Synchronous Testing of Multimodal Systems: an Operational Profile-Based Approach

Laya Madani, Catherine Oriat, Ioannis Parissis
Laboratoire LSR-IMAG
BP 53 - 38041 Grenoble Cedex 9
Forename.Name@imag.fr

Jullien Bouchet, Laurence Nigay
Laboratoire CLIPS-IMAG
BP 53 - 38041 Grenoble Cedex 9
Forename.Name@imag.fr

Abstract

In this paper we present a method for automatically testing interactive multimodal systems[1]. The proposed approach was originally dedicated to synchronous programming which is mainly used for real-time systems. Nevertheless, the behaviour of real-time systems, consisting of cycles starting by reading an external input and ending by issuing an output, is to a certain extent similar to the one of interactive systems. So considered, this paper investigates the use of the Lutess testing environment dedicated to synchronous software for automatically testing multimodal interactive (not necessarily real-time) systems. More precisely, we focus on test data generation based on operational profiles. The main benefit of this approach is that it increases the ability to automatically test an interactive system over long input sequences, according to various use profiles.

1. Introduction

The area of multimodal interaction has expanded rapidly and since the seminal "Put that there" demonstrator [1] that combines speech, gesture and eye tracking, significant achievements have been made in terms of both modalities and real multimodal systems in various domains including medical and military ones [2]. Moreover, multimodal interfaces are now playing a crucial role for mobile systems since multimodality offers the required flexibility for variable usage contexts, as shown in our empirical study of multimodality on PDA [11]. Although several real multimodal systems have been built, their development still remains a difficult task. On the one hand, the power and versatility of multimodal interfaces result in an increased complexity of the software to be developed. On the other hand, tools dedicated to multimodal interaction are currently few and limited in scope. As a consequence, the software is complex and mainly developed manually.

In this article, we address the problem of testing multimodal systems. As any interactive system, the latter require several interaction scenarios to be executed corresponding to several user behaviours. But with multimodality, the number of such scenarios is huge since a multimodal system allows the combined usage of multiple modalities. Many formal proof-based approaches have been proposed in the past, such as the Formal System Modelling (FSM) analysis [7], the Lotos Interactor Model (LIM) [16], the Interactive Cooperative Object (ICO) formalism based on Petri Nets [13] as well as a Lustre-based approach for validation [5]. These approaches imply a formal description of the interactive system as an abstract model on which properties are checked.

In this paper we investigate a testing approach for multimodal interactive systems originally developed for synchronous software. The synchronous programming paradigm is widely used in safety critical applications and makes specification, simulation and proof of such software easier and more reliable [8]. We expect that applying synchronous software verification methods to interactive multimodal software will result in usable and powerful tools. We are particularly interested in the Lutess testing environment [6], [14]. Lutess handles a partial specification of the interactive system to test written in Lustre [8], a synchronous dataflow language. In contrast to the above mentioned proof-based approaches, Lutess does not require the entire system to be formally specified nor does it intend to formally prove properties. Lutess requires a non-deterministic specification of the user behaviour as well as a description of the software properties. Lutess then automatically builds a simulator that will feed with inputs the software under test (i.e., the multimodal user interface). Test oracles can also be written in Lustre to encapsulate the properties to check and to detect software failures. The test data generation can be a purely random simulation of the user behaviour, but this seldom results in realistic interaction scenarios. Operational profiles are supported by Lutess [12] and can be used to improve the relevance of the produced test data, as well as their ability to detect failures, making them correspond to various realistic user behaviours.

The structure of the paper is as follows: first, we define the different forms of multimodal usage and in particular the combined usage of several modalities. We then briefly present the Lutess environment and show how it can be

[1] This work is part of the VERBATIM research project, supported by the French National Network for Research on Telecommunications (Réseau National de Recherche en Télécommunications).

used for testing multimodal systems. We finally present an example that illustrates the test of a multimodal system using Lutess and operational profiles. This example is based on Memo, a mobile multimodal system, whose main features are presented in the next section.

2. An illustrative example: Memo

Memo [3] is an input multimodal system allowing users to annotate physical locations with digital post it-like notes, which have a physical location and are then read/carried/removed by other mobile users. The Memo user of Figure 1 (left) is equipped with a head mounted display (HMD). Its semi-transparency enables the fusion of computer data (the digital notes) with the real environment as shown in Figure 1 (right). In addition a GPS and a magnetometer are worn by the user, enabling the system to compute the location and orientation of the user.

We consider three tasks in Memo which are possible using different modalities: (1) orientation and localization of the mobile user so that the system is able to display on the HMD the visible notes according to the current position and orientation of the mobile user (2) manipulation of a note (get, set and remove a note) and (3) exit the system. For the manipulation of notes using Memo, the mobile user can get a note that will then be carried by her/him while moving and be no longer visible in the physical environment. The user can carry one note at a time. S/he can set a carried note to appear at a specific place. Issuing the set command without carrying a note has no effect. The user can finally remove a note that is carried by her/him or a note visible in her/his physical environment. If the user is carrying a note and is also next to a note, a command remove will delete the note in the physical world: Indeed priority is given to the manipulation of notes attached to the physical world. If the user is carrying a note and has no note around her/him in the physical environment, then the carried note is deleted.

To perform the tasks, Memo supports five input active and passive modalities. For inputs, active modalities are used by the user to issue a command to the computer (e.g., a voice command). Based on our definition of a modality [11] as the coupling of a physical device with an interaction language, the three active modalities in Memo are: (Mouse, Button Commands), (Microphone, Speech Commands) and (Keyboard, Key Commands). Such active modalities are used by the user for manipulating a note and for quitting the system. Passive modalities are used to capture relevant information for enhancing the realization of the task, information that is not explicitly expressed by the user to the computer such as eye tracking in the "Put that there" demonstrator [1] or location tracking for the Memo mobile user. The two passive modalities in Memo are: <Magnetometer, Three orientation angles in radians> and <Localization sensor GPS, 3D location>. Such passive modalities are used to display the notes on the HMD, as well as to select a note.

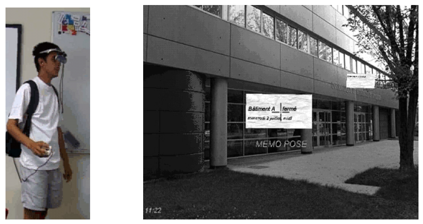

Figure 1 : Left: A Memo user, equipped with the HMD and holding a mouse. Right: A View through the HMD: The user is in front of a building and can see two digital notes

3. Multimodality

Although each modality can be used independently within a multimodal system, the availability of several modalities in a system naturally leads to the issue of their combined usage. The combined usage of multiple modalities opens a vastly augmented world of possibilities in multimodal user interface design that we studied in light of the four CARE properties in [4],[11]. While Equivalence and Assignment express the availability and respective absence of choice between multiple modalities for performing a given task, Complementarity and Redundancy describe relationships between modalities for performing a given task. We illustrate the CARE properties with the Memo system:

Localization and orientation of the user are two passive modalities used in a *complementary* way for computing the position of the mobile user.

Speech, keyboard and mouse commands are *equivalent* active modalities for manipulating notes. The user has therefore the choice amongst the three modalities for the commands get/set/remove a note. A command specified using speech, keyboard or mouse is applied to the note that the user is looking at. As a consequence, the two passive modalities, localization and orientation, that enable the system to determine the note that the user is looking at, are *complementary* to one of the three equivalent modalities (speech, keyboard or mouse commands).

Memo also supports *redundant* usage of modalities. Redundancy corresponds to the case where two modalities convey redundant pieces of information that are close in time. In such a case, one of the two user's actions must be ignored. Using Memo, speech, keyboard and mouse commands can be issued in a redundant way. For example, the user can use two redundant modalities, voice and mouse commands, for removing a note: the user issues the voice command "remove" while pressing the mouse button. Because the corresponding pieces of information are redundant and the two actions (speaking and pressing) produced nearly in parallel or close in time, only one command will be executed and therefore only one note will be removed. If the two remove actions were not produced close in time, there is no redundancy detected and two remove commands will therefore be executed.

For quitting the system, the user has no choice and must use a special key on the keyboard. The task of quitting the system is therefore assigned to typing.

Because the CARE properties have been shown to be useful concepts for the design and evaluation of multimodal interaction [4], we decided to reuse those concepts for formally testing multimodal systems using the Lutess environment.

4. Lutess: A testing environment for synchronous programs

Lutess [14][6] is a testing tool for functional testing of synchronous software. Lutess enables the automatic generation of input sequences for a program with respect to some environment constraints of the program under test. The environment constraints correspond to assumptions on the possible behaviours of the environment of the program under test. Lutess automatically builds a test data generator and a test harness. The latter links the generator, the software under test and the oracle, coordinates their execution and records the sequences of input-output values and the associated oracle verdicts. The program must be synchronous, and the environment constraints must be written in Lustre, a synchronous programming language.

Lustre [8] is a language designed for programming reactive synchronous systems. With a synchronous program, computation is performed during the actual time that an external process occurs, in order that the computation results can be used to control, monitor or respond in a timely manner to the external process. Assuming the time is divided in discrete instants defined by a global clock, a synchronous program, at instant *t*, reads inputs i_t, computes and issues outputs o_t. The synchrony hypothesis states that the computation of o_t is made instantaneously, at instant *t*. A synchronous Lustre program is structured into nodes.

Let us consider the following Lustre program:

```
node Never (A : bool) returns (never_A : bool);
let
never_A = not A -> (not A and pre (never_A));
tel
```

This program has one boolean data as input and one boolean data as output. At any time, the output is true if and only if the input has never been true since the beginning of the program execution. For instance, the program produces the output sequence (true, true, true, false, false) for the input sequence (false, false, false, true, false).

A Lustre node consists of a set of equations defining outputs as functions of inputs and local variables. A Lustre expression is made up of constants, variables as well as logical, arithmetic and Lustre-specific operators. There are two Lustre-specific temporal operators: "pre" and "->". "pre" makes it possible to use the last value an expression has taken (at the last tick of the clock). "->", also called "followed by", is used to assign initial values (at t = 0) to expressions:

- If E is an expression denoting the sequence (e0, e1, ..., en, ...), **pre** E denotes the sequence (nil, e0, e1, ..., en-1, ...) where nil is an undefined value. In other words, pre E returns, at a moment t, the value of the expression E at the moment t-1.
- If E and F are expressions denoting, respectively, the sequences (e0, e1, e2, ..., en ...) and (f0, f1, f2, ..., fn ...), E -> F denotes the sequence (e0, f1, f2, ..., fn, ...).

Basic logical and temporal operators expressing invariants or properties can be implemented in Lustre. For example, once_from_to(A,B,C) states that property A must hold at least once between the instants where events B and C occur.

As shown in Figure 2, Lutess requires the environment description and a test oracle of the software under test. The test is operated on a single action-reaction cycle: The generator randomly selects an input vector and sends it to the software, which 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 properties are violated. The test data generator is automatically built by Lutess from an environment description written in Lustre while the software and the oracle are both executable programs (possibly written in LUSTRE). During a test run, at each execution cycle (or step), the Lutess generator randomly selects an input vector consistent with the environment description assuming that the data distribution is uniform. Additional strategies are supported by Lutess, consisting in guiding the test data generation by means of operational profiles [12], behavioural patterns [6] or according to the likelihood to violate safety properties [15]. In this paper we focus on the use of operational profiles to express relevant interaction scenarios for multimodal interactive applications.

5. Testing interactive multimodal systems with Lutess

5.1 Hypotheses

Although Lutess is a testing environment originally dedicated to synchronous software, we propose its use for testing interactive systems. Indeed, in theory, a synchronous software satisfies the synchrony hypothesis stating that outputs are computer instantaneously. But in practice, this hypothesis holds when the software is able to take into account any evolution of its external environment. Hence, a multimodal interactive system can be viewed as a synchronous program as long as all the user's actions and external stimuli are caught. This means that, during the test operation, test data will be issued only when the software under test is ready to catch them.

Lutess focuses on the control part of the software under test. In other words it checks the ability of the software to successfully transform an input event sequence into adequate outputs. Hence, considering an interactive multimodal system as the software under test, the aim of the test will be to check that a sequence of user's events (represented as boolean events) is adequately processed and results in an appropriate output sequence of events.

5.2 Motivations

Although multiple modalities and forms of multimodality enhance the flexibility, robustness and efficiency of the interaction, they also increase the complexity of the software that must consequently be able to handle a huge variety of input sequences. For testing such software, the number of input event sequences to be considered is therefore increased and motivate our approach of automatic test. Moreover the software and especially the fusion mechanism [11] depends on the temporal window within which the user events occur. For example when two modalities are used in a complementary or redundant way, the resulting events are combined based on a temporal window [11]. Such temporal aspects of the interaction can be tested with Lutess, as it is shown, for instance, in section 6.4.2. To summarize, Lutess makes possible the automatic generation of several and long context aware input sequences, and therefore can be a powerful tool to test multimodal systems.

5.3 Main issues

5.3.1 Connecting Lutess to a multimodal system

Linking a multimodal system and Lutess sets the level of abstraction of the user's events generated by Lutess. Indeed the level of abstraction of the events will determine which component within the multimodal system will be connected to Lutess. If we consider the PAC-Amodeus software architecture for multimodal systems presented in [2],[11] three components can be candidates to receive the input sequences from Lutess as shown in Figure 2. Indeed, since Lutess cannot generate physical actions, the Physical Interaction component is not a possible candidate for plugging Lutess.

A first solution is therefore to connect Lutess with the Logical Interaction component. As a consequence, Lutess should send low-level device dependent event sequences to the multimodal system under test. For example, in the case of Memo, Lutess should send events corresponding to a mouse button press. A second solution consists in connecting Lutess to the fusion mechanism. Events generated by Lutess are therefore modality (device and language) dependent. For example for testing Memo, Lutess can send events such as <Mouse-get> or <Speech-remove>. A third solution is to connect Lutess to the Dialog Controller. Events sent by Lutess to the multimodal system will therefore be complete commands such as <remove note 3>. For the experiment presented in the paper, the second solution has been chosen (see section 6.1)

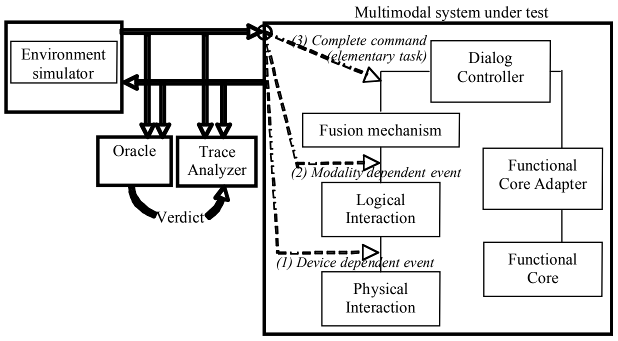

Figure 2: Three solutions for linking a multimodal system organized along the PAC-Amodeus software architecture and Lutess.

5.3.2 Developing the specifications

In order to test a multimodal system with Lutess, we need:

- The system to test, as an executable program. An *event translator* must be added to the program, translating the program input and output events to boolean events handled by Lutess.
- A test oracle describing the properties that the system must meet (such as CARE properties).
- The Lustre specification of the external environment behaviour. This specification describes the stimuli captured by the interactive system, typically the user behaviour. For the case of a context-aware interactive system where the physical environment of the user has an impact on the system, the specification may correspond to variable contexts in addition to user's behaviour.

5.3.3 Guiding by means of operational profiles

With the above specifications, the interactive system can be tested by randomly simulating the user behaviour. However, the user behaviour is seldom random and usually consists of sequences of actions intending to accomplish a precise task. To simulate such, more realistic, user behaviours, Lutess offers several test data generation techniques. In this paper we focus on operational profile–guided generation [12].

According to [10], the construction of an operational profile involves five steps. The first four steps consist in determining the customer, the user, the system-mode and the functional profiles, while the fifth step is the actual construction of the operational profile. We focus on the last step which comprises five main tasks: dividing execution into runs, defining the input space, partitioning the input space and finally associating occurrence probabilities with each partition. The definition of the input space corresponds, in our case to the environment specification. The latter is a set of invariant Lustre temporal logic formulas and provides us with a concise representation of the input space which corresponds to a (potentially infinite) set of sequences. The probability assignment supported by Lutess is of two kinds: unconditional and conditional [12].

Note that specification methods associating occurrence probabilities with input values according to their past values have been proposed, for instance, in [17][18]. In [17] only the last value taken on by the input variables is taken into account while the method proposed in [18] allows probabilities association to history classes : each class correspond to several sequences of input values. Lutess supports a probability association according to any past value taken by the inputs. Moreover, Lustre, which is used for the environment specification, is also used for the specification of the conditions on which depend the probabilities.

6. Testing MEMO with Lutess

We illustrate our testing approach by considering the test of the multimodal system Memo with Lutess.

- Section 6.1 presents the implementation issues related to linking Lutess and Memo.
- Section 6.2 exposes the test oracle expressed in Lustre for Memo.
- Section 6.3 is dedicated to the Lustre specification of the environment of Memo.
- Section 6.4 shows how various operational profiles can be built with conditional and unconditional occurrence probabilities associated with input values. Occurrence probabilities are also used to test modality fusion related issues, as they can force inputs events to occur in the same temporal window.
- Finally, section 6.5 provides commented experimental results, including various operational profile definitions, from the Memo case study.

6.1 Linking Memo and Lutess

The point of contact between Memo and Lutess consists of a Java class *MemoLutess* responsible for translating Lutess outputs into Memo inputs and vice-versa. Because we focus on testing the multimodal interaction with Memo, we set the level of abstraction of events generated by Lutess at the modality level. It corresponds to case (2) of Figure 2. Inputs generated by Lutess and received by the Fusion components of Memo are the following: (1) *Localization* is a boolean vector which indicates the user's movements along the x, y and z axes. For instance, Localization[xplus]=true means that the user's x-coordinate increases. For the case of Memo, we fix the decrement/increment equal to 5 cm (current position +/- 0.05 along x, y or z axis). (2) *Orientation* is a boolean vector, which indicates the changes in the user's orientation along the three orientation angles: yaw, pitch and roll. For instance, Orientation[pitchplus] indicates that the user is bending one's head. (3) *Mouse*, *Keyboard* and *Speech* are boolean vectors corresponding to a get, set or remove command specified using speech, keyboard or mouse. For instance, Mouse[get] indicates that the user has pressed the mouse button corresponding to a get command.

The state of the Memo system is observed through five boolean outputs: (1) *memoSeen*, which is true when at least one note is visible and close enough to the user to be manipulated, (2) *memoCarried*, which is true when the user is carrying a note, (3) *memoTaken*, which is true if the user has got a note during the previous action-reaction cycle, (4) *memoSet*, which is true if the user has set a carried note to appear at a specific place during the previous cycle, (5) *memoRemoved*, which is true if the user has removed a note during the previous cycle.

The class MemoLutess includes a constructor, creating a new instance of a Memo system. A main method creates a new instance of MemoLutess and links it to Lutess.

```
/* Main method */
static public main(String[] args) {
   MemoLutess m = new MemoLutess();
   m.connectLutess(); }
```

The *connectLutess* method consists of an infinite loop which (1) gets a sequence of boolean values specified by Lutess, (2) sends the corresponding events to the Memo system, (3) waits for Memo to execute the resulting commands, (4) gets the Memo current state (5) and finally issues the obtained sequence of boolean values that will be used in turn by Lutess to produce a new sequence for the following action-reaction cycle.

```
/* Main interaction loop */
void connectLutess() {
  while (true) {
  readInputs(); // Get an input from Lutess
  memoApp.sendEvents(); // Send events to Memo
  wait(500); // wait for Memo to do the commands
  memoApp.getState(); // Get the new state of Memo
  writeOutputs();}} // Issue results to Lutess
```

6.2 Test oracle

The test oracle consists of the required system properties. The Memo properties hereafter are functional. First, we require that notes are taken, set or removed only with appropriate commands:

- After a note has been seen and before it has been taken, a "get" command has to occur at an instant when the note is seen (i.e., the note is close enough to the user to be manipulated).

```
   once_from_to(cmdget and pre memoSeen, memoSeen,
memoTaken)
```

- After a memo has been seen or carried and before it has been removed, a "remove" command must occur.

```
   once_from_to(cmdremove and (pre memoSeen or pre
memoCarried), memoSeen or memoCarried,memoRemoved)
```

Moreover the state of the Memo system cannot change except by means of suitable input events:

- Between the instant the user is seeing a note and the instant there is no note in her/his visual field, the user has moved or specified a "get" or "remove" command.

```
   once_from_to(move or (cmdget or cmdremove) and
pre memoSeen, memoSeen, not memoSeen)
```

- Between the instant when no note is visible and the instant when a note is visible, the user has moved or has specified a "set" command.

```
   once_from_to(move or (cmdset and pre
memoCarried), not memoSeen, memoSeen)
```

- If a note is carried, then a "get" command has previously occurred.

```
   once_from_to(cmdget and pre memoSeen, not
memoCarried, memoCarried)
```

- Only a "set" or a "remove" command can cause a carried note to be dropped.

```
   once_from_to(cmdset or cmdremove, memoCarried,
not memoCarried)
```

CARE related properties can also be specified in the test oracle by means of temporal operators, as it has been shown in [9].

6.3 Environment and user's behaviour

Input data are generated by Lutess according to formulas defining assumptions about the external environment of Memo, i.e. the user's behaviour. The below specifications exclude actions that the user cannot perform. For example the user cannot move along an axis in both directions at the same time. The corresponding formulas are:

```
not (Localization[xminus] and Localization[xplus])
not (Localization[yminus] and Localization[yplus])
not (Localization[zminus] and Localization[zplus])
```

Similarly, we can also specify that the user cannot turn around an axis in both directions at the same time.

Moreover, Lutess sends data to Memo at the modality level and not at the device level. Since there is one abstraction process per modality, only one data along a given modality can be sent at a given time. Three commands (Get, Set, Remove) can be performed using speech, keyboard or mouse. We therefore have the following formulas:

```
AtMostOne(3,Mouse); AtMostOne(3,Keyboard);
AtMostOne(3,Speech).
```

6.4 Guiding the test data generation by means of operational profiles

6.4.1 Associating probabilities with inputs

As opposed to usual reactive systems, very few restrictions can be set to user behaviour. This means that, according to the environment specification of section 6.3, a random simulation of the user's actions cannot result in realistic interaction scenarios. Indeed, every input event has the same probability to occur. This means, for instance, that Localization[xminus] will occur as many times as Localization[xplus] and, as a result, the user's position will hardly change. To test Memo in a more realistic way, the data generation can be guided by means of operational profiles and more precisely, by means of unconditional or conditional occurrence probabilities associated with inputs.

Unconditional probabilities can be used to force the simulation to correspond to a particular case, for example that the user is turning one's head to the right:

```
    proba((Orientation[yawminus], 0.80),
          (Orientation[yawplus], 0.01),
       (Orientation[pitchminus], 0.01),
        (Orientation[pitchplus], 0.01),
        (Orientation[rollminus], 0.01),
          (Orientation[rollplus, 0.01))
```

Conditions can be associated with probabilities. For instance, one can specify that a "get" command has a high probability to occur when the user has a note in her/his visual field (close enough to be manipulated):

```
proba((Mouse[get], 0.8, pre memoSeen),
     (Keyboard[get], 0.8, pre memoSeen),
     (Speech[get], 0.8, pre memoSeen))
```

As another example, we can specify that, when there is no note visible, the user will very probably move:

```
proba((Orientation[yawminus], 0.9, not pre
MemoSeen)…).
```

6.4.2 Checking the fusion of multimodal events

When two modalities are used in a complementary or redundant way, the resulting events are combined according to their occurrence instant position in a temporal window. Let T be the duration of this temporal window (this is a parameter of the interactive system) and let C be the duration of an execution cycle of the Lutess test generator (that is the time separating the issue of two successive inputs). C is empirically determined and it is constant for a given generation type. Therefore, if N = T div C, then N is approximately the number of execution cycles included in the temporal window. As a result, for an input event to occur within the temporal window, its occurrence probability must be greater or equal to 1/N.

For example, to specify that Mouse[get] and Speech[get] will both be issued in that order in the same temporal window, we can write:

```
proba(Speech[get], 1/N, after(Mouse[get]) and pre
always_since(not Speech[get], Mouse[get]));
```

Indeed, this formula means that if at least a Mouse[get] event has occurred in the past and if no Speech[get] event occurred since the last Mouse[get] occurrence, then the Speech[get] occurrence probability is equal to 1/N. Since the temporal window starts at the last occurrence of Mouse[get] and lasts N ticks, Speech[get] will very probably occur at least once before the end of the window. The experimental results presented in the next section use the above principle to check the validity of the CARE properties for the Memo interactive system (section 6.5.2).

6.5 Commented experimental results

6.5.1 Random simulation

We first tested Memo with a random simulation (i.e. random generation of inputs consistent with the environment specification of section 6.3). Figure 3 shows an excerpt from the resulting trace. The last column contains the value of the oracle (1 means "true"). We use the following abbreviations:

- (*ya*, *p*, *r*) respectively for yaw, pitch and roll,
- (*mG*, *mS*, *mR*) respectively for Mouse[get], Mouse[set] and Mouse[remove],
- similarly (*kG*, *kS*, *kR*) for the keyboard modality and (*sG*, *sS*, *sR*) for speech,
- *Se* for memoSeen, *Car* for memoCarried,
- *Tak* for memoTaken and *Rem* for memoRemoved

```
x-  z-  -   -   -   -   -   ya+ -   p+  -   r+  -   -   -   -   -   -   -   -   -   Se  -   -   -   1
-   -   -   -   -   -   -   -   -   -   -   -   mG  -   -   -   -   -   -   -   -   Se  Car Tak -   1
-   -   -   z+  y+  -   -   ya+ p-  -   -   -   -   -   -   -   -   -   -   -   -   Se  Car -   -   1
-   -   -   -   -   -   -   -   -   -   -   -   -   -   -   kG  -   -   -   -   sR  -   -   Tak Rem 1
x-  z-  -   -   -   y-  -   -   -   -   r-  -   -   -   -   -   -   -   -   -   -   Se  -   -   -   1
-   -   -   -   -   -   -   -   -   -   -   -   -   mS  -   -   -   -   -   -   sR  Se  -   -   Rem 1
x-  z-  -   -   y+  -   ya- -   p-  -   -   -   -   -   -   -   -   -   -   -   -   Se  -   -   -   1
-   -   -   -   -   -   -   -   -   -   -   -   -   mS  -   -   kS  -   sG  -   -   -   Car Tak -   1
-   -   x+  -   y+  -   ya- -   -   -   -   r+  -   -   -   -   -   -   -   -   -   -   Car -   -   1
-   -   -   -   -   -   -   -   -   -   -   -   mG  -   -   kG  -   -   sG  -   -   -   Car -   -   1
x-  -   -   -   -   y-  -   -   p-  -   r-  -   -   -   -   -   -   -   -   -   -   -   Car -   -   1
-   -   -   -   -   -   -   -   -   -   -   -   -   -   mR  -   -   kR  -   -   sR  -   -   -   Rem 1
```

Figure 3 : Excerpt from Memo random simulation.

6.5.2 Using operational profiles

We next tested Memo using operational profiles. As shown by the following experimental results, we focused on the CARE properties. *Complementarity* has been extensively tested since the manipulation of a note implies two fusions to be performed. A fusion takes place to combine the information from the localization and orientation modalities, in order to determine the selected note. A second fusion is then performed in order to combine the selected note and the command issued by speech or by using the keyboard or mouse. As shown in the execution traces presented hereafter, *equivalence* has also been considerably tested: the three equivalent modalities based on mouse, keyboard and speech for issuing one of the commands (Get, Set, Remove) have been frequently simulated. *Redundancy* has been tested in our third experiment. *Assignment* has not been tested, since it only concerns the Memo exit command.

First experiment

In this first experiment, we choose probabilities such that the user is likely to move when no note is in his visual field, and likely to issue a "get" command otherwise.

```
proba(
(Mouse[get], 0.9, pre memoSeen),
(Clavier[get], 0.7, pre memoSeen),
(Speech[get], 0.5, pre memoSeen),
(Localisation[xminus], 0.5, not pre memoSeen),
(Localisation[zminus], 0.5, not pre memoSeen),
(Localisation[xplus],  0.8, not pre memoSeen),
(Localisation[zplus],  0.8, not pre memoSeen),
(Localisation[yplus],  0.5, not pre memoSeen),
(Localisation[yminus], 0.8, not pre memoSeen));
```

In the following excerpt of the resulting trace, we can note that, when no note is visible, the user moves, and when a note is visible (*Se* occurs in the previous step) the user takes it (*Tak*):

```
 1         :  -  -  -  -  -  -  -  -  -  Se -
 2         :  -  -  -  -  -  -  mG kG sG -  Tak
 3         :  -  z- x+ -  y+ -  -  -  -  -  -
 4         :  x- -  -  z+ -  y- -  -  -  -  -
 5         :  -  -  x+ z+ y+ -  -  -  -  -  -
 6         :  -  z- x+ -  y+ -  -  -  -  -  -
 7         :  -  z- -  -  -  -  -  -  -  -  -
 8         :  -  -  -  -  -  -  -  -  -  Se -
 9         :  -  -  -  -  -  -  mG -  sG -  Tak
10         :  x- z- -  -  -  -  -  -  -  -  -
11         :  -  z- -  -  y+ -  -  -  -  -  -
12         :  x- z- -  -  -  y- -  -  -  Se -
13         :  -  -  -  -  -  -  mG kG sG Se Tak
14         :  -  -  -  -  -  -  mG -  -  Se Tak
15         :  -  -  -  -  -  -  mG kG sG -  Tak
16         :  x- z- -  -  y+ -  -  -  -  Se -
17         :  -  -  -  -  -  -  mG kG -  -  Tak
18         :  x- -  -  z+ y+ -  -  -  -  -  -
19         :  x- z- -  -  y+ -  -  -  -  -  -
20         :  -  -  x+ z+ y+ -  -  -  -  -  -
21         :  x- z- -  -  y+ -  -  -  -  -  -
22         :  -  z- x+ -  -  y- -  -  -  -  -
23         :  -  -  x+ z+ y+ -  -  -  -  -  -
```

Second experiment

For this second experiment, we have first set a few notes along the x-axis. The aim is that the user, moving along the x-axis, removes these notes. We choose to let the user continue his move in the same direction with a high probability. When a note is visible, there is a high probability that the user will remove it. Finally, when a note is removed, there is a high probability that the user will change his direction.

```
proba(
  (Localisation[xplus], 0.9, pre always_since(
    not Localisation[xminus], Localisation[xplus])),--(1)
  (Localisation[xplus], 1, pre always_since(
    not Localisation[xplus],
    Localisation[xminus]) and pre memoRemoved),      --(2)
  (Localisation[xminus], 0.9, pre always_since(
    not Localisation[xplus], Localisation[xminus])),--(3)
  (Localisation[xminus], 1, pre always_since(
    not Localisation[xminus],
    Localisation[xplus]) and pre memoRemoved),       --(4)
  (Mouse[remove], 0.9, pre memoSeen ),               --(5)
  (Clavier[remove], 0.9, pre memoSeen),              --(6)
  (Speech[remove], 0.9, pre memoSeen));              --(7)
```

Line `(1)` means that there is a high probability to move along (x+), if the user has not moved along (x-) since the last occurrence of (x+) (the last move was x+). Line (2) means that there is a high probability to change the user direction when a note has just been removed. Lines (3), (4) are similar to (1), (2).

In the following excerpt from the resulting trace, we can note that generally the user moves towards the same direction until finding a note (step 64: event memoSeen *Se*). Then s/he removes it (step 65: event memoRemoved *Rem*). Then the user changes his direction, and so on.

```
53         :  -  -  mR kR sR -  -
54         :  -  x+ -  -  -  -  -
55         :  -  -  mR kR -  -  -
56         :  -  x+ -  -  -  -  -
57         :  -  -  -  kR -  -  -
58         :  -  x+ -  -  -  -  -
59         :  -  -  -  -  -  -  -
60         :  -  x+ -  -  -  -  -
61         :  -  -  -  -  -  -  -
62         :  -  x+ -  -  -  -  -
63         :  -  -  mR -  -  -  -
64         :  -  x+ -  -  -  Se -
65         :  -  -  -  kR sR -  Rem
66         :  x- -  -  -  -  -  -
67         :  -  -  -  -  sR -  -
68         :  x- -  -  -  -  -  -
69         :  -  -  mR -  sR -  -
70         :  -  x+ -  -  -  -  -
71         :  -  -  -  -  -  -  -
72         :  -  x+ -  -  -  -  -
73         :  -  -  -  -  -  -  -
74         :  -  x+ -  -  -  -  -
75         :  -  -  -  -  -  -  -
76         :  -  x+ -  -  -  -  -
77         :  -  -  -  kR -  -  -
78         :  -  x+ -  -  -  Se -
79         :  -  -  -  -  -  -  -
80         :  -  x+ -  -  -  -  -
81         :  -  -  -  -  sR -  -
82         :  -  x+ -  -  -  Se -
83         :  -  -  mR kR sR -  Rem
```

Third experiment

The third example describes a redundant usage of two modalities: mouse and speech. We have reconfigured the Memo system for allowing "redundancy[2]" between mouse and speech. In this mode, to execute a command, one event from every redundant modality is necessary and both events must occur in the same temporal window.

We first consider a high probability to issue a "get" command by using mouse and speech when a note is visible:

```
proba (  (Speech[get], 0.9, pre memoSeen),
         (Mouse[get],  0.9, pre memoSeen));
```

Here is an excerpt from the resulting trace:

```
1          :  -  -  Se -   -
2          :  -  -  Se -   -
3          :  mG sG Se Car Tak
4          :  -  -  Se Car -
5          :  mG -  Se Car -
```

We start the test in a state where two notes are close to the user. Step 2 contains the event memoSeen (*Se*), implying that one or several notes are close to the user. In step 4, the two simultaneous events mouseGet and speechGet (*mG* and *sG*) cause, because of the redundancy, one note to be taken (*Tak*). Thus, a note is still visible (*Se*), and the user carries one note (*Car*). Note that in step 5, the single event mouseGet (*mG*) does not cause any reaction, because in this mode we need two events to accomplish the task.

Let's now assume now that the following situation must be tested: when a note is visible, a "get" command with mouse and a "get" command with speech are issued in the same temporal window, but not at the same instant. Such a scenario checks the redundancy fusion mechanism. For this, we give a probability *pr* to issue `Speech[get]` when a memo is visible and `Speech[get]` has not yet occurred since the last occurrence of `Mouse[get]`. Because we do not want both events to occur at the same instant, we give the probability 0 to the `Mouse[get]` condition.

```
proba (
  (Speech[get], pr, pre memoSeen and
   after (Mouse[get]) and
   pre always_since(not Speech[get], Mouse[get])),
  (Mouse[get], 0, pre memoSeen and
   after (Mouse[get]) and
   pre always_since(not Speech[get], Mouse[get])) );
```

The value of *pr* is chosen as follows (see section 6.4.2). Let T=5000ms be the duration of the temporal window and let C=1000ms be the duration of a cycle of execution (i.e. the frequency of the input events generated by Lutess). If *pr* = 1000/5000 = 0.2, then `Speech[get]` will occur about one time every 5 cycles of execution when the precondition is true (`Mouse[get]` has occurred and `Speech[get]` has not occurred since `Mouse[get]`). If we wish both events to

[2] As opposed to the original definition of Redundancy provided in section 3, the two modalities are here required to accomplish a task. Redundancy-Equivalence (see fourth experiment) requires only one of several modalities.

be closer, we have to increase *pr*. We have processed this example with different values of *pr*:

Here is an excerpt from the resulting trace for *pr* = 0.2:

```
   1        :  mG -  Se -   -
   2        :  -  -  Se -   -
   3        :  -  -  Se -   -                    T
   4        :  -  -  Se -   -
   5        :  -  -  Se -   -
   6        :  -  -  Se -   -
   7        :  -  -  Se -   -
   8        :  -  -  Se -   -
   9        :  -  sG Se -   -
  10        :  -  -  Se -   -
  11        :  -  sG Se -   -
  12        :  -  -  Se -   -
  13        :  -  -  Se -   -
  14        :  -  -  Se -   -
…………
```

The temporal distance between the two redundant events mouseGet and speechGet (*mG*, *sG*) is equal to 8 cycles (8000 ms), which is more than T, so no task is executed. We can note that, considering the whole trace, the average distance between *mG* and *sG* is about 5 cycles (5000ms).

Here is an excerpt from the resulting trace for pr = 0.8:

```
 145        :  -  -  Se -   -
 146        :  -  -  Se -   -
 147        :  mG -  Se -   -
 148        :  -  -  Se -   -
 149        :  -  sG Se Car Tak                  T
 150        :  -  -  Se Car -
 151        :  -  -  Se Car -
```

The average distance between *mG* and *sG* is lower. We can observe that the events *mG* and *sG*, which occur in the same temporal window, cause one note to be taken and the other to remain in the physical field.

Fourth experiment

In the last experiment, we use the same expressions of probabilities used in the previous experiment, but we have configured the Memo system in order to work in the Redundancy-Equivalence mode. In this mode, the application uses the modes Redundancy and Equivalence at the same time: for two events occurring in the same temporal window and carrying the same information, there is only one executed task (Redundancy mode). However, a single event can cause the task execution (Equivalence mode).

We first choose the probabilities as follows:

```
proba (  (Speech[get], 0.9, pre memoSeen),
         (Mouse(get],  0.9, pre memoSeen) );
```

Here is an excerpt from the resulting trace for N = T/C = 10 000/1000 = 10

```
…
  10        :  -  -  Se -   -
  11        :  mG sG Se Car Tak
  12        :  -  -  Se Car -
  13        :  mG -  Se Car -
  14        :  -  -  Se Car -
  15        :  mG sG Se Car -
  16        :  -  -  Se Car -                    T
  17        :  mG sG Se Car -
  18        :  -  -  Se Car -
  19        :  mG sG Se Car -
  20        :  -  -  Se Car -
  21        :  mG -  -  Car Tak
  22        :  -  -  -  Car -
  23        :  -  -  Se -   -
  24        :  -  -  Se -   -
  25        :  mG sG Se -   -
  26        :  -  -  Se -   -
  27        :  mG sG Se -   -
  28        :  -  -  Se -   -
  29        :  mG sG Se -   -
```

As in the previous example, we first set two notes close to the user. Step 11 shows that, because of the redundancy mode, the two simultaneous events mouseGet and speechGet (*mG* and *sG*) cause one note to be taken (*Tak*). Thus, one note is still visible (*Se*), and one note is carried (*Car*). After that, all the events in the same temporal window are ignored. In step 21, because of the equivalence mode, the single event mouseGet (*mG*) cause a digital note to be taken.

Assume that we wish to check the fusion mechanism. For this, we choose the probabilities as in the third experiment: we assign a probability *pr* to `Speech[get]` when a memo is visible and `Speech[get]` has not yet occurred since the last occurrence of `Mouse[get]`. We assign the probability 0 to `Mouse[get]` to avoid both events to occur at the same instant.

```
proba (
(Speech[get], pr, pre memoSeen and
   after(Mouse[get]) and
   pre always_since(not Speech[get], Mouse[get])),
(Mouse[get], 0, pre memoSeen and
   after(Mouse[get] and
   pre always_since(not Speech[get], Mouse[get])) );
```

Here is an excerpt from the trace with *pr* = 0.8 and N = 5000/1000 = 5 cycles:

```
   1        :  -  -  -  -   -
   2        :  -  -  Se -   -
   3        :  -  -  Se -   -
   4        :  -  -  Se -   -
   5        :  -  -  Se -   -
   6        :  -  -  Se -   -
   7        :  -  -  Se -   -
   8        :  mG -  Se Car Tak
   9        :  -  -  Se Car -
  10        :  -  -  Se Car -
  11        :  -  sG Se Car -
  12        :  -  -  Se Car -
  13        :  -  sG -  Car Tak
  14        :  -  -  -  Car -
  15        :  -  -  -  Car -
  16        :  -  -  -  Car -
  17        :  -  sG Se -   -
  18        :  -  -  Se -   -
  19        :  -  -  Se -   -
  20        :  -  -  Se -   -
  21        :  mG -  Se -   Tak
  22        :  -  -  Se -   -
  23        :  -  sG Se -   -
  24        :  -  -  Se -   -
```

Note that at step 8, because of equivalence, the event mouseGet (*mG*) causes one note to be taken. Then, the event speechGet (*sG*) at step 11 is ignored, because of redundancy, and so on.

7. Conclusion and future work

In this article, we have presented a method for automatically testing multimodal systems by means of the Lutess environment, initially designed for synchronous software. Our hypothesis is that the behaviour of an interactive multimodal system is to a certain extent similar to the one of a synchronous system. Based on this hypothesis, we used the Lutess testing environment to test a prototype multimodal application, Memo. We focused on multimodal interaction in light of the CARE properties.

Although a more thorough empirical evaluation of the approach is necessary, this experiment has shown that Lutess, especially when used in the operational profile based generation mode, can simulate relevant interaction scenarios, involving modality fusion. The occurrence probability of the input modalities involved in a fusion can be easily computed and depends on the length of the temporal window and the duration of an execution step.

In future work, we plan to enhance the Lutess simulation engine in order to handle occurrence probabilities associated with logical expressions (rather than with single variables). Such an extension would make possible the expression of more complex execution scenarios and operational profiles.

8. References

[1] Bolt, R. Put That There: Voice and Gesture at the Graphics Interface. Proc. of SIGGRAPH'80. ACM Press (1980) 262-270.

[2] Bouchet, J., Nigay, L., & Ganille, T. ICARE Software Components for Rapidly Developing Multimodal Interfaces. Proc. of ICMI'04. ACM Press (2004) 251-258.

[3] Bouchet, J., Nigay, L. ICARE: A Component-Based Approach for the Design and Development of Multimodal Interfaces. Proc. of CHI'04 extended abstract. ACM Press (2004) 1325-1328.

[4] Coutaz, J., Nigay, L., Salber, D., Blandford, A., May, J., & Young, R. Four Easy Pieces for Assessing the Usability of Multimodal Interaction: The CARE properties. Proc. Of INTERACT'95. Chapman et Hall (1995) 115-120.

[5] d'Ausbourg, B. Using Model Checking for the Automatic Validation of User Interfaces Systems. Proc. of DSVIS'98. Springer Verlag (1998) 242-260.

[6] du Bousquet, L., Ouabdesselam, F., Richier, J.-L., & Zuanon, N. Lutess: a Specification Driven Testing Environment for Synchronous Software. Proc. of ICSE'99. ACM Press (1999) 267-276.

[7] Duke, D., Harrison, M. Abstract Interaction Objects. Proc. of Eurographics'93. North Holland (1993) 25-36.

[8] Halbwachs, N. Synchronous programming of reactive systems, a tutorial and commented bibliography. Proc. of CAV'98, LNCS 1427. Springer Verlag (1998) 1-16.

[9] L. Madani, L. Nigay, I. Parissis. Testing the care properties of multimodal applications by means of a synchronous approach. Proc. of IASTED Int'l Conference on Software Engineering, (2005).

[10] J. Musa. Operational Profiles in Software-Reliability Engineering. IEEE Software (1993), 14–32.

[11] Nigay, L., Coutaz, J. A Generic Platform for Addressing the Multimodal Challenge. Proc. of CHI'95. ACM Press (1995) 98-105.

[12] F. Ouabdesselam, I. Parissis. Constructing Operational Profiles for Synchronous Critical Software. Proc. of 6th Int'l Symposium on Software Reliability Engineering (1995).

[13] Palanque, P., Bastide, R. Verification of Interactive Software by Analysis of its Formal Specification. Proc. of INTERACT'95. Chapman et Hall (1995) 191-197.

[14] Parissis, I., Ouabdesselam, F. Specification-based Testing of Synchronous Software. Proc. of ACM SIGSOFT Fourth Symposium on the Foundations of Software Engineering. ACM Press (1996) 127-134.

[15] I. Parissis, J. Vassy. Thoroughness of Specification-Based Testing of Synchronous Programs. Proc. of 14th. IEEE International Symposium on Software Reliability Engineering (2003) 191-202.

[16] Paterno, F., Faconti, G. On the Use of LOTOS to Describe Graphical Interaction. Proc. of HCI'92. Cambridge University Press (1992) 155-173.

[17] J. Whittaker. Markov chain techniques for software testing and reliability analysis. Thesis, University of Tenessee (1992).

[18] D. Woit. Specifying Operational Profiles for Modules. Proc. of the International Symposium on Software Testing and Analysis (1993) 2–10.

[19] Zouinar, M. et al. Multimodal Interaction on Mobile Artefacts. Chapter 4 in Communicating with smart objects. Hermes Penton Science/Kogan Page Science (2003).