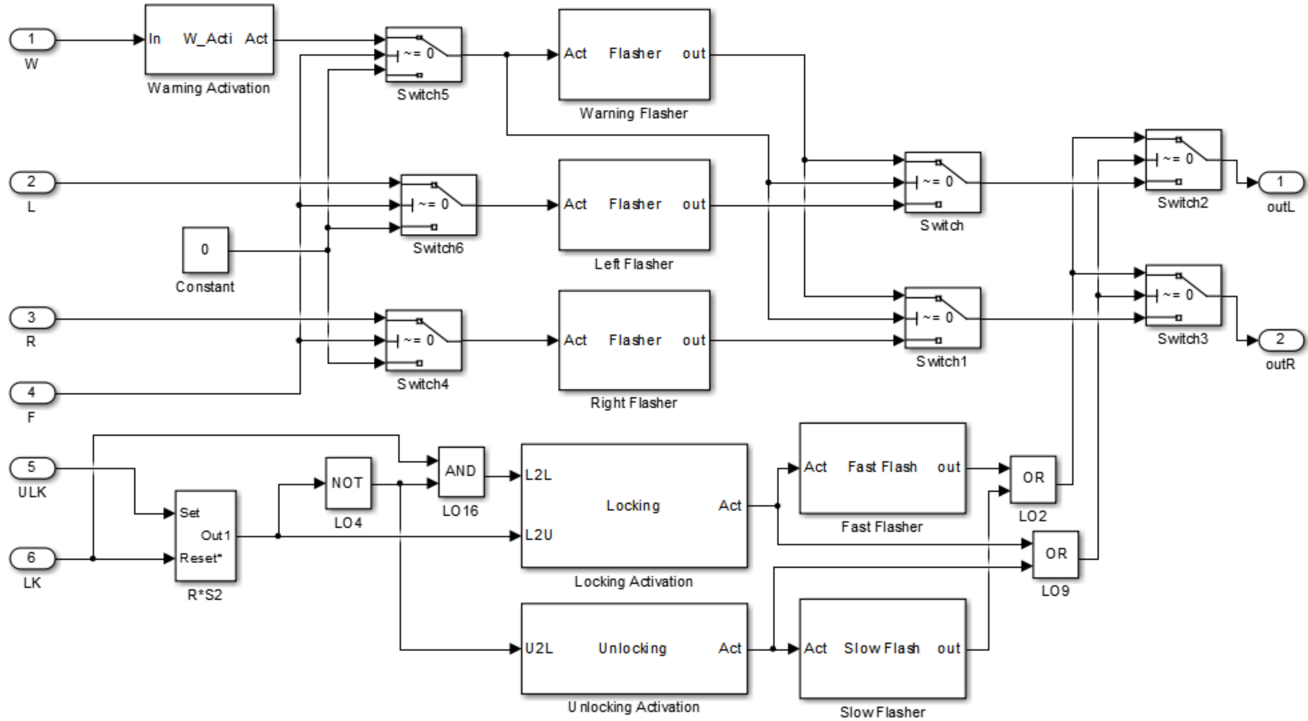


Fig. 12: Simulink model of the Flasher Manager

b) Warning: When the warning is on, both lights flash with a 6 time-units period as for a direction change (producing the sequence [111000]). The *W* input is a toggle button, meaning that only rising edges can activate/deactivate the warning function. A first rising edge activates the warning and a second one deactivates it. This function has priority over the regular flashing functions.

c) Locking and unlocking: It is also possible to lock and unlock the doors using a 2-buttons RF-key: the *locking* button (LK) and the *unlocking* button (ULK). The behavior of the flasher w.r.t both buttons is dependent on the current state of the car (locked or unlocked). As an example, if one presses LK while the car is not already locked, then the lights shall be lit for 10 time-units and then be off for the next 10 time-units, thus producing the sequence [11111111110000000000]. In the initial state, the doors are locked.

Furthermore, the flasher manager has two assumptions on the inputs. The first is that the L and R inputs cannot be triggered at the same time and the second states that the inputs LK, ULK and W inputs are push buttons, meaning the trigger lasts only 1 cycle.

In [9], the authors considered 4 properties of the flasher manager component. For this paper, we will consider the fourth one. This property states that "the lights should never remain lit infinitely". Being a liveness property, it can not be checked using Simulink's model-checker. We will consider a bounded version of the property, by setting a maximum number of cycles during which the lights shall not remain lit. In [9], the authors checked the property for bounds of 10, 50, 100 and more (up to 1,600 cycles). They found one counterexample for every bound, but did not discuss the underlying behavior (their focus was on benchmarking model-checking algorithms). So, we know for sure that we will have counterexamples to analyze and we will determine why the property fails. Furthermore, as we are working at a debugging level, we will start with a

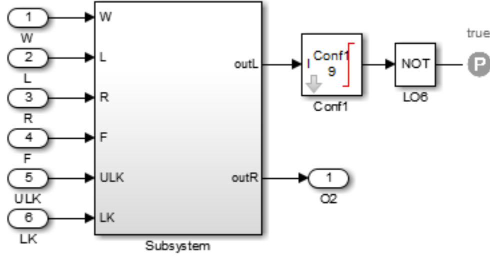


Fig. 13: Flasher Manager MUV with it property

bound of 10. Doing so allows us shorter model-checking time and potentially shorter causes.

Figure 13 shows the diagram of the property. It consists of a single *Confirmator* operator, that outputs 1 only if there has been *n+1* consecutive 1 on its input, where *n* is a parameter (nine in our case, meaning that it detects ten consecutive 1 on its input). For readability purposes, the modeling of the assumptions on the environment is not shown. Also, for simplification, we only check the behavior of the left light, the behavior of the right light being exactly the same.

B. Results

The tool gave us 9 causes each pertaining to a different counterexample. However, after analysis, we managed to categorize those counterexamples into three equivalent classes.

1) First counterexample: The model-checker found a counterexample where all inputs have been activated at least once (except for R, since we are only checking the left light). However, the colored model gives us a huge hint about what is really happening. Indeed, only the paths containing the *Unlocking Activation* and *Slow Flasher* components are involved. This first impression allows us to consider the F, L and W activation as noise, they are thus not relevant for our diagnosis. The associated key events give us even more clues: the falling edge of ULK at time 2 and the rising edge of LK at the same time. The key events also confirm us that F, L and W are not relevant for this counterexample.

What happened is that the activation of ULK unlocked the car and the following LK input locked it back. Now, looking at the behavior of LK in the specification, locking the car when it is in an unlocked state triggers a [11111111100000000000] sequence, hence the violation. This counterexample is not really interesting as it is an artifact of our choice of 10 cycles for checking the property. Fortunately, the next counterexamples are more enlightening.

2) Second counterexample: The colored model of the counterexample shows a larger number of active areas in the model compared to the previous one. This time, in addition to the ULK and LK activation blocks (*Unlocking Activation* and *Locking Activation*, respectively), the areas for W and L are also colored, making them candidates for the responsibility of the violation.

By analyzing the cause, we can determine that the user can lit the light forever, by repeatedly acting on the warning button and on the left direction change. More precisely, in the counterexample, the user activates the W input at cycle 2 (lighting the left light during the cycles 2,3 and 4). Then, at cycle 5, the warning is deactivated and the L input is activated, lighting the lights for another 3 time units (cycles 5, 6 and 7). When the left flasher ends, the W input is activated again keeping the lights on during cycles 8, 9 and 10. Finaly, the L input is activated again to have the light up during cycle 11. At the end, the lights have been on for 10 cycles (from 2 to 11).

The fifth, seventh and eighth counterexamples are variants of this counterexample: user inputs still alternate between warning and left flasher but vary as regards the time to deactivate the warning.

3) Third counterexample: This counterexample is interesting as it is a mix of the first and the second counterexamples. All parts of the model are colored, so we need to rely on the key events to understand what happened.

We see that the activation of W at cycle 2 is responsible for turning the light on during cycles 2 and 3. On cycle 3, the car is unlocked and is locked back on cycle 4. This part reminds of the first counterexample but now exhibits a new case combining the warning and the lock.

The fourth, sixth and ninth counterexamples are variants of this counterexample, with a different number of warning cycles and different relative times for the unlock before the lock.

C. Conclusion

With these counterexamples, we learned that it is possible to keep a light on with the continuous intervention of the user. We did not find any sequence of inputs that allows the system to remain lit by itself without user interaction.

We introduce an assumption constraining the behavior of the user: there must be at least 4 cycles between consecutive user actions. Setting the maximum lighting limit to 20 (instead of 10) gets rid of the first counterexample, and the added assumption allows the model-checker to end successfully, discovering no new counterexamples.

VI. RELATED WORK

Related work addresses the understanding of counterexamples to facilitate debugging. The interactive visualization of counterexamples is addressed in [10] [11]. [12] and [13] combine visualization and analysis of counterexamples. [12] uses the structure of the property to display information on when the property fails, and with which subconditions. [13] uses the structure of both the model and the property, and annotates counter-examples with proofs explaining the model-checking result. In software model-checking, [14] and [15] consider neighboring correct and incorrect executions to localize the error in the code. [16] identifies all counterexamples in one pass and uses this information to identify code segments that are likely to cause the violation. [17] combines model-checking and dynamic analysis of programs. In particular, local coverage information is used to guide the state-space search.

Comparing our work to the above ones, the closest approaches are the ones on software model-checking [14], [15]

[16] [17]. Like them, we analyze the execution of counter-examples in terms of structural locations. However, our work is dedicated to data-flow models. It induces significant differences in how an execution connects to the structure. The execution of an imperative program activates one path, the executed constructs being the ones along the path. In a data-flow model, all constructs are executed at the same time, and a set of temporal propagation paths is activated. So, neighboring executions have to be defined in terms of neighboring sets of paths; temporal propagation conditions (like the ones in our secondary constraints) replace reachability ones.

Outside the model-checking community, structural analysis of data-flow models is performed in testing and debugging work. Ludic [8] 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. Structural testing of Lustre models has been addressed by [18] [7]. Particularly, [7] proposed a definition of path activation conditions in data-flow programs, which we re-used in our work.

Finally, work on concolic testing [5] [6] was also a source of inspiration to us. We retained the principle of analyzing concrete execution data to derive constraints for covering new paths. But again, the framework of our approach (data-flow models) is quite different from the one of imperative programs.

VII. Conclusion and perspectives

In this paper, we presented a path-based approach for the generation of new counterexamples from a model-checker of data-flow models. It is based on the analysis of the causes of the counterexample, *i.e.* of the activated paths in the model that were responsible for property violation. Knowing the causes of a counterexample, the approach targets paths that have not yet been involved in a violation. To this end, we developed a set of instrumentation blocks that are used to constrain the value of certain arcs so that new paths are covered. The process is rerun on the instrumented models to find new counterexamples. This approach is implemented in the STANCE tool.

We first considered an academic case study with two known bugs. The approach allowed us to retrieve them, yielding one counterexample for each. We then applied our approach to an industrial-sized case study taken from [9]. We knew from previous work on this case study that the considered property does not hold, but the authors did not analyze why. Our approach gave 9 different counterexamples that we categorized into three classes by analyzing their causes. With this analysis, we understood what went wrong and proposed the addition of an assumption that makes the property successfully checkable.

So far, our approach has focused on analyzing the Boolean and temporal structure of the models, leaving apart the numeric parts. The extraction of the causes assumes that all inputs of a numeric operator (e.g., an arithmetic operator) are propagated, and numeric arcs are not the targets of any instrumentation. Future work will study constraint-based approaches to refine the analysis with respect to the numeric aspects. In particular, approaches extracting subsets of constraints, like Minimal Unsatisfiable Subsets (MUS) or Minimal Correction Sets (MCS) could be considered [19].

VIII. Acknowledgements

This work is funded in part by the French Space and Aeronautic Sciences and Technologies foundation (STAE).

References

[1] Esterel Technologies, "SCADE Suite product page." http://www.esterel-technologies.com/products/scade-suite/. (2014-10-24).

[2] The Mathworks, Inc, "Matlab/Simulink website." http://www.mathworks.fr/products/simulink/. (2014-10-24).

[3] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels, "Model checking flight control systems: The Airbus experience," in *31st Int. Conf. on Software Engineering, ICSE 2009, Vancouver, Canada*, pp. 18–27, IEEE, 2009.

[4] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels, "Paths to property violation: A structural approach for analyzing counter-examples," in *12th IEEE High Assurance Systems Engineering Symposium, HASE 2010, San Jose, USA*, pp. 74–83, IEEE Computer Society, 2010.

[5] N. Williams, B. Marre, and P. Mouy, "On-the-fly generation of k-path tests for C functions," in *19th IEEE Int. Conf. on Automated Software Engineering (ASE 2004), Linz, Austria*, pp. 290–293, IEEE Computer Society, 2004.

[6] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," in *10th European Software Engineering Conference, Lisbon, Portugal* (M. Wermelinger and H. Gall, eds.), pp. 263–272, ACM, 2005.

[7] A. Lakehal and I. Parissis, "Structural coverage criteria for LUSTRE/SCADE programs," *Softw. Test., Verif. Reliab.*, vol. 19, no. 2, pp. 133–154, 2009.

[8] F. Maraninchi and F. Gaucher, "Step-wise + Algorithmic debugging for Reactive Programs: Ludic, a debugger for Lustre," in *AADEBUG*, 2000.

[9] H. Collavizza, N. L. Vinh, O. Ponsini, M. Rueher, and A. Rollet, "Constraint-based BMC: a backjumping strategy," *STTT*, vol. 16, no. 1, pp. 103–121, 2014.

[10] C. Artho, K. Havelund, and S. Honiden, "Visualization of concurrent program executions," in *31st Annual International Computer Software and Applications Conference, COMPSAC 2007, Beijing, China*, pp. 541–546, IEEE Computer Society, 2007.

[11] H. Aljazzar and S. Leue, "Debugging of dependability models using interactive visualization of counterexamples," in *5th Int. Conf. on the Quantitative Evaluation of Systems (QEST 2008), Saint-Malo, France*, pp. 189–198, IEEE Computer Society, 2008.

[12] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. J. Trefler, "Explaining counterexamples using causality," *Formal Methods in System Design*, vol. 40, no. 1, pp. 20–40, 2012.

[13] M. Chechik and A. Gurfinkel, "A framework for counterexample generation and exploration," *STTT*, vol. 9, no. 5-6, pp. 429–445, 2007.

[14] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *Model Checking Software, 10th Int. SPIN Workshop. Portland, USA* (T. Ball and S. K. Rajamani, eds.), vol. 2648 of *Lecture Notes in Computer Science*, pp. 121–135, Springer, 2003.

[15] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," in *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, USA* (A. Aiken and G. Morrisett, eds.), pp. 97–105, ACM, 2003.

[16] C. W. Keller, D. Saha, S. Basu, and S. A. Smolka, "Focuscheck: A tool for model checking and debugging sequential C programs," in *11th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2005, Edinburgh, UK* (N. Halbwachs and L. D. Zuck, eds.), vol. 3440 of *LNCS*, pp. 563–569, Springer, 2005.

[17] A. Groce and R. Joshi, "Extending model checking with dynamic analysis," in *9th Int. Conf. on Verification, Model Checking, VMCAI 2008, San Francisco, USA* (F. Logozzo, D. Peled, and L. D. Zuck, eds.), vol. 4905 of *LNCS*, pp. 142–156, Springer, 2008.

[18] B. Marre and A. Arnould, "Test sequences generation from LUSTRE descriptions: Gatel," in *ASE*, p. 229, 2000.

[19] M. Bekkouche, H. Collavizza, and M. Rueher, "LocFaults: A new flow-driven and constraint-based error localization approach," in *30th Symposium on Applied Computing, SAC'15, Salamanca, Spain*, 2015.