Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 111 (2005) 93–111
www.elsevier.com/locate/entcs

Test Selection Strategies for Lustre Descriptions in GATeL

Bruno Marre and Benjamin Blanc

Laboratoire Sûreté des Logiciels
CEA/DRT/DTSI/SOL
91191 Gif sur Yvette CEDEX
{Bruno.Marre|Benjamin.Blanc}@cea.fr

Abstract

We describe various test selection techniques from Lustre descriptions using the tool GATeL. The Lustre language is declarative and describes synchronous data-flow computations. Our test generation tool interprets the language constructs as boolean and integer interval constraints. Test sequence generation is automated using constraint logic programming techniques. GATeL provides various mechanisms to allow testers to define their own selection strategies. They are illustrated on an simple example.

Keywords: Lustre, constraint logic programming, test generation, data-flow computation

1 Introduction

It is now well established that the use of a formal specification in a development process is of great interest even if, due to lack of time or resources, it is not subsequently used. Indeed a formal specification provides a concise and precise document for testing purposes: it gives solid basis for oracle decisions, and, when executable, it allows the automation of test data generation and submission. A substantial amount of work has been carried out around on model-based testing from classical specification languages, e.g.: Loft for axiomatic specifications [12], [6] for VDM, [17] for Z, [19] for Promela, BZ-Tools for the B Method [11], TGV for input/output automata [7]. Here, we present a model-based testing technique from Lustre descriptions, supported by the tool GATeL[13].

1571-0661
doi:10.1016/j.entcs.2004.12.010

Lustre is a specification and programming language for the description of synchronous data-flow computations (cyclic computations). It is used for reactive control/command systems, mainly for electrical power production applications. The final code may be either generated automatically or hand coded (for efficiency reasons, or in order to reuse existing code). We assume that we work in the second context, where Lustre is merely used as a specification language.

While this language benefits from powerful verification tools (using model-checking techniques such as in Lesar [10]), there is still a demand for adequate testing techniques. Several methods and tools have been proposed for the testing of synchronous reactive systems. They include those based on a Lustre model such as Lutess [4,5], and Lurette [16], and the statistical method presented in [18]. Lutess and Lurette both consider the testing of reactive programs in a black-box framework. Lustre is not imposed for the system specification but is used for the specification of the environment in which the program is embedded. Inputs of the program under test at each cycle must then conform to this specification. The computed outputs are then checked against an oracle defined by invariant (safety) properties. Lurette handles boolean and numeric data flows, while Lutess only deals with boolean data. However, Lutess allows the description of a richer environment (operational profiles and behavioral patterns). Another testing method [18] makes it possible to generate random test sequences which guarantee a statistical coverage of the structure of a Lustre program. Input distributions are defined on the control automata derived from the program under test. These distributions are defined so as to balance branch probabilities, and then taken into account during random generation of test sequences.

GATeL[13] is another tool supporting test sequence generation from Lustre descriptions. As in Lutess or Lurette, a specification of the environment describing the possible evolution of the inputs can be considered during test sequence generation. These inputs flows are specified by invariant properties (with the `assert` directive of Lustre) on inputs and past outputs, so as to reflect the reaction of the environment. Test sequence generation can also focus on test objectives as in classical approaches to protocol testing (test purposes, [19,7]). A test objective can be a safety property (an invariant expressed with an `assert`) or a declarative characterization of some interesting states of the system under test, expressed with a `reach` directive specific to GATeL. The characterization of the states to be reached is a boolean property expressed in Lustre, so it is possible (with the help of temporal operators) to test any property that can be expressed as an observation of the past. These components are translated into a constraint system using an interpretation of the

language constructs as boolean, integer interval constraints and guarded constraints (for the handling of control and temporal operators). Test sequence generation is then automated using constraint logic programming techniques. The use of a specialized constraint solver for the generation of test data is already the basis of several test generation tools, either for functional testing as in [11,15,20], or for structural testing as in [2,8,9,14].

In most model-based approaches, a particular testing strategy is defined according to the underlying model. The strategy usually relies on a predefined coverage criterion chosen according to the structure of the model. For instance, if the model belongs to the finite state machine family, the criterion can be to cover all states, or all transitions, or both [19,7,18]. Concerning state based models, it can be a coverage of boundary values of state components [6,17,11], or a coverage of operators sub-cases (e.g. branches of if-then-else statements [12,6,20,11]). Each criterion addresses a particular class of faults. However, in the case of a combinatorial explosion due to the coverage of a predefined criterion, it may become useful to change the criterion. Moreover, generating numerous test cases is useless when the resources devoted to the testing stage do not allow the execution and result evaluation of each of them. Finally, for the certification of highly critical systems, certification agencies must apply the diversification paradigm when they have to validate such systems. In the context of software testing, this means that they must be able to apply their own test selection strategies. For these reasons, we preferred in GATeL to provide the user with the basic mechanisms allowing the definition of customized selection strategies for the application under test.

After a brief presentation of Lustre and the resolution procedure used in GATeL, we will present these basic selection mechanisms in the rest of the paper. The first one relies on the notion of test objective, and we will take a microwave oven controller as a simple example. We then explore on the same example the technique of interactive domain splitting by exploring sub-cases of the initial constraint system. This technique is based either on predefined operator sub-cases or user-defined integration scenarios.

2 Lustre

Lustre [10] belongs to the synchronous data-flow language family. It was developed at the Grenoble IMAG institute. Lustre is not just another academic language but is effectively used in industry due to its numerous advantages (e.g. Schneider Electric, Aerospatiale). It provides both a textual and a graphical notation, the latter being similar to those used in hardware design (block diagrams, operators net, etc.).

The underlying model of a Lustre program is that of a time-driven automaton. It describes cyclic behavior between two consecutive ticks of a global clock. At each tick, it gets all its input data and computes them so as to define the corresponding outputs. Each input (resp. output) data-flow is the sequence built from the successive data received (resp. emitted) during the temporal run of the program. Synchronization of all processes is specified by the fact that the computation time is bounded by the following tick. Thus, at each cycle, all the data flows have the same length.

Programs and sub-programs blocks are called *nodes*. Each computed variable (output or local) is defined by a single equation where its name occurs on the left-hand side, the right-hand side being its defining expression. This expression can refer to past values of the defined variable and to other variables (present or past values).

```
const cycle_duration = 1; -- cycle duration set to one second

node oven (start, abort, open: bool; duration: int)
returns (remaining_time: int; cooking, bell: bool) ;
var
  running: bool;
let
  running = if (open or abort)
            then false
            else if start
                 then true
                 else (false -> pre(running));

  remaining_time =
       if running
       then (duration ->
             if start
             then if pre(remaining_time) = 0
                  then duration                    -- new cooking
                  else pre(remaining_time)         -- restart
             else if (pre(remaining_time) > cycle_duration)
                  then (pre(remaining_time) - cycle_duration)
                  else 0)
       else if abort
            then 0
            else (0 -> pre(remaining_time));

  cooking = running and (remaining_time > 0);

  bell = false ->
         ((remaining_time = 0) and (pre(remaining_time) > 0));
tel;
```

Let us consider the simplified model of a microwave oven controller. The

reaction is described with four inputs and three outputs. The boolean input data flows `start` and `abort` correspond respectively to the start and abort buttons of the oven, they are true when the user pushes the corresponding button. The input `open` is set by a sensor on the oven door, it is true while the door remains open. The input `duration` is the cooking duration in seconds programmed by the user. The output parameter `remaining_time` is the remaining time (in seconds) until the end of cooking which is displayed on the screen of the oven. The boolean output `cooking` is connected to a symbol highlighted on the screen when cooking is in progress. Finally, the `bell` parameter triggers the ringing of a bell when cooking is finished or aborted.

Let us describe the definition of the local variable `running`. It is true when cooking has been started (at the current cycle or at a previous one) and has never been aborted and the door never been opened since the beginning of cooking. The expression used for its definition uses two particular temporal operators: `->` and `pre`. The operator `pre` returns its parameter value at the previous cycle. Thus `running` is recursively defined over a discrete time. The operator `->` is used for initializations and returns at the first cycle the value of its first parameter, and at any later cycle the value of its second parameter. Other Lustre operators, e.g. `if then else`, boolean operators (`and`, `not`,...), arithmetic operators (`-`,...) and comparisons (`>`,...), are more classic. They refer only to the values of their arguments at the current cycle.

Invariant properties can be stated with the `assert` directive, which operates on a boolean expression that must be satisfied at each cycle. Let us come back to the above example. It can be extended with the assertion given below. This assertion states that the oven is never started and aborted in the same cycle.

```
assert  not (start and abort) ;
```

3 GATeL prerequisites

A testing method usually follows three classical steps: test case *selection* and input data *generation*, *submission*, *oracle*. Within Lustre these steps are adapted as follows:

- *selection* of test cases and input data *generation* : for each selected test case, test inputs are sequences of equal length, with values of adequate data types;
- test *submission*: the program under test is executed on test sequences in order to obtain the corresponding outputs;

- the *oracle* compares computed outputs from the program with the outputs expected from the specification.

GATeL provides mechanisms allowing a tester to define his own selection strategy. It completely automates the generation of input sequences for each test case derived from the selection strategy. GATeL also provides the information needed to construct an oracle. Our tool systematically computes from the Lustre model, inputs, outputs and truth values of the test objective at each cycle. These evaluated outputs constitutes the expected ones which should be compared to actual outputs of the program under test, and thus represents a partial oracle. The mechanisms proposed to assist the definition of selection strategies use some of the control features of the resolution procedure involved in test sequences generation. Before going further in their presentation, we first describe the principles of our resolution procedure.

3.1 GATeL kernel

The kernel of GATeL is a resolution procedure for constraints built from an interpretation of Lustre constructions over boolean variables, variables with integer intervals (real numbers or floating-point number arithmetic are not considered yet), and a special synchronization constraint for the status of each cycle (whose value is either`initial`, or `non_initial`). Resolution proceeds by successive elimination of all constraints. A non-deterministic instantiation procedure (called "labelling" in the logic programming community) instantiates the variables involved in the constraints. In order to avoid erroneous valuations, a constraint propagation mechanism continuously checks constraint satisfiability. The instantiation of a variable "awakes" the propagation of related constraints, which can disappear (when solved) or awake/create other constraints.

The temporal operators make it impossible to always predict how long each test sequence will need to be. However, this length is bounded by a parameter tunable by the user. When a constraint needs the value of a variable at a previous cycle (operator `pre`), whose status is unknown, the status of the current cycle (attached to the constraint) is instantiated to `non_initial` and a previous cycle is created.

The equation defining a variable is introduced as a constraint only when needed, i.e. when some constraint needs the value of this variable. This "lazy" insertion of constraints makes it possible to minimize the average number of constraints and thus the amount of memory needed (memory also depends on the number of "labelling" steps).

When all constraints are solved, we get a partial instantiation of input

data flows (some input values may not be needed during resolution). Ground test sequences are then computed by a random instantiation of the remaining input variables (inside the interval bounds for integer variables). The expected output sequences (for the oracle) are then computed by a simple evaluation step.

GATeL efficiency relies on several specialized heuristics, for the choice of the variable to be instantiated during the labelling steps or consistency checking during propagation. Classically, the chosen variable must awake the maximal number of constraints, while minimizing the average branching of the resolution tree (variable with the smallest domain). To facilitate this choice, the global constraint system is structured into smaller independent ones according to constraint dependencies. Consistency checks rely on the usual arc-consistency refined with refutation mechanisms for boolean constraints, and abstraction of constraint relationships for integer ones. For a more complete description of our tool, please refer to [13].

4 Defining a test objective

Test sequences are generated from a testing description involving three Lustre components: a model of the program under test, a specification of its environment and a test objective.

The specification of the environment contains assertions about current/past inputs and possibly about past outputs (since the program under test reacts with its environment). The point here is to filter out from all the possible behavior of the model the behaviors corresponding to realistic reactions. Each statement should thus be carefully checked. Moreover, since several assertions may involve the same variables, the consistency of the model gets harder to ensure as it gets larger. In order to check an environment specification the model can be animated using simple random simulations for a limited number of cycles. At each cycle, assertions coming from the environment are introduced as constraints on input values, which are then randomly instantiated inside their restricted domain so that the computed outputs also remain within valid domains. Indeed, for integer variables this computation could lead to values outside the authorized bounds so it is controlled by a formal integer interval arithmetic.

The test objective states some important expected properties of the program under test to be checked. Such properties must be consequences of the model restricted by the environment specification (we will see later how refutation can be used to ensure this point). Generally, these properties correspond to the formalization of information found in the requirements doc-

uments. They can be either invariant properties or reachability properties. They may involve inputs/outputs (and if necessary, local variables). Invariant properties are stated with the `assert` directive. The properties that must be satisfied in at least one cycle (in fact, in the last cycle of sequences built by GATeL) are stated by `reach` directives. We have added this `reach` directive to Lustre in order to exercise input/output properties which are not invariant, but satisfied only at some specific point in the execution. If the same property has to be proved it would be expressed with the state to be reached as a precondition. However, for test purposes, we are only interested in the case in which the precondition is true. The `reach` directive allow us to make it explicit. The full Lustre syntax is available for the argument of a `reach`. We can thus express any observation of the past.

With such a testing description, test sequences are generated as solutions of the constraint system built from the conjunction of constraints derived from each component.

4.1 Example

Now let us illustrate the definition of a test objective for our microwave oven example. The invariant given in section 2 is a part of a description of the environment. Here are further environment constraints and their corresponding Lustre translation:

- Cooking duration is strictly positive and less than or equal to one hour.
- Start is not possible during cooking.
- Abort and door opening cannot occur simultaneously, abort is possible if the door is closed or was already opened at previous cycle.
- Similarly, start and door closing cannot occur simultaneously: start is allowed when the door was previously closed (except in the first cycle) and is still closed.
- Cooking duration may be modified only when a start occurs after a normal end of cooking or an abort (in both cases `pre(remaining_time) = 0`).

```
assert (duration > 0) and (duration <= 3600);
assert implies(start, (true -> not(pre(cooking))));
assert implies(abort and open, (true -> pre(open)));
assert implies(start, not(open) and (true -> not(pre(open))));
assert (true  ->
         (if duration = pre(duration)
          then true
          else  (start and (pre(remaining_time) = 0))));
```

Now we can try out our environment constraints with random simulations. The length of the sequences is bounded by a parameter which is tunable by the user (here 10). Cycles are numbered backwards (0 is the last cycle). Even though the behavior of the environment seems realistic, some interesting typical situations are not reached. For instance (see following table), a standard use of the oven is almost never tried out, since it implies avoiding an `abort` for several cycles in a row.

#Cycle	start	abort	open	duration	rem time	cooking	bell
9	false	false	true	2153	0	false	false
8	false	true	true	2153	0	false	false
7	false	false	false	2153	0	false	false
6	false	true	false	2153	0	false	false
5	false	true	false	2153	0	false	false
4	true	false	false	1026	1026	true	false
3	false	false	false	1026	1026	false	false
2	false	true	true	1026	0	false	true
1	false	true	true	1026	0	false	false
0	false	true	true	1026	0	false	false

Random simulation as a way to generate test sequences could be further refined, however this was not its primary goal in GATeL. Some other tools have developed refinements of this simple method (Lutess or Lurette).

An interesting objective to reach would be a completed cooking session, that is an occurrence of `bell` while `start` occurred at least once and without an `abort` since the last `start` occurred. The temporal operators `once_at_least` and `never_since_last` observe the past of their parameters and may be defined in Lustre just like any other nodes.

After grouping the environment assertions together in a node `env_oven`, the whole test description can be assigned to a single test node `obj1_oven` as follows. In this node, the `reach` directive is specially commented (with `(*! !*)`) so as to ensure compatibility of the file with other Lustre tools.

```
node obj1_oven (start, abort, open: bool; duration: int)
returns (remaining_time: int; cooking, bell: bool) ;
let
  assert env_oven(start,abort,open,cooking,duration,remaining_time);

  (remaining_time, cooking, bell) =
                oven(start, abort, open, duration);

  (*! reach bell and
            once_at_least(start) and
            never_since_last(abort,start,false) !*)
tel;
```

Loading the test description above in GATeL leads to this partial sequence:

#Cycle	start	abort	open	duration	remaining_time	bell
?	...	...	...	...	...	...
1	_	false	false	1..3600	1	_
0	false	false	false	1..3600	0	true

Indeed, due to the test objective, the values of some flows are known at the last cycle (cycle 0), and at a previous cycle. These values and the existence of at least two cycles are direct consequences of the testing description deduced by the initial deterministic propagation step.The generation of a sequence for this test objective can give for instance the completed cooking session of 4 cycles duration:

#Cycle	start	abort	open	duration	rem_time	cooking	bell
4	true	false	false	4	4	true	false
3	false	false	false	4	3	true	false
2	false	false	false	4	2	true	false
1	false	false	false	4	1	true	false
0	false	false	false	4	0	false	true

Due to random instantiations during labelling steps and constraint propagation, another run would lead to sequences of different length with different values. Thus, we cannot guarantee minimality of the generated test sequences. However, we can exhibit instances of behavior that can further be differentiated (see section 5).

4.2 Refutation of invariant properties

The previous test objective was the Lustre expression of a reachability property. The generation of one test sequence shows that this property is effectively a consequence of the testing description. If the test objective is an expected invariant property, it is important to check its real invariance w.r.t. the testing description. The resolution procedure may also be used to prove or disprove invariant properties of Lustre models using refutation. Given a model and an expected invariant property P, a test objective is defined by `reach not(P)`. Then GATeL is asked to generate a test sequence for this objective. If a sequence is found, it is a counter-example of the invariance of P. On the other hand, due to the bounded completeness of our resolution procedure, when no sequence is found, one can only deduce that within the bounds defined by global parameters (maximal number of past cycles, size of the initial interval for integer variables), this property is invariant. The resolution procedure is used here as a semi-algorithm. However, after an analysis of error messages given by GATeL, one can often conclude that this failure is not caused by these global parameters but by the property itself, thus ensuring invariance.

Thus, refutation can be used to check the consistency of any invariant, and helps during the elaboration of a Lustre model. For example, we can check that it is not possible for the oven to cook while the door is open. This property must be invariant and can be exercised with the following objective:

```
(*! reach (cooking and open) !*)
```

When introduced with the environment and the model of the oven, GATeL immediately detects the unsatisfiability of the corresponding constraint system.

5 Assistance in the design of test selection strategies

At this point we have just shown how to select one test case using a test objective. Keeping only one solution of the constraints system amounts to considering that each solution has an equal interest. In other words, we assume that any test sequence reaching the objective has the same power to reveal faults. Such *uniformity hypotheses* [3] are common but often too strong. Test cases are smaller domains on which theses hypotheses get more realistic.

There are two different ways to describe test cases in GATeL. The first one relies on an interactive unfolding of Lustre operators of the current constraint system. In this case, test cases are defined by a structural decomposition of the initial constraint system. This method allows a fine grained coverage of Lustre expressions, and thus is best used within unit testing. The second one uses predefined functional scenarios attached to variables. A scenario can be seen as a high level splitting method, since the user identifies which parts of the domain to explore, only exhibiting particular instances of behavior among all possible ones. This method can be used during the integration phase, since scenarios may involve the composition of several nodes.

5.1 Unfolding of Lustre operators

The initial constraint system is built from assertions at each known cycle, from properties occurring in `reach` directives at the last cycle, and from necessary data flow variables definitions. This system characterizes a domain of validity of the uniformity hypotheses. We propose a splitting technique of this domain adapted from the unfolding technique of the tool Loft [12]. It makes it possible to recursively split sub-domains according to predefined sub-cases of Lustre operators.

For example, for a constraint "$\texttt{S}_i$ `= if` $\texttt{Cond}_i$ `then ExpThen else ExpElse`" where $\texttt{Cond}_i$ is a variable at the cycle i, by unfolding of `if then else` we can derive two sub-domains. The first sub-domain includes all test sequences such that $\texttt{Cond}_i$ is true, while the second sub-domain includes all test sequences such that $\texttt{Cond}_i$ is false. These two sub-domains are characterized by the constraints systems obtained after propagation of $\texttt{Cond}_i$ valuations. They can be

split again by unfolding an operator occurring in their constraints.

At each unfolding step, GATeL shows the operators that can be unfolded (top-level operators of an expression whose evaluation is needed). Thus, the user can interactively and dynamically tune the kind and number of sub-domains. Furthermore, domain splitting can be applied to the constraints system defining an output chosen by the user. In this case, unfolding builds the path predicates of the selected output at an arbitrary cycle. In this way a *structural* coverage of the expressions involved in the computation of the selected output can be obtained. This technique can thus also be used when the Lustre description is the program itself.

Here is an excerpt of the list of operators which can be unfolded by GATeL (and their sub-cases).

- $\texttt{A}_i$ `= not(Exp)`: 2 cases corresponding to $\texttt{A}_i$ valuations;
- $\texttt{A}_i$ `=` $\texttt{Exp}_1$ `and` $\texttt{Exp}_2$: 3 possibilities tunable by user,
  · sequential `and` (default): 2 cases corresponding to $\texttt{A}_i$ valuations;
  · lazy `and`: 3 cases,
  (i) $\texttt{A}_i \leftarrow \texttt{true} \wedge \texttt{true} = \texttt{Exp}_1 \wedge \texttt{true} = \texttt{Exp}_2$,
  (ii) $\texttt{A}_i \leftarrow \texttt{false} \wedge \texttt{false} = \texttt{Exp}_1$,
  (iii) $\texttt{A}_i \leftarrow \texttt{false} \wedge \texttt{false} = \texttt{Exp}_2$;
  · normal `and`: 4 cases corresponding to its truth table;
- $\texttt{A}_i$ `=` $\texttt{Exp}_1$ `=<` $\texttt{Exp}_2$: 2 cases ($\texttt{A}_i$ valuations) or 3 cases ($=$, $<$ or $>$), tunable by user,
- $\texttt{A}_i$ `=` $\texttt{Exp}_1$ `->` $\texttt{Exp}_2$:
  (i) the status of cycle i is `initial`,
  (ii) the status of cycle i is `non_initial`.

If a complete coverage is sought, a systematic unfolding could be undertaken. This would lead to an "all paths" coverage criterion. However, due to the presence of temporal operators, an infinite unfolding (in fact bounded by a global parameter) would be possible. It is because of this specificity of Lustre that we believe interactive operation is more appropriate.

5.2 Unfolding tools in GATeL

Suppose one wants to observe two different kinds of sequences for the test objective given in the previous section: those where the door remains closed during cooking, and others where the door has been opened during cooking, both without abort since the beginning of cooking. We will try to characterize some test sequences for both cases by successive unfolding of Lustre operators.

After loading the former testing description, a first way to proceed consists

in a selection of unfoldable operators from the Lustre definition of "active" variables (whose definition has already been propagated). Several operators can be selected at each step, each operator can be unfoldable at several cycles and in several cases. Since all values were set at cycle 0, no further operator can be selected for this final cycle. However, the upper-most `->` of the definition of variable `remaining_time` is unfoldable at cycle 1. Selecting this operator, we get the two sub-domains corresponding to the possible status of cycle 1. If cycle 1 is initial, we get a partial sequence of two cycles where the door was not opened. Otherwise (if cycle 1 is not initial) another previous cycle is created.

In this second case, the definition of `remaining_time` at cycle 1 is reduced to the second argument of the previous `->` (the non-initial case). Thus, the `if then else` operator at the top-level of this expression becomes unfoldable. This unfolding defines a splitting between sequences where no start occurred at cycle 1 and others where it did (`start` being the condition of this operator). In order to get sequences where this `start` is a re-start (which is the situation we want to focus on), we can simply indicate that the remaining time was not set to 0 at the previous cycle. This can be done with a last unfolding of the next `if then else` of the definition in this case.

The definition of the variable `remaining_time` in the main window of GATeL after this splitting process is shown in Figure 1. The color code for operators is the following: blue when unfoldable (a menu appears in order to choose the cycle and the case in which unfolding should take place), green when just selected, grey when previously selected and no longer selectable.(Clearly, the reader need to be imaginative enough to guess these colors on the grey level interpretation of this screenshot!)

```
remaining_time =
   if running
   then ( duration
            ->
            if start
            then if pre(remaining_time) = 0
                 then duration
                 else pre(remaining_time)
            else if (pre(remaining_time) > cycle_duration)
                 then (pre(remaining_time) - cycle_duration)
                 else 0)
   else if abort
        then 0
        else (0 -> pre(remaining_time));
```

Fig. 1. Unfolded and unfoldable operators

We decide to stop domain splitting and ask for the generation of one test

sequence for each of these four sub-domains.

(i) The first test sequence reach the objective when cycle 1 is set to be the initial cycle:

#Cycle	start	abort	open	duration	rem_time	cooking	bell
1	true	false	false	1	1	true	false
0	false	false	false	1	0	false	true

(ii) The second test sequence reaches the objective when `start` is true at cycle 1 and `remaining_time` is set to 0 at the previous cycle. Thus, this occurrence of `start` begins a new cooking.

#Cycle	start	abort	open	duration	rem_time	cooking	bell
2	false	false	false	1	0	false	false
1	true	false	false	1	1	true	false
0	false	false	false	1	0	false	true

(iii) The third test sequence reach the objective when this `start` is a re-start (`remaining_time` is maintained). Thus, `open` must occur at cycle 2 to create this situation. This sequence is then finished so as to complete the test objective.

#Cycle	start	abort	open	duration	rem_time	cooking	bell
4	true	false	false	1	1	true	false
3	false	false	true	1	1	false	false
2	false	false	false	1	1	false	false
1	true	false	false	1	1	true	false
0	false	false	false	1	0	false	true

(iv) The last sequence corresponds to a normal cooking with any duration time.

#Cycle	start	abort	open	duration	rem_time	cooking	bell
2	true	false	false	2	2	true	false
1	false	false	false	2	1	true	false
0	false	false	false	2	0	false	true

Even though this splitting process was not obvious and requires a good understanding of the unfolding of Lustre operators, it only involved a single definition at one cycle. Another way to proceed allows the user to unfold operators not from their definition but directly in the constraint system defining a sub-domain. Two tools are then available in GATeL: the constraint system for each test case, and a test tree showing the splitting process.

Going back to the initial propagation step after loading the test description, the definition of the variable `remaining_time` is contained in the constraint system and simplified as follows (where $\texttt{pre(remaining\_time}_1\texttt{)}$ is replaced by $\texttt{remaining\_time}_2$) [1] :

[1] Compare with the original definition in section 2

```
remaining_time1 =
    duration1
    ->
    if start1
    then if remaining_time2 = 0
         then duration1
         else remaining_time2
    else if (remaining_time2 > cycle_duration)
         then (remaining_time2 - cycle_duration)
         else 0
```

We can then follow the same splitting process as before on the successive versions of this definition at cycle 1. Each new version is calculated according to the unfolding calculus given in the previous section. The difficult point here is to clearly recognize which case corresponds to the propagation of which value. A test tree is designed for this purpose, illustrating the splitting process and also allowing to directly select operators from popup menus attached to its leaves. For instance, at the end of this process, we get the tree of Figure 2.

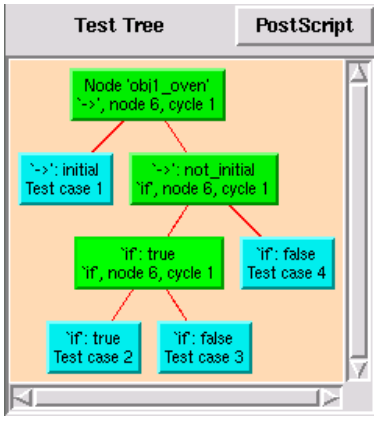

Fig. 2. Test tree

5.3 Functional scenarios

Unfolding is a simple answer to the need of splitting the domain of a test objective. However for methodological reasons this process may not be applicable on complex examples. When the decomposition sought implies several variables at several cycles, the choice of the right operator to unfold may become harder. This is also the case even on simpler decompositions, when many operators are unfoldable. Moreover, as we saw on the above example, the splitting process may create auxiliary test cases which complicate the examination of the generated test sequences. For instance, only two cases in four were really needed (*iii* and *iv*), while the other two are cumbersome.

For these reasons, we also propose a `split` directive in GATeL to attach predefined functional scenarios to one variable. Each scenario represents one expected case of the decomposition, and is defined as any boolean Lustre observation of the past. This directive follows the syntax:

```
(*!  split var with [case_1,...,case_n] !*)
```

When activated, this directive splits the global system into n cases containing respectively each boolean expression $case_i$ constrained to be true. Notice that to be activated, this directive needs the definition of its attached variable to have been introduced into the constraint system (either due to direct constraint propagation or due to an interactive unfolding).

Let us consider again the decomposition of the previous section. In order to force this decomposition with a split directive, two scenarios are defined: one where the door remains closed during cooking, and another one where the door has been opened during cooking, both without abort since the beginning of cooking. With these scenarios the test objective is simplified as follows:

```
(*! reach bell !*)
(*! split bell with [
          bell and once_at_least(start)
               and never_since_last(abort, start, false)
               and never_since_last(open,start,false),
          bell and once_at_least(start)
               and never_since_last(abort, start, false)
               and current_when_bool(
                      (duration > remaining_time) and
                      (remaining_time > 0),
                      start)] !*)
```

The `current_when_bool` operator is the GATeL definition of the combination of the Lustre operators `current` and `when`, which are not directly implemented in GATeL. It states that when the `start` variable is true, then `remaining_time` is less than initial duration and strictly positive, which correspond to a restart configuration.

When the corresponding Lustre description is loaded, the `split` directive is directly unfoldable. When selected, two test cases are created according to each predefined scenario. The generation of test sequences then gives for instance two sequences similar to sequences 3 and 4 above.

This directive may also be seen as an integration testing technique. When declared in an embedded node, it defines the integration strategy of related outputs very early in the development process, by stating functional scenarios for them. In such situations, unfolding techniques are complementary since

the activation of `split` directives requires the attached variable to be present in the current constraint system.

6 Performances

GATeL performances allow complex Lustre descriptions to be treated. These performances are illustrated on one of the test objectives of the SRIC case study [1]. This objective needs test sequences of at least 1000 cycles with complex constraints: boolean constraints involving comparisons and timers whose parameters are constrained by the same constraints at previous cycles. The system is composed of at least 7000 constraints - this number represents the maximum number reached, since it dynamically evolves during resolution - over a thousand cycles. It takes GATeL 10 seconds and 7 mega-bytes of memory to compute adequate test sequences (on a PC 1.4GHz Linux platform with 512 MB of memory). The first 7 seconds are used to build and propagate the initial constraints system creating one thousand cycles with adequate valuations of inputs, then test generation takes only the remaining 3 seconds, of which 2.9 seconds are used for the evaluation of the expected outputs.

These performances were confirmed on another complex case study treated by the IRSN (the French nuclear verification authority) in the BE-SECS European project. One of the test objectives is reached in 80 cycles, each containing 247 procedures call, and involving 10,000 constraints altogether. The generation of test sequences takes 1 minute and 18 mega-bytes of memory.

7 Conclusion

GATeL provides several basic mechanisms to define selection strategies: generation of a sequence leading to a test objective, interactive domain splitting from predefined operators sub-cases or user-defined subdomains. As discussed in the introduction, unlike other approaches in model-based testing [6,11,17,19], no particular strategy has been defined. Our choice was to allow the users to finely tune their own selection strategies according to the testing context. Classical strategies for unit or integration testing can nevertheless be implemented through unfolding of operators and split directives. Branch testing or bounded path testing can be achieved by a systematic unfolding of boolean operations. A systematic unfolding of comparisons and boolean decisions is a way to provide some boundary testing. Concerning integration testing, sub-cases defined by split directives give a functional decomposition which abstracts the behavior of the integrated components.

On the other hand, when the tester suspects some special types of behav-

ior to be badly implemented, it is important to provide him or her with the means to focus on the relevant parts of the model. The fine-tuning of predefined operator case analysis combined with interactive unfolding and split directives make it possible to discard many irrelevant details during test case selection. We are aware that the use of GATeL requires some understanding of the propagation mechanism, particularly during interactive domain splitting. This difficulty has been alleviated by the many ways in which information from GATeL's constraint store is made available to the user (analysis and pretty-printing of current constraints, choice tree presentation, ...) and by navigation facilities between the various representations.

The handling of real and floating point numbers in GATeL is still being studied. This point introduces difficulties of various sorts. The kernel has to be extended so as to manage constraints on these new types. (A French national research action, V3F is focused on the design of a constraint solver for floating point number arithmetic.) Moreover, these data types introduce observability problems for the oracle step: computation accuracy may change outputs at any cycle, and may cause some temporal shifts (e.g.: comparisons of floating-point numbers as timing parameters).

The activation of split directives may require many operators to be unfolded leading to undesired leaves in the test tree. In further work, we will study the means to automate this unfolding process in order to get only the single leaf concerning this activation. This would allow deeply embedded critical components to be thoroughly tested.

In GATeL, a test case is characterized by a partial instantiation of relevant data flows and a constraint system. Given a test sequence provided by an external source, it is possible to check whether such a test case is covered or not by this sequence. This can be achieved through sequence overlapping and constraint resolution. Generalized to several test cases and sequences, this could be used to qualify these sequences for acceptance testing or certification purposes.

References

[1] Collective contribution. Opération 2 : vérification et génération de tests pour un système de comptage de neutrons. *Action Forma, first year report*, Paris, January 1998.

[2] F. Baray, P. Codognet, D. Diaz, H. Michel. Code-based test generation for validation of functional processor descriptions. *Proc. TACAS'03*, LNCS 2619, Springer-Verlag, Jan. 2003.

[3] G. Bernot, M.C. Gaudel, and B. Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[4] L. du Bousquet and N. Zuanon. An overview of Lutess, a specification-based tool for testing synchronous software. *14th IEEE International Conference on Automated Software Engineering*, USA, October 1999.

[5] L du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: A Specification-Driven Testing Environment for Synchronous Software. *21st International Conference on Software Engineering*. ACM,May 1999.

[6] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME'93: Industrial-Strength Formal Methods, First International Symposium of Formal Methods Europe*, Springer Verlag, *LNCS* volume 670, pages 268–284, Odense, Denmark, April 1993.

[7] C. Jard, T. Jéron. TGV: theory, principles and algorithms. *Sixth World Conference on Integrated Design & Process Technology (IDPT'02)*, June 2002

[8] A. Gotlieb, B. Botella, M. Rueher. A CLP Framework for Computing Structural Test Data. *Constraints Stream, First International Conference on Computational Logic (CL2000)*, July 2000.

[9] S.-D. Gouraud, A. Denise, M.-C. Gaudel and B. Marre. A new way of automating statistical testing methods. *Sixteenth IEEE Int. Conf. on Automated Software Engineering (ASE 2001)*,pages 5–12, IEEE Computer Society Press, November 2001.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The Synchronous Data Flow Programming Language Lustre. *Proceeding of the IEEE*, 79(9): 1305–1320, September 1991. pages 229–237, IEEE Computer Society Press, 1991.

[11] B. Legeard, F. Peureux. Generation of functional test sequences from B formal specifications - presentation and industrial case study *Proc. of ASE'01 , International Conference on Automated Software Engineering*,pages 377-381, IEEE Computer Society Press,2001.

[12] B. Marre. Toward automatic test data set selection using algebraic specifications and logic programming. *ICLP'91, Eighth International Conference on Logic Programming*, pages 25–28, Paris, France, 1991. MIT Press.

[13] B. Marre, A. Arnould. Test Sequences Generation From Lustre Descriptions: GATeL. *ASE'00, Fifteen IEEE Int. Conf. on Automated Software Engineering*, pages 229–237, IEEE Computer Society Press, 1991.

[14] B. Marre, P. Mouy, N. Williams. On-the-fly generation of structural tests for C functions. *Proc. ICSSEA'03* ,Paris, December 2003.

[15] A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming *Proc. Formal Approaches to Testing of Software (FATES'01)*, pages 47-60, August 2001.

[16] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic Testing of Reactive Systems. *19th IEEE Real-Time Systems Symposium*, IEEE, 1998.

[17] P. Stocks, D.A. Carrington. Test Templates: A Specification-Based Testing Framework. *ICSE'93, Fifteen Int. Conf. on Software Engineering*, pages 405-414, IEEE Computer Society / ACM Press, 1993

[18] P. Thevenod-Fosse. Unit and Integration Testing of Lustre Programs: A Case Study From the Nuclear Industry. *9th European Workshop on Dependable Computing (EWDC-9)*, pages 121–124, May 1998. LAAS report 98078.

[19] R.G. de Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer*, 2(4): 382-393, March 2000.

[20] L. Van Aertryck, M. Benveniste, and D. Le Metayer. Casting: a formally based software test generation method. *proc. IEEE Int. Conference on Formal Engineering Methods*, pages 101–111, 1997.