# Towards the Integration of Visual and Formal Models for GUI Testing *

Ana C. R. Paiva[1], João C. P. Faria[1,2], Raul F. A. M. Vidal[1]

[1] Engineering Faculty of Porto University, [2] INESC Porto,
Rua Dr. Roberto Frias, s/n, 4200-465 Porto, Portugal
{apaiva, jpf, rmvidal}@fe.up.pt

**Abstract.** This paper presents an approach to diminish the effort required in GUI modelling and test coverage analysis within a model-based GUI testing process. A familiar visual notation – a subset of UML with minor extensions – is used to model the structure, behaviour and usage of GUIs at a high level of abstraction and to describe test adequacy criteria. The GUI visual model is translated automatically to a model-based formal specification language (e.g., Spec#), hiding formal details from the testers. Then, additional behaviour may be added to the formal model to be used as a test oracle. The adequacy of the test cases generated automatically from the formal model is accessed based on the structural coverage of the UML behavioural diagrams.

**Keywords:** GUI modelling, GUI testing, model-based testing, UML, Spec#.

## 1 Introduction

Software systems usually feature Graphical User Interfaces (GUIs). They are mediators between systems and users and their quality is a crucial point in the users' decision of using them. GUI testing is a critical activity aimed at finding defects in the GUI or in the overall application, and increasing the confidence in its correctness. Currently it is extremely time-consuming and costly, and very few tools exist to aid in the generation of test cases and in evaluating if the GUI is adequately tested. In fact most of the tools that have been developed to automate GUI testing do not address these two aspects. Capture/replay tools, like WinRunner (*www.mercury.com*), are the most commonly used for testing GUI applications. They facilitate the construction of test cases through the recording of user interactions into test scripts that can be replayed later, but they still require too much manual effort and postpone testing to the end of the development process, when the GUI is already constructed and functional. They are useful mainly for regression testing. Unit testing frameworks of the XUnit family (e.g., *www.junit.org*) used together with GUI test libraries (e.g., *jemmy.netbeans.org*) automate test execution but not test generation. Random input testing tools generate and execute test cases randomly [10].

* Work partially supported by FCT (Portugal) and FEDER (European Union) under contract POSC/EIA/56646/2004.

Recently, model-based approaches for software testing have deserved an increasing attention due to their potential to automate test generation and the increasing adoption of model driven software engineering practices. Some examples of model-based tools developed specifically for GUI testing have been reported [2,8,11]. But the usage of unfamiliar modelling notations, the lack of integrated tool environments, the effort required to construct test-ready models, the test case explosion problem, and the gap between the model and the implementation may barrier the industrial adoption of model-based GUI testing approaches.

In previous work, the authors have developed several extensions to the model-based testing environment Spec Explorer [3] to foster its application for GUI testing and address some of the above issues: techniques and helper libraries for modelling GUIs in Spec# [1]; a GUI Mapping Tool to automate the mapping between the GUI model and the implementation [12]; and a tool to avoid test case explosion taking into account the hierarchical structure of GUIs [13]. Further details can be found in [14]. The results achieved have received significant interest from modellers and testers.

However, the reluctance of modellers and testers to write textual formal specifications that resemble programs is an obstacle to the dissemination of approaches of the above type. Another problem is the lack of support of coverage criteria best adapted for GUI testing, such as coverage of navigation maps.

To address these problems we propose in this paper an approach for model-based GUI testing that aims at combining the strengths of visual modelling (usability) and formal modelling notations (rigor). The basic idea is to provide a familiar visual modelling *front-end*, based on UML [5], on top of a formal specification language (such as Spec#), with the following goals: diminish the need to write textual specifications; hide as much as possible formalism details from the testers/modellers; use the visual models as the basis for test adequacy/coverage criteria. A subset of UML is selected and several extensions are defined to facilitate GUI modelling and enable the automatic translation to Spec#, where additional behaviour can be added if required. The approach tries to circumvent some UML limitations: difficulty to model several particularities of interactive systems [6], inconsistency problems [4], and the lack of an integrated concrete language for querying and updating the state.

Sections 2 to 5 contain the main contributions of this research work: section 2 describes the overall model-based GUI testing process proposed, integrating visual and formal models; section 3 presents guidelines and stereotypes developed to construct test-ready (and Spec# translatable) GUI models with UML; section 4 describes rules developed to translate the UML behavioural diagrams (namely protocol state machines) into Spec# (namely pre/post-conditions of the methods that trigger state machine transitions); section 5 describes test case generation and coverage analysis of the UML diagrams. Related work is described in section 6; finally, some conclusions and future work are discussed. The Windows Notepad text editor is used as a running example.

## 2 Overview of the Model-Based GUI Testing Process

Fig. 1 summarizes the activities and artifacts involved in the model-based GUI testing process proposed, which comprises the following steps:

1. **Construction of the visual model** – a set of UML diagrams and additional stereotypes are used to model the usage (via optional use case and activity diagrams), structure (via class diagrams) and behaviour (via state machine diagrams) of the GUI under test.
2. **Visual to formal model translation** – an initial formal model in Spec# is obtained automatically from the UML model according to a set of rules; class diagrams are mapped straightforwardly; state machine diagrams are mapped to pre and post-conditions of methods that model atomic user actions; activity diagrams are translated into methods that model high-level usage scenarios composed of atomic user actions.
3. **Refinement of the formal model** – the Spec# specification resulting from the translation process is completed with method bodies (called model programs in Spec#), in order to obtain an executable model that can be used as a test oracle. The added (explicit) specifications should be consistent with the initial (implicit) specifications, i.e., post-conditions should hold. Consistency may be checked by theorem proving or by model animation during model exploration.

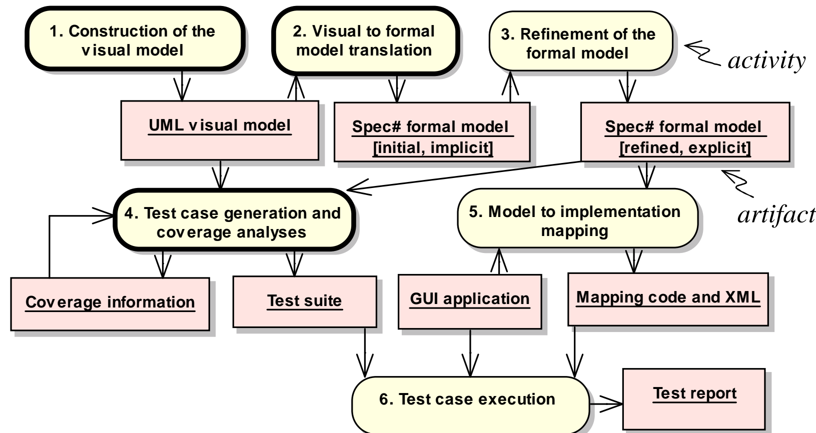


**Fig. 1**. Simplified view of the model-based GUI testing process proposed.

4. **Test case generation and coverage analysis** – test cases are generated automatically by Spec Explorer in two steps: first, a finite state machine (FSM) is extracted from the explicit Spec# specification by bounded exploration of its usually infinite state space; then, test cases are generated from the FSM according to FSM coverage criteria that is usually set to full transition coverage. In the approach proposed, test adequacy is checked by analysing the degree of coverage of the UML behavioural diagrams, which can also be used to limit the exploration.
5. **Model to implementation mapping** – a GUI Mapping Tool [12] allows the tester to interactively relate the abstract user actions defined in the model with concrete

actions on physical GUI objects in the application under test, generates a XML file describing the physical GUI objects and methods to simulate the concrete user actions, and binds such methods to the abstract ones for conformity testing.

6. **Test case execution** – Finally, test cases are executed automatically on the implementation under test and all inconsistencies found are reported.

Steps 4 and 5 and the explicit Spec# model constructed in step 3 are described in [12,13]. The focus of this paper is on the new steps 1 and 2, and the modified step 4.

# 3 Building translatable visual GUI models with UML

In order to be used as the basis for black-box GUI testing, the GUI model should describe user requirements with enough detail and rigor to allow a person or a machine to decide whether an implementation, as perceived through its GUI, obeys the specification. In addition, it should be easily constructed and analysed. Besides modelling the structure and behaviour of the GUI to support state based testing, we also model the GUI usage to capture high-level requirements and support scenario based testing. Hence, the GUI model is structured as follows:

- **Usage sub-model** (optional) – describes the purpose and typical usages of the application; modelled in UML with use case and activity diagrams;
- **Structure sub-model** – describes the structure of GUI windows; modelled in UML with class diagrams;
- **Behaviour sub-model** – describes inter/intra component behaviour; high-level behaviour is modelled in UML with state machine diagrams; detailed behaviour is modelled by Spec# method bodies.

UML **use case diagrams** are used optionally to describe the main functionalities and features of the GUI application, as illustrated in Fig. 2. Use cases can be structured as task trees, where higher level tasks are specialized, decomposed (with the «include» UML stereotype) or extended by lower level ones.

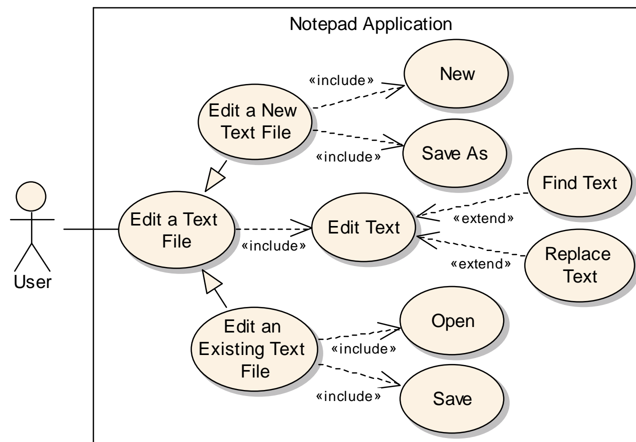


**Fig. 2**. Example of a partial use case diagram for the Notepad application.

UML **activity diagrams** are used optionally to detail use cases/tasks in a way that translates directly to test-ready Scenario methods in Spec#. Besides the user steps, they may have parameters that correspond to user inputs, pre/post-conditions (describing use case intent) and assertions, as illustrated in Fig. 3.

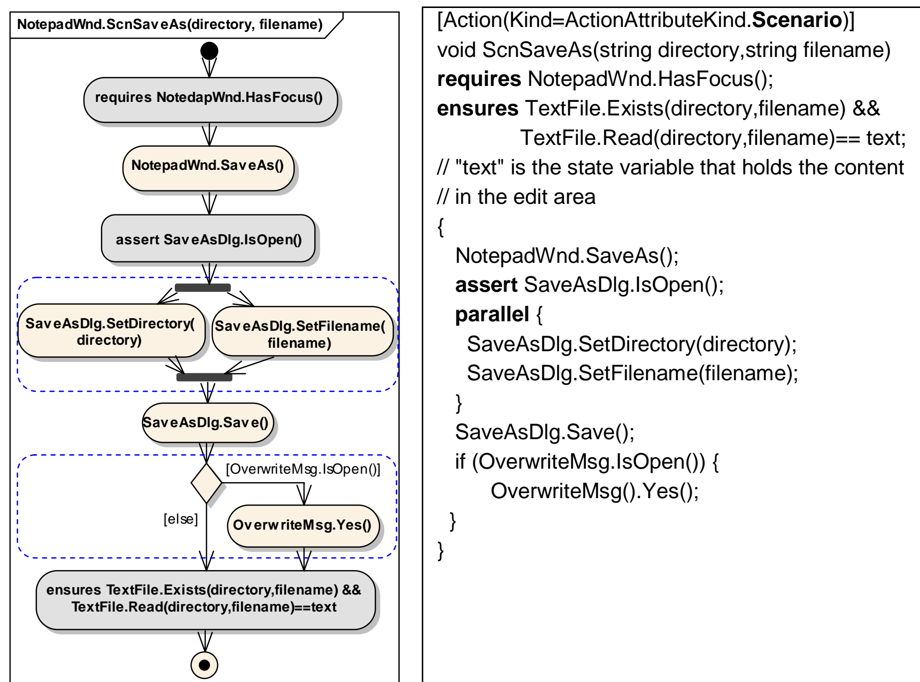


**Fig. 3**. Example of a test-ready activity diagram detailing the SaveAs use case from a user perspective (left) and the corresponding Spec# translation (right).

UML **class diagrams** are used to describe the static structure of the GUI under test at the level of abstraction desired. In most situations, it is appropriate to model top-level windows as objects (instances of classes), while the interactive controls that are contained in those windows are best modelled more abstractly by state variables and/or methods of the containing windows. In the case of the Spec# target language, one can also represent a singleton window by a set of global variables and methods grouped under a common namespace, corresponding to the concept of a module. Several annotations were defined to give special meaning to several UML elements for GUI testing, as shown in Fig. 4. An example of a class diagram is shown in Fig. 5.

The basic behaviour of the several types of windows described in Fig. 4 (including switching the input focus between modeless windows) is handled by a reusable window manager. Information and choice message boxes are on-the-fly windows with a simple structure that need only be represented in state machine diagrams, as illustrated in Fig. 6. A test ready model usually has to include some domain layer model, below the user interface layer. In the case of the Notepad application, it would take care of (non interactive) text file management.

| Annotation | Applies to | Description |
|---|---|---|
| «ModalWnd» | class, state | Represents a modal window. When open, the other windows of the application are disabled. |
| «ModelessWnd» | class, state | Represents a modeless window. When open, it's still possible to interact with other windows of the same application. |
| «InfoMsg» | class, state | Represents an on-the-fly message box that presents information to the user that must be acknowledged. |
| «ChoiceMsg» | class, state | Represents an on-the-fly message box that requires the user to choose among a limited range of options. |
| «navigate» | association | Models navigation among windows modelled as classes. |
| «Action» | method | Describes the effect of an atomic user action (e.g., press a button, enter text) on the GUI state; maps to [Action] annotation in Spec#. |
| «Probe» | method | Models the observation of some GUI state from the user eyes (e.g., see the text in a textbox); maps to a Probe annotation in Spec#. |
| «Property» | method | Models a control with a value that can be set and read by the user; it's a shorthand for the combination of a state variable, and set (Action) and get (Probe) methods. |
| «Scenario» | method | Describes how a user should interact with the GUI to achieve some goal, by a composition of lower level scenarios and atomic actions; maps to a Scenario annotation in Spec#. |
| {navigationMap} | state machine | Represents the overall navigation map of the application. |

**Fig. 4.** Stereotypes and other annotations developed for GUI modelling with UML.

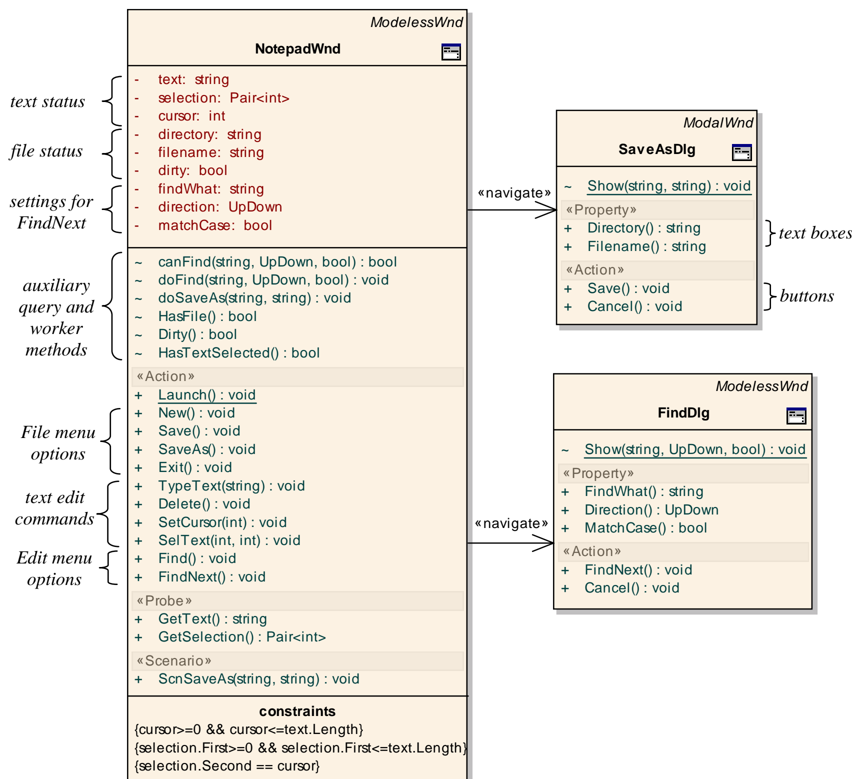


**Fig. 5.** Example of a partial UML class diagram for the GUI of the Notepad application.

UML **state machine diagrams** are adequate to model the reactive behaviour of GUIs, showing GUI states at different levels of abstraction, the user actions available at each state, their effect on the GUI state, and therefore the possible sequences of user actions. For model-based GUI testing, we advocate the usage of UML *protocol* state machines [5], instead of behavioural state machines, for the following reasons: they are more abstract (effects are specified implicitly via post-conditions, instead of explicitly via system actions); they promote the separation of concerns between the visual model (pre/post-conditions) and the refinements to introduce in the textual model (executable method bodies); they have the right level of abstraction to express black-box test goals. Each state of a protocol state machine can be formalized by a Boolean condition on the state variables (also called *state invariant* in [5]). Each transition has a triggering event that, in our case, is the call of a method annotated as Action, representing a user action, and may additionally have pre and post-conditions on state variables and method parameters, according to the syntax *[pre]event/[post]*.

The hierarchical structure of GUIs leads naturally to hierarchical state machines. The overall navigation map of the application can be represented by a top level state machine diagram annotated as {navigationMap}, as illustrated in Fig. 6.

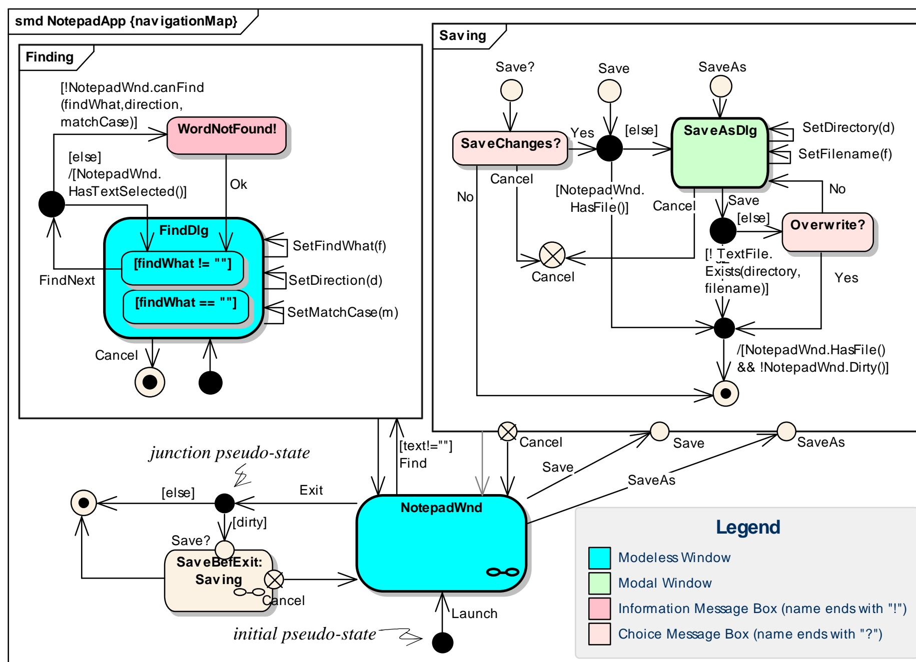


**Fig. 6.** Example of a partial navigation map of the Notepad application.

In multiple window environments, as is the case of GUIs, each state in the navigation map typically represents a situation where a given window (identified by the state name) has the input focus. Transitions represent navigation caused by user actions available in the source window/state of each transition. When one does not want to model the input focus, but only which windows are enabled, orthogonal

(concurrent) states/regions [5] can be used to model multiple modeless windows enabled at the same time. Fig. 6 also illustrates the usage of an intermediate level of abstraction, between the application and the window level, to group together states and transitions related to some use case or task (Finding and Saving composite states). Task reuse via submachine states is also illustrated (SaveBefExit submachine state). Junction pseudo-states [5] are used to factor out common parts in transitions.

The internal behaviour of each window shown in the navigation map can be detailed by a lower-level state machine diagram as illustrated in Fig. 7. States in this case represent different window modes or conditions (as identified by state names and formalized by state invariants), distinguished according to display status, user actions available, effect of those actions, test conditions, etc. Transitions are triggered by user actions available in the enclosing window. Orthogonal regions can be used to describe independent state components. For example, in the Notepad main window, the three state components indicated in Fig. 5 (top left) could be described in orthogonal regions. The same event occurrence can fire multiple transitions in orthogonal regions. For example, the TypeText user action can simultaneously cause a transition from HasTextSelected to !HasTextSelected and from !Dirty to Dirty.

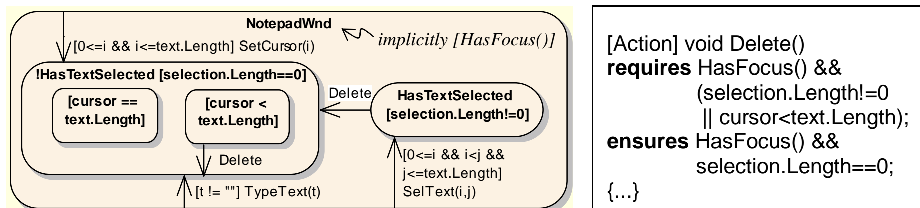


**Fig. 7.** Example of state machine describing internal window behaviour in the Notepad application (left) and partial translation to Spec# (right, simplified).

# 4 Translation to Spec#

A set of rules where developed to translate UML protocol state machines into pre/post-conditions of the Spec# methods that trigger state machine transitions. Some of the rules are presented in Fig. 8, and an application is illustrated in Fig. 7. In some cases (see, e.g., rules R2 and R3), post-conditions may need to refer to the old values of state variables (i.e., the values before method execution). Complex states are reduced by a flattening process (see, e.g., rules R4 and R5). A Boolean simplification post-processing step may be applied (see, e.g., Fig. 7).

The translation of UML activity diagrams into Spec# Scenario methods was already illustrated in Fig. 3. In general, a pre-processing step is required to discover standard structured activities (as illustrated by the dashed boxes in Fig. 3), from which appropriate control structures can then be generated, without jump instructions and control variables whenever possible. The details are outside the scope of this paper.

The translation of the UML class diagrams annotated as described in section 3 into Spec# is reasonably straightforward.

| Rule | UML protocol state machine | Translation to Spec# / Reduction |
|---|---|---|
| R1.<br>Simple<br>transition | S1<br>[cond1(svars)]<br>[pre(svars,params)]m(params)<br>/[post(svars,params)]<br>S2<br>[cond2(svars)] | [Action] m(params)<br>requires cond1(svars)<br>&& pre(svars,params);<br>ensures post(svars,params)<br>&& cond2(svars);<br>{ ... }<br>("svars" – state variables) |
| R2.<br>Multiple<br>transitions<br>with the<br>same trigger | S1 [p1]m1/[q1] S2<br>S3 [p2]m1/[q2] S4<br>...<br>(S1 and S3 may be orthogonal states) | [Action] m1(params)<br>requires (S1 && p1)<br>\|\| (S3 && p2)<br>\|\| ... ;<br>ensures (S1$_{old}$ && p1$_{old}$ ==> q1 && S2)<br>&& (S3$_{old}$ && p2$_{old}$ ==> q2 && S4)<br>&& ... ; |
| R3.<br>Junction<br>pseudo-state<br>(branch with<br>"else" case) | S1 [p1]m1 /[q1]<br>[p2]/[q2] S2<br>[p3]/[q3] S3<br>...<br>[else]/[qn] Sn | [Action] m1(params)<br>requires S1 && p1;<br>ensures q1 && (p2$_{old}$? q2 && S2 :<br>p3$_{old}$? q3 && S3 :<br>...<br>qn && Sn);<br>( "?:" is the C ternary operator) |
| R4.<br>Composite<br>state<br>(S)<br>(case with<br>initial<br>pseudo-state<br>and final<br>state) | S3<br>[p3]m3/[q3]<br>S<br>/[q] S1 t1 S2<br>[p2]m2/[q2]<br>/[q4]<br>S4 | S3<br>[p3]m3/[q3 && q]<br>S && S1 t1 S && S2<br>[p2]m2/[q2 && q4]<br>S4 |
| R5.<br>Submachine<br>states<br>(S1 ... Sn) | t S /[q]<br>t1 S1: S /[q1]<br>...<br>tn Sn: S /[qn] | ...<br>t t1/[S1] tn/[Sn]<br>S<br>[else]/[q] [S1]/[q1] [Sn]/[qn]<br>... |

**Fig. 8.** Translation rules from UML protocol state machines into Spec#. State and pre/post-conditions are abbreviated after rule R1. Due to space limitations, rules for other features (e.g., fork, join, entry, exit and merge pseudo-states) are omitted.

## 5 Test Case Generation and Coverage Analysis

With Spec Explorer, the exploration of a Spec# model to generate a finite test suite (see section 2) is based on parameter domain values provided by the tester, but there is no feedback mechanism to evaluate the quality of the test cases obtained based on appropriate GUI test adequacy criteria. To overcome this problem, we propose the structural coverage of the UML state machine model as a test adequacy criterion. To report the degree of coverage achieved and bound the exploration, the exploration process is extended as follows. Every time an action is explored with actual parameter values, it is checked if there is a corresponding transition (or set of transitions from orthogonal states) in the UML model (by evaluating their pre/post-conditions and source/target state conditions with the actual values of parameters and state variables) and, in that case, their colour is changed; otherwise, a consistency warning (if the state does not change) or error is reported. The exploration can be stopped as soon as the specified degree of coverage of the UML state machine model is achieved. At the end of the exploration, the tester knows if the coverage criteria defined were satisfied and can use the information provided to refine the domain values or the models.

A complementary technique to generate test cases is based on scenarios' coverage. In fact, the methods annotated as Scenario, describing *typical* ways (but not *all* the possible ways) of using the system, can be used as parameterized test cases for scenario based testing. Symbolic execution and constraint solving may be used to generate a set of parameter values that guarantee exercising those scenarios to a certain degree of structural coverage [16]. Scenario based testing leads to a small number of test cases, that are more adapted to test GUIs usability and more useful as acceptance tests, but does not cover the GUI functionality to the same extent as the above state based testing approach can potentially do. Scenarios can also be constructed to exercise behaviour that is difficult to cover with state based testing.

## 6 Related Work

There are few examples of model-based GUI testing tools. The main characteristics of two of them will be presented next.

IDATG [2] (Integrated Design and Automated Test Case Generation Environment) provides an editor to assist and facilitate the construction of the GUI specification as atomic user actions and as (task-oriented) test scenarios. Test cases may be generated to cover the functionality of the GUI from the former model and to check the usability of the GUI from the latter. Although it may be more pleasant to construct the model using an editor, IDATG does not provide a way to model different features of the GUI, i.e., different views of the lower level model, and to assure consistency among those views. In addition, IDATG does not provide support for test case execution, which requires a change of environment, for instance, using WinRunner.

GUITAR (GUI Testing Framework) provides a dynamic reverse engineering process to construct the model of already existing GUIs as a way to reduce the time and effort needed in their construction. The GUI model comprises an event flow graph to model intra-component behaviour and an integration tree to model

inter-component behaviour [7]. These models can be viewed graphically. However, they are not editable and cannot be constructed manually. As a consequence, its industrial applicability beyond regression testing is questionable.

There are also examples of graphical notations for modelling GUIs in the context of user interface design. Several notations are based on UML and its extension mechanisms. E.g., UMLi (UML for Interactive Applications) is an extension to UML aiming to integrate the design of applications and their user interfaces [15]. It introduces a graphical notation for modelling presentation aspects, and extends activity diagrams to describe collaboration between interaction and domain objects. Another example is the UML profile defined in the Wisdom approach [9]. However, most of these extensions are not sufficient to describe GUIs with the detail and rigour required by model-based testing tools.

# 7 Conclusions and Future Work

We have presented an approach to foster the adoption of model-based GUI testing approaches by diminishing the effort required in GUI modelling and test coverage analysis. It consists of a familiar UML-based visual notation, a translation mechanism into Spec#, and a test adequacy analysis technique. The visual notation is used to model GUIs at a high-level of abstraction and at the same time to describe test adequacy criteria. The translation mechanism aims to hide formalism details as much as possible. The adequacy of the test cases generated is accessed and reported graphically to the user on the visual models.

We are currently extending the model-based testing tools developed in previous work to support the techniques described in this paper. The prototype under development is able to manipulate UML models represented in the XML Metadata Interchange (XMI) format (*www.omg.org/technology/documents/formal/xmi.htm*).

As future work, we intend to:

- extend the tools with round-trip engineering capabilities, following the principle that the UML diagrams are partial views over the formal model;
- explore other visual behaviour modelling techniques, such as UML behavioural state machines, in order to completely hide the Spec# model at the price of a more detailed visual model (with procedural actions on transitions), and/or produce less coupled models (with the exchange of signals between concurrent state machines) at the price of a more complex execution model;
- reduce even further the modelling effort for already existing GUIs by extracting a partial GUI model by a reverse engineering process;
- use temporal logic to express additional test goals and model-checking to generate test cases, taking advantage of the IVY platform (*www.di.uminho.pt/ivy*);
- validate the approach in an industrial environment.

## References


1. M. Barnett, K. R. M. Leino, and W. Schulte, "The Spec# Programming System: An Overview", in Proceedings of CASSIS'04 – International workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices, Marseille, 10–13 Mar, 2004.
2. A. Beer, S. Mohacsi, and C. Stary, "IDATG: An Open Tool for Automated Testing of Interactive Software", in Proceedings of COMPSAC'98 – The Twenty-Second Annual International Conference Computer Software and Applications, 19–21 Aug, 1998.
3. C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes, "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer", Microsoft Research, MSR-TR-2005-59, May, 2005.
4. H. Fecher, J. Schönborn, M. Kyas, and W. P. Roever, "New Unclarities in the Semantics of UML 2.0 State Machines", in Proceedings of ICFEM'05, Manchester, UK, Nov, 2005.
5. O. M. Group, "Unified Modeling Language: Superstructure, version 2.0, formal/05-07-04", (*www.uml.org),* conferred in Dec 2006.
6. P. Markopoulos and P. Marijnissen, "UML as a representation for Interaction Design", in Proceedings of OZCHI'00, 2000.
7. A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing", in Proceedings of WCRE2003 – The 10th Working Conference on Reverse Engineering, Victoria, British Columbia, Canada, 13–16 Nov, 2003.
8. A. M. Memon, "A Comprehensive Framework for Testing Graphical User Interfaces", Pittsburgh, 2001
9. N. J. Nunes and J. F. e. Cunha, "Towards a UML profile for interaction design: the Wisdom approach", in Proceedings of the Third International Conference, A. E. a. S. K. a. B. Selic (Eds.), York, UK, Oct, 2000.
10. N. Nyman, "Using Monkey Test Tools", in *STQE – Software Testing and Quality Engineering Magazine*, 2000.
11. T. Ostrand, A. Anodide, H. Foster, and T. Goradia, "A Visual Test Development Environment for GUI Systems", in Proceedings of ISSTA'98, Clearwater Beach Florida, USA, 1998.
12. A. C. R. Paiva, J. C. P. Faria, N. Tillmann, and R. F. A. M. Vidal, "A Model-to-implementation Mapping Tool for Automated Model-based GUI Testing", in Proceedings of ICFEM'05, Manchester, UK, Nov, 2005.
13. A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Vidal, "Modeling and Testing Hierarchical GUIs", in Proceedings of ASM 2005 – 12th International Workshop on Abstract State Machines, Paris – France, 8–11 Mar, 2005.
14. A. C. R. P. Pimenta, "Automated Specification Based Testing of Graphical User Interfaces", PhD, Engineering Faculty of Porto University, Department of Electrical and Computer Engineering, 2006 (submitted, *www.fe.up.pt/~apaiva/PhD/PhD7Nov06.pdf*)
15. P. P. d. Silva and N. W. Paton, "UMLi: The Unified Modeling Language for Interactive Applications", in Proceedings of the Third International Conference UML2000 – The Unified Modeling Language. Advancing the Standard, 2000.
16. N. Tillmann and W. Schulte, "Parameterized Unit Tests with Unit Meister", in Proceedings of the Joint 10th European Software Engineering Conference (ESEC) and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-13), Lisbon, Portugal, 5–9 Sep, 2005.