Automated Specification-based Testing of Interactive Components with AsmL

Ana C. R. Paiva, João C. P. Faria, and Raul F. A. M. Vidal

Abstract — It is presented a promising approach to test interactive components, supporting the automatic generation of test cases from a specification. The relevance and difficulties (issues and challenges) associated with the testing of interactive components are first presented. It is shown that a formal specification with certain characteristics allows the automatic generation of test cases while solving some of the issues presented. The approach is illustrated with an example of automatic testing of the conformity between the implementation of a button, in the .Net framework, and a specification, written in the AsmL language, using the AsmL Tester tool. The conclusion discusses the characteristics of the tool and gives directions for future work.

Index Terms — Formal Methods, Interactive Systems Testing.

1 Introduction

The development of high-quality interactive systems and applications is a difficult and time-consuming task, requiring expertise from diverse areas (software engineering, psychology). Current IDE's are not powerful enough for specifying/modeling, building and testing those systems in an effective way. The development of interactive systems and applications based on reusable interactive components is the key to achieve higher quality and productivity levels. Improving the quality of interactive components should have a major impact in the quality of interactive systems and applications built from them, and should contribute to their increased reuse.

In this paper, it is presented a promising approach to the testing of interactive components. By interactive components we mean reusable controls or widgets or interactors, capable of both input from the user and output to the user, written in a general-purpose object-oriented language, such as Java or C#. Interactive components range from the more basic ones (such as buttons, text boxes, combo boxes, list boxes, etc.) to the more sophisticated ones (calendars, data grids, interactive charts, etc.) built from simpler ones. The overhead incurred in testing reusable interactive components pays-off, because of their wider usage and longevity, when compared to special purpose and short lived "final" user interfaces.

The paper is organized as follows: next section (section 2) presents some important issues and challenges of testing interactive components. Section 3 explains the type of test automation that is envisioned (automated specification-based testing), discusses the type of formal specification required, and discusses its costs and benefits. Section 4 presents an example of performing automated specification-based tests using the AsmL language and the AsmL Tester tool. Some conclusions and future work can be found in the final section.

- *All authors are with the Electrical Engineering Department, Engineering Faculty of Porto University. E-mail: apaiva@fe.up.pt; jpf@fe.up.pt; and rmvidal@fe.up.pt.*

2 Issues and Challenges of Testing Interactive Components

Testing interactive components is particularly difficult because it shares and combines the issues and challenges of testing object-oriented systems [1], component-based systems [2], and interactive systems. Some of the main issues and challenges are identified and described next.

2.1 Complex Event-driven Behaviour

Interactive components (and interactive applications and systems in general) have complex event-driven behaviour, difficult to analyze and predict, and, consequently, also difficult to test and debug. Even basic interactive components, such as buttons and text boxes, may react to and generate dozens of events. Most of us have already experienced "strange" behaviours (blocked interfaces, dirty displays, etc.) apparently at random when using wide-spread interactive applications and systems. This should not be a surprise given their complex event-driven behaviour.

2.2 Highly-configurable (or Customizable) Behaviour

Reusable interactive components usually have a highly-configurable (or customizable) behaviour. This can be done statically or dynamically by setting configuration properties or attributes, by adding event-handlers or by defining subclasses and method overriding. Testing an interactive component in all the configurations allowed is almost impossible because of the huge set of possible configurations and the difficulty to predict the customized behaviour.

2.3 Multiple Interfaces

Interactive components have both a user interface (GUI) and an application interface (API). The application interface is used for customizing and composing them, and for linking them with the underlying application logic. Different kinds of inputs and outputs occur via these different interfaces. Adequate testing of an interactive component cannot look at just one of these interfaces in isolation, and has to

take into account all these kinds of inputs and outputs in the definition of test cases and in test execution.

2.4 GUI Testing is Difficult to Automate

Automating the testing of graphical user interfaces poses well-known challenges:

1. How to properly simulate inputs from the user (mouse, keyboard and other higher-level events that are generated by the user)?
2. How to check the outputs to the user without excessive sensitivity to formatting and rendering details?

2.5 API's with Callbacks and Reentrance

The designer of a reusable interactive component defines its methods but does not know in advance which kind of applications will make use of them. Method calls between an interactive component and an application occur in both directions:

1. The application (or test driver) may call methods of the interactive component. From the testing perspective, inputs are methods invoked with parameters while outputs are the values returned by those methods. This is the traditional situation in unit testing.
2. The interactive component may generate events (originated from the user or internally generated) that cause the invocation of methods in the application (or test stub), by some kind of callback mechanism (event handlers, or subclassing and method overriding). Again, from the testing perspective, the outputs are the events and parameters passed to the application, while inputs are returned parameters.

Testing the second kind of interaction (callbacks) poses specific issues and challenges, as already noted in [3]:

1. An application method invoked in a callback may, in turn, invoke methods of the interactive component (reentrancy situation) and have access or change its intermediate state. Hence, the internal state of the interactive component when it issues a callback is not irrelevant. Moreover, some restrictions may have to be posed on the state changes that an application may request when processing a callback.
2. During testing, one has to check that: (1) the appropriate callbacks are being issued; (2) when a callback is issued, the interactive component is put in the appropriate internal state; (3) during the processing of a callback, the application doesn't try to change the state of the interactive component in ways that are not allowed.

2.6 Operating System Interference

Interaction with the user is mediated by the operating system in non trivial ways (often, several layers are involved), introducing more dimensions of configurability, and complicating the analysis and prediction of its behaviour, as well as the testing and debugging tasks.

2.7 Insufficient Documentation

The documentation supplied with interactive components is usually scarce and not rigorous enough for more advanced uses, such as advanced customization and thorough testing. For example, from the documentation, it is difficult to know precisely:

1. when are events signalled and by what order;
2. what is the internal state of a component when it signals an event;
3. what is safe for an event handler to do;
4. what interactions exist between events.

This usually leads to a trial-and-error style of application programming and poor application quality, and also complicates the design of test cases.

2.8 Poor Testability

Testing of interactive components is usually difficult and time-consuming due to:

1. the lack of rigorous, unambiguous and comprehensive documentation;
2. the reduced observability (capability to observe the internal state, display produced, and events raised);
3. the deficient controllability (capability to simulate user input).

Some of the issues and challenges described in this section will be addressed by our testing approach and discussed in the next sections.

3 Automated Specification-based Testing

Manual testing of GUIs and interactive components is labour-intensive, frequently monotonous, time-consuming and costly. Some of the reasons are the existence of varied possibilities for user interaction and a large number of possible configurations for each component, and other issues described in section 2, making it impractical the satisfaction of adequate coverage criteria by manual testing. It is necessary to use some type of automation to perform those tests.

3.1 Degree of Automation Envisioned

The degree of automation we envision is the automatic generation of test cases (inputs and expected outputs) from specification, and not just the type of automation that is provided by unit testing frameworks and tools, such as JUnit (www.junit.org) or NUnit (www.nunit.org), or the type of automation provided by capture and replay tools, such as WinRunner (www.mercure.com) and other tools (www.stlabs.com/marick/faqs/t-gui.htm).

Unit testing frameworks and tools are of great help in organizing and executing test cases, particularly for API testing, but not in generating test cases from a specification.

Capture and replay tools are probably the most popular tools for GUI testing, but don't support the automatic generation of test cases. With these tools, it is possible to record the user interactions with a graphical user interface (mouse input, keyboard input, etc.) and replay them later. Capture and replay tools are of great help in several scenarios, but also have widely recognized limitations (see e.g. the lesson "capture replay fails" in [4]). In this type of test automation, there is no guarantee of test coverage, and there is an excessive dependency on the "physical" details of the user interface.

From a higher perspective, these different approaches and types of automation are complementary and not opponents.

The automatic generation of test cases from specification requires some sort of formal specification of the software to be tested, that can be used to generate concrete input values and sequences, as well as the expected outputs (as a test oracle).

It is possible to design test cases (for black-box testing) from informal specifications, but not in an automated way. At most, inputs can be generated automatically based on the signatures of methods and events (their calling syntax), but expected outputs can only be generated based on a formal specification of their semantics.

3.2 Type of Specification Needed

A popular type of specification of object-oriented systems is based on the principles of design by contract [5], by means of invariants and pre and post-conditions, as found in Eiffel (www.eiffel.com), ContractJava [6] or JContract (www.parasoft.com). An invariant is a condition that restricts the valid states of an object, at least on the boundaries of method calls. A pre-condition of a method is a condition on the input parameters and the internal state of the object that should hold when a method is called. On the opposite side, a post-condition of a method is a condition on the input parameters, initial state of the object (when the method is called), final state of the object (when the method returns), and value returned that should hold at the end of the method execution.

Although with limitations, some test tools, such as JTest (www.parasoft.com), have the capability of generating unit tests based on the specification of pre and post-conditions. While pre and post-conditions are a good mean to restrict the allowed behaviours of an object, they are not adequate, in general, to fully specify their intended behaviour, particularly when callbacks are involved. As already noted by Szyperski in his book [3], the semantics of components that issue callbacks, as is the case of interactive components (see section 2), cannot be captured only by means of pre and post-conditions (at least with the meaning presented above).

The Object Constraint Language (OCL) (see www.uml.org) goes a step further, by allowing the specification, in the post-condition of a method, of messages that must have been sent (method calls and sending of signals) during its execution. However, in general, it is not possible to specify the order by which messages are sent and the state of the object when each message is sent (important because of re-entrance, as explained in section 2). The definition of post-conditions in OCL has another advantage over its definition in Java or Eiffel, because OCL is a higher-level formal language supporting formal reasoning and automation.

AsmL [7] (http://research.microsoft.com/fse/asml), a formal specification language developed at Microsoft Research, tightly integrated with the .Net framework, bridges over the limitations found in OCL by means of "model programs". A "model program" is an executable specification of a method. A model program may be organized in a sequence of steps. For example, if a method issues a callback in the middle of its execution, three steps should be defined: a first step to lead the object to the appropriate state before issuing the callback; a second step where the callback is issued; a third step to lead the object to the appropriate final state and return. These steps facilitate the definition of restrictions on sequences of actions/events that are common to find in user interface modelling and are not easy to express using just post-conditions. Each step comprises one or more non-contradictory "model statements" that are executed simultaneously. Model statements are written in a high-level action language with primitives to create new objects, assign new values to the attributes of an object, and call other methods. Model programs may be used in combination with pre and post-conditions, usually dispensing the later. Examples of specifications written in AsmL will be presented in section 4.

3.3 Conformity Checks

With appropriate tool support (as is the case of the AsmL Tester tool), model programs can be used as executable specification oracles [1]. That is, the results and state changes produced by the execution of model programs (executable specifications written in AsmL) can be compared with the results produced by the execution of the corresponding implementation under test (written in any .Net compliant language in this case). Any discrepancies found are reported by the tool. Mappings between actions and states in the specification and the implementation have to be defined, either explicitly or implicitly (based on name equality). Although this is not the only way of performing conformity checks between a specification and an implementation (see [8] for a discussion of other possible ways), it is a feasible way.

3.4 Finite State Machine Model and Test Case Generation

For the generation of test cases, the AsmL Tester tool first generates a FSM (Final State Machine) from the AsmL specification, and then generates a test suite (with one or more test cases) from the FSM, according to criteria provided by the user. Since the number of possible object states (possible combinations of values of instance variables) is usually huge, the states in the FSM are an abstraction of the possible object states, according to some criteria provided by the user.

It is well known that state machine models are appropriate for describing the behaviour of interactive systems (and reactive systems in general), and a good basis for the generation of test cases, but usually there is not a good integration between the object model and the state machine model. AsmL and the AsmL Tester tool solve this problem with the generation of the FSM from the specification (formal object model).

3.5 Advantages of the Formal Specification of Interactive Components

When compared to other testing techniques, automated specification-based testing has the disadvantage of requiring a formal specification (to achieve a higher degree of automation). But the investment in the formal specification of reusable interactive components may be largely compensated by the multiple benefits it can bring:

1. Formal specifications and models are an excellent complement to informal specifications and documentation, because ambiguities are removed and inconsistencies are avoided.
2. Formal specifications allow the automation of specification-based testing, as described in this paper.
3. Besides being useful as the basis for the generation of test cases, FSM's can also be used to automatically prove required properties of a system, with model-checking tools that exhaustively search the state space. The properties are written in temporal logic. For example, Campos, in [9], uses model checking tools to prove usability properties of user interfaces.
4. Desired properties of a system (with a finite or infinite state space) may be proved in a semi-automated way, given a formal specification or model of the system, and a formal description of those properties [10].
5. Executable specifications (or models) of user interfaces and interactive systems may be used as fully functional prototypes. Problems in specification and design can be discovered and corrected before implementation begins.
6. In restricted domains, and with appropriate tool support (see for example [11]), formal specifications or models of user interfaces can be used as the basis for the automatic generation of an implementation in some target platform, according to refinement or translation rules. The generated implementations are correct by construction, and conformity tests are not needed.

Overall, higher rigor in the description and verification of interactive components is important to gain confidence on their correctness and encourage their reuse [10].

4 EXAMPLE

In this example, the AsmL Tester tool is used to test the conformity between the implementation of the button control in the .Net framework (`System.Windows.Forms.Button` class) and a specification of a small part of its behaviour (related to mouse and keyboard events only) in the AsmL language. The example is small but was selected mainly to illustrate the testing process and turnarounds to some difficulties, and not the power of the AsmL language. The approach presented can easily scale to be used in larger interactive controls.

4.1 Formal Specification of a Button in AsmL

The button specification has instance variables (Specification 1), events (Specification 2), and methods (Specification 3).

Two types of events should be distinguished: events received by the button (`MouseUp`, `MouseDown`, `KeyUp` and `KeyDown`), and events generated by the button in response to the previous ones (`Click`). Both these types of events may be sent by the button to the application via event handlers. There are more button events in the .Net platform but only these events are considered in this example.

Instance variables `Text`, `Enabled` and `Focused` (Specification 1) model the internal state of the button, and correspond to properties existing in the `Button` class in the .Net framework.

Instance variables `MouseDownFlag` and `KeyDownFlag` represent event flags that were added to check that appropriate sequences of mouse and keyboard events are received by a button (`MouseUp` after `MouseDown`, `KeyUp` after `KeyDown`).

Instance variable `ClickedFlag` was added to tell whether the `Click` event was generated by the button in response to the last keyboard or mouse event received. This instance variable is reset each time an event is received (method `ResetGeneratedEventFlags`).

```
class Button extends Object
  var Text as String
  var Enabled as Boolean
  var Focused as Boolean = false
  var MouseDownFlag as Boolean = false
  var KeyDownFlag as Boolean = false
  var ClickedFlag as Boolean = false
```

Specification 1 - Button instance variables.

Events are defined in a way similar to the .Net framework (Specification 2). For each event named *EventName*, there is an instance variable named *EventNameHandlers* that stores the event handlers registered with the event. Operations `add` and `remove` are responsible to register and unregister event handlers. In an AsmL specification, it is possible to use types defined in the .Net framework. That's the case of the type `EventHandler` in Specification 2, and types `EventArgs`, `KeyEventArgs` and `MouseEventArgs` in Specification 3.

```
private var ClickHandlers as Set of EventHandler={}
  event Click as EventHandler
    add     add value to ClickHandlers
    remove  remove value from ClickHandlers
// similar statements omitted for events
// KeyDown, KeyUp, MouseDown, and MouseUp
```

Specification 2 – Button events.

For each event named *EventName*, there is a method named *OnEventName* that executes when the event occurs. The method is responsible for calling the event handlers that were registered with the event (Specification 3). Methods are public by default. In the case of events received by the button (`MouseUp`, `MouseDown`, `KeyUp` and `KeyDown`), the reception of the event may be simulated (for testing purposes) by calling the method *OnEventName* from the outside of the class. This explains why those methods are marked as public. In the case of events generated internally by the button (`Click`), the method is marked as protected, because it should only be called internally.

Some of the methods have a pre-condition indicated with the keyword `require`. The pre-condition restricts the states where the method can execute. For instance, the method `OnClick` can only execute when the property `Enabled` is true. After the pre-condition, comes the model program (described in section 3). In some cases, the model program is organized in steps.

```
protected OnClick(e as EventArgs)
  require Enabled = true
  forall handler in ClickHandlers handler(me,e)
  ClickedFlag := true

OnKeyDown(e as KeyEventArgs)
  require Enabled = true and Focused = true and
    KeyDownFlag = false
  step ResetGeneratedEventFlags()
  step forall handler in KeyDownHandlers
    handler(me,e)
  KeyDownFlag := true

OnKeyUp(e as KeyEventArgs)
  require Enabled = true and Focused = true and
    KeyDownFlag = true
  step ResetGeneratedEventFlags()
  step forall handler in KeyUpHandlers handler(me,e)
  step if (e.KeyData =
    System.Windows.Forms.Keys.Space)
      then OnClick(new EventArgs())
  KeyDownFlag := false

OnMouseDown(e as MouseEventArgs)
  require Enabled = true and MouseDownFlag = false
  step ResetGeneratedEventFlags()
  step forall handler in MouseDownHandlers
    handler(me,e)
  Focused := true
  MouseDownFlag := true

OnMouseUp(e as MouseEventArgs)
  require Enabled = true and Focused = true and
    MouseDownFlag = true
  step ResetGeneratedEventFlags()
  step forall handler in MouseUpHandlers
    handler(me,e)
  step OnClick(new EventArgs())
  MouseDownFlag := false

shared Button(s as String, b as Boolean)
  Text := s
  Enabled := b
  add me to BInstances

ResetGeneratedEventFlags()
  ClickedFlag := false
```

Specification 3 – Button methods.

4.2 Extension of the Button Implementation Class for Testability

In order to overcome the limited testability of the `Button` class in the .Net framework, a `TButton` (*Testable Button*) class, inheriting from `Button`, was created in C# (Implementation 1).

Event flag `ClickedFlag` is similar to the one used in the specification. This flag was marked as public, to facilitate the mapping of states between the specification and the implementation.

Method `OnClick` overrides a protected method with the same name defined in the `Button` class, in order to manipulate the added event flag and record the occurrence of the `Click` event. It calls the base method, so that the event handlers are called (i.e., so that the event is sent to the application).

For each event received (from the user) with name *EventName*, it was added a public method name *UOnEventName* (U-User) that can be called from outside the class to simulate user events.

```
public class TButton: Button  {
  public Boolean ClickedFlag = false;
  public TButton(String s,Boolean b):base() {
    Text = s;
    Enabled = b; }
  protected override void OnClick(EventArgs e) {
    base.OnClick(e);
    ClickedFlag = true; }
  public void UOnKeyDown(KeyEventArgs e) {
    ClickedFlag = false;
    OnKeyDown(e); }
  // similar methods omitted for events
  // KeyUp, MouseDown, and MouseUp
}
```

Implementation 1 – Implementation of a testable button class (TButton).

4.3 Specification and Implementation of a Test Container

In order for a user to interact with a button, it must be made visible by putting it inside some window or container. With this purpose, a class `TBForm` was created both at the specification level (Specification 4) and at the implementation level (Implementation 2). Due to limitations of the test tool, auxiliary methods `TBMouseDown`, `TBMouseUp`, `TBKeyDown` and `TBKeyUp` had to be created to simulate user events that are sent to the button contained in the form. These methods are selected to trigger the transitions in the state transition diagram (Figure 1). Each test case will be constructed as a sequence of calls to these methods.

```
class TBForm
  var TB as Button = new Button("TButton",true)
  TBMouseDown(e as System.Windows.Forms.MouseEventArgs)
    TB.OnMouseDown(e)
  TBMouseUp(e as System.Windows.Forms.MouseEventArgs)
    TB.OnMouseUp(e)
  TBKeyDown(e as System.Windows.Forms.KeyEventArgs)
    TB.OnKeyDown(e)
  TBKeyUp(e as System.Windows.Forms.KeyEventArgs)
    TB.OnKeyUp(e)
  shared TBForm()
    // insert this instance in the set of instances
    // to be tested
    add me to instances
```

Specification 4 – Container class TBForm.

```
using System;
using System.Drawing;
using System.Windows.Forms;
namespace TBForm
{
  public class TBForm : Form {
  private TButton TB;
  public TBForm()      {
    InitializeComponent();
    this.TB = new TButton("TButton",true);
    this.TB.Location =
      new System.Drawing.Point(32, 38);
    this.TB.Name = "TB";
    this.Controls.Add(this.TB);
    this.Show(); // mandatory
  }
```

```
  #region Windows Form Designer generated code
  private void InitializeComponent() {
    this.AutoScaleBaseSize =
      new System.Drawing.Size(5, 13);
    this.ClientSize =
      new System.Drawing.Size(146, 106);
    this.Name = "TBForm";
    this.Text = "TBForm";
    this.ResumeLayout(false);
  }
  #endregion
  [STAThread] static void Main() {
    Application.Run(new TBForm());
  }
  public void TBKeyDown(
    System.Windows.Forms.KeyEventArgs e) {
      TB.UOnKeyDown(e);
    }
  private void TBKeyUp(
    System.Windows.Forms.KeyEventArgs e) {
    TB.UOnKeyUp(e);
  }
  private void TBMouseDown(
    System.Windows.Forms.MouseEventArgs e) {
    TB.UOnMouseDown(e);
  }
  private void TBMouseUp(
    System.Windows.Forms.MouseEventArgs e) {
    TB.UOnMouseUp(e);
  }
 }
}
```

Implementation 2 – Container class TBForm.

4.4 Generation of the Finite State Machine Model

The AsmL Tester tool was used to automatically generate the finite state machine (FSM) from the AsmL specification, based on the following configuration information:

1. List of state variables - all the event flags defined in the specification of the `Button` class (`MouseDownFlag`, `KeyDownFlag` and `ClickedFlag`);
2. List of actions that trigger transitions – the constructor and methods defined in the `TBForm` class (`TBMouseDown`, `TBMouseUp`, `TBKeyDown` and `TBKeyUp`).
3. Domains (values to consider) for the previous state variables and actions' arguments. For each state variable the domain is `{true, false}`. In the case of the mouse actions, a single argument was provided, specifying the left mouse button and a position inside the form button, more precisely, the value `{new System.Windows.Forms.MouseEventArgs (System.Windows.Forms.MouseButons.Left, 1,1,0,1)}`. In the case of the keyboard actions, a single argument was provided, specifying the 'A' key, that is, the value `{new System.Windows.Forms.KeyEventArgs(System.Windows.Forms.Keys.A)}`.

The FSM obtained is shown in Figure 1. Each state corresponds to a combination of values of the state variables. Apart from the start state (greyed), the FSM has 6 states. This means that two of the possible combinations of values of the three state variables cannot occur (`*, true, true`). The possible transitions departing from each state are constrained by the methods' pre-conditions (`require` clause).

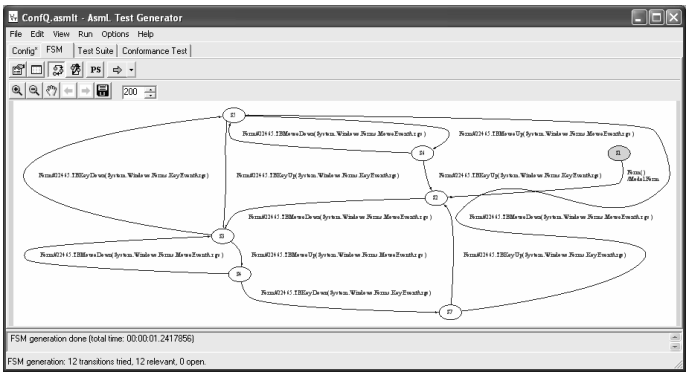

Figure 1 – Finite State Machine (FSM) diagram.

4.5 Definition of Mappings between the Specification and the Implementation

To perform conformity tests it is necessary to define mappings (conformance relations) between specification and implementation methods and data (state) [8]. These relations can be established manually or in an automated way. It was defined a relation between the classes `Model.TBForm` (specification) and `WindowsApplication1.TBForm` (implementation). After this, the methods with the same names and arguments in both classes are automatically related. In the current version of the test tool, data relations have to be defined manually. In this example, only the `ClickedFlag` instance variable was mapped, as shown in Figure 2. Conformance tests will execute related methods in both levels (specification and implementation) and will compare results obtained from both and also compare the related data.

Figure 2 – Conformance relations configuration.

4.6 Generation of the Test Suite

After generating the FSM, it is possible to generate a test suit automatically (Figure 3), based on coverage criteria selected by the user. In this case, it was used the default criteria that ensures full coverage of states and transitions. A test suite with a single test case (sequence) was sufficient to cover all the transitions. This test suit will be used as the input to conformance testing.

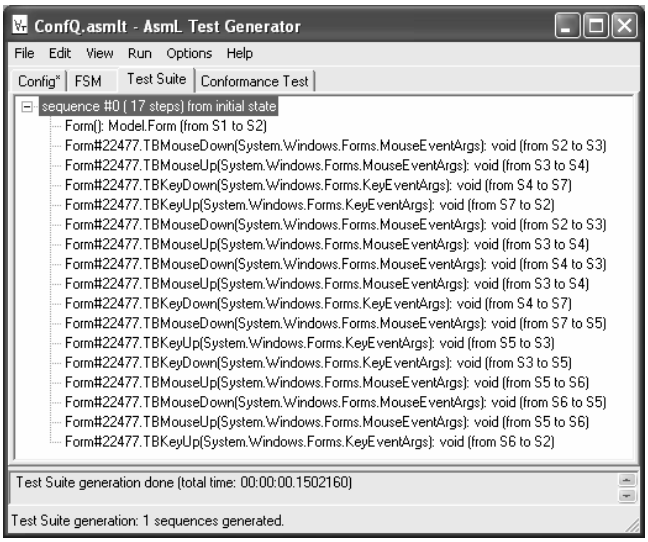

Figure 3 – Test suit generated.

4.7 Test Execution and Results

As soon as the conformity relations are defined and the FSM and the test suit are generated, it is possible to execute conformance tests. Every time there is an inconsistence, the tool stops and reports the error.

The tool reports a conformance error when the sequence of events `MouseDown`, `KeyDown`, and `KeyUp` is executed (Figure 4), with key 'A'. The error is an inconsistency between the value of the `ClickedFlag` value at the implementation (the value is *true*) and the specification (the value is *false*). This means that the implementation (the `Button` class in the .Net framework) generates a `Click` event, when it receives from the user the sequence of events `MouseDown`, `KeyDown`, and `KeyUp`. According to the documentation of the .Net framework, this should only happen when the key pressed is the spacebar (which is not the case here).

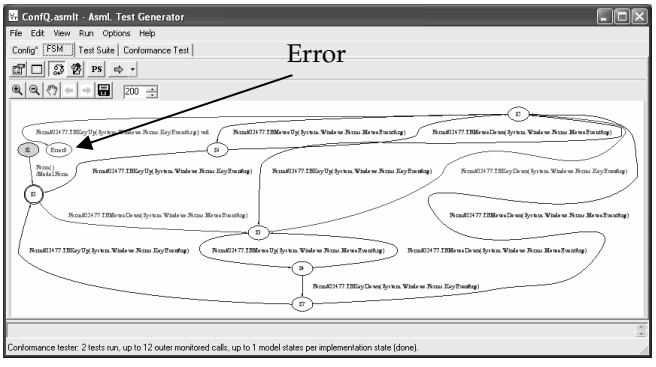

Figure 4 – Conformance test inconsistency (the path to the error is shown in red).

To reproduce this abnormal behaviour manually it is necessary to press the left mouse button on a .Net button, and press and release a keyboard key without releasing the mouse button. This will have the effect of selecting the button and executing the action associated with it. According to the documentation, this should only happen with the spacebar key.

5 CONCLUSION

An approach to test interactive components, with the automatic generation of test cases from a specification was described. In comparison with others, the approach presented in this paper requires a formal specification with demonstrated benefits in the development and verification of interactive components. In the past, formal specification and verification techniques have been used mainly in the development of critical systems, but, from our point of view, they also have a major role to play in the development and verification of reusable components, as is the case of interactive components.

It was presented an example of automatic testing the conformity between the implementation of a button, in the .Net framework, and a specification, written in the AsmL language, using the AsmL Tester tool. Some test code was needed to overcome testability limitations of the target code. Although, only a small part of the behaviour of a button was specified and tested, the tests were successful, that is, a bug was detected. A larger example could be used since the approach can easily scale but it would be difficult to explain that example in few pages.

However, in its current state, the AsmL Tester tool also has some limitations:

1. It still requires too much user intervention.
2. While the tight integration with the .Net framework has some advantages, one of its shortcomings arises from the fact that the level of abstraction of the specification is not as high as should be.
3. Interactive components can have lots of states and actions or events that can be hard to manipulate and test. The AsmL Tester tool allows the selection of which actions should appear in the FSM diagram (and in the test cases generated from the FSM). Consequently, it is possible to test separately parts of the behaviour of the object or component under test. But a rigorous method is needed to define those parts and "sum" the results obtained in each part to take coverage criteria conclusions.

The approach presented in this paper has to be extended and matured in several directions:

1. Use the approach presented in larger examples.
2. Explore other ways to generate test cases from the FSM model – some criteria to generate specification-based tests can be found at [12].
3. Define additional check points – for instance, when a callback is issued and on return.
4. Model Checking – integrate the approach with model checking techniques to prove properties about the model.
5. Verification of the user interface contract – particularly challenging is the problem of checking that the outputs sent by an interactive component to the user obey to some kind of specification or contract. For example, the user interface contract of a textbox is to allow the user to insert and visualize a string through a small

window.

Above points and possible others will be subject of future work.

ACKNOWLEDGMENT

The authors wish to thank the anonymous reviewers for their comments and suggestions.

REFERENCES

[1] R. V. Binder, *Testing Object-Oriented Systems: Models, Patterns and Tools*: Addison-Wesley, 2000.

[2] J. Z. Gao, H.-S. J. Tsao, and Y. Wu, *Testing and Quality Assurance for Component-Based Software*: Artech House Publishers, 2003.

[3] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*: Addison-Wesley, 1999.

[4] C. Kaner, J. Bach, and B. Pettichord, *Lessons Learned in Software Testing: A Context-Driven Approach*: John Wiley & Sons, 2002.

[5] B. Meyer, "Applying Design by Contract," *IEEE Computer*, pp. 40-51, 1992.

[6] F. Findler, "Contract Soundness for Object-Oriented Languages," presented at Object-Oriented Programming Systems, Languages and Applications (OOPSLA), 2001.

[7] Microsoft, "Introducing AsmL: A Tutorial for the Abstract State Machine Language," Foundations of Software Research, 2002.

[8] A. C. Paiva, J. P. Faria, and R. M. Vidal, "Specification-based Testing of User Interfaces," presented at 10th DSV-IS Workshop - Design, Specification and Verification of Interactive Systems, Funchal - Madeira, 2003.

[9] J. Campos and M. D. Harrison, "Model Checking Interactor Specifications," in *Automated Software Engineering*, vol. 8, 2001.

[10] I. MacColl and D. Carrington, "User Interface Correctness," presented at Human Computer Interaction - HCI'97, 1997.

[11] M. D. Lozano, "Entorno Metodológico Orientado a Objectos para la Especificación y Desarrollo de Interfaces de Usuario," in *Sistemas Informáticos y Computación*. Valencia: Universidad Politécnica de Valencia, 2001.

[12] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, "Generating test data from state-based specifications," *Software Testing, Verification and Reliability*, vol. 13, pp. 25-53, 2003.

Ana C. R. Paiva received M.Sc degree in Electrical and Computers Engineering from Engineering Faculty of Porto University (FEUP) and a degree in Information Systems Engineering from Minho University of Portugal in 1997 and 1995 respectively. She is currently developing hers doctorate in formal methods applied to user interfaces at FEUP, Electrical and Computers Engineering Department, where she is an Assistant Lecture since 1999.

João C. P. Faria received a Ph.D. in Electrical and Computer Engineering from the Engineering Faculty of Porto University (FEUP) in 1999, and a degree in Electrical Engineering from FEUP in 1985. He is an Assistant Professor at FEUP, Electrical and Computers Engineering Department, Informatics.

Raul F. A. M. Vidal received a Ph.D. in Digital Electronics at UMIST in 1978, an M.Sc in Communication Engineering at UMIST in 1974 and a degree in Electrical Engineering at Engineering Faculty of Porto University (FEUP) in 1972. He is an Associate Professor at FEUP, Electrical and Computers Engineering Department, Informatics.