Electronic Notes in Theoretical Computer Science 88 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume88.html

Testing against some eventuality properties of synchronous software: a case study

L. du Bousquet F. Ouabdesselam J.-L. Richier [1]

Laboratoire LSR-IMAG
BP 72, 38402 St Martin d'Hères, France

N. Zuanon [2,3]

Actel
7 chemin des Prés, 38240 Meylan, France

Abstract

In this article, we study a tentative approach to the problem of software validation against some eventuality properties in a synchronous context. This approach is based on an automated functional testing tool whose various testing methods are well-adapted to statistical predictions. The main results are drawn from a telephone feature validation benchmark for feature interaction detection.

1 Introduction

During the last decade, the growing interest in synchronous languages from large companies has initiated significant contributions to the practical validation problem of synchronous software. Contrary to many other areas, and thanks to the rigorous mathematical semantics of this approach, much of current synchronous software testing theory and practice is not built on wishful thinking: several specification-based testing methods have been designed, implemented and have shown to be effective at revealing errors [2,11,5,1,9]. Furthermore, all these methods allow to automate the test data generation process.

In this context, our previous works on testing against properties written in Lustre (viewed as a temporal logic) have mainly concerned safety constraints.

[1] Email: {lydie.du-bousquet,farid.ouabdesselam,jean-luc.richier}@imag.fr
[2] Email: nicolas.zuanon@actel.com
[3] This study was carried out while N. Zuanon's PhD work was supported by a contract between France Telecom R&D and the LSR laboratory.

©2003 Published by Elsevier Science B. V.

Indeed, testing to detect failures amounts to demonstrating safety property violation on finite program executions, which is a manageable process.

This paper examines a proposal for testing against *eventuality* formulas constructed with the "leads to" temporal operator [7]. These eventuality properties (of the form *requested* $\leadsto$ *served*) mean that if a process has requested a service then it is eventually served.

Tackling this problem with a testing approach is quite a challenge: a definitive verdict on the violation of such a property formally requires infinite executions. We propose a probabilistic approach in which the decision that an eventuality property is probably not met relies on some finite observations of the program under test behaviors and a comparison of the time lapses necessary to complete the property satisfaction. The proposal is exemplified on the detection of telephone service interactions. This paper concerns the empirical and statistical study of a particular telephone network model. The ways the results can be exploited to a more general approach to testing against eventuality properties is beyond the scope of this paper.

Section 2 is devoted to the presentation of the case study context. In section 3, we introduce the principle of our approach to test against eventuality properties. Section 4 describes Lutess, our testing tool, and its adaptation to test against eventuality properties. Section 5 briefly sets out the case study results and section 6 concludes.

2 Case study

2.1 Context: validation of telephone features

In this study, we make use of the same context as in previous experiments we have conducted on telephone service (feature) interaction detection [3]. A feature is a modification of the Plain Old Telephone Service (POTS), which is built on top of POTS.

The feature interaction problem

Incompatibility between features is referred to as feature interaction. Feature interaction occurs when the behavior of a new feature modifies or inhibits the behavior of one well-functioning existing feature and/or similarly when the pre-existing features prevent the new feature to behave as expected.

Applying a synchronous approach

To find feature interactions at a specification level, we modeled the whole telephone network, the POTS and the features as a synchronous reactive system, and ran this executable model. We produced several synchronous units which correspond to the POTS alone, the POTS with one feature, and the POTS with two features. All these programs were written in Lustre [4]. Their environments are composed of 4 users (see Fig. 1). Details about those programs

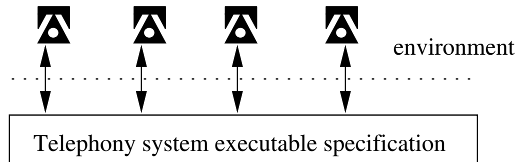

Fig. 1. Executable model

can be found in [3].

In order to detect incompatibilities among features, we expressed the service requirements as properties. They are stated also in Lustre. An interaction between two features A and B is declared if feature A (resp. B) alone with POTS satisfies its properties and if features A and B together with POTS do not satisfy anymore the conjunction of A and B properties. This feature interaction detection process has been carried out using a functional black-box testing approach; it has been implemented with Lutess [2].

The relevance of the whole approach has been demonstrated during the "First Feature Interaction Detection Contest" which was sponsored by the *Fifth Feature Interaction in Telecommunication and Software Conference*, in 1998 [6,3]. The goal of the contest was to compare different automated tools for detecting interactions from the feature requirements.

In this article, we use Chisel diagrams to describe the feature and the POTS behaviors. Chisel is a language for defining requirements for communication services [6]. Short descriptions of both the Chisel diagram principle and the POTS specification are given in appendices.

2.2 Does one really need eventuality properties ?

The First Feature Interaction Detection Contest provided the requirements for 12 features. We translated the requirements into executable specifications (synchronous automata) and properties. All the properties we expressed then were typically *safety* properties. For example, let us consider the Terminating Call Screening (TCS) and the Call Number Delivery (CND) features. TCS allows a subscriber to screen calls based on the originating number. A typical TCS safety property is that "a TCS subscriber *never* receives a call from a number which is in his screening list". CND enables the subscriber's telephone to receive and to display the number of the originating party on an incoming call. A safety property one can expect is that "when a CND subscriber receives a call, the number of the originating party is *always* displayed".

Let us study now two new features: Call Completion to Busy Subscriber (CCBS) and Return Call on Busy (RCB).

CCBS (see Fig. 2)
Let A be a CCBS subscriber. If user A tries to call user B when he is busy, user A can choose to *activate* the feature (he has to dial the "CCBS-code"). Then, as soon as both lines are idle, CCBS tries to establish the communication

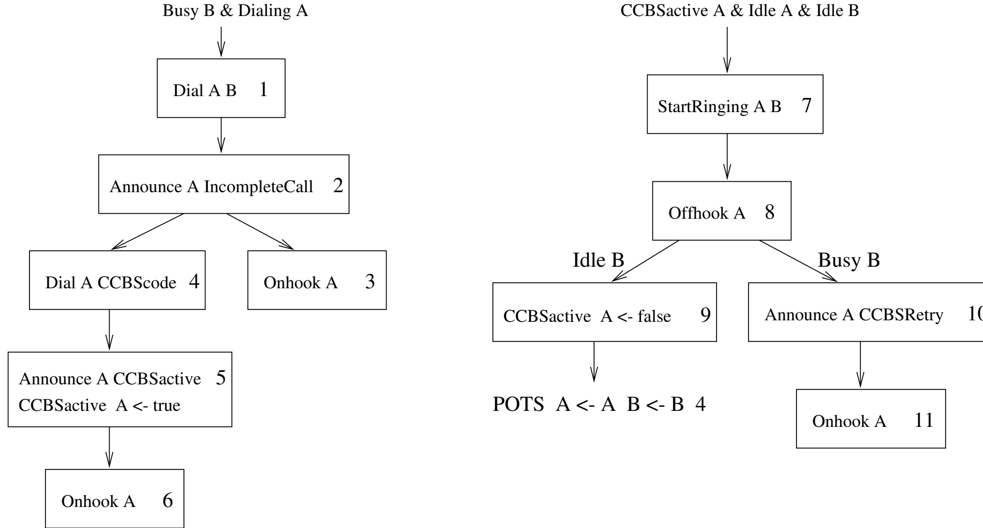

Fig. 2. Call Completion to Busy Subscriber feature specification

between A and B; the CCBS invocation condition is *Idle A and Idle B*. A CCBS invocation consists in several steps. First, user A's line starts ringing. When A off-hooks, if B is still idle, B's line starts ringing. At this stage, the CCBS invocation is considered as *successful*, and CCBS is automatically deactivated. If B is busy, CCBS remains active and a new CCBS invocation will be performed as soon as possible. For sake of simplification, we consider that CCBS feature can not be activated twice without being deactivated in between.

RCB (see Fig. 3)
Let B be a RCB subscriber. The feature automatically registers the incoming call numbers when B is busy. It tries to establish the call as soon as possible, that is when the condition *Idle A and Idle B* is fulfilled. RCB is similar to CCBS, since they both share the same invocation condition, but RCB activation is automatic (user B does not dial a code to activate the feature).

CCBS and RCB interaction: informal overview
In the executable specification, when CCBS (resp. RCB) is alone with POTS, the feature seems to work correctly (for a single CCBS subscriber). The elapsed time between the feature activation and a successful invocation can be long, but it is always finite. This time is counted as the number of ticks of the basic synchronous clock (equivalent to the number of execution cycles). However, when the features are put together with POTS, the elapsed time between the CCBS feature activation and a successful invocation "seems" to be infinite [4].

Indeed, let A be a CCBS subscriber and B a RCB subscriber. Let A call

[4] The observations were done on finite but long traces (1 000 000 execution cycles).

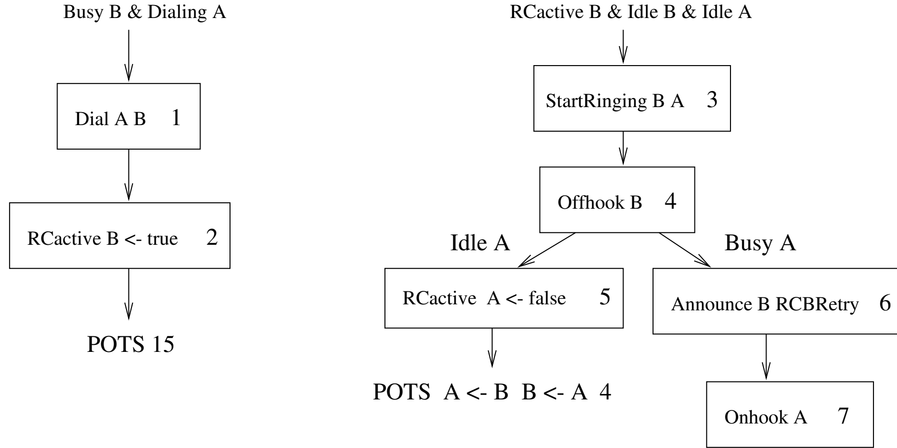

Fig. 3. Return Call on Busy feature specification

B when B is busy. Let A activate the CCBS feature (RCB is automatically activated). As soon as A and B are idle, both RCB and CCBS are invoked *simultaneously*. Since B (resp. A) is always busy when CCBS (resp. RCB) is in the Chisel diagram state number 8 (resp. 4), then the CCBS (resp. RCB) feature invocation never succeeds. The interaction in our executable specification appears in the form of a livelock.

Note that there is an interaction in our executable model because the features are invoked simultaneously. This interaction may not occur under different hypotheses (for instance if a priority between the features is set). In real world, one can consider that CCBS and RCB interact if successful invocations of CCBS (resp. RCB) are *sometimes* delayed.

CCBS and RCB interaction: formal detection

From CCBS and RCB requirements, it is possible to assert that both services will be invoked "as soon as possible". For instance, one can state the following safety properties:

- *CCBSactive(A) and Idle(A) and Idle(B)* $\Rightarrow$ *StartRinging(A,B)*
- *RCBactive(B) and Idle(A) and Idle(B)* $\Rightarrow$ *StartRinging(B,A)*

But these safety properties are not the relevant ones to indicate that the CCBS or RCB feature invocations will succeed in the end. Indeed, it is not possible *a priori* to fix how long it will take between the activations and the corresponding successful invocations.

Therefore, we need eventuality properties to mean that a feature will always lead to a successful invocation after it has been activated. The "leads to" temporal operator [8] ($\leadsto$) was adequate for this example[5].

For both CCBS and RCB features, one can write a property of the form: "*activation* $\leadsto$ *successful_invocation*". The CCBS feature is activated when the

[5] $p \leadsto q$ means that an event p should be followed by q in the future.

event *"Announce A IncompleteCall"* occurs, and the successful invocation is detected when the condition *"Off A and StartAudibleRinging A"* is observed. This condition is sufficient because, in the POTS specification, the environment event *"Off A"* produces either the *"DialTone A"* or the *"Talking A"* output event. Thus, we can state the following property:

$$\mathcal{P}_{ccbs} : \textit{Announce A IncompleteCall} \leadsto (\textit{Off A and StartAudibleRinging A})$$

3 Testing against eventuality properties: principle

3.1 What is the problem ?

We recall that a synchronous program has a cyclic execution. This execution can be observed on an execution trace. A trace is a sequence of pairs of the program related input and output values. A sequence is time-ordered from the first cycle up: each pair corresponds to one cycle number.

The truth value of any temporal property T to be satisfied by a program P can only be evaluated on P behavior. In a functional black-box testing approach, P behavior is represented by some execution traces. Usually the purpose of the functional testing techniques is to reveal errors rather than to prove that the system under test is correct.

Testing against a safety-like property [6] S can lie on a purely random or statistical generation of input data. It can also be based on input selection of the data which have the highest probability of leading the program under test into states where S can be violated. Testing against S is usually carried out on finite traces. Indeed, as soon as S is violated, a definitive verdict can be issued. This viewpoint is well-compatible with the main trait of safety-like properties: they are used to mean *"bad things which should not occur"*. If defaults are suspected, the primary goal is to detect them. On the opposite, demonstrating that P meets S by testing requires infinite execution traces. The verdict can be constructed on finite traces only if some hypotheses are stated and verified on both the program behavior and the input generation algorithm. The fair generation of all possible violation-prone inputs can be such a guarantee.

Most eventuality properties are used to state that *"something good will occur"*. Contrary to safety-like properties, these properties L can only be evaluated on infinite traces. Therefore, testing against L with respect to finite traces is a highly uncertain procedure. Let us take as an example, a formula based on the "leads to" temporal operator. The formula $p \leadsto q$ asserts anytime p is true, q is true then or at some later time [7].

On a finite trace, if the premise has been fulfilled, two cases are possible:

[6] These include all invariant properties, and eventuality properties the future of which is bounded by the occurrence of an event.
[7] "$p \leadsto q$ is equivalent to $\Box(p \Rightarrow \Diamond q)$" (whenever p is true, p will eventually become true).

- the conclusion is also observed. This allows to conclude that the property is only locally true; indeed, in the future, a new occurrence of p could not be followed by an occurrence of q.
- the conclusion is still not established at the end of the trace. It is not possible to know whether a longer trace would lead to the property satisfaction.

Thus, only a “partial” verdict of the property observance can be issued from the analysis of finite traces. Our objective is to determine a predictive and probabilistic verdict from both partial verdicts and levels of confidence in these latter pieces of information.

3.2 Proposal

For several reasons, a simple probabilistic prediction process fits rather well an eventuality property truthfulness evaluation based on the analysis of the program under test execution traces.

- Firstly, as long as the left-hand side condition is not met, the verdict is clear: the property is trivially true. Uncertainty appears only after the premise has been observed: as long as the right-hand side condition is not met, the verdict is unknown. However, the premise occurrence is an event which defines the origin of the sub-trace to be examined, and thus the starting point of a statistical analysis.
- Secondly, a tentative and temporary verdict can be issued in various circumstances, by reference to controlled situations. For example, one can predict the verdict with respect to some data which are provided by the software requirements document or which result from the software previous executions. At worst, one may assume that the very naive intuition that the longer the delay to get the conclusion the lower the probability for that conclusion to occur applies.
- Thirdly, these reference data are associated with some sort of “time to service”, i.e. the elapsed time in between the premise and the conclusion respective observances. This notion of “time to service” is directly inspired from typical applications of the temporal expression $\mathit{request} \Rightarrow \Diamond\ \mathit{response}$ which links the requested and served states of a service.

Therefore, we propose to tentatively measure the confidence in the truthfulness of the eventuality property on the remaining trace of the program under test execution which follows each premise occurrence[8]. This means of determining the level of confidence could be complemented and strengthened by pinpointing some additional conditions. These conditions would represent intermediary constraints which would mark out the progression of the service request towards its conclusion. In this case study, no such additional

[8] A premise or a conclusion occurrence corresponds to the instant at which the premise or conclusion condition becomes true.

conditions have been used.

The main problem then is to design a valid confidence index which would define the probability that a partial verdict is correct. To this end, we regard the "time to service" as a random variable which takes on its values over a discrete set of time points (each point corresponds to a tick of the synchronous clock). We also assume that, for each temporal property *request* $\Rightarrow \Diamond$ *response* to be evaluated, there exists a distribution law whose probability density function measures, at each instant during the testing process, the probability that this property is true, i.e. the probability that the response is going to be observed. This distribution law is a means to estimate the "time to service" of a pending request.

By simplification, we consider this variable as a continuous one. The time origin is the instant at which the request has been issued. Since the property is trivially true as long as the request is not observed, the density is considered as null over $]-\infty, 0[$. On the opposite, in order to keep consistent with the concept of liveness which says that "no situation is hopeless", the density integration over $[0, \infty[$ is equal to 1. The distribution function $F(t) = \int_0^t f(x)dx$ gives the estimated truth value of the property at time t. Therefore, the probability that the property is true, at time t, when the response has not been observed is measured by $1 - F(t) = \int_t^{+\infty} f(x)dx$.

Our last hypothesis is that this approach can be generalized from the analysis of "*request* $\Rightarrow \Diamond$ *response*" formula to the evaluation of "leads to" properties. Indeed, we can consider that the confidence decreases as long as the response is not observed, since it is expected that any new request occurrence does not modify this phenomenon (this assumption has been validated by the case study experiment).

Our goal is to get the data to construct such a distribution law in order to read the value of the confidence index directly of the curve. Ideally, the distribution law equation is provided using past experiences. Otherwise, the distribution curve is built from experimental data; in our case study, it is built from the validation of an isolated feature. This curve is used as a basis to determine whether there is an interaction among several feature including the validated one.

Thus, the whole approach, which is based on statistical predictions from sampling, makes sense and is well-founded if several conditions hold:

(i) inputs which drive the program execution during testing are "representative" of the program uses,

(ii) these uses guarantee that each value from the program input space will eventually be selected,

(iii) if appropriate samples are used for the test set, results on these samples stand in for future program behaviors,

(iv) program executions guarantee that all program operations that are possible eventually will be executed.

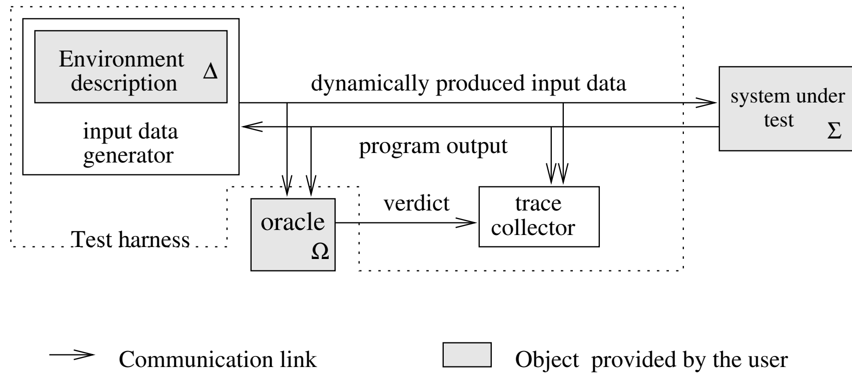

Fig. 4. Lutess

The second and the fourth conditions are fairness requirements which have to be met respectively by the test data generation process and the program behavior. To be complete, this approach must also make precise the type of fairness (strong or weak) to be implemented.

4 Using Lutess to test against eventuality properties

4.1 Lutess testing tool: an overview

Lutess [2,10] is the testing tool which we developed to validate reactive synchronous software. It requires three elements: an environment description written in Lustre (Δ), a program under test (Σ) and an oracle (Ω) providing the program requirements (fig. 4). Lutess builds a random generator from the environment description and constructs automatically a test harness which links the generator, the program under test and the oracle. Lutess coordinates their executions and records the sequences of input-output relations and the associated oracle verdicts, thanks to the trace collector. Components are just connected to one another and not linked into a single executable code.

The program under test and the oracle are both synchronous executable programs, with boolean inputs and outputs. Optionally, they can be supplied as Lustre programs.

The test is operated on a single action-reaction cycle, driven by the generator. The generator randomly selects an input vector for the program under test and sends it to this latter. The program under test reacts with an output vector and feeds back the generator with it. The generator proceeds by producing a new input vector and the cycle is repeated. The oracle observes the program inputs and outputs, and determines whether the software specification is violated. The testing process is stopped when the user-defined length of the test sequence is reached.

Basically, the Lutess generator selection algorithm chooses a valid [9] input

[9] An input is valid if and only if it is complying with the environment description.

vector in an equally probable way. In each environment state, any valid input vector has the same probability to be selected.

This method is not powerful enough when it comes to test systems in a complex environment: the realistic behaviors might be a small part of all possible behaviors with respect to the environment specification. For instance, we noticed that the use of this method for the contest results in each user dialing his own number as often as any other number. In reality, this behavior is quite rare, though possible.

To overcome this drawback, Lutess offers various facilities to guide the generation [2]. Several methods are proposed:

- a behavioral pattern-based guiding, which allows the user to define some classes of scenarios; the selection algorithm will favor sequences of inputs that match a scenario;
- an operational profile-based guiding, which allows the user to define input statistical (partial) distribution; the selection algorithm will produce the inputs according to the given distribution.

4.2 Testing against eventuality properties with Lutess

Let us examine how testing against the property: "*premise* $\leadsto$ *conclusion*" can be implemented.

To build the time distribution between premise and conclusion occurrences, a "counting program" can be plugged-in in the place of the oracle (Ω). This program returns an integer value which represents the number of instants in between the premise and the conclusion occurrences (`timetoserve`). This value is egal to zero most of the time. A local integer variable (`tts`) is used as a counter, which is incremented at each cycle, between the occurrence of the logical events `premise` and `conclusion`. The counter is reset to zero when `conclusion` is true. A simplified Lustre node illustrating the counting program built for CCBS validation [10] is given in figure 5.

Statistical predictions from sampling using Lutess is valid since the samples are appropriate:

- each guiding technique offered by Lutess can be used to associate occurrence weights to the inputs; the operational-profile based technique allows to represent a profile of actual or anticipated use of the software; therefore, the samples which are produced are representative of these uses, even when using random testing alone,
- the test data generation procedures integrated in Lutess guarantee that the inputs are taken without bias and that each input is infinitely often selected; even the scenario-based technique leaves room to random generation

[10] We recall that here the CCBS feature can not be activated twice without being deactivated in between.

of events which are not in the scenario and does not force the observance of the conclusion of an eventuality property after the premise has occurred.

Lutess drives executions which are long enough not to prevent some behaviors of continuously operating programs to take place.

```
node CountingPgm(input_0,..,input_n, outputs_0,..,outputs_m: bool)
returns (timetoserve: int);
var  premise, conclusion: bool; tts: int;
let
     premise = Announce_A_IncompleteCall ;
     conclusion = Off_A and StartAudibleRinging_A;
     tts= 0 -> if between(premise,conclusion) then (pre(tts)+1) else 0;
     timetoserve = 0 -> if conclusion then (pre(tts)) else 0;
tel;

node between(inf,sup: bool) returns (btwn: bool);
let
     btwn=false -> if inf then true else if sup then false else pre btwn;
tel;
```

Fig. 5. The counting program built for CCBS

5 Results of the case study

5.1 Discovering interaction using eventuality property

This first experiment has been conducted using only Lutess uniform (equally likely) random test data generation. It was composed of three stages:

(i) Collecting relevant data by operating an executable specification of POTS and CCBS:
- the number of premise occurrences
- the number of conclusion observances
- the elapsed time between each premise occurrence and its conclusion.

(ii) Collecting the same three categories of data by using an executable specification of POTS, CCBS and RCB.

(iii) Analyzing and comparing both data sets, in order to conclude about feature interaction.

Figure 6 was elaborated from the analysis of several traces of CCBS alone. The curve displays the distribution of the elapsed time between CCBS activations and their corresponding successful invocations. It allows to deduce a MTTS [11] of approximately 42 cycles.

[11] Mean Time To Service; a notion named after the Mean Time Between Failure concept in the reliability domain.

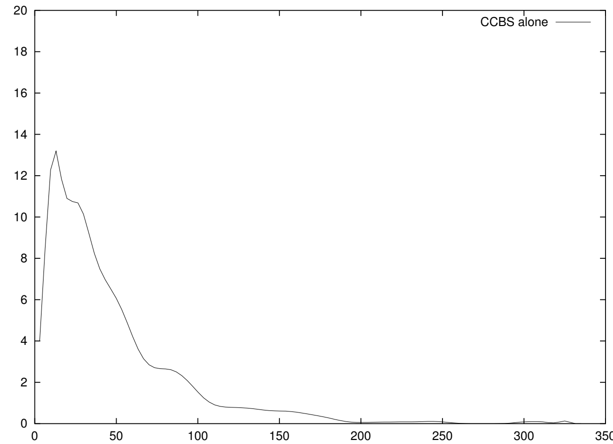

Fig. 6. Elapsed time distribution between CCBS activations and their corresponding successful invocations

For CCBS and RCB together, six long traces were produced. For all those traces, it was not possible to observe any successful invocation even after more than 100 000 steps following the last activation of CCBS.

Comparing both results, one can conclude that the CCBS eventuality property is not satisfied when CCBS is composed with POTS and RCB. Thus, there is a feature interaction, which has been previously explained (see section 2.2).

5.2 Statistical analysis

Several statistical inference test techniques have been applied to the observational material (Fig. 6). The elapsed time distribution does not correspond to any known law (Exponential, Weibull, Poisson, Normal, ...).

The p-quantiles of the elapsed time distribution have been studied for p=90%, 95%, 97.5%, and 99%, on the basis of 78 execution sequences. Using the Anderson-Darling test, one concludes that the distribution of these empirical p-quantiles is the normal law. So, the estimate of the expected time to service ranges from 103 cycles (for p=90%) to 202 cycles (for p=99%). This means that the probability that a request gets an answer after 202 execution cycles is 1%.

6 Conclusion and perspectives

For software applications in continuous operation, some requirements may take the form of *a request should always be followed by a response*. When the response delay is not restricted, this kind of requirement can be expressed as an eventuality property ($request \leadsto response$).

This paper has addressed the problem of validating synchronous applications against eventuality properties with a testing method. Our approach consists in measuring statistically the response delay and determining the predictive verdict of the property satisfaction together with its associated confi-

dence index.

We have applied this approach to telephony feature specification validation, and more precisely to detect feature interaction between the *Call Completion on Busy Subscriber* (CCBS) and the Call Return on Busy (RCB) features.

We are currently trying to determine if the CCBS elapsed time distribution follows a classical distribution law or a combination of such laws. In that case, indeed, we could be more precise in the confidence index definition. Furthermore, we could use this index as a stopping criterion: in the particular case of software testing against an eventuality property, one may decide to stop the test when this index has too small a value (0.01%, for instance).

References

[1] L. Arditi, A. Bouali, H. Boufaied, G. Clave, M. Hadj-Chaib, L. Leblanc, and R. de Simone. Using Esterel and Formal Methods to Increase the Confidence in the Functional Validation of a Commercial DSP. In *ERCIM workshop on Formal Methods for Industrial Critical Systems*, Trento, Italy, 1999.

[2] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess: a specification-driven testing environment for synchronous software. In *21st International Conference on Software Engineering*. ACM Press, May 1999.

[3] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Feature interaction detection using synchronous approach and testing. *Computer Networks and ISDN Systems*, 11(4):419–446, 2000.

[4] N. Halbwachs. Synchronous programming of reactive systems, a tutorial and commented bibliography. In *Tenth International Conference on Computer-Aided Verification (CAV)*, Vancouver (B.), June 1998. LNCS 1427, Springer.

[5] L.J. Jagadeesan, A. Porter, C. Puchol, J.C. Ramming, and L. Votta. Specification-based Testing of Reactive Software: Tools and Experiments. In *19th International Conference on Software Engineering*, 1997.

[6] K. Kimbler and L.G. Bouma, editors. *Feature Interactions in Telecommunications Systems V*. IOS Press, 1998.

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

[8] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(2):872–923, May 1994.

[9] B. Marre and A. Arnould. Test Sequences Generation from Lustre Descriptions: GATeL. In *15th IEEE International Conference on Automated Software Engineering*. IEEE, September 2000.

[10] F. Ouabdesselam and I. Parissis. Testing Synchronous Critical Software. In *5th International Symposium on Software Reliability Engineering*, USA, 1994.

[11] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium (RTSS)*, 1998.

A Chisel diagrams and POTS

A Chisel diagram is a formalism used to present POTS and feature specifications during the "feature interaction detection contest", organised for the *fifth Feature Interaction in Telecommunication and Softare conference*, in 1998 [6]. The POTS is presented Figure A.1.

A Chisel diagram is a directed acyclic graph. A diagram node (one of the rectangles) contains a number, which uniquely identifies the node within the feature, and one or more events and variable assignments. The nodes are connected by directed edges (arrows in the diagrams). Multiple events in a node are separated by verticals bars (|||). A node containing such multiple events is equivalent to the sequence diagram representing any possible sequence of those same events (i.e. A|||B means {AB, BA}).

For the contest, the telephony system was mainly composed of a switch and several telephone devices. A user can emit the following events: Off-hook, On-hook, Dial *user_number_or_code*. Those events are the system inputs. The system can produce the following events: DialTone, Start_AudibleRinging, Start_Ringing, Stop_AudibleRinging, Stop_Ringing, LineBusyTone, Disconnect.

A feature may use internal variables. For instance, the variable **Busy A** is true between an Off-hook A event and the next On-hook A event; between a Start Ringing A B event and the next Stop Ringing A B event, if no Off-hook A intervenes; or between a Start Ringing A B event and the next On-hook A. All of the POTS event sequences start and end with Busy A = False (Idle A = True).

B A trace analysis of CCBS alone with POTS

Figure B.1 presents an excerpt of a 10 000 steps trace for CCBS alone with POTS. It was obtained with the random seed 72535. During the 10 000 steps, CCBS was activated and sucessfully invoked 66 times. We choose to present the 26th activation of the trace. The successful invocation appears 4 steps after the activation. It is the shorter time lapse between an activation and a successful invocation in this trace. The longer time lapse is 241 and the MTTS value here is 45.

- Step 3030, C goes on the hook (input "On C"). All users are idle (phone state "Id").
- Step 3035, A goes off the hook (input "Off C"). He hears the dialing tone (output "DT").
- A dials C number (step 3036), but this one is busy (he goes off the hook step 3031). A can activate the CCBS feature (output "Ann IC" = CCBS announce Incomplete Call).
- Step 3038, A dials the CCBS code (input "Dial A ccbs"). He receives a message from CCBS annoucing the activation is OK (output "Ann AcOK").
- A goes on the hook step 3039.
- Step 3040, A and C are both Idle, and CCBS starts invocation procedure: user A's phone starts ringing (input "StaR").
- A goes of the hook at step 3042, while C is still idle. C's phone starts ringing (input StaR) while A's phone starts audible ringing tone (input StaAur).
- Step 3042 the feature CCBS has been sucessfuly invoked. The delay between the activation and the invocation is displayed.
- Step 3043, C goes off the hook. Both audible ringing and ringing tones stop (outputs "StoAur" and "StoR"). A and C are "talking" (phone states "Tk").

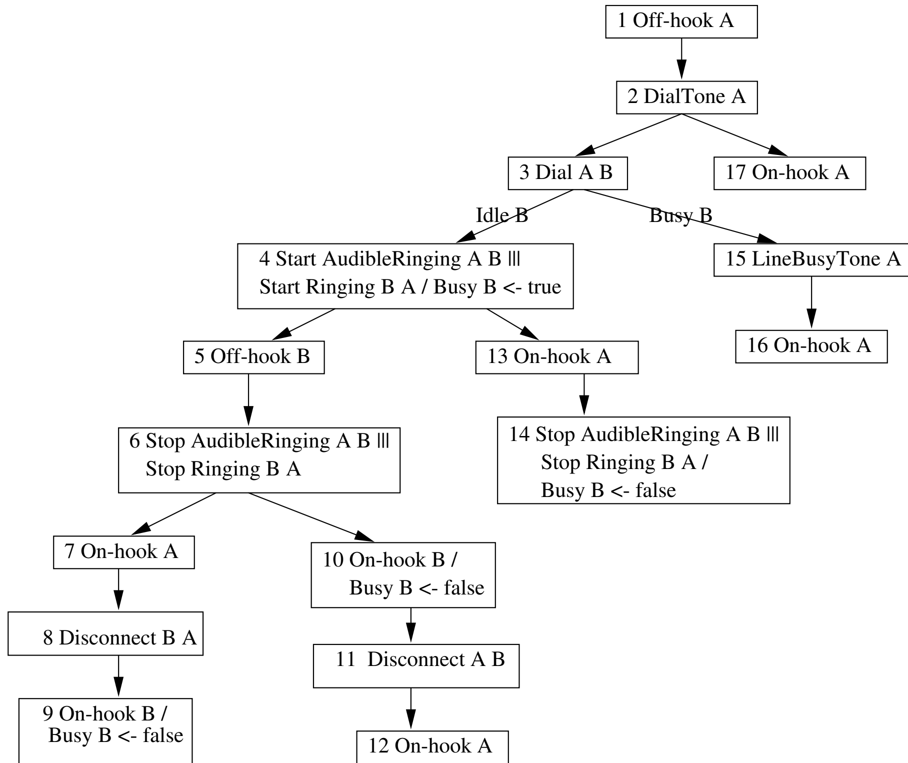

Fig. A.1. POTS formal description (Chisel diagram)

step	system input	system output				phone state				delay
		A	B	C	D	A	B	C	D	
3030	On C	-	-	-	-	Id	Id	Id	Id	0
3031	Off C	-	-	DT	-	Id	Id	Di	Id	0
3032	Dial C B	-	StaR	StaAur	-	Id	Rg	Al	Id	0
3033	Off B	-	StoR	StoAur	-	Id	Tk	Tk	Id	0
3034	-	-	-	-	-	Id	Tk	Tk	Id	0
3035	Off A	DT	-	-	-	Di	Tk	Tk	Id	0
3036	Dial A C	Ann IC	-	-	-	Ex	Tk	Tk	Id	0
3037	On C	-	Disc	-	-	Ex	Ex	Id	Id	0
3038	Dial A ccbs	Ann AcOK	-	-	-	Ex	Ex	Id	Id	0
3039	On A	-	-	-	-	Id	Ex	Id	Id	0
3040	Off D	StaR	-	-	DT	Rg	Ex	Id	Di	0
3041	On D	-	-	-	-	Rg	Ex	Id	Id	0
3042	Off A	StaAur	-	StaR	-	Al	Ex	Rg	Id	4
3043	Off C	StoAur	-	StoR	-	Tk	Ex	Tk	Id	0

Fig. B.1. A trace excerpt of CCBS alone with POTS executable specification