29

A tool for testing synchronous software

I. Parissis
Laboratoire de Génie Informatique, Institut IMAG
BP 53 38041 Grenoble Cedex 9, France
Phone : (33) 76 82 72 57, Fax : (33) 76 82 72 87
E-mail : Ioannis.Parissis@imag.fr

Abstract

We present a testing tool for synchronous reactive software, an interesting subclass of critical real-time software. Its aim is to provide a new means for critical software validation since formal verification techniques are often impracticable. For the particular domain of synchronous reactive software, we have designed specific testing techniques which are loosely related to techniques proposed for sequential programming languages. One of the main characteristics of synchronous reactive software is the importance of the environment behavior in the validation process. Moreover, software requirements are usually expressed by means of temporal properties. The tool comprises random testing and specification-based testing techniques which have been designed to take into account these particularities. A structure-based testing technique well adapted to data-flow languages is also described.

Keywords

Software testing, formal methods, automated validation, quality of synchronous software.

1 INTRODUCTION

With the increased use of software controls in critical real-time systems, the need for rigorous methods at each step of the development process has become clear to all companies working on trustworthy systems. The aim of such methods is to provide confidence in the critical software, for instance by ensuring that it will behave in obedience to some properties (typically safety properties).

This paper is concerned with *synchronous reactive* critical software. A reactive software continuously reacts with its environment and must satisfy temporal constraints so that it can take into account all the events issued by this environment. The *synchrony hypothesis* (Benveniste and Berry, 1991) for a software requires that every reaction of the software to a set of

S. Bologna et al. (eds.), *Achieving Quality in Software*

inputs is theoretically instantaneous. In fact, this requirement is filled by any reactive software for which it is possible to prove that its environment is invariant during every reaction.

An important feature of reactive software is that it is developed under assumptions about the possible environment behavior. When such assumptions are formally expressed they can be easily integrated in an automated software verification process.

Special programming languages have been designed for the development of synchronous software, such as Esterel (Boussinot and De Simone, 1991) and LUSTRE (Halbwachs et al., 1991a). When a software is implemented in one of these languages it is often possible to perform a formal verification of some safety properties by model-checking. This verification technique, which can be fully automated, requires the software to be portrayed as a logical model and the safety properties to be expressed as logical formulae. Then, if the formulae are true in the model, it is assumed that the safety properties hold in the software. For instance, LESAR (Halbwachs et al., 1992) is a tool dealing with LUSTRE programs allowing automatic verification of safety properties. This tool uses LUSTRE as an executable specification language (for describing the model) as well as a linear temporal logic (Pilaud and Halbwachs, 1988) (for expressing the safety properties).

The main drawback of such formal verification techniques is that they often require prohibitive memory and time amounts. When they fail (because of lacks of memory and/or time) they do not provide any information about the satisfaction of the safety properties. Thus, it is clear that new - complementary - verification and validation approaches have to be developed.

The use of rigorous software testing techniques is the approach proposed here. Testing usually requires less memory and time amounts than formal verification and, hence, it is often the only means to perform software validation when formal verification is impracticable. Moreover, testing can reveal discrepancies between the model on which formal verification is carried out and the real world.

The tool presented in this paper provides a formal framework for testing synchronous software. Although it is based on the use of the LUSTRE language, most of the facilities it offers do not require the software to be implemented in that specific language. Therefore, the tool, of which the principles have been presented in (Ouabdesselam and Parissis, 1994a) and (Ouabdesselam and Parissis, 1994b), can be used for a wide range of synchronous applications. It offers three main testing facilities : constrained random testing, specification-based testing and structure-based testing. They are respectively presented in sections 2, 3 and 4 while section 5 describes how the tool should be used.

2 CONSTRAINED RANDOM TESTING

Random software testing consists in executing the software with input values which are randomly selected from the software input domain. The input domain is usually considered to be equal to the cartesian product of the domains of the input variables of the software.

Such a definition of the input domain is not suited to the test of most reactive software since it overlooks the environment description.

Let's consider the following example of a temperature control system developed in (Atlee and Gannon, 1993). This system is composed of a heater, an air conditioner, temperature sensors and an on/off switch. For the present paper, we introduce a reactive software for controlling that system.

Table 1 SCR Requirements for the Temperature Control Software.

Current Mode	*Running*	*BelowDesired Temp*	*TempOK*	*AboveDesired Temp*	*New Mode*
OFF	@T	**f**	t	**f**	INACTIVE
	@T	t	**f**	**f**	HEAT
	@T	f	**f**	t	AC
INACTIVE	@F	-	-	-	OFF
	t	@T	**@F**	**f**	HEAT
	t	**f**	**@F**	@T	AC
HEAT	@F	-	-	-	OFF
	t	**@F**	@T	**f**	INACTIVE
AC	@F	-	-	-	OFF
	t	**f**	@T	**@F**	INACTIVE

The software has four boolean inputs and four boolean outputs :

- The input *Running* is true when the on/off switch of the device is to the 'on' position.
- The inputs *BelowDesiredTemp*, *TempOk* and *AboveDesiredTemp* are mutually exclusive signals issued by temperature sensors; they are true when the current temperature is respectively lower, equal or higher than the desired temperature.
- *OFF*, *INACTIVE*, *HEAT* and *AC* are the four mutually exclusive outputs and correspond to the current *mode* of the system. *OFF* is true when the system is turned off, *INACTIVE* is true when the system is on but neither the heater nor the air conditioner is on, while *HEAT* and *AC* are true when the heater or the air conditioner, respectively, is on and controlling the temperature.

The software specifications are given in (Atlee and Gannon, 1993) as SCR tabular requirements (see table 1). Every row in the table 1 is associated with a *transition* : the left column contains the current mode of the system, while the right column contains the new mode computed with respect to the current input values given in the central columns of the table. @T means that at the instant when the new mode is computed, the associated input value raises from false to true. Similarly, @F specifies the point in time when the value becomes false. A lower case letter 't' (resp. 'f') means that the associated input value is true (resp. false) at this instant and at the instants immediately preceding and following it. For example, the eighth row of the table 1 states that when the system is turned on, if its current mode is *HEAT* and the temperature becomes equal to the desired temperature (i.e. *TempOK* becomes true) then the new system mode is *INACTIVE*.

The input values given in bold characters do not belong to the initial specification but are computed according to the *environment constraints* : the first assumption on the environment is that (I) exactly one of the three inputs *BelowDesiredTemp*, *TempOK* and *AboveDesiredTemp* is true at the same time. The second less obvious assumption on the environment can be deduced from the requirements specification : (II) the temperature cannot raise (resp. fall) from a value below (resp. above) the desired temperature to a value above (resp. below) it

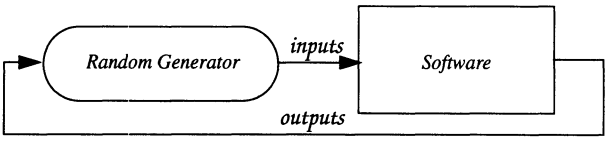

Figure 1 Random test data generator.

without reaching in the meantime the desired temperature. Indeed, no transition is specified in table 1 allowing the system to progress directly from the *HEAT* (resp. *AC*) mode to the *AC* (resp. *HEAT*) mode. As shown below, these environment constraints are easily expressed in LUSTRE, used as a temporal logic :

(I) *(AboveDesiredTemp* ***or*** *BelowDesiredTemp* ***or*** *TempOK)* ***and***
#(AboveDesiredTemp, BelowDesiredTemp, TempOK)

(II) *once_from_to(TempOK, BelowDesiredTemp, AboveDesiredTemp)* ***and***
once_from_to(TempOK, AboveDesiredTemp, BelowDesiredTemp)

where # is a built-in boolean operator ensuring that no more than one of its arguments is true at the same time, while *once_from_to(A, B, C)* is a user-defined in LUSTRE temporal operator returning a true value when the event *A* has occurred at least once between two subsequent occurrences of *B* and *C*. The definition in standard LUSTRE of such temporal operators can be found in (Halbwachs et al., 1992).

It is easy to conclude that the actual software input domain is much smaller than the cartesian product of the domains of input variables. The property (I) restricts the valid input values for *(AboveDesiredTemp, TempOK, BelowDesiredTemp)* to *(0, 0, 1)*, *(0, 1, 0)* or *(1, 0, 0)*. Moreover, the property (II) restricts the valid *sequences* of input values. The random generator the construction principles of which are outlined below performs a random generation of test data among all those defined by the environment constraints (e.g. (I) and (II)). It is used as a simulator of the software environment, as shown in Figure 1.

The first step of the generator construction consists in compiling the constraints into a finite state automaton. This automaton recognizes all input and output value sequences satisfying the constraints (in the same way that an automaton recognizes words from a language). Thanks to a process used by the LUSTRE compiler (Halbwachs et al., 1991b) (Bouajjani et al., 1990), the generated automaton is minimal. Moreover, only a symbolic representation of the automaton is actually generated in which states are represented by a set of variables, and transitions by boolean functions.

A state corresponds to one or more different values of the state variables. The next value of each state variable is computed from the current values of the state, input and output variables by means of the associated boolean function. The computation of the next state is based on the use of all these boolean functions.

A boolean function is also associated with the automaton. It is used to check whether the environment constraints are satisfied for a given state and a given value of the inputs and outputs. This boolean function is implemented by a binary decision diagram (Akers, 1978) (Bryant, 1986).

Such a symbolic representation of the automaton needs generally a quite smaller amount of memory than the usual representation in which states and transitions are explicitly generated.

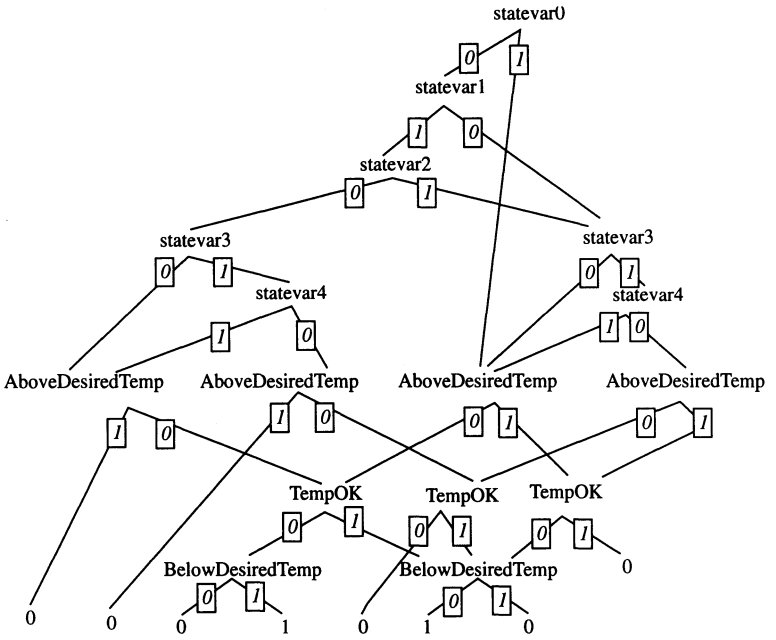

Figure 2 Binary decision diagram for the environment of the Temperature Control System.

For instance, the automaton associated with the constraints (I) and (II) of the above example has five boolean state variables. The binary decision diagram describing the environment constraints is given in Figure 2 (where *1* and *0* respectively hold for true and false). Each node of the diagram carries a variable and each of its outgoing branches is labelled with the value taken by that variable. It must be noted that the variables are ordered. The lower order variables are the state variables. They appear at the top of the diagram; for example in the diagram in Figure 2 the state variables are *statevar0*, ..., *statevar4*. Next come output variables while input variables occur at the bottom of the diagram. In the same example (Figure 2), there are no output variables.

The test data generator uses the above diagram to randomly generate input values for the temperature control software. It consists of a loop composed of the following steps :

1. Locate, in the diagram describing the environment constraints, the subdiagram corresponding to the current values of the state and the software output variables.
2. Generate a random value for the software inputs satisfying the boolean function associated with that diagram.
   (The first two steps are carried out in only one pass on the diagram)
3. Read the new software outputs.
4. Compute the next state by computing the next value of each state variable.

In other words, the generator searches in the diagram associated with the constraints a path leading to *1*. In the example (Figure 2), when *statevar0* is true, the generator will search a path in the subdiagram given in Figure 3. Possible values for the input variables

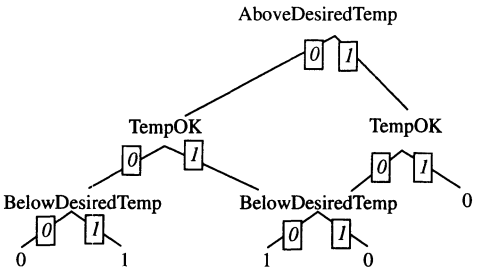

Figure 3 Subdiagram for statevar0 = 1.

(AboveDesiredTemp, TempOK, BelowDesiredTemp) are *(0, 0, 1)*, *(0, 1, 0)* and *(1, 0, 0)*. Each of these values will be generated with the same probability (*1/3* for the example of Figure 3).

3 SPECIFICATION-BASED TESTING

The test data generation technique presented in section 2 is not concerned with the problem of detecting errors occurred during the actual test operation. In the particular case of critical software, we're often interested in tracking down errors related to violation of safety properties. LUSTRE is a high level programming language which can be interpreted as a temporal logic, the most usual and natural means for specifying safety properties. In the example of the temperature control software, some safety properties supplied by the authors are :

1. *OFF* $\Rightarrow$ *~Running*
2. *INACTIVE* $\Rightarrow$ *(Running & TempOK)*
3. *HEAT* $\Rightarrow$ *(Running & BelowDesiredTemp)*
4. *AC* $\Rightarrow$ *(Running & AboveDesiredTemp)*
5. *(Running & BelowDesiredTemp)* $\Rightarrow$ *(HEAT|O(HEAT))*
6. *(Running & AboveDesiredTemp)* $\Rightarrow$ *(AC|O(AC))*

   where *O* is the 'next state' operator : *O(A)* means that *A* will occur at the next system state.

The translation in LUSTRE of the first four properties is straightforward :

1. ***not** OFF **or not** Running*
2. ***not** INACTIVE **or** (Running **and** TempOK)*
3. ***not** HEAT **or** (Running **and** BelowDesiredTemp)*
4. ***not** AC **or** (Running **and** BelowDesiredTemp)*

On the contrary, the translation of the last two properties is more difficult because they use the 'next state' temporal operator. Indeed, in LUSTRE only references to the past values of an expression are possible by means of the ***pre*** operator, denoting the value of the expression at the previous system state. However, since only two successive system states are involved in the property, it is possible to translate the properties to equivalent LUSTRE expressions :

5. ***not pre** (Running **and** BelowDesiredTemp) **or** (**pre** HEAT **or** HEAT)*
6. ***not pre** (Running **and** AboveDesiredTemp) **or** (**pre** AC **or** AC)*

```
node TempControlOracle(
        Running,                        -- ON/OFF switch position
        BelowDesiredTemp,               -- current temp. < desired temp.
        TempOK,                         -- current temp. = desired temp.
        AboveDesiredTemp : bool         -- current temp. > desired temp.
        OFF,                            -- off mode
        INACTIVE,                       -- stand-by mode
        HEAT,                           -- heating mode
        AC: bool                        -- air conditioning mode)
returns(  PropertiesOK : bool);
let
  PropertiesOK = (not OFF or not Running) and
                 (not INACTIVE or (Running and TempOK)) and
                 (not HEAT or (Running and BelowDesiredTemp)) and
                 (not AC or (Running and BelowDesiredTemp)) and
                 (not pre (Running and BelowDesiredTemp) or (pre HEAT or HEAT)) and
                 (not pre (Running and AboveDesiredTemp) or (pre AC or AC))
tel;
```

Figure 4 Test oracle for the temperature control system.

Once the safety properties have been specified, a test oracle can be automatically constructed in a straightforward manner as a LUSTRE program of which the unique output is the conjunction of the above safety properties (see Figure 4). A violation of the safety properties (i.e. an error) is detected when this output takes a false value during the software execution.

However, the actual aim of the specification-based testing is to analyze these properties and to automatically generate relevant input values (that is, input values which are more appropriate for tracking down errors related to the violation of safety properties).

In order to illustrate the definition of relevant input values, let's consider the simple property ***not A or B*** (meaning $A \Rightarrow B$), where A is a software input and B is a software output, stating that the output B must be true every time that the input A is true. One can easily notice that input values setting A to false cannot detect a violation of the property, since in that case ***not A or B*** will be true for any B. In other words, only input values for which A is true are *relevant* with respect to that property. Let R be the predicate characterizing the relevant input values; $R(\textbf{not } A \textbf{ or } B) = A$.

A similar, more complex but automated, analysis can be carried out on every LUSTRE boolean expression E. This is in particular true for all the user-defined in Lustre temporal operators (e.g. *once_from_to* used in section 2). It results in a new LUSTRE boolean expression $R(E)$ which will be true only when the values of the input variables are relevant with respect to the property E.

It is then possible to automatically generate relevant input values with respect to a property E. The generation process is similar to the one used in section 2 for constrained random testing. Indeed, the latter consists in building a generator of input values according to a set of environment invariant properties, while the former builds a generator according to the expression $R(E)$ characterizing the relevant input values for the property E.

Another use of the above definition of relevant input values is the assessment of the adequacy of the input values used for testing the software. Indeed, it is very easy to write a LUSTRE program counting the number of input values for which the formula characterizing the relevant ones has been true. At the end of the testing process, the ratio between the relevant and the total input values could be a useful measure of the quality of the testing process.

4 STRUCTURAL TESTING OF LUSTRE PROGRAMS

The approaches presented in sections 2 and 3 are black-box testing techniques and should be used for the detection of design errors resulting in a violation of the specifications (which are safety properties in the case of the technique of the section 3). On the contrary, implementation errors are usually tracked down by means of structural testing techniques. We consider that specifying functional features with LUSTRE is a kind of programming activity resulting in programs which are not exempt from errors. Hence, it is natural to adapt structure-based techniques to the particular case of LUSTRE programs.

Control graphs are the most common representations for programs written in sequential programming languages. Every node of the control graph consists of a sequence of successive instructions without branch statements; the latter are associated with arcs. Such a graph is a suitable representation of the control flow of the program. Structure-based testing consists in defining coverage criteria on the above graph. For instance, instruction coverage consists in running the program until all the nodes of the control graph has been executed, while branch coverage requires all the arcs of the graph to be executed (see for example (Ntafos, 1988)).

Due to the data-flow nature of the language, LUSTRE programs are represented by an operator net instead of a control graph. Indeed, a LUSTRE program (also called *node*) is an unordered set of equations defining the data flow of the program as invariant relations between inputs and outputs. Therefore, an operator net is a more natural representation of the program structure. We believe that defining structure-based criteria on such a net, by analogy with those defined on a control graph, could be useful for detecting implementation errors in LUSTRE programs.

The formal definition of such structure-based criteria can be found in (Ouabdesselam and Parissis, 1994b).

Let's consider the LUSTRE program with its associated operator net given in Figure 5.

```
node Edge(X: bool) returns(EDGE: bool);
let
  EDGE = X -> (X and not pre(X));
tel
```

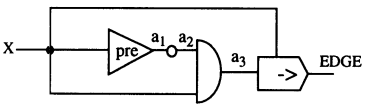

Figure 5 A node example and its operator net.

The operator **->** (the "followed-by" operator) allows to set a variable to an initial value : *EDGE = X -> (X **and not pre**(X))* means that the output variable *EDGE* takes at the initial state (the first execution step) the value of the input *X*. At every other state, *EDGE* takes a true value every time that the value of *X* raises from false to true (when *X* is true and ***pre** X* is false).

A *path* is defined as a finite sequence of successive arcs from the net. Its first arc is a net input and its last arc is a net output. For example, three paths are defined on the net of the node *Edge* : p_1 = *(X, EDGE)*, p_2 = *(X, a_1, a_2, a_3, EDGE)* and p_3 = *(X, a3, EDGE)* where a_1 = ***pre**(X)*, a_2 = ***not** a_1* and a_3 = a_2 ***and** X*. A *path predicate at instant t* associated with a path *p*, noted *PP(p, t)*, is the condition for the last arc to be computed at instant *t using* the path

p. In other words, it is the condition for the path p to be *traversed* by the data-flow when its last arc is traversed at instant t. Path predicates for the node *Edge* are :

$PP(p_1, t) = PP((X, EDGE), t) = init$,

$PP(p_2, t) = PP((X, a_1, a_2, a_3, EDGE), t) = X$ ***and not*** $init$

$PP(p_3, t) = PP((X, a3, EDGE), t) =$ ***not pre***(X) ***and not*** $init$ where $init =$ ***true -> false***

Structure-based test data selection criteria can be defined on such a net. For instance the *operator coverage* is satisfied by a test data set if at least an output arc of each operator belongs to one of the resulting traversed paths (i.e the operator has been activated and its results has been used for computing the output of the path). Similarly, *arc coverage* is satisfied if every arc of the operator net belongs at least to one resulting traversed path. *Path coverage* is generally impossible to satisfy since the number of paths may be infinite. However, the number of paths is finite for a *fixed length* of the test data sequences. Note that paths are defined in such a manner that errors occurring during program execution will be propagated to the output values, even several executions cycles after their effective occurrence.

Once structural test data selection criteria has been defined, we suggest a method for computing test data in order to satisfy these criteria. This computation performs a symbolic evaluation of the associated path predicates.

The method is based on previous work made on formal verification of LUSTRE programs. A verification tool, LESAR (Halbwachs et al., 1992), has been developed, allowing to automatically prove that a property always holds on a LUSTRE program. An interesting feature of LESAR is that it provides a counter-example for properties which do not hold (i.e. an input sequence leading the program to a state violating the property). The method comprises three steps :

1. Computing a finite set of paths which must be executed in order to satisfy the criterion. Let $(p_i)_{i=1...n}$ be these paths.
2. Computing the path predicates $(PP(p_i, t_i))_{i=1...n}$ associated with the above paths where t_i has a fixed value for every path.
3. Attempting to prove with LESAR that ***not*** $PP(p_i, t_i)$ for $i=1...n$ always hold. In case of success, the path p_i is infeasible at instant t_i (another path or another instant must be chosen). Otherwise LESAR provides an input sequence which can be executed to cover the path p_i. The length of the generated input sequences will be minimal, since LESAR will search the shortest counter-example. It is however possible to generate longer input sequences. For this, consider the following LUSTRE equations : $X1 =$ ***true***; and for every $n > 1$, $Xn =$ ***false*** $->$ $Xn\text{-}1$ ***and pre***$(Xn\text{-}1)$. In other words a variable Xn denotes a sequence of boolean values of which the $n\text{-}1$ first terms are equal to ***false*** while all the other terms are equal to ***true***. Attempting to prove with LESAR that ***not*** $(PP(p_i, t_i^0)$ ***and*** $Xt_i^0)$ always hold, where t_i^0 is the length of the required test data sequence, will result, if the path p_i is not infeasible, in an input sequence of which the length will be at least equal to t_i^0. Indeed, thanks to the definition of Xn, every property ***not*** $(P$ ***and*** $Xn)$ will always be true during the first $n\text{-}1$ execution cycles. Hence, the counter-example provided by LESAR, if any, will be a sequence longer than $n\text{-}1$.

It must be noted that the verification tool LESAR is used here in a quite different manner than for its original purpose which is formal verification. Indeed, when a formal verification is performed, the entire automaton (i.e. all possible states) must be explored in order to prove that properties hold at every state. On the contrary, the operation of test data generation consists in

searching a counter-example of the negation of the path predicate. This is usually a very short operation (unless, of course, the path predicate is infeasible). Thus, there is no contradiction in the use of a formal verification tool for testing purposes.

5 USING THE TOOL

The testing tool includes all the testing facilities described in the previous sections. In this section we present the parameters that the user should specify for each kind of testing. We also suggest a methodology for performing test in a progressive way.

Random testing requires the user to fix at least two parameters : the environment specification as a LUSTRE boolean expression and the software to test (executable form). Other optional parameters are the number and the length of the generated input sequences.

Specification-based testing also requires two parameters : the specification of the safety properties as LUSTRE boolean expressions and the software to test. The length and the number of the generated sequences can also be specified.

Finally, structure-based testing requires the user to specify the implementation in LUSTRE of the software to test. An executable version of the software is also required. Three mutually exclusive coverage types are available : operator, arc and path coverage. The required coverage rate can be selected (the default value is 100%). If path coverage is selected, the maximum length of the paths to cover can be specified.

For any of the three kinds of testing a test oracle can be specified. Moreover, it is possible to combine the environment specification (for example the one used for random testing) with the specification-based testing or the structure-based testing. When an environment specification is provided for this techniques, the generated test data will also satisfy this specification.

At the end of any of the above testing operations the test results are stored in a file. They are composed of structured sequences of the input and output values produced during the test operation. For each input and output value the result of the oracle (if any) is also stored. The user can browse these results by listing the entire result file, by selecting specific sequence numbers or by listing sequences for which the oracle (if any) has taken a given value.

The user can choose one or more of the testing facilities in any order. However, we suggest the following methodology :

First, the test oracle and the environment specification must be written out.

Then, constrained random testing should be performed in order to get confidence in the test oracle and the environment specification. Indeed, errors detected during this stage are often caused by a bad specification of the environment constraints or by an erroneous oracle. Random testing should continue until the user is confident enough in the correctness of environment and test oracle specification.

When no more errors are discovered by random testing, specification-based testing should be performed in order to detect discrepancies between the software behavior and the specification of the safety properties. The LUSTRE expression of the safety properties of the software must be written (unless they have already been expressed for the test oracle). The environment constraints and test oracle developed for random testing are also used during specification-based testing.

Finally, structure-based testing can be performed if an implementation in LUSTRE of the software is available. Structure-based testing is useful to track down implementation errors,

not necessarily related to safety properties, since is it is based on a more precise specification of the software intended behavior (the LUSTRE program).

Note that the environment constraints can be omitted if the user is interested in the software behavior in cases when the environment does not respond correctly.

6 CONCLUSION AND FURTHER WORK

We have presented in this paper a testing tool dealing with synchronous reactive software. It provides a new formal framework for critical software validation, complementary to the current formal verification techniques which are often impracticable.

Several testing techniques have been proposed in the literature, generally for sequential programming languages (see for example (Ntafos, 1988) (Dauchy and Marre, 1991)). Just a few works have been conducted in the particular domain of reactive software; they are more concerned with a testing methodology than testing techniques per se (Richardson, 1992). The techniques we have designed are specific in the sense that they deal with synchronous reactive software. Indeed, for such software the environment behavior is extremely important. Moreover, software requirements are usually expressed by means of temporal properties. Thus, specific random testing and specification-based testing techniques have been devised to cope with these particularities. Equally specific is the structure-based testing technique which is adapted to data-flow languages.

It should be noted that the last of the three proposed techniques requires the software to be implemented in LUSTRE. On the contrary, constrained random testing and specification-based testing can be applied to reactive software implemented in any programming language. Indeed, although they both use LUSTRE for the description of the environment or for the expression of the safety properties, they do not require this particular programming language to be used for the software implementation.

The extension of the proposed techniques to software with numerical inputs and outputs is the main challenge for future work. Although many reactive software handle boolean signals, such an extension would allow to enlarge the application field of the tool.

Another interesting extension of the tool that we're currently studying is the introduction of reliability estimation features. This will allow to measure the software failure probability which is the main attribute of critical software quality.

Finally, more theoretical work is needed for comparing testing with formal verification according to the required memory and time amount and the resulting software quality.

7 ACKNOWLEDGMENTS

We would like to thank Pascal Raymond of Verimag for his contribution to the modification of the LUSTRE compiler and for adapting to our needs the binary decision diagram library.

8 REFERENCES

Akers, S. (1978). Binary Decision Diagrams. *IEEE Transactions on Computers*, C-27:509–516.

Atlee, J. and Gannon, J. (1993). State-Based Model Checking of Event-Driven System Requirements. *IEEE Transactions on Software Engineering*, pages 24–40.

Benveniste, A. and Berry, G. (1991). The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9):1270–1282.

Bouajjani, A., Fernandez, J., and Halbwachs, N. (1990). Minimal model generation. In *Workshop on Computer-Aided Verification*, Rutgers (N.J.).

Boussinot, F. and De Simone, R. (1991). The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304.

Bryant, R. (1986). Graph-based algorithms for boolean functions manipulation. *IEEE Transactions on Computers*, pages 667–692.

Dauchy, P. and Marre, B. (1991). Test data selection from algebric specifications : application to an automatic subway module. In *3rd European Software Engineering Conference*, pages 80–100, Milan, Italy. Springer-Verlag L.N.C.S. 550.

Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. (1991a). The Synchronous Data Flow Programming Language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320.

Halbwachs, N., Lagnier, F., and Ratel, C. (1992). 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*, pages 785–793.

Halbwachs, N., Raymond, P., and Ratel, C. (1991b). Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany).

Ntafos, S. (1988). A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, pages 868–874.

Ouabdesselam, F. and Parissis, I. (1994a). Testing Safety Properties of Synchronous Reactive Software. In *7th International Software Quality Week*, San Francisco, USA.

Ouabdesselam, F. and Parissis, I. (1994b). Testing Synchronous Critical Software. In *5th International Symposium on Software Reliability Engineering*, pages 239–248, Monterey, USA.

Pilaud, D. and Halbwachs, N. (1988). From a synchronous declarative language to a temporal logic dealing with multiform time. In *Symposium on Formal Techniques in Real Time and Fault Tolerant Systems*, Warwick. Springer Verlag.

Richardson, D. (1992). Specification-based Test Oracles for Reactive Systems. In *14th Int'l Conf.on Software Engineering*, pages 105–118, Melbourne, Australia.

9 BIOGRAPHY

I. Parissis received the D.E.S.S. en Génie Informatique (Master's degree in Software Engineering) from Université Joseph Fourier, Grenoble, France in 1990 and the D.E.A. en Informatique (Master's degree in Computer Science) from Institut National Polytechnique de Grenoble in 1993. He is currently a Ph.D candidate (under a grant of the French Ministry of Research). He holds a teaching assistantship at Université Joseph Fourier and a research assistantship at Laboratoire de Génie Informatique of the IMAG Institute. His research interests include software V&V, software testing and reliability and formal methods for the development of safety critical and synchronous software.