Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Counter-Example Generation in Symbolic Abstract Model-Checking

Gordon Pace * **, Nicolas Halbwachs**** **, and Pascal Raymond*****

Vérimag†, Grenoble, France

Abstract. The boundaries of model-checking have been extended through the use of abstraction. These techniques are conservative, in the following sense: when the verification succeeds, the verified property is guaranteed to hold; but when it fails, it may result either from the non satisfaction of the property, or from a too rough abstraction. In case of failure, it is, in general, undecidable whether an abstract trace corresponding to a counter-example has any concrete counterparts. For debugging purposes, one usually desires to go further than giving a "yes/no" answer (actually, a "yes/don't know" answer!), and look for such concrete counter-examples. We propose a solution in which we apply standard test-pattern generation technology to search for concrete instances of abstract traces.

* gpace@cmst.org.mt
** Nicolas.Halbwach@imag.fr
*** Pascal.Raymond@imag.fr
† Verimag is a joint laboratory of Université Joseph Fourier (Grenoble I), CNRS and INPG. see www-verimag.imag.fr

1 Introduction

The ultimate goal of program verification techniques is to certify that an implementation satisfies a number of properties or matches a given specification. Recent progress in model-checking techniques has made it possible to apply automatic verification not only to large hardware systems, but also to critical software. The key techniques are *symbolic model-checking* [BCM+90, CMB90] and *abstraction* [CGL94, GL93]. Symbolic techniques allow large state space systems to be dealt with by avoiding state enumeration: sets of states are characterized globally by Boolean formulas, possibly encoded with BDDs, and reachable state traversal is performed through symbolic computations over formulas. Abstraction is especially important for software verification, since it extends automatic verification to infinite state systems, like programs with numerical variables: by simply abstracting away numerical aspects of the program, the model-checker is still able to provide *conservative results* about a wide class of properties, including all safety properties. When such a property is verified on the abstract model, it is sure to hold on the concrete one, but the converse is false: it may be the case that a state violating the property seems to be reachable in the abstract model, while being actually forbidden by the behavior of numerical variables.

In this paper, we address the problem of reporting a meaningful explanation when the verification of a safety property fails. Of course, even in the finite state case, explaining the source of the error to the user is very important for identifying the error, which can lie either in the program or in the specification of the property. But it is especially important when abstraction is used, since one has to decide whether such a failure corresponds to an actual error, or whether it only results from a weakness of the abstraction. In the case of safety properties, the natural explanation consists of a counter-example — a trace leading from an initial state of the system to a state violating the property. Although exhibiting such an erroneous trace is straightforward in enumerative model-checking of finite state systems, it is a bit less trivial in symbolic model checking, and it is generally undecidable in abstract model checking of infinite state systems. In this case, we propose to use a test-case generation tool to allow the user to search for a concrete counter-example, within the apparently erroneous traces identified by the model-checker. This approach has been implemented in the framework of the verification of reactive programs written in the synchronous language Lustre [HCRP91], using the model-checker Lesar [HLR92] and the test-case generation tool Lurette [RWNH98], and experimented on several real examples extracted from the flight control software of the Ariane-V rocket.

2 Verification and Diagnosis with Lesar

2.1 Synchronous machines

We are interested in the verification of synchronous systems, which can be modeled as transition systems with inputs and outputs (generalized Mealy Machines). Such a machine starts from an initial state and performs an infinite loop, by selecting a transition from its current state according to the current values of its inputs. Performing a transition consists in assigning the corresponding values to the outputs and moving to the target state of the transition, i.e., assigning the corresponding values to state variables. Since input, output, and state variables can be either Boolean or of infinite types, the set of states can be infinite. Such a machine can be described by giving an initial state (or a formula characterizing the set of initial states), and a set $\{t_k | k = 1..n\}$ of transitions, each of which being a guarded command of the form:

$$t_k : g_k(s,i) \rightarrow o := \omega_k(s,i); s := \sigma_k(s,i)$$

where $g_k(s,i)$ is a condition on the current state s and the current input i, ω_k and σ_k are functions, respectively giving the values of outputs and next state. We consider deterministic systems, hence ω_k and σ_k are functions. Moreover, these systems are supposed to always accept inputs, so these functions will also be total. For the same reasons, the guards are assumed to be pairwise exclusive, and their disjunction $\bigvee_k g_k$ is assumed to be true.

Note that, in general, such a "program" results from the compilation of a higher level language. As an important consequence, *state variables are generally meaningless for the user*, and should not be used in the diagnosis. They result from the encoding of internal processes control points and internal memories.

The more conventional transition relation can be defined from this guarded transition formulation as follows:

$$s \xrightarrow{i} s' \stackrel{df}{=} \exists k \cdot g_k(s,i) \wedge s' = \sigma_k(s,i)$$

2.2 Model checking synchronous Programs with Lesar

A standard way of specifying a safety property is by giving an automaton recognizing the erroneous traces. In synchronous programming, such an automaton may be given as a program (i.e., a machine as before), called a *synchronous observer*, receiving as inputs the inputs and outputs of the program to be verified, and computing a single Boolean output variable, which is true as long as the property is satisfied. Moreover, since the considered programs interact with an environment, we are generally interested in proving properties under some known *assertion* about the behavior of the environment. An assertion is also a safety property, which can be described by an observer which detects whenever the environment behaves unrealistically. The general verification problem consists in considering the program and the two observers, connected as shown by Fig. 1, and to check that, whatever the sequence of inputs, either the output "*ok*" of the property observer is always true, or the output "*realistic*" of the assertion observer becomes at some point false.

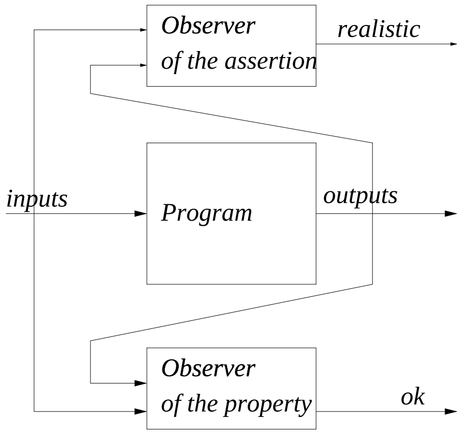


Fig. 1. Verification program

Lesar takes such a compound program (in Lustre), and applies standard symbolic model-checking techniques, with abstraction of numerical variables, to perform the verification. The abstraction consists in ignoring all operations on numerical variables, and in considering conditions involving numerical variables as additional Boolean inputs. If, in this abstract model, the output "*ok*" can go low, with the output "*realistic*" always high, Lesar will find a sequence of (symbolic) states which leads to a bug. The sequence is guaranteed, by virtue of its construction, to be the shortest possible. More precisely, the model-checking procedure starts from the formula F_1

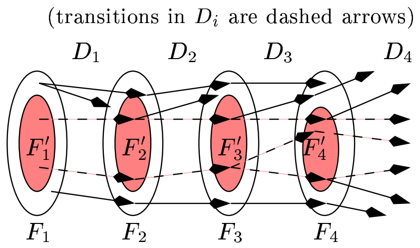


Fig. 2. Diagnosis construction

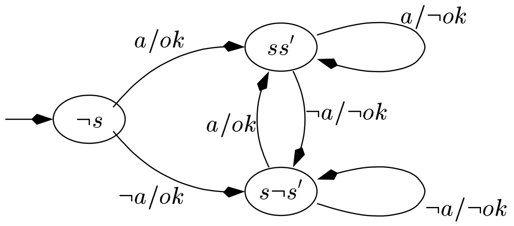


Fig. 3. Automaton of example 1

characterizing the initial states of the program, and computes forward a sequence of formulas $F_2, F_3, \ldots$, where $F_{j+1} = Post(F_j)$, and where $Post(F)$ is the characteristic formula of states directly reachable from states satisfying F by transitions satisfying the assertion:

$$Post(F) \stackrel{df}{=} \lambda s' \cdot \exists s, i \cdot realistic(s,i) \wedge F(s) \wedge s \xrightarrow{i} s'$$

If, at some step n, F_n allows some transitions violating the property, an error is found. Then the diagnosis is built backward (see Fig 2): let D_n be the characteristic formula of the erroneous transitions starting from states in F_n:

$$D_n(s,i) = F_n(s) \wedge realistic(s,i) \wedge \neg ok(s,i)$$

and let F'_n be the characteristic formula of the source states of transitions in D_n ($F'_n = \exists i, D_n$). Lesar computes backward the sequence of formulas $D_{n-1}, D_{n-2}, \ldots, D_1$, where each D_j characterizes the transitions — i.e., the pairs (state, inputs) — which belong to an erroneous trace:

$$D_j(s,i) = F_j(s) \wedge realistic(s,i) \wedge \exists s' \in F'_{i+1}, s \xrightarrow{i} s'$$
$$F'_j = \exists i, D_j$$

2.3 Diagnosis Without Abstraction

As a first step, we will look at problems arising within systems with no abstractions applied — programs using only Boolean data.

Consider the following simple program (with observers inside — *ok* being the output of the observer, and *real* being that of the assertion about the environment) which checks that, apart from the initial step, the input a is true and its previous value is false (a property clearly not satified, since it is unsatisfiable!):

Example 1:

init $s = s' = false$;

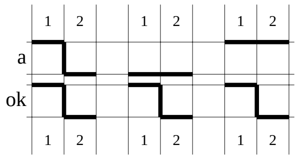
transitions

$\neg s \;\rightarrow\; real := true; ok := true; s := true; s' := a;$

$s \wedge s' \;\rightarrow\; real := true; ok := false; s := true; s' := a;$

$s \wedge \neg s' \;\rightarrow\; real := true; ok := a; s := true; s' := a;$

Fig. 3 shows the corresponding interpreted automaton. Clearly, the shortest counter-examples are of length 2 — the ones shown in the following diagram:



As explained above, Lesar builds first the sequence of state formulas:

$$F_1 = \neg s \wedge \neg s' \qquad F_2 = s$$

It finds that in F_2, ok is false when $s' \vee \neg a$, and computes backward the sequence:

$$D_2 = s \wedge (s' \vee \neg a) \qquad D_1 = \neg s \wedge \neg s'$$

This sequence perfectly describes the set of all the shortest executions leading to the violation of the property. Now, the formulas D_j involve state variables (s, s'), which are meaningless for the user. To solve this problem, one might try to get rid of state variables by existential quantification in the formulas. But, in our example, it would produce $D_1' = true$, $D_2' = true$ which gives no information at all.

2.4 Diagnosis of Programs With Abstraction

When the program being verified uses variables ranging over infinite types, Lesar reduces the state space to a finite one by removing these variables and taking the Boolean conditions dependent on these variables as inputs of the system. These conditions are controlled by an unconstrained environment, meaning that their behavior need not be according to the original program.

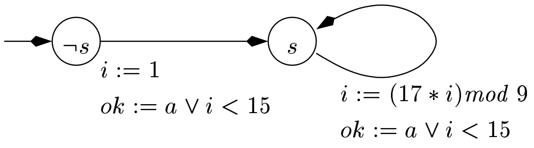


Fig. 4. Automaton of example 2

Consider the following example, pictured by Fig. 4, where a is again an input :

Example 2:

init $\neg s$;

transitions

$\neg s \rightarrow i := 1; ok := a \vee (i < 15); s := true;$

$s \rightarrow i := (17 * i) mod\ 9; ok = a \vee (i < 15); s := true;$

It should be (intuitively) clear that the condition $i < 15$ is always satisfied, and thus ok is always true. However, the abstraction performed on the program only gives: $true \rightarrow ok = a \vee cond$; where $cond$ is an auxiliary variable introduced to stand for the numerical condition $i < 15$.

Upon analyzing this, the model-checker would point out a counter-example of length 1, with inputs a and $cond$ both having value false. This counter model clearly has no counterpart in the concrete program behavior. It is thus desirable to help the user to try and map back the behavior to the concrete model, thus verifying whether or not it is a real counter-example. However, this is to be done with caution, since the non-existence of a concrete instance of the counter-example does not mean that the program is correct (e.g., if the constant 15 in the above example is modified to 5, the same counter-example would be given, which still has no counterpart in the concrete node. However, *ok* can become false on the second step).

3 The Proposed Solution

3.1 A First Solution

If we consider only unabstracted programs, producing counter traces from the internal symbolic representation is not very difficult. If the shortest counter-example is of length n, the model checker returns a symbolic counter-example as n formulas $D_1, \ldots, D_n$ over state and input variables. The selection of an explicit counter-example[1] $i_1, \ldots, i_n$ can be done by the following procedure:

s := init;

for j:= 1 to n do

$i_j = \mathit{Choose}(D_j, s)$; $s := \sigma(s, i_j)$;

where σ stands for the function σ_k such that $g_k(s, i_j)$ holds, and $\mathit{Choose}(D, s)$ is any procedure returning a variable assignment i such that (s, i) satisfies the formula D.

The main problem is the implementation of *Choose*. In the Boolean case, and in particular when D is encoded by a BDD, its implementation is straightforward and simple. However, it is not easily extended to deal with abstractions. Furthermore, one can imagine various strategies to choose assignments (interactive, one random shortest counter-example, enumerate all shortest counter-examples, etc). It would thus make more sense to generate enough information through the model checker to be used by another tool. The information we need is the sequence $D_1, \ldots, D_n$.

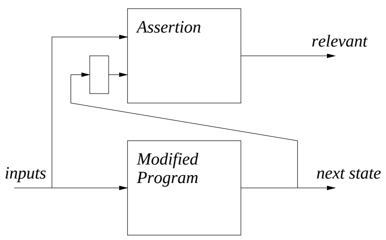
[1] Since we talk only about deterministic automata, a counter-example can be readily represented as a sequence of inputs.

3.2 Data Sequence Selection Under Constraints: The Testing Tool Lurette

The sequence $D_1, \ldots, D_n$ of formulas can be viewed as a sequence of constraints (on state and input variables) which must be successively satisfied by a valid counter-example. The general problem is then the generation of sequences under such constraints. This is precisely what can be done using the test generation tool Lurette [RWNH98, HR99]: given a program and two observers, just as in Fig. 1, Lurette randomly generates sequences of input data of a chosen length, such that the assertion observer always returns *true*, while checking that the property observer is satisfied. The principle is as follows: from the current state of the assertion observer, Lurette deduces a constraint to be satisfied by the current inputs. When this constraint only involves Boolean variables and *linear restraints* about numerical variables, Lurette can select — randomly, or according to various heuristics — a solution to the constraints. The selected data is then given to the program for a single reaction; the returned outputs are used to check the current value of the property, and to perform one transition of the assertion observer, thus computing its new state for the next step.

3.3 Using Lurette to Generate Counter-Examples

Recall that the formulas D_j involve input and state variables, and that state variables are internal to the program. In order to use these formulas in the assertion given to Lurette, the state variables must be output by the program, which requires a straightforward modification of the program. Lurette can be applied to the program shown below.



Note that we have no observer to check for the correctness of the trace, since, by virtue of the way in which the assertion is constructed, only counter-examples can be produced. Whenever a trace of the required length is generated, it is a counter-example.

The assertion itself only selects the current formula to be satisfied, according to the current step:

$init\ step = 1;$

$transitions$

$step = 1 \rightarrow relevant := D_1; step := 2;$

$step = 2 \rightarrow relevant := D_2; step := 3;$

...

$step = n \rightarrow relevant := D_n;$

3.4 Adding Variables of Infinite Types

If the program has variables of an infinite type, Lesar first abstracts these by using Boolean input conditions. Let us come back to the program *Example 2*:

Example 2:

$init\ \neg s;$

$transitions$

$\neg s \rightarrow i := 1; ok := a \vee (i < 15); s := true;$

$s \rightarrow i := (17 * i) mod\ 9; ok = a \vee (i < 15); s := true;$

As explained before, Lesar abstracts it into "$true \rightarrow ok = a \vee cond;$" and finds a counter-example of length 1: $D_1 = (\neg a \wedge \neg cond)$. Note that the value of the abstracted condition $cond$ is determined by the constructed counter-example, despite the fact that it is not a real input of the concrete program. In our framework, Lurette would be required to try and find a solution to both the input constraints and the condition constraints. However, the definitions of the abstract condition constraint and the variables on which it depends are known, and can therefore be placed inside the assertion observer[2]:

$$true \rightarrow relevant := (\neg a \wedge \neg cond \wedge (cond = (i < 15)) \wedge ((\neg s \wedge i = 1) \vee (s \wedge i = (17 * i) mod\ 9)))$$

[2] Here, since the shortest counter-example is of length 1, satisfying "*relevant*" immediately falsifies "*ok*" in the initial program. But remember that, more generally, "*relevant*" is first assigned D_1, which contains all the first transitions which may start an execution leading to "$ok = false$"

3.5 An Overview

To summarize the procedure for debugging:

1. Once the model-checker finds a counter-example, it must create a special diagnosis file which contains:
   - A description of the transformed program, which outputs the values of the next state.
   - An assertion forcing the inputs and state variables to stay in the abstract counter-example.

   It also reports the length of the shortest counter-example.
2. Lurette is run on the diagnosis file generated by Lesar, to generate test vectors of the required length.

4 Applications of the Tool

4.1 A Trivial Boolean Example

Consider once again our *Example 1*:

Example 1:

$init\ s = s' = false;$

$transitions$

$$\neg s \rightarrow real := true;\ ok := true;\ s := true;\ s' := a;$$

$$s \wedge s' \rightarrow real := true;\ ok := false;\ s := true;\ s' := a;$$

$$s \wedge \neg s' \rightarrow real := true;\ ok := a;\ s := true;\ s' := a;$$

Running the program through Lesar, we are told that there are counter-examples of length 2. Remember that the symbolic counter-example is $D_1 = \neg s \wedge \neg s'$, $D_2 = s \wedge (s' \vee a)$. A new file is generated with the main node modified to output state variables, and the following assertion, to allow only the generation of counter examples:

$init\ step := 1;$

$transitions$

$$step = 1 \rightarrow relevant := true;\ step := 2;$$

$$step = 2 \rightarrow relevant = s \wedge (s' \vee a);$$

From this information, Lurette immediately finds the three input traces listed in Section 2.3.

4.2 A Numerical Example

To illustrate the approach with a slightly bigger example, we apply this technique to a small train specification from [BCDPV99].

Consider a system consisting of two trains moving along the same tracks in the same direction. We would like to control the second train, making sure that we do not run into the train in front of us. We realistically assume that we cannot control the current distance between trains or the velocity of our train directly, but can only do so indirectly by changing our acceleration. The model is parameterized by three constants: the initial distance between the trains *init_d*, the maximum acceleration that the trains can achieve *a_max*, and the braking acceleration *brake*.

The system receives two inputs: the acceleration and velocity of the other train (*a_other* and *v_other* respectively), and calculates three outputs: the acceleration, velocity and train distance (*a*, *v* and *d* respectively):

$init\ pre\_d = init\_d \wedge pre\_v = 0;$

$transitions$

$true \rightarrow a := if\ danger\ then\ -brake\ else\ a\_other;$

$v := max\{0, pre\_v + a\};$

$d := pre\_d + v\_other - v;$

$pre\_d := d;$

$pre\_v := v;$

danger is used as shorthand for the condition expressing a potentially dangerous situation:

$$(pre\_v + a\_max \geq brake) \vee (pre\_v + a\_max > pre\_d)$$

To avoid repetition, we express some of the values of the outputs and states in terms of previously defined outputs. These can be eliminated by replacing them by the appropriate expressions from the previous definitions.

The observer simply makes sure that the trains never bump into each other: $d > 0$. We can also add the property that the velocity never exceeds *brake*:

$$true \rightarrow ok := d > 0 \wedge v \leq brake$$

Clearly, this is not satisfiable unless we constrain the environment and constants. This is usually one of the most difficult stages in expressing a specification. Some restrictions may be clear, but it is always a dilemma whether a proof failed to go through because of a bug or because of a weakly specified environment. In this case, we start by adding a number of restrictions on the inputs and constants. The restrictions on the constants are the following:

- The braking acceleration is not negative: $brake \geq 0$
- The trains start off some distance from each other: $init\_d > 0$
- The train can brake faster than it can accelerate: $brake > a\_max$

An observer is also added to make sure that the other train never reverses into us:

$init\ s = true;$

$transitions$

$s \rightarrow real := v\_other = 0; s := false;$

$\neg s \rightarrow real := v\_other >= 0; s := false;$

When the system is run through Lesar, we get a counter-example of length 1. Obviously, due to the abstraction applied for the verification, it is unclear as to whether there are concrete counter-examples corresponding to the abstract one. Lurette answers this in the affirmative, by generating the following concrete counter-example:

$$a\_max = 1, brake = 2, init\_d = 1, a\_other = 2, v\_other = 0$$

It is immediately clear what is happening: the other train is exceeding the maximum acceleration and since (i) the train under our control mimics the acceleration of the other train, and (ii) the safety distance is based on the maximum acceleration we believed we could achieve, we bump straight into the other train. Adding the extra constraint that $a\_other \leq a\_max$ solves this problem, and Lurette can no longer generate concrete counter-examples corresponding to the abstract one. The whole system can then be proved using a refined abstraction function.

4.3 Industrial Application

This tool has been applied in the verification of a Lustre model of the flight control software of the ARIANE-V rocket. As a typical example, for one of the properties which was not confirmed by Lesar, Lurette produced 16 counter-examples of length 23, involving more than 40 variables (including 2 numerical variables) in a matter of a few seconds. The counter-examples showed that the problem with the code lay (as is often the case) with an environment assertion which is too weak to guarantee correctness.

If a case-study is small enough to run through a model-checker, generation of the counter-model can be done in linear time (with respect to the size of the BDDs representing the sets of states leading to a counter example). The constraints implied by the resultant automaton are then to be solved by the trace generator (which can be done by running a Simplex program). Overall, the complexity of generating counter-examples, depends primarily on the model-checking and constraint solving algorithms. In practice, we are thus limited by the current state-of-the-art algorithms for these problems.

5 Related Work

Abstraction is now accepted as a necessary means to verify realistically sized systems. However, abstractions introduce new information and may also introduce spurious counter-examples which make it very difficult for non-expert users to use tools which use abstraction effectively. Translating abstract traces into concrete ones can be particularly difficult. Surprisingly, the amount of work on how this translation can be automated is rather sparse.

In [PDV01], Păsăreanu et al extend the Java PathFinder (JPF) model-checker so as to generate concrete counter-examples in the presence of abstractions. Two techniques are used: (i) searching for a deterministic counter-example, which clearly would correspond to a concrete one, and (ii) using simulation to verify whether the generated abstract counter-example, has at least one related concrete one. The first technique is clearly not complete. The second one works because the systems they analyze are *closed*, i.e., they do not interact with the environment. In our case, the counter-examples correspond to actual behavior only if the inputs take carefully chosen values. Since the inputs may range over infinite data types, straightforward simulation does not always work. A similar constraint is applied in SLAM [BR00], where the aim is to model check *boolean programs* (programs in which variables and procedure parameters are always boolean). Feasability of abstract paths is checked using *path simulation* — in which a heuristic decision procedure is used to try and decide whether the path is a feasible one. In general, this is undecidable and thus the decision procedure may sometime report "don't know".

A more general approach has been presented by Clarke et al [CGJ+03], in which the abstract interpretation is refined if the one in use is not fine enough to allow for a definite answer. An algorithm is presented, which allows the calculation of a concrete set of paths from a set of abstract ones. If the resultant set is empty, clearly, the abstract counter-examples have no counterpart in the concrete world. The method is dependent on the computability of the inverse of the abstraction function and also on the decidability of whether the set of traces is empty or not. This is clearly not always possible (for example, in the case of non-linear numeric constraints). These restrictions also hold in a similar approach presented in [LBBO01] and used in InVeSt. When the inverse of the abstraction function cannot be calculated, an upper approximation is used. In [CCK+02], the feasibility of abstract counter-examples is checked using a SAT-solver — which works only in the finite-state case. The approach is still refined in [CGKS02], which handles numerical variables through the use of Integer Linear Programming techniques. This is surely more powerful and complete than the constrained random searching performed by Lurette, but also surely much more expensive. However, this last approach could fruifully be integrated in the one described here (which was initially published earlier [PHR01]).

6 Conclusion

We have shown how the problem of calculating concrete traces from corresponding abstract traces can be reduced to one of constrained test-pattern generation, thus enabling us to use standard, off-the-shelf tools and algorithms to generate the counter-examples.

The algorithm has been implemented in the Lesar model-checker and can be used in conjunction with the Lurette test-pattern generator. These tools have been successfully applied on a number of examples extracted from the Ariane-V rocket flight control software.

We don't pretend that the resulting tool is better than anything one could do by connecting a model-checker (or an abstract analyzer, like in [JJGM03]) with a constrained sequence generator (like in Concurrent Constraint Programming [Sar90]). We wanted to show that the connection of such tools readily provides some solution to searching concrete counter-examples, and we gave an example of such a connection. However, it is important that the sequence generator be able to take into account an interesting class numerical constraints. This is what Lurette can do with linear inequalities, and which is not that common.

This processing of the counter-example data stream generated is complete, in the sense that no information is lost, and it is thus left to the test-pattern generator to apply — of course, non complete — heuristics when attempting to produce the concrete traces. This gives us the advantage of being able to try to generate concrete traces even when the abstraction function is too complex for other approaches to be applicable.

One major problem is when no concrete counter-examples correspond to the shortest counter-example under abstraction, but still, longer counter-examples do exist. One solution is to prevent the model-checker to stop as soon as the property is violated, letting it explore the whole set of reachable states. Since this can slow down the model-checking, an optimization consists in exploring only those states which can be reached without following an erroneous transitions: as a matter of fact, if none of the erroneous transitions on pathes leading to a given state have concrete counter-examples, this state is actually unreachable. Now, another problem is that we are no longer able to bound the length of searched counter-examples (they can follow loops in the abstract model, and so, be arbitrary long). So, once again, we cannot get a complete procedure (this is not surprizing, since the problem is undecidable), and we can only try to find longer and longer counter-examples.

Other than for debugging purposes, abstract trace feasibility is an essential component of abstraction refinement techniques and it would be interesting to explore the use of our approach with such techniques.

References

[BCDPV99] S. Bensalem, P. Caspi, C. Dumas, and C. Parent-Vigouroux. A methodology for proving control programs with Lustre and PVS. In *Dependable Computing for Critical Applications, DCCA-7, San Jose*. IEEE Computer Society, January 1999.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Fifth IEEE Symposium on Logic in Computer Science, Philadelphia*, 1990.

[BR00] T. Ball and S. K. Rajamani. Checking temporal properties of software with boolean programs. In *Workshop on Advances in Verification (with CAV 2000)*, 2000.

[CCK+02] P. Chauhan, E.M. Clarke, J. Kukula, S. Sapra, H. Veith, and D. Wang. Automated abstraction refinement for model checking large state spaces using SAT based conflict analysis. In *Formal Methods in Computer Aided Design (FMCAD)*, November 2002.

[CGJ+03] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. *JACM*, 50(5), September 2003. Preliminary version in CAV'2000, LNCS 1855, Springer-Verlag.

[CGKS02] E.M. Clarke, A. Gupta, J. Kukula, and O. Strichman. SAT based abstraction-refinement using ILP and machine learning techniques. In *CAV'02*, July 2002.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5), 1994.

[CMB90] O. Coudert, J. C. Madre, and C. Berthet. Verifying temporal properties of sequential machines without building their state diagrams. In R. Kurshan, editor, *International Workshop on Computer Aided Verification*, Rutgers (N.J.), June 1990.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Fifth Conference on Computer-Aided Verification, CAV'93*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language Lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, September 1992.

[HR99] N. Halbwachs and P. Raymond. Validation of synchronous reactive systems: from formal verification to automatic testing. In *ASIAN'99, Asian Computing Science Conference*, Phuket (Thailand), December 1999.

[JJGM03] E. Jahier, B. Jeannet, F. Gaucher, and F. Maraninchi. Automatic state reaching for debugging reactive programs. In *AADEBUG'2003 – Fifth International Workshop on Automated Debugging*, Ghent, September 2003.

[LBBO01] Y. Lakhnech, S. Bensalem, S. Berezin, and S. Owre. Incremental verification by abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Lecture Notes in Computer Science. Springer-Verlag, 2001.

[PDV01] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding feasible counter-examples when model checking Java programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[PHR01] G. Pace, N. Halbwachs, and P. Raymond. Counter-example generation in symbolic abstract model-checking. In *6th International Workshop on Formal Methods for Industrial Critical Systems, FMICS'2001*, Paris, July 2001. Inria.

[RWNH98] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[Sar90] Vijay A. Saraswat. The paradigm of concurrent constraint programming. In Peter Warren, David H.D.; Szerdei, editor, *Proceedings of the 7th International Conference on Logic Program-*

ming (ICLP '90), pages 777–778, Jerusalem, June 1990. MIT Press.