Formal Testing of Multimodal Interactive Systems

Jullien Bouchet, Laya Madani, Laurence Nigay, Catherine Oriat and Ioannis Parissis

Laboratoire d'Informatique de Grenoble (LIG)
BP 53 38041 Grenoble Cedex 9 - FRANCE,
{Forename.Name}@imag.fr

Abstract. This paper presents a method for automatically testing interactive multimodal systems. The method is based on the Lutess testing environment, originally dedicated to synchronous software specified using the Lustre language. The behaviour of synchronous 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. Under this hypothesis, the paper presents our method for automatically testing interactive multimodal systems using the Lutess environment. In particular, we show that automatic test data generation based on different strategies can be carried out. Furthermore, we show how multimodality-related properties can be specified in Lustre and integrated in test oracles.

1 Introduction

A multimodal system supports communication with the user through different modalities such as voice and gesture. Multimodal systems have been developed for a wide range of domains (medical, military, …) [5]. In such systems, modalities may be used sequentially or concurrently, and independently or combined synergistically. The seminal "Put that there" demonstrator [4] that combines speech and gesture illustrates a case of a synergistic usage of two modalities. The design space described in [25], based on the five Allen relationships, capture this variety of possible usages of several modalities. Moreover, the versatility of multimodal systems is further exacerbated by the huge variety of innovative input modalities, such as the phicons (physical icons) [14]. This versatility results in an increased complexity of the design, development and verification of multimodal systems.
Approaches based on formal specifications automating the development and the validation activities can help in dealing with this complexity. Several approaches have been proposed. As a rule, they consist of adapting existing formalisms in the particular context of interactive systems. Examples of such approaches are the Formal System Modelling (FSM) analysis [10], the Lotos Interactor Model (LIM) [23] or the Interactive Cooperative Objects (ICO), based on Petri Nets [21]. The synchronous approach has also been proposed as an alternative to modelling and verifying by model-checking of some properties of interactive systems [8]. Similarly to the previous approaches, the latter requires formal description of the interactive systems such as Lustre [13] programs on which properties, also described as Lustre programs, are checked. However, its applicability is limited to small pieces of software, since it seems very hard to fully specify systems in this language.

As opposed to the above approaches used for the design and verification, this paper proposes to use the synchronous approach as a framework for testing interactive

multimodal systems. The described method therefore focuses on testing a partial or complete implementation. It consists of automatically generating test data from enhanced Lustre formal specifications. Unlike the above presented methods, it does not require the entire system to be formally specified. In particular, the actual implementation is not supposed to be made in a specific formal language. Only a partial specification of the system environment and of the desired properties is needed.

The described testing method is based on Lutess [9, 22], a testing environment handling specifications written in the Lustre language [13]. Lutess has been designed to deal with synchronous specifications and has been successfully used to test specifications of telecommunication services [12]. Lutess requires a non-deterministic Lustre specification of the user behaviour. It then automatically builds a test data generator that will feed with inputs the software under test (i.e., the multimodal user interface). The test generation may be purely random but can also take into account additional specifications such as operational profiles or behavioural patterns. Operational profiles make it possible to test the system under realistic usage conditions. Moreover, they could be a means of assessing usability as has been shown in [24] where Markov models are used to represent various user behaviours. Behavioural patterns express classes of execution scenarios that should be executed during testing.

A major interest of synchronous programming is that modelling, and hence verifying, software is simpler [13] than in asynchronous formalisms. The objective of this work is to establish that automated testing based on such an approach can be performed in an efficient and meaningful way for interactive and multimodal systems. To do so, it is assumed, according to theoretical results [1], that interactive systems can, to some extent, be assimilated with synchronous programs. On the other hand, multimodality is taken into account through the type of properties to be checked: we especially focus on the CARE (Complementarity, Assignment, Redundancy, Equivalence) [7, 18] properties as well as on temporal properties related to the use over time of multiple modalities.

The structure of the paper is as follows: first, we present the CARE and temporal properties that are specific to multimodal interaction. We then explain the testing approach based on the Lutess testing environment and finally illustrate the application of the approach on a multimodal system developed in our laboratory, Memo.

2 Multimodal interaction: the CARE properties

Each modality can be used independently within a multimodal system, but the availability of several modalities naturally raises the issue of their combined usage. Combining modalities opens a vastly augmented world of possibilities in multimodal user interface design, studied in light of the four CARE properties in [7, 18]. These properties characterize input and output multimodal interaction. In this paper we focus on input multimodality only. In addition to the combined usage of input modalities, multimodal interaction is characterized by the use over time of a set of modalities.

The CARE properties (Equivalence, Assignment, Redundancy, and Complementarity of modalities) form an interesting set of relations relevant to characterization of multimodal systems. As shown in Fig. 1, while Equivalence and Assignment express the availability and respective absence of choice between multiple modalities for a given task, Complementarity and Redundancy describe relationships between modalities.

- Assignment implies that the user has no choice in performing a task: a modality is then assigned to a given task. For example, the user must click on a dedicated button using the mouse (modality = direct manipulation) for closing a window.
- Equivalence of modalities implies that the user can perform a task using a modality chosen amongst a set of modalities. These modalities are then equivalent for performing a given task. For example, to empty the desktop trash, the user can choose between direct manipulation (e.g. shift-click on the trash) and speech (e.g. the voice command "empty trash"). Equivalence augments flexibility and also enhances robustness. For example, in a noisy environment, a mobile user can switch from speech to direct manipulation using the stylus on a PDA. In critical systems, equivalence of modalities may also be required to overcome device breakdowns.
- Complementarity denotes several modalities that convey complementary chunks of information. Deictic expressions, characterised by cross-modality references, are examples of complementarity. For example, the user issues the voice command "delete this file" while clicking on an icon. In order to specify the complete command (i.e. elementary task) the user must use the two modalities in a complementary way. Complementarity may increase the naturalness and efficiency of interaction but may also provoke cognitive overload and extra articulatory synchronization problems.
- Redundancy indicates that the same piece of information is conveyed by several modalities. For example, in order to reformat a disk (a critical task) the user must use two modalities in a redundant way such as speech and direct manipulation. Redundancy augments robustness but as in complementary usage may imply cognitive overload and synchronization problems.

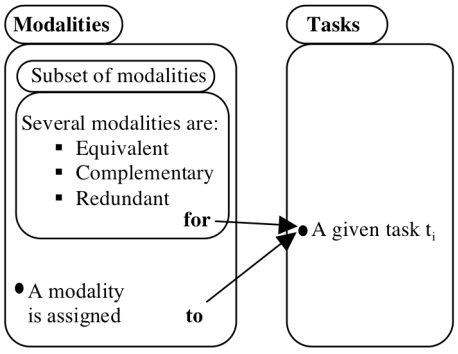

Fig. 1: The CARE relationships between modalities and tasks.

Orthogonal to the CARE relationships, a temporal relationship characterises the use over time of a set of modalities. The use of these modalities may occur simultaneously or in sequence within a temporal window Tw, that is, a time interval. Parallel and sequential usages of modalities within a temporal window are formally defined in [7]. The key point is that the corresponding events from different modalities occur within a temporal window to be interpreted as temporally related: the temporal window thus expresses a constraint on the pace of the interaction. Temporal relationships are often used by fusion software mechanisms [18] to detect complementarity and redundancy cases assuming that users' events that are close in time are related. Nevertheless, distinct events produced within the same temporal window through different modalities are not necessarily complementary or redundant. This is the case for example when the user is performing several independent tasks in parallel, also called concurrent usage of modalities [18]. This is another source of complexity for the software.

The CARE and temporal relationships characterise the use of a set of modalities. They highlight all the diversity of possible input event sequences specified by the user and therefore the complexity of the software responsible for defining the tasks from the captured users' actions. Facing this complexity, we propose a formal approach for testing the software of a multimodal system that handles the input event sequences. In [7], we study the compatibility between what we call system-CARE as defined above and user-CARE properties for usability assessment based on cognitive models such as PUM [3] or ICS [2]. In our formal approach for testing, we focus on system-CARE properties.

3 Formal approach for testing multimodal systems

Our approach is based on the Lutess testing environment. In this section, we first present Lutess and then explain how it can be used for testing multimodal systems. In [16] we presented a preliminary study showing the feasibility of our approach and a first definition of the CARE properties that we simplify here. Moreover in [17], we presented in the context of a case study, one way to generate test data, namely the operational profile strategy. In this section, we present the complete approach with three different ways of generating test data.

3.1 Lutess: A testing environment for synchronous programs

Lutess [9, 22] is a testing environment initially designed for functional testing of synchronous software with boolean inputs and outputs. Lutess supports the automatic generation of input sequences for a program with respect to environment constraints. The latter are assumptions on the possible behaviours of the program environment. Input data are dynamically computed (i.e. while the software under test is executed) to take into account the inputs and outputs that have already been produced.

Lutess automatically transforms the environment constraints into a test data generator and a test harness. The latter:

- links the generator, the software under test and the properties to be checked (i.e. the oracle), and

- coordinates the test execution and records the sequences of input/output values and the associated oracle verdicts (see Fig. 2).

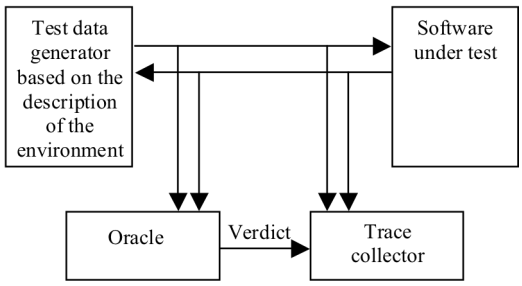

Fig. 2: The Lutess environment.

The test is operated on a single action-reaction cycle. The generator randomly selects an input vector and sends it to the software under test. The latter 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.

In addition to the random generation, several strategies, explained in Section 3.2.4, are supported by Lutess for guiding the generation of test data. In particular, operational profiles can be specified as well as behavioural patterns. The test oracle observes the inputs and the outputs of the software under examination, and determines whether the software properties are violated. Finally the collector stores the input, output and oracle values that are all boolean values.

The software under examination is assumed to be synchronous, and the environment constraints must be written in Lustre [13], a language designed for programming reactive synchronous systems. A synchronous program, at instant t, reads inputs i_t, computes and issues outputs o_t, assuming the time is divided in discrete instants defined by a global clock. The synchrony hypothesis states that the computation of o_t is made instantaneously at instant t. In practice, this hypothesis holds if the program computes the outputs within a time interval that is short enough to take into account every evolution of the program environment.

A Lustre program is structured into nodes. 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. For instance, the following program returns a “true” value everytime its input variable passes from "false" to "true" (rising edge).

```
node RisingEdge(in:bool;) returns(risingEdge:bool);
let
       risingEdge  =  false -> in and not pre in;
tel
```

An interesting feature of the Lustre language is that it can be used as a temporal logic (of the past). Indeed, basic logical and/or temporal operators expressing

invariants or properties can be implemented in Lustre. For example, OnceFromTo(A, B, C) specifies that property A must hold at least once between the instants where events B and C occur. Hence, Lustre can be used as both a programming and a specification language.

3.2 Using Lutess for testing multimodal systems

3.2.1 Hypotheses and motivations

The main hypothesis of this work is that, although Lutess is dedicated to synchronous software, it can be used for testing interactive systems. Indeed, as explained above, the synchrony hypothesis states that outputs are computed instantaneously but, in practice, this hypothesis holds when the software is able to take into account any evolution of its external environment (the theoretical foundations of the transformation of asynchronous to synchronous programs are provided in [1]). Hence, a multimodal interactive system can be viewed as a synchronous program as long as all the users' actions and external stimuli are caught. In a different domain than Human-Computer Interaction, Lutess has been already successfully used under the same assumption of testing telephony services specifications [12].

To define a method for testing multimodal input interaction we focus on the part of the interactive system that handles input events along multiple modalities. Considering the multimodal system as the software under test, the aim of the test is therefore to check that a sequence of input events along multiple modalities represented are correctly processed to obtain appropriate outputs such as a complete task. To do so with Lutess, one must provide:

1. *The interactive system as an executable program*: no hypothesis is made on the software implementation. Nevertheless, in order to identify levels of abstraction for connecting Lutess with the interactive system, we will assume that the software architecture of the interactive system is along the PAC-Amodeus software architecture [18]. Communication between Lutess and the interactive system also requires an event translator, translating input and output events to boolean vectors that Lutess can handle. We have recently shown [15] that this translator can be semi-automatically built assuming that the software architecture of the interactive system is along PAC-Amodeus [18] and developed using the ICARE component-based environment [5, 6]. In this study [15], we showed that the translator between Lutess and an interactive system can be built semi-automatically having some knowledge about the executable program and in our case the ICARE events exchanged between the ICARE components. Such a study can be done in the context of another development environment: our approach for testing multimodal input interaction is not dependent on a particular development environment (black box testing), as opposed to the formal approach for testing that we described in [11], where we relied on the internal ICARE component structure (white box testing). Indeed in [11], our goal was to test the ICARE components corresponding to the fusion mechanism.
2. *The Lustre specification of the test oracle*: this specification describes the properties to be checked. Properties may be related to functional or multimodal interaction requirements. Functional requirements are expressed as properties

independent of the modalities. Multimodal interaction requirements are expressed as properties on event sequences considering various modalities. We focus on the CARE and temporal properties described in Section 2. For instance, a major issue is the fusion mechanism [18], which combines input events along various modalities to determine the associated command. This mechanism relies on a temporal window (see Section 2) within which the users' events occur. For example, when two modalities are used in a complementary or redundant way, the resulting events are combined if they occur in the same temporal window; otherwise, the events are processed independently.

3. *The Lustre specification of the behaviour of the external environment of the system*: from this specification, test data as sequences of users' events are randomly generated. In the case of context-aware systems, in addition to a non-deterministic specification of the users' behaviour, elements specifying the variable physical context can be included. Moreover, additional specifications (operational profiles, behavioural patterns) make it possible to use different generation strategies.

In the following three sections, we further detail each of these three points, respectively, the connection, the oracle and the test data generation based on the specification of the environment.

3.2.2 Connection between Lutess and the interactive multimodal system

Testing a multimodal system requires connecting it to Lutess, as shown in Fig. 3. To do so, the level of abstraction of the events exchanged between Lutess and the multimodal system must be defined. This level will depend on the application properties that have to be checked and will determine which components of the multimodal system will be connected to Lutess.

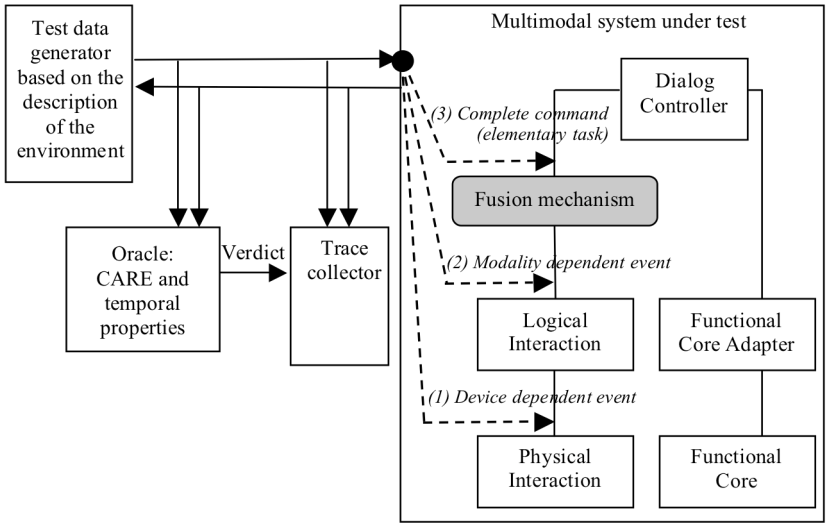

Fig. 3: Connection between Lutess and a multimodal system organized along the PAC-Amodeus model: three solutions.

In order to identify the levels of abstraction of the events exchanged between Lutess and the multimodal system, we must make assumptions on the architecture of the multimodal system being tested. We suppose that the latter is organized along the PAC-Amodeus software architectural model. This model has been applied to the software design of multimodal systems [18]. According to the PAC-Amodeus model, the structure of a multimodal system is made of five main components (see Fig. 3) and a fusion mechanism performing the fusion of events from multiple modalities. The Functional Core implements domain specific concepts. The Functional Core Adapter serves as a mediator between the Dialog Controller and the domain-specific concepts implemented in the Functional Core. The Dialog Controller, the keystone of the model, has the responsibility for task-level sequencing. At the other end of the spectrum, the Logical Interaction Component acts as a mediator between the fusion mechanism and the Physical Interaction Component. The latter supports the physical interaction with the user and is then dependent on the physical devices. Since our method focuses on testing multimodal input interaction, three PAC-Amodeus components are concerned: the Physical and Logical Interaction Components as well as the fusion mechanism. By considering the PAC-Amodeus components candidates to receive input events from Lutess, we identify three levels of abstraction of the generated events:

1. Simulating the Physical Interaction Component: generated events should be sent to the Logical Interaction Component. In this case, Lutess should send low-level device dependent event sequences to the multimodal system like selections of buttons using the mouse or character strings for recognized spoken utterances.
2. Simulating the Physical and Logical Interaction Components: generated events sent to the fusion mechanism should be modality dependent. Examples include <mouse, empty trash> or <speech, empty trash>.
3. Simulating the fusion mechanism: generated events should correspond to complete commands, independent of the modalities used to specify them, for instance <empty trash>.

Since we aim at checking the CARE and temporal properties of multimodal interaction and the associated fusion mechanism, as explained in Section 2, the second solution has been chosen: the test data generated by the Lutess test generator are modality dependent event sequences.

3.2.3 Specification of the test oracles

The test oracles consist of properties that must be checked. Properties may be related to functional and multimodal interaction requirements. Examples of properties related to functional requirements are provided in Section 4. In this section we focus on multimodality-related requirements and consider the CARE and temporal properties defined in Section 2: we show that they can be expressed as Lustre expressions and then can be included in an automatic test oracle (see [16] for a preliminary study on this point).

Equivalence:

Two modalities M_1 and M_2 are equivalent w.r.t. a set T of tasks, if every task $t \in T$ can be activated by an expression along M_1 or M_2. Let E_{AM1} be an expression along

modality M_1 and let E_{AM2} be an expression along M_2. E_{AM1} or E_{AM2} can activate the task $t_i \in T$. Therefore, equivalence can be expressed as follows:

```
OnceFromTo (E_AM1 or E_AM2, not t_i, t_i)
```

We recall (see Section 3.1) that OnceFromTo(A, B, C) specifies that property A must hold at least once between the instants where events B and C occur. Therefore, the above generic property holds if at least one of the expressions E_{AM1} or E_{AM2} has been set before the action t_i occurs.

Redundancy and Complementarity:

In order to define the two properties Redundancy and Complementarity that describe combined usages of modalities, we need to consider the use over time of a set of modalities. For both Redundancy and Complementary, the use of the modalities may occur within a temporal window Tw, that is, a time interval. As Lustre does not provide any notion of physical time, to specify the temporal window, we consider C to be the duration of an execution cycle (time between reading an input and writing an output). The temporal window is then specified as the number of discrete execution cycles:

N = Tw div C.

Two modalities M_1 and M_2 are redundant w.r.t. a set T of tasks, if every task $t \in T$ is activated by an expression E_{AM1} along M_1 and an expression E_{AM2} along M_2. The two expressions must occur in the same temporal window Tw: abs(time(E_{AM1}) - time(E_{AM2}) < Tw. Considering N = Tw div C, and the task $t_i \in T$, the Lustre expression of the redundancy property is the following one.

```
Implies (t_i,
        abs(lastOccurrence(E_AM1)- lastOccurrence(E_AM2))<= N
        and atMostOneSince(t_i, E_AM1) and atMostOneSince(t_i, E_AM2))
```

where:

- Implies(A, B) is the usual logic implication (not A or B).
- lastOccurrence(A) returns the latest instant that A occurred.
- atMostOneSince(A, B) is true when at most one occurrence of A has been observed since the last time that B has been true.

Two modalities are used in a complementary way w.r.t. a set T of tasks, if every task $t \in T$ is activated by an expression E_{AM1} along M_1 and an expression E_{AM2} along M_2. The two expressions must occur in the same temporal window Tw. We therefore get the same Lustre expression as for redundancy. Indeed Complementarity and Redundancy correspond to the same use over time of modalities and the difference relies on the semantic of the expressions along the modalities. While complementarity implies expressions with complementary meaning for the task considered (e.g. speech command "open" while clicking on an icon using the mouse), redundancy involves expressions conveying the same meaning (e.g., speech command "open paper.doc" while double-clicking on the icon of the file named paper.doc using the mouse). The meaning of the conveyed expressions is defined by the Lutess user (i.e. tester). Consequently, the same oracle is defined for redundancy and complementarity.

3.2.4 Strategies for generating test data

The automatic test input generation is a key issue in software testing. In the particular case of interactive systems, such a generation relies on the ability to model various users' behaviours and to automatically derive test data compliant with the models. Lutess provides several generation facilities and underlying models.

Constrained Random Generation:

The user is represented by a set of invariants specifying all its possible behaviours. The latter are randomly generated on an equal probability basis. More precisely, at every execution step, one of the input vectors satisfying the invariants will be fairly chosen among all the possible vectors.

Operational profiles:

Although the random generation is operated in a fair way, the resulting behaviour is seldom realistic. To cope with this problem, operational profiles can be defined by means of occurrence probabilities associated with user actions [19]. Occurrence probabilities can be conditional (that is, they will be taken into account during the test data generation only when a user-specified condition holds) or unconditional. Random generation is performed w.r.t. these probabilities.

An interesting feature of this generation mode is that it makes possible to issue events in the same temporal window and, hence, to check the fusion capabilities of a multimodal system. As we have shown in [19], one has to associate with the input events a probability computed from the temporal window duration to ensure that events will occur in the same temporal window. Let N be the number of discrete execution cycles corresponding to the full duration of the temporal window (computed as in Section 3.2.3). 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 A and B will both be issued in that order in the same temporal window, we can write:

```
proba(A, 1/N, after(B) and pre always_since(not A, B));
```

Indeed, this formula means that if at least a B event has occurred in the past and if no A event occurred since the last B occurrence, then the A occurrence probability is equal to 1/N. Since the temporal window starts at the last occurrence of B and lasts N ticks, A will very probably occur at least once before the end of the window.

Behavioural patterns:

Behavioural patterns make possible to partially specify a sequence of user actions. As opposed to the above operational profile-based generation mode, a behavioural pattern involves several execution instants. Behavioural patterns enable the description of executions that may not be easy to attained randomly and are hard to specify with occurrence probabilities. The random test input generation takes into account this partial specification of user actions.

4 Illustration: the Memo multimodal system

Memo [4] is an input multimodal system aiming at annotating physical locations with digital post it-like notes. Users can drop a note to a physical location. The note can then be read/carried/removed by other mobile users.

Fig. 4: A sketched view through the HMD: The Memo mobile user is in front of the computer science teaching building at the University of Grenoble and can see two digital notes.

A Memo user is equipped with a GPS and a magnetometer enabling the system to compute her/his location and orientation. The memo user is also wearing a head mounted display (HMD). Its semi-transparency enables the fusion of computer data (the digital notes) with the real environment as shown in Fig. 4.

In [17], we fully illustrate our testing method by considering the test of Memo using an operational profile-based approach for generating the test data. In order to illustrate all the strategies for generating test data, we consider here three tasks, namely "get a post-it", "set a post-it" and “remove a post-it” with Memo. For the manipulation of Memo notes, 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. As a consequence if s/he tries to get a note while already carrying one note, the action will have no effect. S/he can set a carried note to appear at a specific place. Issuing the set command without carrying a note has no effect. To perform the three tasks "get", "set" and "remove", the user has the choice between three equivalent modalities: issuing voice commands, pressing keys on the keyboard or clicking on mouse buttons. A command "get" or “remove” specified using speech, keyboard and mouse is applied to the notes that the user is looking at (i.e., the notes close to her/him). Memo can also be set to support redundant usage of modalities. 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 expressions are redundant and the two actions (speaking and pressing) produced nearly in parallel or close in time, the command will be executed and as a result the corresponding note will be deleted. If the two "remove" actions were not produced close in time, there is no redundancy detected and the remove command will therefore not be executed.

In the following sections and considering the three tasks "get", "set" and "remove", we illustrate our method by first explaining the connection between Lutess and

Memo. We then define the test oracle for Memo and finally explain how we automatically generate test data using different strategies.

4.1 Connection between Lutess and Memo

The connection between Memo and Lutess is made by a Java class, MemoLutess, in charge of translating Lutess outputs into Memo inputs and vice-versa. As explained in Section 3.2.1, we developed a method for semi-automatically generating this translator that we describe in [15] as an extension of the ICARE platform. For Memo, the code has been written manually without the ICARE platform. So the class MemoLutess has been written by hand. This class 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 is made of an infinite loop which (1) reads a sequence of inputs issued by the Lutess test data generator and (2) sends the corresponding events to the Memo system; then, it (3) waits for Memo to execute the resultant commands, (4) obtains the new Memo state (5) and sends the computed output vector to the Lutess generator.

```
/* Main interaction loop */
void connectLutess() {
  while (true) {
    readInputs();             // Read test inputs
    memoApp.sendEvents() ;    // Send corresponding events to Memo
    wait(N);                  // wait N ms for Memo to react
    memoApp.getState() ;      // Get the new state of Memo
    writeOutputs();}}         // Write outputs
```

As explained in Section 3.2.2, the level of abstraction is set at the modality level. Generated events are hence received by the fusion component of Memo. For the "get" "set" and "remove" tasks, the following events are involved in the interaction:

- *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. Similarly *Orientation* is a boolean vector, which indicates the changes in the user's orientation. For instance, Orientation[pitchplus] indicates that the user is bending one's head.
- *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 four boolean outputs:

- *memoSeen*, which is true when at least one note is visible and close enough to the user to be manipulated,
- *memoCarried*, which is true when the user is carrying a note,

- *memoTaken*, which is true if the user has get a note during the previous action-reaction cycle,
- *memoSet*, which is true if the user has set a carried note to appear at a specific place during the previous cycle,
- *memoRemoved,* which is true if the user has removed a note during the previous cycle.

4.2 Memo test oracle

The test oracle consists of the required Memo properties. First we consider functional properties. For example the state of Memo 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" command.

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

Moreover we specify that notes are taken or set only with appropriate commands. For example, 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.

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

Furthermore if a note is carried, then a "get" command has previously occurred.

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

In addition to functional properties, multimodality-related properties are specified in the test oracle, as explained in Section 3.2.3. For instance, to check that the task `memoTaken` takes place only after the occurrence of the redundant expressions `Mouse[get]` and `Speech[get]`, we should write the following test oracle:

```
node MemoOracle(-- application inputs and outputs
               )
               returns(propertyOK:bool);
let
       propertyOK =
               Implies (memoTaken,
                          abs(lastOccurrence(Mouse[get])-
                          lastOccurrence(Speech[get]))<= N
                          and
                          atMostOneSince(memoTaken i, Mouse[get]) and
                          atMostOneSince(memoTaken, Speech[get]));
tel
```

The above node states that (1) memoTaken occurs only when (1) Mouse[get] and Speech[get] occur in the same temporal window (of duration N) and that (2) in that case memoTaken occurs only once.

4.3 Memo test input generation

4.3.1 Modelling the environment and the users' behaviour

Input data are generated by Lutess according to formulas defining assumptions about the external environment of Memo, i.e. the users' behaviour. We here describe 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 also specify by three formulas 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. Since there is one abstraction process per modality, only one data along a given modality can therefore be sent at a given time. The commands "get", "set" and "remove" can be performed using speech, keyboard or mouse. We therefore get the following formulas*:

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

4.3.2 Guiding the test data generation

Random generation and operational profiles:

A random simulation of the users' actions results in sequences in which every input event has the same probability to occur. This means, for instance, that Localization[xminus] will occur as many times as Localization[xplus]. As a result, the users' position will hardly change. To test Memo in a more realistic way, the data generation can be guided by means of operational profiles (set of conditional or unconditional probabilities definition). Unconditional probabilities are 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)).
```

Conditional probabilities are used, for instance, to 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))
```

The following expression states that, when there is no note visible, the user will very probably move:

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

---

* Mouse is a boolean table of three elements indexed by "get", "set" and "remove": AtMostOne(3, Mouse) means that at most one of the elements of the table is true.

Behavioural patterns:

A pattern is a sequence of actions and conditions that should hold between two successive actions. During the random test data generation, inputs matching the scenario have a higher occurrence probability. Let us consider the scenario corresponding to the sequence of commands presented in Fig. 5: the user performs twice the "get" command, then a "set" command. The scenario also specifies that in between the first two "get" commands, the user does not perform a "set" command and similarly between the two "get" and "set" commands, no "get" command.

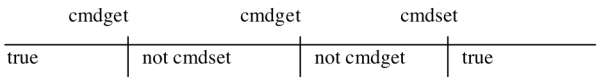

Fig. 5: An example of a scenario for guiding the generation of test data.

This scenario can be described in Lutess as follows:

```
cond(          (Mouse[get] or Keyboard[get] or Speech[get]),
               (Mouse[get] or Keyboard[get] or Speech[get]),
               (Mouse[set] or Keyboard[set] or Speech[set]));
intercond(     true,
               not(Mouse[set] or Keyboard[set] or Speech[set]),
               not(Mouse[get] or Keyboard[get] or Speech[get]),
               true);
```

Let us consider a second scenario. It describes a redundant usage of two modalities: mouse and speech. The scenario starts in a state where notes are visible (`pre memoSeen`). The user first takes one note in a redundant way, with mouse and speech at the same instant. The user then removes a second note by using again mouse and speech in a redundant way but at two different instants belonging to the same temporal window. The scenario is expressed as follows:

```
cond(          pre memoSeen and (Speech[get] and Mouse[get]) and
               not (Speech[remove] or Mouse[remove]),
               Mouse[remove] and not Speech[remove],
               Speech[remove] and not Mouse[remove]);
intercond(     true,
               not Speech[remove],
               not Mouse[remove]);
```

```
[line 1]    -    -    -    -    Se   -    -   -
[line 2]    mG   -    sG   -    Se   Car  Tak -
[line 3]    -    mR   -    -    Se   Car  -   -
[line 4]    -    -    -    sR   Se   Car  -   -
[line 5]    -    -    -    -    -    Car  -   Rem
```

Fig. 6: An excerpt from a Memo trace.

Fig. 6† shows an extract of trace which matches this second scenario. In this trace, the first line contains the event memoSeen (*Se*), implying that one or several notes are

† mG, mR, sG, SR stand for Mouse[get], Mouse[remove], Speech[get] and Speech[remove]
Se, Car, Tak, Rem stand for memoSeen, memoCarried, memoTaken, memoRemoved.

close to the user. In the second line, the two simultaneous events Mouse[Get] and Speech[Get] (*mG* and *sG*) cause one note to be taken (event *Tak line 2*). memoSeen is still set, which means that another note is visible. Lines 3 and 4 contain the events Mouse[remove] and Speech[remove] (*mR* and *sR*), which cause the visible note to be removed (event *Rem* line 5) since the two events (mR and sR) belong to the same temporal window.

5 Conclusion and future work

In this article, we have presented a method for automatically testing multimodal systems based on Lutess, a testing environment originally designed for synchronous software. Multimodality is addressed through the software properties that are checked: the CARE and temporal properties. Testing the satisfaction of the CARE and temporal properties with Lutess requires (1) expressing the properties in Lustre to build a test oracle and (2) generating adequate test input data. We have shown that the expression of the CARE and temporal properties in Lustre is possible, since the language is a temporal logic of the past and makes it possible to specify constraints on event sequences. The test data generation relies on a users' model including invariants and guiding directives (i.e. operational profiles, behavioural patterns). We have shown that by specifying operational profiles it is possible to generate test data corresponding to the combined usage of modalities, and that scenarios are also useful for the expression of functional properties.

In future work, we will explore further the guide-types for generating the test data, and in particular behavioural patterns that correspond to usability scenarios. To do so, we plan to use information from the task analysis in order to define the behavioural patterns. This work will be done in the context of our platform ICARE-Lutess that supports a semi-automatic generation of the translators between Lutess and the multimodal system developed using ICARE. Since an ICARE diagram is defined for a given task, we will first link our ICARE platform with a task analysis tool such as CTTE [20]. We will then exploit the task tree for defining behavioural patterns used for guiding the test. Extending our ICARE-Lutess platform in order to be connected to a task analysis tool will lead us to define an integrated platform from task to concrete multimodal interaction for designing, developing and testing multimodal systems.

6 Acknowledgments

Many thanks to G. Serghiou for reviewing the paper. This work is partly funded by the French National Research Agency project VERBATIM (RNRT) and by the OpenInterface European FP6 STREP focusing on an open source platform for multimodality (FP6-035182).

7 References

1. Benveniste, A., Caillaud, B., & Le Guernic, P. From synchrony to asynchrony. Proc. of CONCUR'99, Concurrency Theory, Springer Verlag (1999) 162-177.
2. Barnard, P., & May, J. Cognitive Modelling for User Requirements. Computers, Communication and Usability: Design issues, research and methods for integrated services, Elsevier: Amsterdam (1993) 101-146.
3. Blandford, A., & Young, R. Developing runnable user models: Separating the problem solving techniques from the domain knowledge. Proc. of HCI'93, People and Computers VIII, Cambridge University Press (1993) 111-122.
4. Bolt, R. Put That There: Voice and Gesture at the Graphics Interface. Proc. of SIGGRAPH'80, ACM Press (1980) 262-270.
5. Bouchet, J., Nigay, L., & Ganille, T. ICARE Software Components for Rapidly Developing Multimodal Interfaces. Proc. of ICMI'04, ACM Press (2004) 251-258.
6. 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.
7. 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.
8. d'Ausbourg, B. Using Model Checking for the Automatic Validation of User Interfaces Systems. Proc. of DSVIS'98, Springer Verlag (1998) 242-260.
9. 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.
10. Duke, D., & Harrison, M. Abstract Interaction Objects. Proc. of Eurographics'93, North Holland (1993) 25-36.
11. Dupuy-Chessa, S., du Bousquet, L, Bouchet, J., & Ledru Y. Test of the ICARE platform fusion mechanism. Proc. of DSVIS'2005, Springer Verlag (2005) 102-113.
12. Griffeth, N., Blumenthal, R., Gregoire, J.-C., & Ohta, T. Feature Interaction Detection Contest. Proc of Feature Interactions in Telecommunications Systems V, IOS Press (1998) 327-359.
13. Halbwachs, N. Synchronous programming of reactive systems, a tutorial and commented bibliography. Proc. of CAV'98, Springer Verlag (1998) 1-16.
14. Ishii, H., & Ullmer, B. Tangible Bits: Towards Seamless Interfaces between People, Bits and Atoms. Proc. of CHI'97, ACM Press (1997) 234-241.
15. Jourde, F. Nigay, L, & Parissis, I. Test formel de systèmes interactifs multimodaux : couplage ICARE – Lutess. Proc. of 19èmes Journées Internationales du génie logiciel (in french).
16. Madani, L., Parissis, I., & Nigay, L. Testing the CARE properties of multimodal applications by means of a synchronous approach. IASTED Int'l Conference on Software Engineering. Innsbruck, Austria, 2/2005.
17. Madani, L., Oriat, C., Parissis, I., Bouchet, J., & Nigay, L. Synchronous Testing of Multimodal Systems: An Operational Profile-Based Approach. Proc. of Int'l Symposium on Software Reliability Engineering (ISSRE'05), IEEE Computer Society (2005) 325-334.
18. Nigay, L., & Coutaz, J. A Generic Platform for Addressing the Multimodal Challenge. Proc. of CHI'95, ACM Press (1995) 98-105.
19. Ouabdesselam, F., & Parissis, I. Constructing Operational Profiles for Synchronous Critical Software. Proc. of Int'l Symposium on Software Reliability Engineering (ISSRE'95), IEEE Computer Society (1995) 286 - 293.

20. Mori, G., Paterno, F. & Santoro, C. CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. IEEE Transactions on Software Engineering (August 2002) 797-813.
21. Palanque, P., & Bastide, R. Verification of Interactive Software by Analysis of its Formal Specification. Proc. of INTERACT'95, Chapman et Hall (1995) 191-197.
22. Parissis, I., & Ouabdesselam, F. Specification-based Testing of Synchronous Software. Proc. of ACM SIGSOFT, ACM Press (1996) 127-134.
23. Paterno, F., & Faconti, G. On the Use of LOTOS to Describe Graphical Interaction. Proc. of HCI'92, Cambridge University Press (1992) 155-173.
24. Thimbleby, H., Cairns, P., & Jones M. Usability Analysis with Markov Models, ACM Transactions on Computer Human Interaction, vol.8, issue 2 (2001) 99-132.
25. Vernier, F., & Nigay. L. A Framework for the Combination and Characterization of Output Modalities. Proc. of DSVIS'2000, Springer Verlag (2000) 32-48.