LaTe, a Non-fully Deterministic Testing Language

Emmanuel Donin de Rosière[1], Claude Jard[2], and Benoît Parreaux[1]

[1] France Télécom R&D,
2 Avenue Pierre Marzin 22307 Lannion, France
emmanuel.doninderosiere@francetelecom.com,
benoit.parreaux@francetelecom.com
[2] ENS Cachan,
Campus de Kerlann, 35 170 Bruz, France
claude.jard@bretagne.ens-cachan.fr

Abstract. This paper presents a case study which is the test of a voice-based service. To develop this application, we propose new functionalities for testing languages and a new language called *LaTe* that implements them.

With LaTe, one testing scenario can describe several different executions and the interpreter tries to find the execution that best fits with the real behavior of the System Under Testing (SUT).

We propose an operational semantics of these non-deterministic operators. Experimental results of the test of the voice-based service are also included.

1 Introduction

The world of testing languages remains complex and dense: there are often more than one language by application domain, e.g. hardware testing [1, 2], protocol testing [3], component testing [4, 5]... Several systems take programming languages in order to use them for testing purpose [6, 7]. One objective of this paper is to experiment with a paradigm that is non-deterministic testing. Despite a more complex interpretation, we will prove that this can increase the quality of black box testing languages (also called functional testing languages) on complex SUT.

This paper focuses on black box testing, i.e. the fact of testing a system where inputs and outputs of functions are known, but internal code structure is irrelevant. It is a form of testing which cannot target specific software components or portions of the code. So, in the rest of this article, we will use the same definitions than TTCN [8] which is the reference in this domain. This language, in its version 3, tries to be as generalist as possible and the most independent of the SUT.

SUT are more and more complex (and sometimes non-deterministic), so we need a testing language that has to be as powerful and expressive as a programming language. This is exemplified by the evolution of TTCN. Another instance

M.Ü. Uyar, A.Y. Duale, and M.A. Fecko (Eds.): TestCom 2006, LNCS 3964, pp. 71–86, 2006.
© IFIP International Federation for Information Processing 2006

is Tela (an UML 1.4 based testing language) [9], which gives lots of control structures like loops, branches, guards, interleaving and so on. However, Tela is more a test description language than a test execution language.

New operators which are resource-greedy can be proposed if they have good qualities because the time of test execution is not something important in our context. For decreasing the cost of learning a new testing language, they must be as easy and generalist as possible in order to test several different SUT with the same testing language (e.g. web services, java SUT etc).

This paper is organized as follows. In the next section, we present *LaTe*, a testing language which implements non deterministic operators. We also give the semantics of these operators. Then, we give some information about the case study: *testing a voice-based service* and the actual difficulties. Section 4 is dedicated to the test architecture of the experiments and to the examples of test cases in order to show the pros and cons of the constructions proposed here. Finally, we discuss the results and conclude.

2 Presentation of *LaTe*

One of the aims of this study is to show that non-deterministic operators can be very useful in testing languages. In order to evaluate these new operators, we have created a new language, but they could be added in a more complete language like TTCN. In this section, the main characteristics of LaTe will be presented. Then a description of the non-deterministic operators will be given. Finally, we will see the power of these operators and LaTe through a small example of unanimity vote.

2.1 Main Requirements

In Late, we try to select some important features:

Genericity: The language should be able to test different SUT. It is unfortunate to have to learn a different testing language for each system you want to test. So, it is better if an unique language can test every SUT (may be through an adaptation layer). As in TTCN, the use of *SUT adapters* can be useful. Nevertheless, unlike TTCN, it is interesting to write only one *SUT adapter* to test different SUT of the same type, e.g. one SUT adapter for testing all the SUT written in JAVA, one SUT adapter for Web services. Actually, three *SUT adaptor factories* have been written: the Java one, the C one and the socket one and. They contain respectively 250, 400 and 100 lines of code. It was very easy to write them because of the use of reflection in Java and the Java Native Interface that allows use C functions in Java code.

Using stubs: It is sometimes useful to develop stubs in order to test some SUT. But, in most languages, the user needs to write a specific stub for each test case. In addition, the interactions between the SUT and the tester and those between the stub and the SUT are often described separately. Thus, it is necessary to add synchronization points in the scenario in order to

express precedence between an action of the tester and one of the stub. With LaTe, like for SUT adapters, you need to write only one *stub constructor* to create stubs for a specific type of SUT and all the interactions between stubs and the SUT are directly written in the global scenario. For example, the following code specifies the creation of a Java stub that implements the interface `MyInterface`.

```
mystub:=createStub("Java","MyInterface");
//You create a stub
callSut(mysut,"register",mystub);
//You send it to the previously created sut
stubcall:=getCalled(mystub);
//You verify that the stub is called by the sut
```

This example shows that you can easily describe the general behavior of the system. In a system where you must separate the behavior of the stub and the tester component, the tester has to send a message to the stub in order to inform it that the call to the SUT have been made and it will receive a call from it. With our scenario, it is more compact and we can easily show the event succession.

Powerful API: SUT become more and more complex and non deterministic. A large collection of APIs is then needed to check that the value returned by the SUT is in accordance with the specification. The easiest way to have a large collection of API is to allow testers to use API from a programming language. In our language, we can easily use the Java APIs.

Dynamic language: Just like above, some SUT can create dynamically PCO (Point of Control and Observation) so we need to discover them during the execution of the scenario and dynamically connect to these PCO.

Something like a scripting language: Sometimes, the tester needs to test in real-time the SUT. For example, when the tester debugs a test scenario, he may need to have a console to execute his own commands. So an interpreted language will therefore be more useful than a compiled one. Moreover, it will be easier to have a dynamically typed language in order to write quickly the scenario and avoid the check and cast of values each line.

2.2 Non-deterministic Operators

As mentioned above, SUT become more complex, and this complexity leads to more non determinism. However the non determinism in a SUT is something quite difficult for a test writer because he has to infer the possible state of the SUT. So he has to add lots of `if ... then ... elsif ...` and if it cannot distinguish two cases, he may have to indicate an inconclusive verdict. Unfortunately, some SUT are intrinsically non deterministic: there are not fully observable, so we cannot know their internal state just from their outputs. To succeed in testing non deterministic SUT, we have hadded two non deterministic operators in the language. These operators are the non deterministic interleaving and the non deterministic choice. They are presented in details in

this section. Note that a similar paradigm has already been used in the procol testing domain in [10].

In a nutshell, the solution of the non deterministic SUT problem is to have different executions at the same time for the same scenario. When a divergence point is found in an execution, several executions are created in order to cover all the possibilities. When a contradiction is detected in an execution, then this execution is stopped and destroyed. The test passes when there is at least one execution that arrives at the end of the scenario, otherwise, it fails.

The first non-deterministic operator we define is noted `||`. It represents a choice in a scenario. For instance, `a:=3||a:=5` means that we have two executions: one where `a` equals 3 and another where `a` equals 5. All statements afterwards will be executed twice, once for each execution.

It can be done because the statements in one execution are independent of the ones of another execution. However, communications between the tester and the SUT do modify the state of the SUT, so precautions must be taken before executing these instructions.

In *LaTe*, when an execution wants to send a message, it always requests it to a component that can view all the current executions.

The other non-deterministic operator is noted `&&`. It represents a non deterministic interleaving. It executes all of the possible interleavings of two given branches. For example, `{A;B}&&{C;D}` is equivalent to:

```
{A;B;C;D}||{A;C;B;D}||{A;C;D;B}||{C;D;A;B}||{C;A;B;D}||{C;A;D;B}.
```

This operator helps to easily describe two independent behaviors. For example, when stubs are used in a scenario, it allows to specify that the behavior of one is independent of the other. Nevertheless, the longer the branches are, the more different executions are evaluated. Thus, in order to decrease this number, we decide that only communication with the SUT statements will cause the evaluation of new execution. For instance, with the following scenario: `{ ?a; assert(a==5) } && { ?b;assert(b==6) }`, only two executions will be evaluated: `?a;assert(a==5);?b;assert(b==6)` and `?b;assert(b==6); ?a;assert(a==5)` instead of the 6 possible interleavings. We can do this because we suppose the instructions that do not communicate with the SUT can be executed in any order without changing anything. This can be true only if a statement of a particular branch cannot influence other branches. In order to prevent this, we implement a locking variable system: if a variable is modified in a particular branch, it cannot be read or written in other branches. With all these restrictions, we can verify the initial hypothesis.

As we said before, we need a component that can view all the executions in order to decide which messages can be sent and when. This component will be called *top level* in this article. Each time an execution wants to send a message, it sends a request to the top level. If all the executions want to send the same message, the top level emits it and wakes up all the executions. If all the executions want to emit different messages then the top level may choose randomly

an execution, sends the corresponding message and destroys the other messages or may stop the evaluation of the scenario and returns an error depending of the configuration of the *LaTe* interpreter.

2.3 Operational Semantics

In the following equations, $\mathfrak{P}(S)$ denotes the set of all subsets of S, $\bullet$ the concatenation operator and $\sqcup\!\sqcup$ the shuffle product.

Let $[\![]\!]$ be the following semantic operator:

$$\begin{aligned} [\![]\!] :& \mathbb{P} \times (\Sigma_r \times \mathbb{R}^+)^\infty \times \mathbb{R}^+ \times \mathfrak{P}(\Sigma_e \times \mathbb{R}^+) \rightarrow \\ & \mathfrak{P}(\mathbb{P} \times (\Sigma_r \times \mathbb{R}^+)^\infty \times \mathbb{R}^+ \times \mathfrak{P}(\Sigma_e \times \mathbb{R}^+) \times \mathfrak{P}(\Sigma_e \times \mathbb{R}^+)) \end{aligned} \tag{1}$$

$\mathbb{P}$ corresponds to the set of all programs formed by waiting, sending and receiving statements and interleaving, alternative and sequence operators. It also contains the null program (P_0).

Σ_r is the set of messages that can be sent by the SUT and Σ_e is the set of all messages sent by the tester.

This operator expresses the set of all possible futures from a program P, a trace σ^i containing the received messages and their arriving times, the current time, and a list of sendable messages. Each future is made up by the remaining program to execute (which can be null), the current time, the list of sendable messages and the set of messages to send. If the remaining program is null, then all the statements were executed else the execution is waiting for the authorization to send a message to the SUT.

The null program can be executed but does not modify the context:

$$[\![P_0, \sigma^i, t, a]\!] = \{(P_0, \sigma^i, t, a, \phi)\} \tag{2}$$

Let $?e^{<w}$ be the statement meaning that the next message received must be e and that it must arrive before w seconds. If it is true ($\sigma^i(0) = e^\tau$ and $t+w > \tau$), then e is deleted from the trace and the clock is moved forward by τ.

$$[\![?e^{<w}, \sigma^i, t, a]\!] = \begin{cases} \{(P_0, \sigma^{i+1}, t+\tau, a, \phi)\} & \text{if } \sigma^i(0) = e^\tau \wedge t + w > \tau \\ \phi & \textit{otherwise} \end{cases} \tag{3}$$

Let $!E$ be the statement that means `E` must be sent to the SUT. First, if E belongs to the set of sendable messages, it is sent. If not, the execution is stopped, and `E` (associated with the current time) is added to the list of messages we want to send. We also verify that any more incoming message does not arrive.

$$[\![!e, \sigma^i, t, a]\!] = \begin{cases} \{(P_0, \sigma^i, t', a \setminus \{e\}, \phi)\} & \text{if } \exists t', t'' \in \mathbb{R}^+, \exists f \in \Sigma | \\ & e^{t'} \in a \wedge \sigma^i(0) = f^{t''} \wedge t'' \geq t' \\ \phi & \text{if } \forall t' \in \mathbb{R}^+, \exists t'' \in \mathbb{R}^+, \exists f \in \Sigma | \\ & e^{t'} \in a \wedge \sigma^i(0) = f^{t''} \wedge t'' < t' \\ \{(!e, \sigma^i, t, a, \{e^t\})\} & \textit{otherwise} \end{cases} \tag{4}$$

In order to execute $P;Q$, P is executed first with the current environment. Then, for each possible future where all the statements were executed, Q is executed with the new environment. For each future where a sending authorization is awaiting, the remaining program is rebuilt.

$$\begin{aligned}\llbracket P;Q,\sigma^i,t,a\rrbracket = &\bigcup\{\llbracket Q,\sigma^j,t',a'\rrbracket|(P_0,\sigma^j,t',a',\phi)\in\llbracket P,\sigma^i,t,a\rrbracket\}\\ &\cup\bigcup\{(P';Q,\sigma^j,t',a',d)|(P',\sigma^j,t',a',d)\in\llbracket P,\sigma^i,t,a\rrbracket\wedge P'\neq P_0\}\end{aligned} \tag{5}$$

The possible futures of $P||Q$ are the union of possible futures of P with those of Q:

$$\llbracket P||Q,\sigma^i,t,a\rrbracket = \llbracket P,\sigma^i,t,a\rrbracket\cup\llbracket Q,\sigma^i,t,a\rrbracket \tag{6}$$

The interleaving operator (&&) is the most complex operator in these semantics.

In order to execute $P\&\&Q$, P and Q are executed separately. To do this, two disjoint sets a_p and a_q are extracted from a. Each set corresponds to the part of a used by each process. $\sigma^i\setminus\sigma^j$ is also divided into two parts thanks to two projections h_p and h_q. If the two processes are fully executed, the execution is well finished and the new current time is the maximum of the two final times. If at least one of them is waiting for a sending authorization, the state of the program is rebuilt at the moment of the send of the message. Therefore, waiting times ($S(t'-t)$ and $S(t''-t)$) have to be added in order to take the passing time into account.

$$\begin{aligned}\llbracket P\&\&Q,\sigma^i,t,a\rrbracket = &\{(P_0,\sigma^j,max(t',t''),a\setminus(a_p\cup a_q),\phi),\exists j\geq i,\\ &\exists h_p,h_q\in\text{Projection},\exists t',t''\in\mathbb{R}^+,\exists a_p,a_q\in\mathfrak{P}(\Sigma_e\times\mathbb{R}^+)\\ &|\sigma^i\setminus\sigma^j\in h_p(\sigma^i\setminus\sigma^j)\sqcup\!\sqcup\, h_q(\sigma^i\setminus\sigma^j)\\ &\wedge(P_0,\sigma^j,t,\phi,\phi)\in\llbracket P,h_p(\sigma^i\setminus\sigma^j)\bullet\sigma^j,t',a_p\rrbracket\\ &\wedge(P_0,\sigma^j,t,\phi,\phi)\in\llbracket Q,h_q(\sigma^i\setminus\sigma^j)\bullet\sigma^j,t'',a_p\rrbracket\\ &\wedge a_p\cap a_q=\phi\wedge a_p\subset a\wedge a_q\subset a\}\\ &\cup\{((S(t'-t);P')\&\&(S(t''-t);Q'),\sigma^j,t,\phi,d_p\cup d_q)\setminus\exists j\geq i,\\ &\exists h_p,h_q\in\text{Projection},\exists t',t''\in\mathbb{R}^+,\exists a_p,a_q\in\mathfrak{P}(\Sigma_e\times\mathbb{R}^+)\\ &|\sigma^i\setminus\sigma^j\in h_p(\sigma^i\setminus\sigma^j)\sqcup\!\sqcup\, h_q(\sigma^i\setminus\sigma^j)\\ &\wedge(P',\sigma^j,t',\phi,d_p)\in\llbracket P,h_p(\sigma^i\setminus\sigma^j)\bullet\sigma^j,t,a_p\rrbracket\\ &\wedge(Q',\sigma^j,t'',\phi,d_q)\in\llbracket Q,h_q(\sigma^i\setminus\sigma^j)\bullet\sigma^j,t,a_q\rrbracket\\ &\wedge a_p\cap a_q=\phi\wedge a_p\cup a_q=a\wedge d_p\cup d_q\neq\phi\}\end{aligned} \tag{7}$$

We have just seen that some executions may enter in a waiting state. In fact, an emission is something that cannot be cancelled. As a result, we have to check that all executions want to send the same message. Indeed, in order to choose what and when an emission can be made, we must have a total view of all the

executions. It is why the use of a *top level* is unavoidable. It is the only element that can see the global state of the tester. This component gets back the emission requests, computes them in order to find if the executions are determinable. After that, it gives the authorization to send messages.

Let TL be semantics of the *top level*:

$$TL : \mathfrak{P}(\mathbb{P} \times (\Sigma_r \times \mathbb{R}^+)^\infty \times \mathbb{R}^+ \times \mathfrak{P}(\Sigma_e \times \mathbb{R}^+) \times \mathfrak{P}(\Sigma_e \times \mathbb{R}^+)) \rightarrow \text{Bool}$$

TL is defined as follow:

$$TL(P_g) = \begin{cases} true \text{ if } \exists \sigma^i \in (\Sigma_r \times \mathbb{R}^+)^\infty, \exists t \in \mathbb{R}^+ | (P_0, \sigma^i, t, \phi, \phi) \in P_g \\ TL(\bigcup\{[\![P, \sigma, t, d]\!] | \exists a \in \mathfrak{P}(\Sigma_e \times \mathbb{R}^+),\ (P, \sigma, t, \phi, a) \in P_g\}) \text{ if } \exists d \\ \in \mathfrak{P}(\Sigma_e \times \mathbb{R}^+)\ ,\ d = \{e^{max(t_i)} | \forall d_i, \exists t_i\ e^{t_i} \in d_i\} \wedge d \neq \phi \wedge P_g \neq \phi \\ false \text{ otherwise} \end{cases} \quad (8)$$

If one possible future finishes its execution, the trace conforms to the testing scenario. Otherwise, all the executions are waiting at least one sending authorization. If the intersection of all of these requests is not empty, we authorize their emission. In any other cases, we cannot find a consensus. Thus, the test is declared false. Denote that the same emission for different executions must be done in the same time (because there can be only one real emission). We emit therefore this message at the maximum time given by all executions.

With these semantics, we do not assure that the emission will immediately take place when the request is made. We use a *best effort* policy. Nevertheless, these semantics can be easily modified in order to emit the message immediately, but it risks compelling too much the executions of the scenario.

Finally, we just have to compute $TL([\![P, \sigma^0, 0, \phi]\!])$ to find if a trace σ satisfies a program P.

2.4 An Example: Unanimity Vote

We point out some of the advantages of these operators through a small example: a system of unanimity vote. Here, the SUT is a java class that can be called to register to the electoral list an object implementing a particular interface. When a question is asked to the SUT, it transmits the question to some electors. These calls can be executed concurrently and if one of the electors replies `false`, the returned value of the SUT will be `false` and all the electors may not be asked to vote (it is a unanimity vote).

This example is quite difficult to test using classical systems because:

- it has to use stubs for electors and has to control each stub;
- some of the interactions are concurrent. We have to verify if the test scenario agrees with all the possibilities of interleaving the SUT interactions;
- it contains lots of non-determinism: if one of the electors replies `false`, the SUT may continue to call all of the electors but it also can stop and directly replies to the tester. The test scenario must describe all of these cases.

With LaTe, it is quite simple to write a testing case for this SUT. For example, for 2 electors, the scenario may be the following:

```
1  mysut:=createStub("Java","SUT");
2  elector1:=createStub("Java","ElectorInterface");
3  callSut(mysut,"register",elector1);
4  //We create a stub and register it to the SUT
5  elector2:=createStub("Java","ElectorInterface");
6  callSut(mysut,"register",elector2);
7  //We create a stub and register it to the SUT
8
9  sutcall:=callSUT(mysut,"ask","Is 42 prime ?");
10 {
11   @<10*sec{stubcall1:=getCalled(elector1)};
12   replySUT(stubcall1,false);
13   //The SUT asks the question to the elector 1 and we reply false
14 }
15 &&
16 {
17   {
18     @<10*sec{stubcall2:=getCalled(elector2)};
19     replySUT(stubcall2,true);
20     //The SUT may ask the question to the elector 2 and we reply true
21   }
22   ||
23   {
24   }
25   //But it is optional
26 };
27 assert(getReply(callSut)==false);
28 //The answer is "false"
```

Listing 1.1. LaTe scenario for the unanimity vote

As mentioned above, LaTe evaluates all the possible interleavings of the scenario. In other words, whatever the order of stub calls by the SUT and whatever the number of called stubs, LaTe is capable of verifying that the execution fits the specification described in the scenario. Time specification can also be easily added by using the `@` operator.

Nevertheless, LaTe may have difficulties to evaluate particular scenarios. For instance, the pseudo code `{ ?a;!A } && { ?b;!B }` is problematic if the SUT sends the two messages `a` and `b`, without waiting for the replies `A` or `B`, because this behavior is extremely non deterministic. All the executions are presented in Figure 1.

In this example, if the SUT sends `a` and waits for `A` before sending `b`, there is also a problem. All the branches beginning by `?b` are not executable because

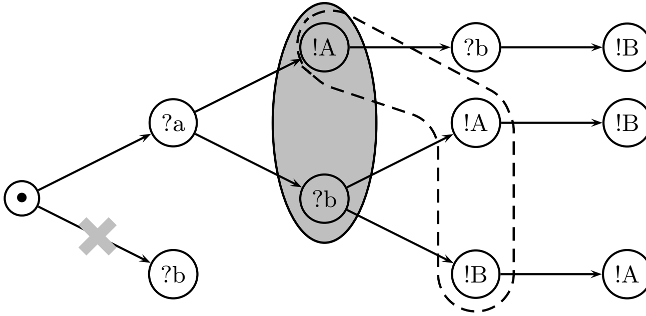

Fig. 1. All the possible interleavings

the first message received by the tester is `a` and the assertion `?b`, which signifies that the next message receive by the tester is `b`, will be broken. Thus, in the next step, two executions are possible: `?b` and `!A`. The second execution (with `?b`) is in a waiting state, because the SUT does not send an additional message and the first execution cannot emit `A`, because there is an execution that does not want to send something. We are in a livelock. In order to resolve this conflict, the scenario must contain information about timeouts in all the *receive* instructions, e.g. if the `?b` lasts more than 5 seconds, this execution will be destroyed, and there will be only one possible execution (the first one). The evaluation of the scenario will continue.

In other case, if the SUT sends the two messages `a` and then `b` without waiting for the replies `A` or `B`, there will be another problem. Three executions will be then possible. They are displayed with dash in Figure 1. Two of these executions want to send `A` and the last intends to emit `B`. This is because of the exact symmetry of the test scenario (there is no difference between the two stubs). In this case, we decide that LaTe may choose randomly an execution and continues it. Then, LaTe prints a warning message in order to warn the tester that the scenario contains conflicts. LaTe may also stop the execution according to its configuration.

Through this example, we can see some of the advantages of the non-determinism operators of LaTe: they permit to easily describe the parallelism and the non-determinism of the SUT. Although this example is an extreme case of non-determinism associated with concurrency, we can test quite easily this behavior. Nevertheless, the tester has to think of all possible executions. In the next sections, a more complex example will be studied in details, the voice-based service that we will just study: the vocal-based telephone directory.

3 Description of the Case Study: Testing a Voice-Based Service

We decided to validate our approach on the test execution of voice-based services and particularly a vocal-activated telephone directory. In this service, the user gives a name and the service seeks this name and then proposes to put the

user in relation with the found number. In the case of homonyms, the service proposes several solutions and requests the user to choose the solution which is appropriate to him.

The new voice-based services use intensively speech synthesis and recognition. These functionalities simplify the access to the voice-based services but complicate their validation. Indeed, they produce lots of non-determinism if we try to automatically test it. For example, the volume and the speed of the voice during two different conversations can change. If we need to recognize automatically the sentences pronounced by the voice-based services using a speech recognition tool, we should make the verdict even more random.

Furthermore, the use of speech recognition for the tester also increases the non-determinism of all the system. Many factors can affect the result of the speech recognition such as the quality of the transmitted messages and the quality of the line and so on. It is possible for a same message to be correctly recognized the first time but not the following times. Thus, the presence of speech recognition and synthesis causes a significant number of *inconclusive* verdicts during automatic tests.

One current solution for this problem is to replace all speech signals emitted and expedted by the platform by DTMF (Dual Tone Multi Frequency) signals. These signals are the sounds produced when you dial a number with your phone. This solution is not completely satisfactory because it requires the modification of the voice-based service. The verdict of the tests can also be deteriorated by these modifications. The other solution is to carry out the tests, taking the risk of obtaining a significant number of *inconclusives* verdicts. No other solution are possible with common testing languages. The language that we propose in this article enables us to bring a new solution to this problem by using non-deterministic operators in the test scenario. So this vocal-based phone directory, although very simple, is enough to clarify the interests of our language.

4 Methodology of the Experiment

In this section, we will see in details how the vocal-based telephone directory was tested using LaTe. First, we will see the test architecture of this experiment, then we study the communication protocol between the tester and the calling platform. Finally, some test scenarios will be presented.

4.1 Test Architecture

In section 3, we discussed in details how works the vocal-based telephone directory testing here. Thus, we develop a special test architecture in order to allow the tester to connect to the voiceXML service. This architecture is represented in Figure 2.

The tester represents the machine where LaTe and the scenarios are executed. It communicates through a socket with the call API. This computer is linked to a call card which can makes calls and conversation on an analogic line. Therefore, this computer can call the voice-based service. As we just said, there is a dialog

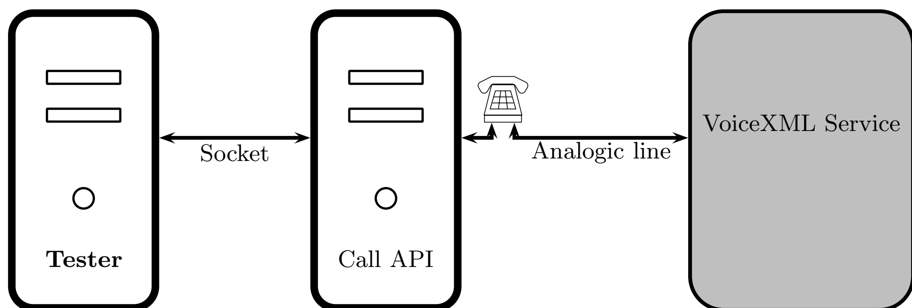

Fig. 2. Test Architecture

between the tester and the call API through a socket. A special protocol has been defined to allow the tester to test the service and obtain information about the conversation. This protocol uses all of these messages:

`BECMDCall num ENDCMD`: this message asks the call API to call the number *num*.
`BECMDDTMF num ENDCMD`: it asks the call API to simulate the pushing of a particular touch (defined by `num`) in the telephone keyboard. This is done by emiting a special sound called *DTMF* during the conversation.
`BECMDWaitDTMF num ENDCMD`: this message asks the API to wait a particular DTMF sent by the directory.
`BECMDTalk file ENDCMD`: it asks to play a particular file during the conversation. This file must be in a PCM format at a good frequency.
`BECMDRecordAll file ENDCMD`: it asks the API to record all of the conversation in the file `file`. This conversation will be saved in a PCM format.
`BECMDHangUp ENDCMD`: sent when the tester wants to hang up.
`BECMDEnd ENDCMD`: sent when the tester wants to stop the communication between it and the calling computer.

For each of this message, the call API sends a corresponding `done` message when the operation executed without any problem or a `not started` when a problem occurs. Moreover, the call API can emit particular messages:

`BEGINFDetect Speech ENDINF`: it is sent when the call API discovers that there is someone who speak during the conversation.
`BEGINFEnd Speech ENDINF`: it is sent when the call API discovers that the speech finishes.
`BEGINFHangUp ENDINF`: sent when the call API discovers that the line has been hanged up.

At first, we tried to allow the API to use vocal synthesis in order to send every possible message. However, we discovered that the speech recognition system of the voiceXML service has lots of difficulties to recognize generated voice. These problems increase the number of cases we have to manage in our scenarios, so we finally preferred to record the sentences we will play during the tests for this reason.

We also imagine using a speech recognition system for our call API. Indeed, if we can know what was pronounced by the service, we can deduct in which state is the SUT. Nevertheless, as for the previous remark, speech recognition has difficulties to recognized generated voices. So lots of recognized sentences were wrong because we were comparing character to character these sentences with the sentences of the specification. We may modify the speech recognition system in order to give for each sentence pronounced by the voiceXML service, the sentence of the specification that may have been pronounced.

With respect to our aim, we develop a *socket SUT adaptor factory.* Thus, in LaTe, you just have to specify the host and the port of the server and LaTe automatically creates the stub adaptor and connects itself to the server. The code of this factory is quite simple but we will not show it here because we are not specially interested by this information in this paper. Just with these information, we allow LaTe to easily test the vocal-based directory. So in the next section, we will see in details some scenarios and further the advantages of the non-deterministic operators on this particular test case.

4.2 Some Test Cases

For all of the following test cases, we defined several LaTe functions to simplify the writing of scenarios. These functions are:

`connection(host,port)`: this function initializes the communication between the tester and the call API.

`call(sut,num)`: for a giving SUT, it send a message for calling the phone number `num`.

`sendCommand(sut,command)`: it sends the corresponding command to the SUT.

`listenSpeech(sut,t1,t2)`: it verifies that a sentence is prononced before the time `t1` and during at most the time `t2`. Otherwise, the execution is destroyed.

`getMessageIn(sut,message,t)`: it waits at most the time `t` for a particular message. If any message arrives or the first message was not `message`, the execution is destroyed.

`getMessage(sut1,message)`: it makes the same thing than the previous function, but without any timeout.

It is very easy to write thess fonctions. For example, the code of the function `listenSpeech` is the following:

```
1  function listenSpeech(sut1,t1,t2)
2  {
3    @<t1{ assert(getMessage(sut)=="BEGINFDetect SpeechENDINF") };
4    @<t2{ assert(getMessage(sut)=="BEGINFEnd SpeechENDINF") };
5  };
```

Listing 1.2. User-defined function

For the first test case, our aim is to connect to the call API, call the directory, pronounce a name and verify that people picks up.

```
1  sut:=connection("l-at7290",4442);
2  call(sut,"123");
3  sendCommand(sut,"RecordAll communication.pcm");
4  listenSpeech(sut,30*sec,30*sec);
5  sendCommand(sut,"Talk testername");
6    {   listenSpeech(sut,50*sec,30*sec); }
7  &&
8    {   getCommandeTimer(sut,"Send talk done",20*sec); };
9  wait(5000);
10 sendCommand(sut,"Talk yes");
11   {   listenSpeech(sut,50*sec,30*sec); }
12 &&
13   {   getMessageIn(sut,"Send talk done",15*sec); };
14 sendCommand(sut,"HangUp");
15   {
16       {  getMessage(sut,"HangUp detected"); }
17     &&
18       {  listenSpeech(sut,50*sec,30*sec); }
19   }
20 ||
21   {   getMessage(sut,"HangUp detected"); };
22   {   getMessageIn(sut,"RecordAll done",15*sec); }
23 &&
24   {   getMessageIn(sut,"HangUp done",15*sec); };
```

Listing 1.3. LaTe scenario for a simple test

The fact that we use an analogic line to connect to the service adds randomness in the receipt order of messages. For example, we do not know in advance if the message `Send talk done` will be received before the beginning of the speech. So we have to set that all of `Send talk done` messages can be interleaved with the listenSpeech function. A lot of executions will be evaluated, but only one will fit the real events.

Moreover, when the tester executes manually the test, he can observe that sometimes the call API detects two different speeches and other times, only one. Thus, in the scenario (lines 15 to 21) we specify that these two cases can occur by using the `||` operator. We have either an `HangUp detected` message, or a `HangUp detected` message interleaved with a speech detection. At the end of the scenario, we also specify that a `HangUp done` is interleaved with a `RecordAll done` message.

5 Experimental Results and Discussion

5.1 Results and Pros

The use of LaTe in this case allowed the tester to semi automatically test the voiceXML service. It was not fully automatic because the test needs an operator which verifies that his phone rings and picks up for some test cases. But compared to the manual testing, this solution reduces the interactions between the tester and the SUT.

If we compare this solution to one based on TTCN, we can observe that TTCN contains an `interleave` statement that specifies that different branches must be executed concurrently. TTCN allows user to write its own functions (internally or externally), nevertheless, they can be used in an `interleave` branche. In this particular case, LaTe is more powerfull than TTCN, because we don't have to inline the `getMessage` and `listenSpeech` functions. Moreover, TTCN does not allow to specify in a test scenario that something is optional. The only statement that can be used for that is the `alternative` one, but the user have to give a guard for each branches of the `alternative` statement which is very difficult in our example.

Another advantage of this testing architecture is that we have to our disposal all the traces of the conversations between the tester and the voiceXML service. So when we discover an error, we can easily locate it thanks to these traces.

5.2 Discussion

As we said previously, this method for testing the vocal-based phone directory is not perfect: the tester does not know exactly what is pronounced during the conversation and he has to pre-record all of the sentences before using it. Thus, one possible evolution of this technique is to use speech recognition. We saw in section 4.1 that a normal system may not work for our example. The recognition is not something perfect and may make mistakes. So some verdicts may be `fail` with any difference between the specification and the SUT. One possible solution is to modify the speech recognition system so that it gives a set of possible sentences that may have been pronounced. With this modification, the speech recognition system will someway be non deterministic: several possible verdicts will be returned. Associated with our non-deterministic operators, it can easily find after few steps which sentence was pronounced thanks to the following sentences. Thus, with only few modifications of the scenarios, this system will explore more deeply the real behavior of the SUT and will be more capable of detecting mistakes.

Another possible improvement of this system is to generate directly the *LaTe* scenarios from the specification. One of our perspective is to modify *TGV* [11] for this aim. TGV allows the generation of an abstract test case from a specification and a test purpose. The generation is done "on-the-fly" on the synchronous product of the specification with the test purpose. It is based on Tarjan's algorithm. During the depth-first search (DFS), TGV performs abstraction and

determinization of this product. The DFS stops when an accepting state of test purpose is reached. During the backtracking, TGV synthesizes the transitions of the test case.

Currently, TGV generates test cases in both BCG [12] and Aut [13] formats, so if it can be modified to directly generate test scenarios in *LaTe* format, we will be able to reduce the work of writing these scenarios.

6 Conclusion

Some SUT are so complex and non-deterministic that usual testing systems and languages have difficulties to evaluate the state of the SUT. Thus, we have defined two particular operators for testing languages, the non-deterministic choice and the non-deterministic interleaving. These operators allow the tester to maintain several different executions at the same time. Each execution is independent of the others and is destroyed when a contradiction is found. Nevertheless, communications between these executions and the SUT must be managed because of the communication cannot be undone and modify the environment of all executions.

In order to show that these non-deterministic constructions may be useful, we have implemented them in a new language, LaTe and we have applied them on a particular case study: testing a vocal-based telephone directory. They were particularly useful on this case because the SUT contains lots of non-deterministic behavior like interleaved events, optional messages etc. Thus, these constructions allowed to increase the automation of this task and also allowed the scenarios to test deeper behavior than usual test scenarios. Finally, we have proposed several enhancements for this particular case study, like adding a speech recognition system in order to increase the power of the system and test deeper voice-based services.

References

1. Thomas, D.E., Moorby, P.R.: The Verilog Hardware Description Language. 3rd edn. Kluwer Academic Publishers (1996)
2. Offerman, A., Goor, A.: An experimental user level implementation of tcp. Technical Report 1-68340-44(1997)07, Delft University of Technology (1997)
3. Aho, A.V., Dahbura, A.T., Lee, D., Uyar, M.U.: An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. IEEE Transactions on Communications **39** (1991) 1604–1615
4. Doong, R.K., Frankl, P.G.: The ASTOOT approach to testing object-oriented programs. ACM Transactions on Software Engineering and Methodology **3** (1994) 101–130
5. Cheon, Y., Leavens, G.T.: A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In: ecoop. (2002)
6. Beck, K., Gamma, E.: Junit test infected: Programmers love writing tests. Technical report, Java Report (1998)
7. Massol, V., Husted, T.: JUnit In Action. Manning (2003)

8. ITU-T Z.140: The Tree and Tabular Combined Notation Version 3 (TTCN-3): Core Language. (2001)
9. Pickin, S., Jard, C., Le Traon, Y., Jézéquel, J., Le Guennec, A.: System test synthesis from uml models of distributed software. In: FORTE'2002, IFIP Int. Conf. on Formal description techniques, Houston, Texas (2002)
10. Ghriga, M., Frankl, P.G.: Adaptive testing of non-deterministic communication protocols. In: Protocol Test Systems. (1993) 347–362
11. J. C. Fernandez, C. Jard, T. Jéron, G. Viho: Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur, Thomas A. Henzinger, eds.: Proceedings of the Eighth International Conference on Computer Aided Verification CAV. Volume 1102., New Brunswick, NJ, USA, Springer–Verlag (1996) 348–359
12. Tock, L.P.: The bcg postscript format. Technical report, INRIA Rhône-Alpes (1995)
13. Fernandez, J.C.: Aldebaran user's manual. Technical report, Laboratoire de Génie Informatique - Institut IMAG (1989)