Automatic State Reaching for Debugging Reactive Programs

Fabien Gaucher, Erwan Jahier, Florence Maraninchi, Bertrand Jeannet

Automatic State Reaching for Debugging Reactive Programs

Fabien Gaucher*,1, Erwan Jahier*,1,
Bertrand Jeannet†,2, Florence Maraninchi*,1

* *VERIMAG, Centre Equation, 2 Av. de Vignate, 38610 Gieres, France*
† *IRISA, Campus de Beaulieu, 35042 RENNES cedex, France*

ABSTRACT

Reactive systems are made of programs that permanently interact with their environment. Debuggers generally provide support for data and state inspection, given a sequence of inputs. But, because the reactive programs and their environments are interdependent, a very useful feature is to be able to go the other way around; namely, given a state, obtain a sequence of inputs that leads to that state. This problem is equivalent to the general verification of safety properties, which is notoriously undecidable in presence of numeric variables. However, a lot of progress has been done in recent years through the development of model checking and abstract-interpretation-based techniques.

In this article, we take advantage of those recent advances to implement a fully automatic state reaching capability inside a debugger of reactive programs. To achieve that, we connect a debugger, a verification tool, and a testing tool. One of the key contributions of our proposal is the proper handling of numeric variables.

KEYWORDS: automated debugging of reactive programs; state reaching; input sequence generation; test; counter-example generation; abstract interpretation

1 Introduction

Debugging reactive programs. A reactive system can be viewed as an infinite loop, in which the program first reads inputs from its environment, then computes and emits some outputs towards the environment, while updating its internal memory. This intrinsic closed-loop behaviour of reactive systems makes the process of debugging particularly difficult, because:

- for each step of the execution, the user must provide values for all inputs, mimicking the behaviour of the environment. This is both tedious and error-prone;
- a reactive system is generally intended to control its environment; therefore the environment may depend on values produced by the program, and the program may depend on values produced by the environment. Hence, producing realistic input sequences is difficult and such sequences can not be generated off-line in general.

[1]E-mail: {Fabien.Gaucher,Erwan.Jahier,Florence.Maraninchi}@imag.fr
[2]E-mail: Bertrand.Jeannet@irisa.fr

From sequences of inputs to states, and vice-versa. A usual feature of debuggers is, *given a program input sequence*, to show the program *internal state* (current instantiation of input, output, and local variables as well as memories) at specific program execution points, possibly expressed as complex conditions on the internal state.

However, a much more challenging task is the other way around: *given an internal state*, how does one find a *sequence of inputs* that drives the program to that *state*? Such a state reaching capability is particularly useful for reactive programs, precisely because providing input sequence is tedious and difficult.

Issues related to automatic state reaching. The state reachability problem is equivalent to the verification of safety properties. Actually, verification tools reduce safety properties into a state reachability problem. And the verification of numeric safety properties is notoriously undecidable. However, a lot of progress has been done recently through the development of model checking [CES86, QS82] and abstract interpretation techniques [CC77, CH78, JHR99].

In this article, we take advantage of those recent advances to implement a fully automatic state reaching capability into a debugger for reactive programs.

Our proposition. The basic idea is the following: we first use an abstract-interpretation-based verification tool and try to prove that the state to reach (or set of states) is unreachable. Since this problem is undecidable, some abstractions are performed. Those abstractions are safe in the sense that, whenever the proof succeeds, then the state is unreachable for sure. But if the proof fails, we can not be sure that the state is indeed reachable and the abstract path leading to the state(s) may have no counterpart in the concrete world. The second idea is then to use a random-based sequence generator that will try to find a concrete path in the abstract one.

In this article, we present how we have added an automatic state reaching capability to Ludic, a debugger of reactive programs written in the Lustre data-flow synchronous language [HCRP91]. As far as we know, there is no automatic support for such a functionality in state-of-the-art debuggers. We have implemented it by connecting three tools together[3].

1. Ludic [MG00], a Lustre debugger that lets one, among other tasks, execute Lustre programs step by step and inspect program states.
2. Nbac [JHR99], an abstract-interpretation-based verification tool that may prove safety properties concerning Boolean and numeric variables of reactive programs (e.g., Lustre programs);
3. Lurette [RWNH98, RJR03], an automatic testing tool for reactive programs that computes sequences of values that are *relevant* according to a formal description of the program environment.

The connection between a proof tool and a test sequence generation tool is not a completely new idea: it has been done already in order to obtain counter-examples when the Boolean model-checker fails [PHR01, CGMZ95]. But as far as we known, it is the first time it is done for numeric variables properly; [PHR01] did handle numeric variables, but by abstracting them away into Boolean variables.

Contributions. The main contribution of the article is therefore the connection between three state-of-the-art tools to provide a completely automatic way of reaching a given program state in the presence of numeric variables. The main advantages of our proposal are the following:

- the tool is fully automatic. The user does not need to know anything about the testing and the verification tool at all (well, at this stage of debugging at least);

[3]Ludic and Lurette are part of the Lustre academic programming and validation environment, developed at Verimag. Nbac has been initially developed by B. Jeannet at Verimag and is now maintained by him at Irisa.

```
node maintain (n : int; val : bool) returns (m : bool) ;
var cpt : int ;
let
    cpt = if val then (0 -> pre(cpt)) + 1 else 0 ;
    m   = (cpt >= n) ;
    assert(n>=5);
tel
```

Figure 1: A Lustre program example

- the connection scheme allows the user to start the "reach a state" capability from any state, and not necessarily the initial one. This kind of functionality has been advocated as a means to reduce the complexity of model-checking [HWKF02];
- we propose heuristics that try to minimise the length of the generated sequences when exploring the abstract paths;
- the connection scheme would work with different tools, provided they have the same kind of interfaces.

Structure of the article. We first present in Section 2 a debugging session illustrating the usefulness of automatic state reaching for debugging reactive programs. Then, we briefly describe the main features of the underlying tools in Section 3 and describe how they are connected to each other in Section 4. We present related work in Section 5. Finally, we present some possible future work and conclude.

2 An illustrating example

Consider the very simple Lustre program of Figure 1. It receives two inputs, an integer variable `n` and a Boolean variable `val`, and computes a single Boolean output `m`. The output `m` is true if the input `val` has been maintained true during the last `n` consecutive steps. The length of such periods is computed with a local counter `cpt`. `pre(cpt)` denotes the value of `cpt` at the previous step; this previous value is initialised to 0 thanks to the "`0 -> pre(cpt)`" expression.

This program makes an assumption about the domain of correct values for input `n`, via the *assertion* "`(n>=5)`". Lustre assertions usually express constraints about the program's physical environment, that are taken into account by verification tools. Assertions may also be useful for the compiler to produce optimised code, and for the debugger to detect spurious node[4] calls dynamically.

Figure 2 shows the diagram of variable values during a possible execution of the program of Figure 1 that has been generated by our tool. From instant `0` to instant `4`, inputs are given manually, and the output `m` keeps the value `false`. At this point, the user invokes the state-reaching functionality, specifying he would like to make `m` true. Then, from instant `5` to instant `11`, inputs are generated automatically, in order to make `m` true at step `11`.

One should note that the generated input sequence is not the shortest one, because the input `val` at step `6` is false, which causes the counter `cpt` to be reset. Moreover, from instant `5` to instant `10`, the only constraint that is applied to the input `n` is the assertion, because there is no value for `n` that satisfies both the assertion and the goal to reach. However, choosing `n = 5` at instant `11` allows to make `m` true, i.e. `cpt` $\geq$ `n` . Section 4 will give the details of the whole process on this particular example.

[4] Lustre nodes are the equivalent of procedures or functions of most languages.

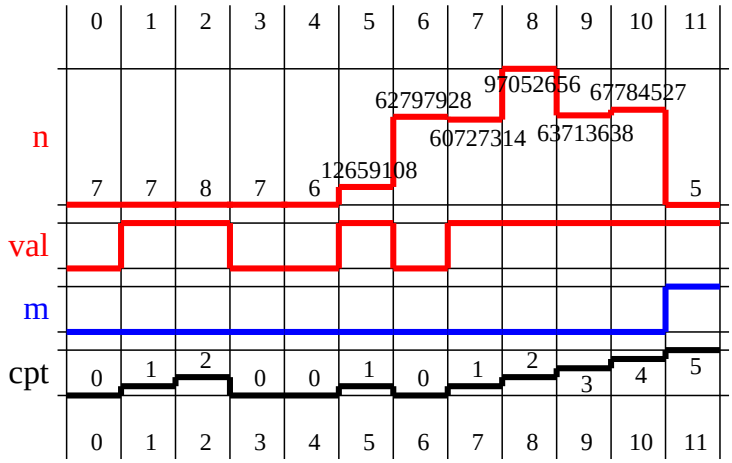

Figure 2: A possible execution of the program of Figure 1

3 Synchronous Development Environments and Tools

3.1 Generalities

In the synchronous approach to the programming of reactive systems, all executable programs have a very simple form. It is an infinite loop: first read inputs from the external world (read values from sensor devices for instance); then compute the corresponding outputs, depending on the current input and on (a bounded abstraction of) the input history; emit the outputs (write values to actuator devices); and do so forever. This very simple code may be run on a processor with no operating system.

The code that computes the outputs is hard to write in a sequential language, because it often has a natural parallel structure. The main motivation for the definition of synchronous languages [Hal93], is to allow users to *think* in parallel. The synchronous languages Esterel [BG92], Signal [LGLL91] and Lustre [HCRP91] all have a compiler into sequential code. Their semantics is deterministic, and they provide syntactic restrictions that rule out infinitely growing memory.

The main application domain being safety-critical embedded systems, a lot of effort has been put on the definition of testing and formal verification tools, in order to guarantee safety properties over program executions. In the following, we recall the central notion of an *observer*, and we briefly describe three of those tools, involved in our automatic state reachability process: the debugger Ludic, the verification tool Nbac, and the testing tool Lurette.

3.2 Observers

The Lustre development environment relies on: a compiler into C, a debugger, a testing tool, several verification tools (model-checkers, and theorem-provers). In all these tools, the user may have to specify *safety properties* (see [Lam77] for a distinction between safety and liveness properties).

Safety properties are described by *synchronous observers* [HLR93]. An observer is a regular synchronous subprogram, which observes the inputs and outputs of the program to be verified, and which outputs a single Boolean o with the following meaning: o is true as long as the sequence of inputs and outputs satisfies the safety property. As soon as the property is not satisfied, o is false forever. Compiling the program to be verified together with its observer provides a single output program, on which one may *prove* that the output is never false.

Of course, observing a deterministic program should not change its behaviour. This is the case with the observer technique, because the communication mechanism used in synchronous languages is the so-called *synchronous broadcast*: adding one or several *listeners* of a signal does not modify the behaviour of the emitter.

In the testing and verification tools, the observer technique is used both for the specification of the property to be tested or proved, and for the description of the environment (see more details below). In the debugger, observers may be used to specify conditional breakpoints.

3.3 The Ludic Lustre debugger

The Ludic debugger takes advantage of the formal semantics of Lustre: it works by interpreting Lustre, not by executing some compiled machine code equipped with traps. This enables some powerful functionalities over execution control or state observation. Especially, the static bounded memory used at execution time can be accessed, hence allowing to store any program state at any step of some execution. Moreover, saving inputs is sufficient to replay executions deterministically, which can be performed in a quite efficient way by storing states periodically. Thus, temporal behaviours of programs can be observed step by step, either forward or backward.

Inside one step, the declarative style of Lustre makes the data dependencies quite hard to understand, because the executed code is the result of a static scheduling of some activities that appear to be parallel at the Lustre level. Using slicing techniques [Wei79] may help programmers cutting parts of code that can affect (or be affected) by some variable's value, even if it does not give richer information about the partial ordering between computations of variables. Ludic implements such slicing algorithms, both static and dynamic [Gau03]. As far as this work is concerned, slicing allows to reduce the size of the program with respect to the set of variables that appears in the property describing the state to be reached.

3.4 The Nbac verification tool

Nbac [JHR99, Jea00] is founded on the theory of abstract interpretation [CC77], which allows to overcome the undecidability of the reachability problem for a large class of programs. Sets of states are represented in an approximate way by abstract values belonging to an abstract domain, and (fixpoint) computations are performed on this abstract domain. This leads to conservative results: if a state is shown unreachable, then it is, for sure.

The "basic" abstract domain used by Nbac is the direct product of the Boolean lattice and the convex polyhedra lattice. More precisely, a set of states is represented by the conjunction of a Binary Decision Diagram [Ack78, Bry86] for Boolean variables, and a convex polyhedra [Jea02] for numerical variables.

A lot of abstract interpretation tools for imperative programs work with the control structure given by the program text, and compute abstract values for each state. For parallel programs, state explosion may occur when building the explicit control structure.

In the declarative style of Lustre, there is no explicit control structure. Some tools use the structure induced by the configurations of all the Boolean variables, but this may also explode. Moreover, such a control structure does not take into account the control aspects induced by the numerical variables. Nbac is original because it starts from a quite declarative program description, in which Boolean and numerical variables play a symmetrical role. There is no natural control structure in such a program, and Nbac is able to build one particular control structure, depending on the property to prove. It starts from a very rough control structure, and then refines it dynamically according to the needs of the analysis. Successive refinements improve the accuracy of results and incrementally remove states that have already been shown unreachable.

The input of Nbac is a synchronous program in which *state* variables have been identified (to be used by the control structure refinement process), and the transition from one instant to the next one is made explicit. Starting from the Lustre program example of Section 2 and a specification of

```
state
  init, pre_m : bool;
  pre_cpt : int;
input
  val : bool;
  n : int;
local
  goal, start : bool;
  m : bool;
  cpt : int;
definition
  start = (not init) and (pre_cpt = 0);
  goal  = if start then false else pre_m;
  cpt   = if val then pre_cpt + 1 else 0;
  m     = cpt >= n;
transition
  pre_cpt' = cpt;
  pre_m' = m;
  init' = false;
assertion
  n>=5;
initial start;
final goal;
```

Figure 3: An Nbac program example

the initial and goal states, Ludic produces the Nbac program of Figure 3. See more details on the translation in Section 4, in particular the specification of an initial and a goal states. For the translation of the original Lustre program, two *state* variables are used: `pre_cpt` and `init` that helps encoding the arrow operator (that initialises flows). The variables are updated at each instant according to equations such as `v' = expr`, where `v'` denotes the value of variable `v` at the *next* instant and `expr` is evaluated in the *current* instant. The assertion on the environment `n >= 5` is inherited from the Lustre program. There is no explicit output because this format is used to specify the following *proof obligation*: prove that there is no execution starting in an initial state and leading to a final state.

Starting the analysis, Nbac will first build the explicit automaton given in Figure 4, by separating initial, final, and other states. Nodes are numbered (for references in the text) and labelled by formulas describing sets of states. Edges are labelled by necessary conditions on state and input variables to move from the source node to the target node within one execution step.

Nbac may answer "yes", which means that the property is true, and there is no path from initial to final. It may also answer "don't know" and provide as a result a new program of the same form. The meaning of the result is the following: Nbac has reduced the concrete state space of the original program, removing the states that cannot belong to a path from an initial state to a final state.

The result of the analysis, for the program of Figure 3, is given in Figure 5. Notice that the number of abstract states has increased, because Nbac has refined the control structure in order to be more precise. The result is still given by means of formulas, and it may be difficult to extract concrete counter-examples from it. That is the reason why we connect the output of Nbac to the testing tool Lurette (see Section 4).

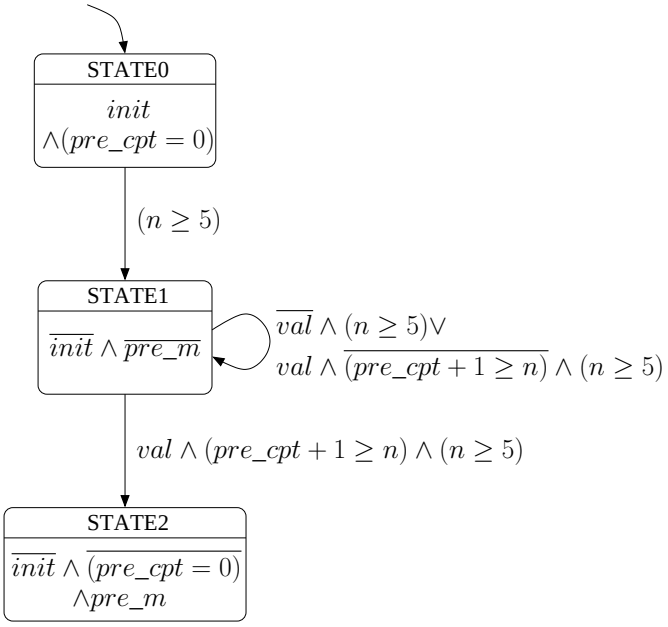

Figure 4: Nbac: the initial control structure

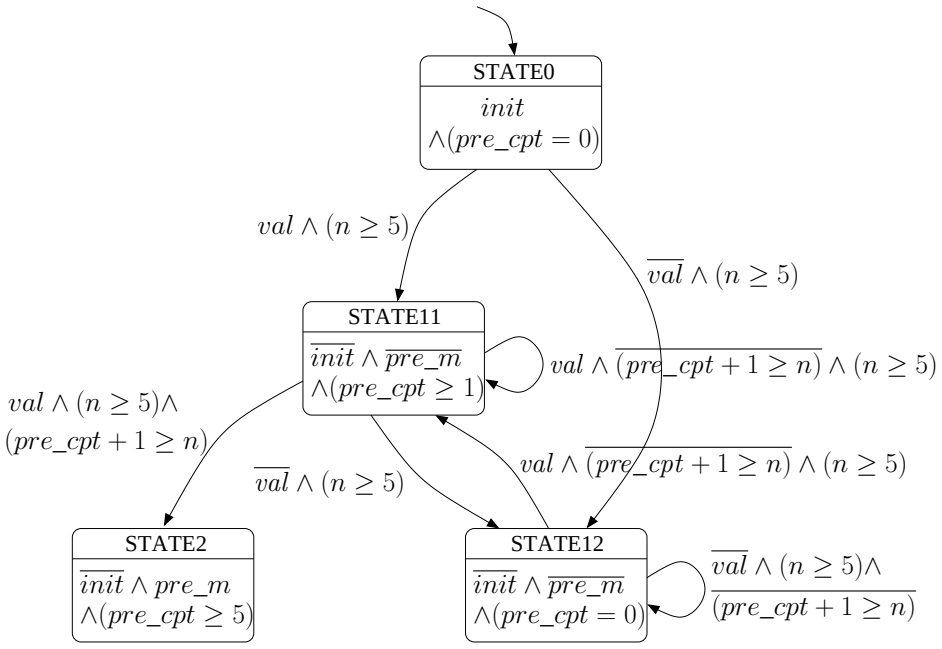

Figure 5: Nbac: the result of the analysis on the automaton of Figure 4

3.5 The Lurette testing tool

Lurette [RWNH98] is a tool that automatically tests reactive programs. One of the key points is the selection of inputs. Indeed, the inputs of a reactive system cannot be chosen randomly. The program is intended to be run inside some *environment*, on which its outputs will have some influence, thus influencing the future inputs of the program itself. For instance, the reactive system may be used to control a physical environment made of a heater and some air; it receives inputs from a thermometer, and sends outputs `on` and `off` to the heater. The heater is supposed to work. Then, if the system sends `on`, the temperature should start increasing; considering sequences of temperature inputs that do not increase from now on, is simply irrelevant. This is even stronger: a model of the physical environment also gives bounds to the variation rate of the temperature.

In Lurette, the testing process is automated in two ways:

- Lurette automatically generates input sequences for the program under test according to a user-given specification of realistic (or "interesting") scenarios modelling the environment. This cannot be done off-line, because the specification of the environment needs to know the outputs of the program. Hence the code of the program, and the specification of the environment, are "executed" together to provide sequences of relevant inputs. Generating values for a program with numerical variables involves general constraint solving techniques. The environment being intrinsically nondeterministic means we have to choose random values, among the solutions of the constraints.

- The test results perusal is also automated; users simply need to provide yet another specification (an oracle), which describes correct behaviours or desired properties of the inputs and outputs sequences. The oracle is also executed together with the program and the environment.

Recently, Lurette has been completely re-implemented, and extended. The main difference, from the user point of view, is the language used to describe the environment. In the first version of Lurette, only Lustre observers are offered, which means that we describe *acceptors* of correct input/output sequences, as Lustre programs.

In the new version, one may also use the Lucky [Jah03, RJR03] or the the Lutin [RR02] languages. Lutin is based on regular expressions, which are sometimes more convenient than Lustre, when *sequences* of behaviours have to be described. Lutin is compiled into Lucky which is basically the automaton form of Lutin regular expressions. Using lucky or Lutin instead of Lustre does not change the underlying synchronous computation model, but it gives a more "operational" style of description, where non-determinism is explicit.

The interesting part, for using this new version Lurette in our context, is the possibility to attach *weights* to the branches of choices. For instance, in the regular expression `e1 + e2`, one may put a large weight on `e1`, and a smaller one on `e2`, meaning that the generator will select the first branch more often. In the automaton form, such weights are associated with the transitions.

The restrictions of Lutin/Lucky are: (1) the constraints on inputs, at a given point of a sequence, may only depend on the *past* values of outputs (as in Lustre); and (2) numeric constraints should be linear. For instance, $x + y > 3$ can be handled, but not $x_1^2 + x_2^2 > 2$ nor $log\ x_1 + sin\ x_2 > e^{x_3}$.

We explain the operational semantics of Lucky on the automaton of Figure 6. This automaton has one Boolean input $heat\_on$, and one float output D. Suppose that the initial node is Off; two transitions are possible from that node.

- If the current value of the input $heat\_on$ is $false$, then it means that the transition that goes to the On node can not be taken (it is labelled by a formula that can not be satisfied). In such a case, only the transition labelled by the formula $-0.1 < D < 0$ can be taken. This formula will therefore be solved, and a solution will be drawn inside its set of solutions (namely, a float between -0.1 and 0) which will be the output of the automaton for the first step.

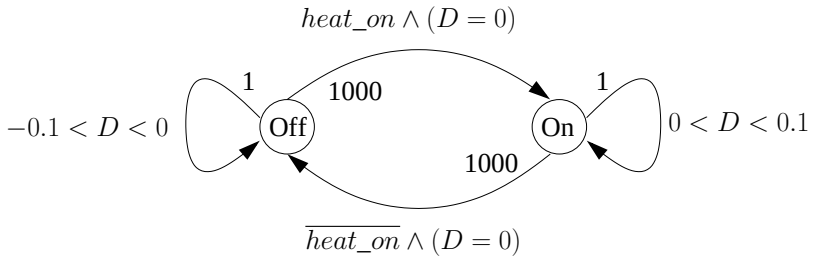

Figure 6: A simple Lucky automaton with a Boolean input $heat\_on$ and a float output D

- If the current value of the input $heat\_on$ is $true$, then two transitions are possible: the previous one, which has the weight 1; and a transition labelled by the weight 1000. The latter will therefore be drawn with a probability of $1000/1001$. In such a case, since the transition is labelled by the formula $heat\_on \wedge D = 0$, the value of the automaton output D will be 0 for the current step. And at the next step, the current node will be On.

The behaviour is symmetric if the current node is On, except that D will increase.

4 Connecting tools

From Ludic to Nbac. Consider again our Lustre program example of Figure 1, and the Nbac program of Figure 3. Remember that we want to generate an input sequence that makes the output variable `m` become `true`, starting from a given step of some execution (not necessarily the first instant). For that purpose, Ludic has to translate the Lustre program into the Nbac format, adding the specification for the *start* and the *goal* states.

The *start* state we are interested in is unique and completely defined by the valuation of the memories of the original program (`init` and `pre_cpt`). Ludic encodes this state into the Nbac format with the variable `start`. The set of *goal* states is usually specified as a safety property through the use of a synchronous observer. In our example, we need an observer for the variable `m`. We are interested in *instants* for which the output `m` is true. This is the same as looking for *states* where `pre(m)` is true. This is encoded by `goal = if start then false else pre_m`, because the goal should not be true at the instant in which we start the analysis.

The encoding of the original Lustre program requires only two variables: `pre_cpt` for the memory of `cpt`, and `init` for the encoding of the arrow initialisation operator. Specifying the final state involved in the proof obligation requires one more variable: `pre_m`, the memory of `m`. It is added to the Nbac program, with the obvious updating equation `pre_m' = m`.

For efficiency purposes, Ludic also performs some static slicing on the Lustre program, with respect to the set of variables involved in the definition of the goal state. Moreover, the front-end of the Lustre compiler is used to perform some network minimisation in order to reduce the number of variables, a crucial issue for the performance of Nbac.

From Nbac to Lurette. The verification goal of Nbac is then asked to show that there is no execution starting from the set of *start* states and leading to the set of *goal* states. The two possible answers of the Nbac analysis are the following:

- the property holds, i.e., the goal state is unreachable and has been removed during the refinement of the control structure. For debugging purposes, this information is as interesting as a counter-example;

- the property may not hold, i.e., there may exist a path inside the control structure leading to the goal state. Lurette will try to generate one or more input sequences. If Lurette generates at least one finite sequence, this sequence can be replayed by Ludic. Otherwise, we are in the problematic case where it is impossible to know, because of the abstractions made by Nbac, whether the state is unreachable or whether an existing path is difficult to reach because of a very low probability.

From Lurette back to Ludic. The control structure delivered by Nbac from our example is given in Figure 5. Here, it is intuitively clear that we should try to avoid the node `STATE12` to generate a short counter-example, which implies to always maintain the input `val` to true. This corresponds precisely to the fact that the input variable `val` should be maintained true long enough to make `m` become true.

Attaching Lucky's weights to this control structure can help finding short paths. Our heuristic is the following: from a given state S, the transitions that may lead to the goal state along the shortest paths have the greater weights. More precisely, we compute for each node n its minimal distance (in number of edges) $\delta(n)$ to the goal node. To increase the weight of edges that make this distance decrease, each edge $n_1 \rightarrow n_2$ is then given the weight $p^{\delta(n_1)-\delta(n_2)}$, where p is a user defined parameter.

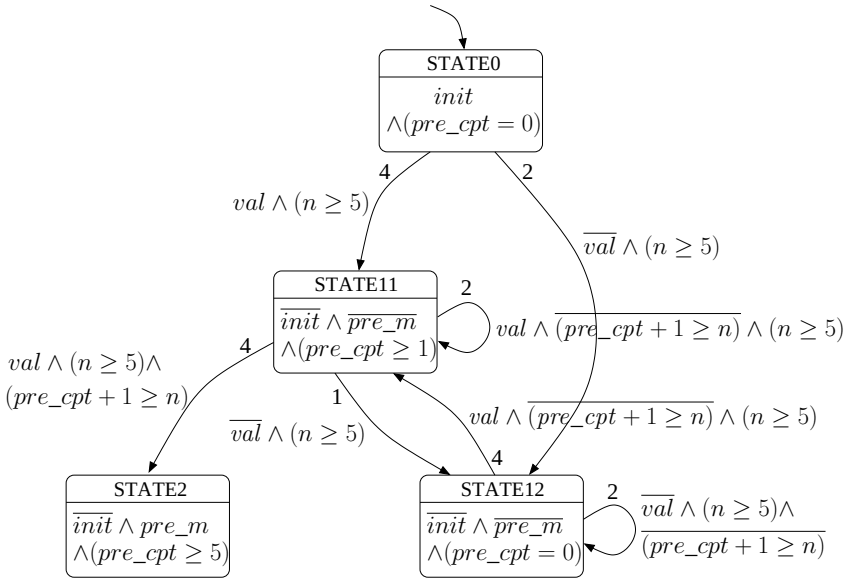

Figure 7: The Lucky automaton: adding weights to the result of Nbac shown on Figure 5

The result of this heuristic with $p = 2$ is shown on Figure 7. Note that it is the same automaton as in Figure 5, but decorated with weights. For `STATE11`, the outgoing transitions have weights 1, 2, and 4. The loop on `STATE11` corresponds to the incrementation of `cpt`, which should be selected at least four times before the transition to the goal state is satisfiable. In the diagram of Figure 2, steps 5 and 6 represent a (useless) transition to `STATE12` and back. Choosing $p = 1000$ instead of 2 would make this behaviour very unprobable. Indeed, when the transition to the goal state is unsatisfiable, the choice is between transitions with weights 1 and 2 if $p = 2$, or 1 and 1000 if $p = 1000$.

Limitation of the approach. There exists cases in which increasing weights does not help in generating shorter paths. Observe, for instance, the automaton of Figure 8. In order to find a concrete path from A to C, the random sequence generator has to "choose" the loop that increments `cpt` exactly 5 times, and then to "choose" the transition from A to B. Whatever the weights on the transitions sourced in state A, this is very unlikely to happen.

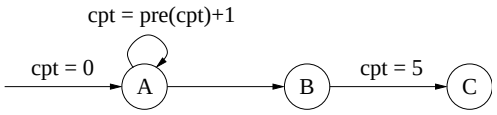

Figure 8: The Lucky automaton of a problematic case

In fact, the automaton of Figure 8 can not be produced by Nbac. Indeed, Nbac performs a backward analysis that enforces a condition on the transition from A to B, i.e., the guard `cpt = 5`. In this case, the "choice" between the two transitions sourced in A can be guided by the satisfiability of conditions on the transitions, not only by weights.

However, in similar cases, Nbac may have to abstract away variables, preventing the backward analysis from enforcing a condition on the transition from A to B. In those complex cases, playing with weights attached to the transitions sourced in A is useless. These cases should be detected somewhere along the chain of tools. This requires further work.

5 Related work

As far as the motivations of our work are concerned, the closest related work deals with automated testing of synchronous circuits. In [ZCR01], for instance, the authors use symbolic simulation in order to generate functional test vectors; since they care about coverage metrics, they have to specify a simulation target, and to try and reach it. The paper mainly discusses various constraint solving problems involved in the symbolic simulation process. Our approach is based on an *approximate* verification tool, which guarantees more reasonable costs. For the "reach a state" functionality to be useful in a debugger, it should not take too long.

The connection between a proof tool and a test case generation tool is not a completely new idea: it has been done already, in order to obtain counter-examples when the Boolean model-checker fails [PHR01]. The problem of error *diagnosis* in symbolic model-checking is well known [CGMZ95], but it is even more difficult in the case of programs with (a lot of) numerical variables, because the verification tool only gives approximate results. More recently, there has been interest in using counter-example generation to refine abstractions [CGJ+00, CCK+02], but in these works only abstraction of (big) finite-state systems are considered, which simplifies several algorithmical aspects.

6 Conclusions and Further work

The first motivation of this work is to provide a "reach-a-state" functionality in a debugger for reactive systems, written in formally defined languages, for which the problem can be expressed clearly. Since implementing this functionality amounts to solving a general model-checking problem, we chose a solution in which several independent tools are connected together, instead of some *ad hoc* coding of model-checking algorithms in the debugger. At first sight, it might not be the most straightforward implementation, but it makes clear what interfaces have to be respected between the various stages, for other tools to be used in the same chain. The set of tools we selected for demonstrating the feasibility of the approach have the following advantages: they all take numerical variables into account; the proof tool favours approximate results because they can be obtained in a reasonable time; the testing tool exploits the weights on the transitions of the automaton in order to provide short sequences.

We tried the chain of tools on medium-size examples, and it seems feasible. As one could expect, the bottleneck lies in the verification tool. In particular, the number of variables has a strong influence on the complexity of the algorithms involved. Further work will be devoted to the translation from Ludic to the input of Nbac, in order to simplify the program as much as possible. We already

used slicing techniques, but we could also apply general constant propagation techniques, or other optimisations based on static analyses.

Further work includes studying carefully the influence of weights on the generation of "short" sequences. For the moment, the weights are chosen automatically according to the structure of the graph, and the length of the paths leading to the goal. We could also think of alternative criteria for choosing the sequences: in a debugger, the sequence provided by our chain of tools is meant to show clearly *how* the program can go from the present state to another one. It might be the case that a sequence in which a minimal number of variables change their values is simpler to understand than a shortest sequence in which all variables change their values at each step.

As a new feature in the debugger Ludic, the ability to reach a state automatically has several applications. In order to automate the exploration of both time and data dependencies of complex programs, Ludic implements an original adaptation of the well-known algorithmic debugging principle [Sha83]. In practice, this technique can be used for a large class of (even big) programs. However, it may be the case that, at a given point of some execution, the value of a variable directly depends on some values computed many steps ago, forcing the tool to explore many previous steps (potentially the whole execution). For such programs, generating a short input sequence that leads to the same state where the bug symptom originally occurred would help locating the bug.

References

[Ack78] S. B. Ackers. Binary decision diagrams. In *IEEE Transactions on Computers*, pages 509–516, 1978.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

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

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.

[CCK+02] P. Chauhan, E. 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'02)*, page 18, November 2002. Paper will appear in the FMCAD 2002 conference.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[CGJ+00] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, CAV'00*, volume 1855 of *LNCS*, July 2000.

[CGMZ95] E. M. Clarke, O. Grumberg, K. L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the 32nd ACM/IEEE conference on Design automation conference*, pages 427–432. ACM Press, 1995.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978.

[Gau03] F. Gaucher. Slicing lustre programs. Technical report, VERIMAG, Grenoble, February 2003.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer Academic Pub., 1993.

[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.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HWKF02] S. Hazelhurst, O. Weissberg, G. Kamhi, and L. Fix. A hybrid verification approach: Getting deep into the design. In *DAC*, New Orleans, Louisiana, June 2002. ACM.

[Jah03] E. Jahier. *The Lucky Reference Manual Release 0.2*, 2003.

[Jea00] B. Jeannet. Dynamic partitioning in linear relation analysis. Technical Report RS-00-38, BRICS, December 2000. `http://www.brics.dk/RS/00/38`, to appear in *Formal Methods and System Design*.

[Jea02] B. Jeannet. *The Polka Convex Polyhedra library Edition 2.0*. IRISA, May 2002. www.irisa.fr/prive/bjeannet/newpolka.html.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In A. Cortesi and G. Filé, editors, *Static Analysis Symposium, SAS'99*, Venice (Italy), September 1999. LNCS 1694, Springer Verlag.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977.

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with signal. *Proceedings of the IEEE*, 79(9):1321–1336, September 1991.

[MG00] F. Maraninchi and F. Gaucher. Step-wise + algorithmic debugging for reactive programs: Ludic, a debugger for lustre. In *AADEBUG'2000 – Fourth International Workshop on Automated Debugging*, Munich, August 2000.

[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.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*. LNCS 137, Springer Verlag, April 1982.

[RJR03] P. Raymond, E. Jahier, and Y. Roux. Weight-labelled transition systems: an operational model for stochastic reactive systems. submitted for publication, January 2003.

[RR02] P. Raymond and Y. Roux. Describing non-deterministic reactive systems by means of regular expressions. In *First Workshop on Synchronous Languages, Applications and Programming, SLAP'02*, Grenoble, April 2002.

[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.

[Sha83] E.Y. Shapiro. Algorithmic program debugging. *MIT Press*, 1983.

[Wei79] Mark Weiser. *Program slices: Formal, psychological, and practical investigations of an automatic program abstraction method*. PhD thesis, University of Michigan, Ann Arbor, MI, 1979.

[ZCR01] Z. Zeng, M. Ciesielski, and B. Rouzeyre. Functional test generation using constraint logic programming. In *11th IFIP International Conf. on VLSI-SOC*. LNCS 137, Springer Verlag, December 2001.

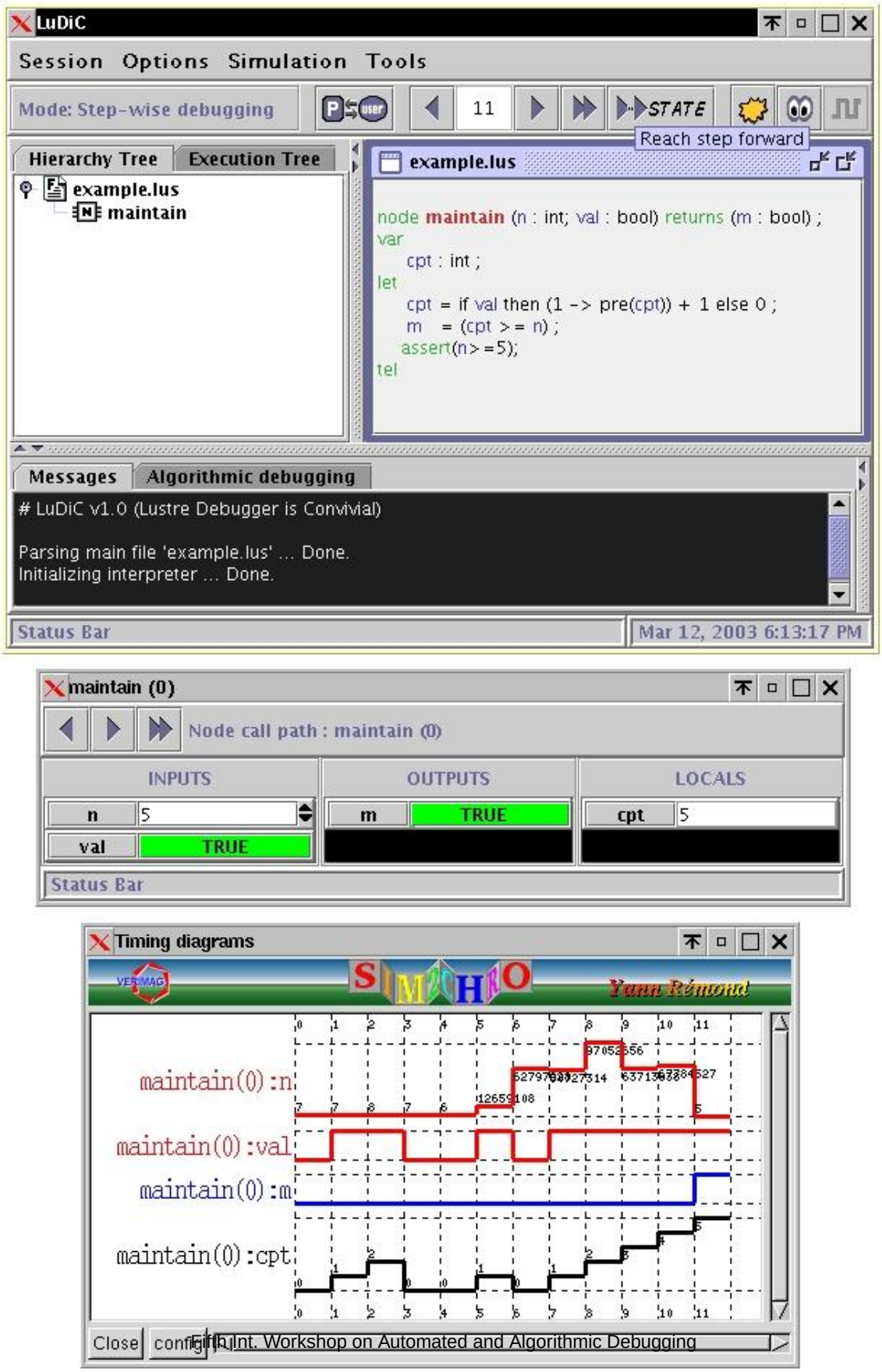

Figure 9: A Ludic snapshot of the automatic state reaching capability in action