TESTING THE CARE PROPERTIES OF MULTIMODAL APPLICATIONS BY MEANS OF A SYNCHRONOUS APPROACH

Laya Madani[1], Laurence Nigay[2], Ioannis Parissis[1]
{Laya.Madani, Laurence.Nigay, Ioannis.Parissis}@imag.fr

[1] Laboratoire LSR-IMAG - BP 53 38041 Grenoble Cedex 9 - FRANCE

[2] Laboratoire CLIPS -IMAG - BP 53 38041 Grenoble Cedex 9 - FRANCE

Abstract

Multimodal interactive applications support several interaction modalities (e.g., voice, gesture), which may be combined. As a result, designing and testing such applications is more complex than classical graphical applications. In this context, formal methods can make the development process more reliable. In this paper we focus on a testing approach dedicated to synchronous software. Although synchronous programming is mainly used in real-time applications, synchronous software are, from a certain point of view, similar to interactive applications: indeed they both have a behaviour consisting of cycles starting by reading an external input and ending by issuing an output. Under this hypothesis, we have been interested in the Lutess testing environment and in the possibility to use it to automatically test multimodal interactive applications. Using Lutess requires providing a specification of the user behaviour as well as a set of properties that the software under test must meet. As a first step, we consider the CARE properties. We provide patterns making possible to express them in the extended Lustre language upon which Lutess is built. Preliminary experimental results on testing a multimodal application are presented.

Keywords

Multimodal applications, automated testing, CARE, synchronous software.

1 Introduction

Multimodal interactive applications are designed to support interaction modalities, which may be used sequentially or concurrently, and independently or combined synergistically [1]. For example the user can issue the voice command "delete that file" while selecting the file using the mouse. This illustrates a case of a synergistic usage of two modalities. The variety of possible usages of modalities makes the design, development and test of such software more complex and time consuming than of usual graphical interactive applications.

Several formal approaches have been studied for the design and development of interactive applications, including the FSM (Formal System Modelling) analysis [4], the LIM (Lotos Interactor Model) approach [10] and the ICO (Interactive Cooperative Object) formalism based on Petri Nets [8]. In these approaches, the interactive application is formally described as an abstract model and the properties that it must meet are checked on this model. They require a rather heavy specification process that many designers cannot afford.

In this paper we apply the synchronous approach for testing multimodal applications against properties specific to multimodality. More precisely, we have been studied how the Lutess testing environment [3] [9] can be adapted to the specific needs of multimodal applications testing. Lutess handles software specifications written in Lustre [5]. Unlike other formal methods (especially [2], where the Lustre language is also used), it does not require the entire application to be specified nor intends to formally prove the property satisfaction. In fact, Lutess requires a non deterministic specification of the user behaviour as well as the description of the software properties. It automatically builds a simulator feeding with inputs the software under test. Inputs are computed dynamically, during the test operation and long sequence of inputs can be generated by Lutess.

As a first step to assess the feasibility of this new approach, we have studied the translation of the CARE (Complementarity, Assignation, Redundancy, Equivalence) properties [1] into an enhanced version of the Lustre synchronous language upon which Lutess is built. The CARE properties define a framework for characterizing different forms of multimodal usages of an interactive application. They are formally defined in [6].

The paper is structured as follows: Section 2 presents the CARE framework. Section 3 introduces useful concepts about synchronous software and the testing environment, Lutess. In section 4 we show how CARE properties of a multimodal application can be tested with Lutess and provide some preliminary experimental results.

2 Multimodality and the CARE properties

A modality is an interaction method that an agent can use to reach a goal. A modality can be defined as a pair <d, l >, where "d" means a physical device and "l" an interaction language. An interaction language is defined as a vocabulary of terminal elements and a grammar. The terminal elements are captured or produced by the input/output devices. This definition characterizes the interchanges (events) between the user and the application. For example, the speech input modality is described by the pair <microphone, pseudo-natural language> where the pseudo-natural language is defined by a specific grammar and a vocabulary of elements. In other words, the modality defines the type and form of the data exchanged between the user and the application.

Multimodality refers to the multiplicity of modalities for a given interactive application. Let us consider a multimodal application that we have developed [7]: a multimodal notebook, a personal electronic book.]. It allows a user to create, edit, browse, and delete textual notes. In particular, to insert a note between two notes, the user can say "Insert a note" while simultaneously selecting the location of insertion with the mouse. To edit the content of a note, one modality only is available: typing. Browsing through the set of notes is performed by clicking dedicated buttons such as "Next" and "Previous" or by using spoken commands such as "Next note". To empty the note book, a "Clear notebook" command may be specified using voice or clicking the mouse on the "Clear" button. The simple notebook application highlights several usages of multiple modalities. For describing such various forms of multimodality, we propose the CARE properties [1]. CARE stands for Complementarity, Assignment, Redundancy, and Equivalence that may occur between the interaction modalities available in a multimodal application. Equivalence expresses the availability of choice between multiple modalities while assignment designates the absence of choice. Redundancy and complementarity go one step further by considering the combined used of multiple modalities under temporal constraints. Redundancy is defined by the existence of redundant information specified along different modalities and complementarity is characterized by cross modality references.

Although each modality can be used independently within a multimodal application, the availability of several modalities in an application naturally leads to the issue of their combined usage. Although the combined usage of multiple modalities opens a vastly augmented world of possibilities in multimodal application design, it also implies that software performs fusion. Fusion involves the combination of data to obtain a complete command/task such as the "Insert a note" in the NoteBook application. The input elementary events to be combined can be produced simultaneously or sequentially within a temporal window. The criteria for triggering fusion are twofold: the complementarity of data, and time. According to the strategy adopted, fusion is made as soon as the input events are captured (early fusion) or is delayed until the end of a temporal window (lazy fusion). In that latter case, events are processed and a task is determined according to the confidence factors associated with the events.

3 Lutess: a testing environment for synchronous programs

3.1 Synchronous programs

A synchronous program behaves as follows: the time is divided in discreet instants defined by a global clock. At instant t the synchronous program reads its inputs i_t from its external environment. Then, it computes and issues outputs o_t. The synchrony hypothesis states that the computation of the output values is made instantaneously, at the same instant t. More realistically, this hypothesis is satisfied when software is able to take into account any external evolution. Hence, an interactive application can be considered as a synchronous program as long as all user initiated actions and all external stimuli can be caught.

3.2 The Lustre language

Lustre [5] is a language designed to write synchronous programs (called nodes). 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 a single boolean input and a boolean output. At any moment, 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, possibly, local variables. A Lustre expression is made up of constants, variables and operators. An operator may be logical, arithmetic or Lustre-specific. The third kind serves to specify temporal constraints. It consists of two main temporal operators. The first, called "pre", makes possible to use the last value an expression has taken (at the last tick of the clock). The second, called "followed by", is used to assign initial values to expressions. These two operators are defined as follows:

- 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.

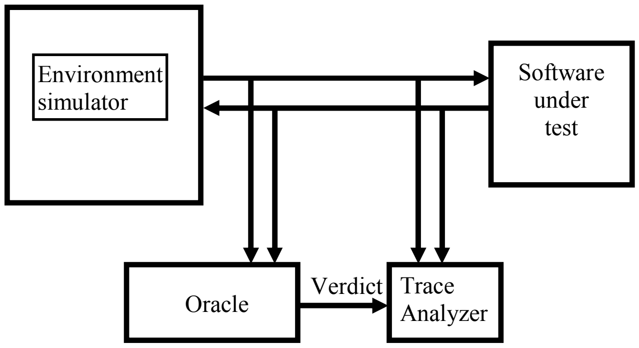
- If E and F are expressions denoting, respectively, the sequences (e0, e1, e2, ..., en, ...) and (f0, f1, f2, ..., fn, ...), E -> F denotes the sequence (e0, f1, f2, ..., fn, ...). Informally, the operator -> makes possible to assign the initial value of an expression (i.e. at time t=0).

Lustre makes possible to implement basic logical and/or temporal operators expressing invariants or safety properties. For example, the temporal operator OnceAFromBToC specifies that property A must hold at least once between the instants where events B and C occur.

3.3 Lutess

Lutess [3] is a tool for functional testing of synchronous software. The main idea is to automatically generate long input sequences and to examine whether a program satisfies some stated properties. These properties are requirements imposed on the program behaviours, such as "a user's phone goes back to its idle state every time the user goes on the hook". An important point is that input sequences are generated under assumptions on the possible behaviours of the environment interacting with the software. For example, it is physically impossible for the user of a telephone to go on the hook twice without going off the hook in between.

Lutess requires three elements: the software under test, its environment description and a test oracle (as shown in the following figure).



Lutess automatically builds a test data generator and a test harness which 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 test is operated on a single action-reaction cycle, driven by the generator. The generator randomly selects an input vector for the software and sends it to the latter. The software 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 requirements are violated. The test data generator is automatically built by Lutess from an environment description written in Lustre. The software and the oracle are both executable programs with boolean inputs and outputs and a synchronous behaviour.

The trace collector stores the input, output and oracle data (boolean values) into specific files and displays the traces in a textual mode, defined by the user.

During a test run, at each cycle (or step), the Lutess generator randomly selects an input vector for the system under test. Basically, the input is selected using the environment description and assuming that the data distribution is uniform. But the user can also define an input statistical distribution or scenarios to guide the test data generation.

4 Testing multimodal applications with Lutess

4.1 Main issues

From the previous section, it results that, in order to test a multimodal application with Lutess, one must provide:

1. The software to test as an executable program. No hypothesis is made on the program implementation, but an event translator must be added to the program, translating the program input and output events to boolean events (that Lutess can handle).

2. The Lustre specification of the behaviour of the external environment of the application (that is, user initiated actions or signals as well as inputs issued by external devices).

3. A Lustre specification of the test oracle describing the properties that the software must meet. Software properties may deal with functional requirements or interaction related requirements or recommendations.

The scope of this paper is limited to the third point and, more precisely, to the Lustre translation of the application CARE properties. In order to build a test oracle able to check at every execution step the validity of the CARE properties, the latter should be expressed as Lustre logical expressions. For every CARE property we propose a generic Lustre formula (see section 4.2,). This formula can be used as a pattern and can be adapted to any multimodal application. As an illustration, we present some CARE properties expressed on an application example in section 4.3.

4.2 Expressing CARE properties in LUSTRE

Assume M to be a modality and *n* the number of application input events associated with M. Let M[0..n-1] be a vector of boolean variables such an entry M[i] corresponds to a unique event of M. At a given instant, at most one entry of M[0..n-1] is set to true.

Let MCF[0..n-1] be a vector of integer values containing the event confidence factors. The confidence factor of the event M[k] is MCF[k].

Output tasks are represented by boolean variables, set to true whenever the tasks occur.

The temporal window involved in fusion is implemented by two boolean variables *start* and *end*, which become true respectively at the beginning and the end of every occurrence of the temporal window. *Start* is defined as follows:

```
 start = true -> pre end
```

This means that temporal windows follow each other.

4.2.1 Complementarity

Complementarity requires at least one event of two different modalities to occur before activating the associated output task. The expression of this property depends on the fusion strategies mentioned in section 2: early and lazy.

Early fusion

The complementary input events which are temporally close are merged and the associated output task is enabled as soon as the required inputs have been identified.

The occurrence of one event of every modality in the current temporal window is enough to enable the output task. It is however possible that several events of the same modality occur in this window. In that case, the task is computed according to the last event of each modality. This is translated in LUSTRE as follows:

```
IsEquivalentTo (TM1jM2k,
          (OnceSince(M1[j], start) and
           OnceSince(M2[k], start) and
            IsLast(j, M1) and
            IsLast(k, M2)));
```

The node IsEquivalentTo implements the ordinary logical equivalence while the task TM1jM2k is supposed to be associated with the events M1[j] (of the modality M1) and M2[k] (of the modality M2). IsLast(i, T), where i is an integer and T a boolean vector, returns true if T[i] is the last element that has taken the value true.

Lazy fusion

The input events which are temporally close and have the highest confidence factors are merged in the end of the temporal window and the associated output task is enabled. Hence, the output task is determined at the end of the temporal window. For each modality involved in the interaction, only the event with the highest confidence factor (among all the events of the same modality) in the temporal window is taken into account. This is translated in LUSTRE as follows:

```
IsEquivalentTo (TM1jM2k,
          (end and OnceFromTo(M1[j], start, end) and
           OnceFromTo(M2[k], start, end) and
     IsTheHighestCFFromTo(M1CF[j], M1, M1CF, start,
end)
        and
     IsTheHighestCFFromTo (M2CF[k], M2, M2F, start,
end)));
```

The node *IsTheHighestCFFromTo* takes as input a confidence factor (integer) of an event, a vector of Boolean variables and a vector of confidence factors, as well as two boolean variables (*start, end*) indicating the beginning and the end of a temporal window. It returns a true value if no event has occurred between *start* and *end* the confidence factor of which has been more elevated than the confidence factor of the first parameter.

4.2.2 Redundancy and Equivalence

If there are several input events, redundancy requires the fusion process to choose one event among those of all the available modalities. Equivalence admits a single input event to be propagated.

For example, if the mouse cursor can be manipulated by two input devices (either the mouse, or the tactile pavement), these two devices are considered as redundant and equivalent. In case of redundancy, the fusion process will choose one of the two events while in case of equivalence, the presence of several events is not required.

Early fusion

An output task is activated as soon as the associated input event is received:

```
IsEquivalentTo (TM1jM2k, (M1[j] or M2[k])) ;
```

Lazy fusion

At the end of the temporal window, the input event with the highest confidence factor is propagated and the adequate output task is activated. The considered event must have the highest confidence factor among all the events of all the modalities, which have occurred within the temporal window. A LUSTRE expression of this property is the following:

```
IsEquivalentTo (
        TM1jM2k,
        (end and (
        (OnceFromTo(M1[j], start, end)  and
          IsTheHighestCFFromTo ( M1CF [j],
                      M1|M2, M1CF|M2CF, start, end)
         ) or
         (OnceFromTo(M2[k], start, end)  and
          IsTheHighestCFFromTo ( M2CF [k],
                      M1|M2, M1CF|M2CF, start, end)))));
```

M1|M2 denotes the concatenation of the two vectors M1 and M2.

4.3 An application example: Geonote

4.3.1 Brief description

Geonote is a prototype multimodal application developed for PDAs. Geonote users can read, write, get and drop virtual digital notes ("post-it"). The digital notes are localized in space. When a user moves from a place to another, his/her PDA catches the virtual notes that other users have dropped. The user can read a virtual note and/or delete it. He/she can also create a new note and drop it.

The Geonote user can issue commands by means of the PDA pen or by voice: two input modalities are therefore available.

When Geonote detects a new "post-it", the task "manage current post-it" becomes available. The user can then read, delete or save the post-it:

- by using the pen on the PDA (for example, by selecting the "delete" button);
- by issuing a voice command (for instance, by saying "read");
- by using the two modalities in a redundant way (for instance, by saying "read" while selecting the "Read" button).

Two CARE properties are relevant for the task "manage current post-it": (1) *equivalence* since the task can be performed by using one of the two available modalities; (2) *redundancy* since the user can use the two available modalities in a redundant way.

4.3.2 LUSTRE expression of CARE properties

In order to provide a LUSTRE expression of these CARE properties, we use two boolean vectors, Pen [0..2] and Voice[0..2]:

- Pen[0] or Voice [0] are set when the user command is "Consult";
- Pen[1] or Voice [1] are set when the user command is "Delete";
- Pen[2] and Voice [2] correspond to the "Save" command.

Assume that we want to check that the user can save a post-it by a vocal command or by pen manipulation (i.e. that the equivalence property holds) or by the two modalities (i.e. that redundancy holds).

If the early fusion strategy is used, this property is simply expressed as follows:

IsEquivalentTo (SaveCPostT, (Pen[2] or Voice[2])).

For the lazy fusion, we need two more vectors (PenCF [0..2], VoiceCF [0..2]) such as PenCF[i] is the confidence factor of Pen[i] (similarly for VoiceCF and Voice).

The LUSTRE expression for the same property is:

```
IsEquivalentTo (
        SaveCPostT,
        ( end and (
        ( OnceFromTo(Pen[2], start, end)  and
         IsTheHighestCFFromTo ( PenCF[2],
             Pen|Voice, PenCF|VoiceCF, start, end)
         ) or
         ( OnceFromTo(Voice[2], start, end)  and
         IsTheHighestCFFromTo ( VoiceCF[2],
             Pen|Voice, PenCF|VoiceCF, start, end)))));
```

The Geonote application has been tested with test oracles built according to that method. In two cases, the application failed to satisfy the associated CARE properties: these failures indeed correspond to software faults.

5 Conclusion and future work

This paper presents a first study towards the definition of a method for automatically testing multimodal applications by means of the synchronous approach. We have focused on the CARE properties that describe different forms of multimodality. Lustre patterns have been defined, making possible to build test oracles to automatically detect a non respect of the properties.

As future work, we plan to study the test data generation. Up to now, the latter has been randomly made. The strategies provided by Lutess should be used to improve the testing effectiveness.

6 References

[1] Coutaz, J., Nigay, L., Salber, D., Blandford, A., May, J. and Young, R. (1995). Four Easy Pieces for Assessing the Usability of Multimodal Interaction: The CARE properties. INTERACT'95, Norway.

[2] B. d'Ausbourg. Using Model Checking for the Automatic Validation of User Interfaces Systems. DSVIS'98, pages 242 -- 260, Abingdon, UK, June 1998.

[3] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. *Lutess: a specification driven testing environment for synchronous software*. In 21st International Conference on Software Engineering, pages 267-276. ACM Press, May 1999.

[4] Duke, D. and Harrison, M. (1993). Abstract Interaction Objects. Computer Graphics Forum. Eurographics'93.

[5] N. Halbwachs. *Synchronous programming of reactive systems, a tutorial and commented bibliography*. In Tenth International Conference on Computer-Aided Verification, CAV'98, LNCS 1427, Vancouver (B.C.), June 1998. Springer Verlag.

[6] Nigay, L., Jambon, F. Coutaz, J. Formal Specification of Multimodality. CHI'95, Workshop on Formal Specification of User Interfaces.

[7] Nigay, L. & Coutaz, J. (1993). A design space for multimodal interfaces: concurrent processing and data fusion. Proc. InterCHI'93 ACM: New York, 172-178

[8] Palanque P., Bastide R. (1995). Verification of Interactive Software by Analysis of its Formal Specification. INTERACT'95, Norway.

[9] I. Parissis, F. Ouabdesselam. Specification-based Testing of Synchronous Software. ACM SIGSOFT Fourth Symposium on the Foundations of Software Engineering, San Francisco, 1996.

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