View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Universidade do Minho: RepositoriUM

ELSEVIER

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 208 (2008) 77–93

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

Model-based User Interface Testing With Spec Explorer and ConcurTaskTrees [⋆]

José L. Silva[1] José Creissac Campos[2]

Departamento de Informática/CCTC, Universidade do Minho, Braga, Portugal

Ana C. R. Paiva[3]

Engineering Faculty of the University of Porto (FEUP), Porto, Portugal

Abstract

Analytic usability analysis methods have been proposed as an alternative to user testing in early phases of development due to the cost of the latter approach. By working with models of the systems, analytic models are not capable of identifying implementation related problems that might have an impact on usability. Model-based testing enables the testing of an implemented software artefact against a model of what it should be (the oracle). In the case of model-based user interface testing, the models should be expressed at an adequate level of abstraction, adequately modelling the interaction process. This paper describes an effort to develop tool support enabling the use of task models as oracles for model-based testing of user interfaces.

Keywords: Model-based testing, ConcurTaskTrees, Spec Explorer

1 Introduction

A range of techniques can be useful in testing a user interface. User testing can provide higher fidelity results when assessing user response to the system, but is not exhaustive in terms of covering all potential user interface problems. Furthermore, user testing is expensive, and access to final users is not always feasible. Analytic methods have been proposed as an alternative to user testing in early phases of development. Using models of the devices, they try to identify potential usability problems that need further consideration during the design process. Being based

[⋆] Work carried out in the context of the IVY project, supported by the Portuguese Foundation for Science and Technology (FCT) and the European Regional Development Fund (FEDER) under contract POSC/EIA/56646/2004.

[1] Email: a38222@alunos.uminho.pt

[2] Email: jose.campos@di.uminho.pt

[3] Email: apaiva@fe.up.pt

1571-0661/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.entcs.2008.03.108

on models, these techniques analyse systems that faithfully implement the models, but may ignore problems that are introduced in the implementation of the system.

An additional family of techniques, from the software testing arena, deals with the problem of testing a system's implementation. Model-based testing, in particular, enables the testing of a software artefact against a model of what it should be (the oracle). This is relevant both in a forward engineering context, where we will be interested in making sure that the implementation does not introduce problems in the proposed design, and in maintenance and redesign contexts, where we will be interested in guaranteeing that changes made to the system have not created unexpected effects. In [11] such an approach has been developed for model-based testing of user interfaces, based on the Spec# [1] specification language and the Spec Explorer tool [3]. Similar work has been proposed in [8], but in that case application specific languages and tools were developed.

When considering user interface testing, however, it is desirable that the models used as oracles are expressed at the same level of abstraction as required during the user interface design process. In fact, rather than create models specifically to aid testing, it would be better to use the models already developed during design. This is not the case of the Spec# models used in [11], nor the models used in [8]. While both models can be considered as dialogue control models (as developed in [6]) the abstraction level of the specification languages used was found to be too low to be practical in a user interface testing context.

Task models are amongst the most commonly used models during interactive systems design. In this paper we address the abstraction level problem identified above by investigating the feasibility of using task models as oracles for model-based testing of user interfaces. We will be using ConcurTaskTrees (CTT) [14] as the task modelling notation. This has a number of potential advantages:

- the models are at a level of abstraction familiar to user interface designers/developers;
- testing will follow the anticipated use of the system;
- and (especially when such models have been produced during the design stages) the cost incurred in developing the oracle is much reduced.

At this point it must be noted that this paper is not about test case generation for user interfaces (see [4] for an example of that type of work). As it will be shown, the test case generation step is performed by the Spec Explorer tool. The subject of interest in the present case is the test oracle that will be used as a measure of the implementation's quality.

The paper is structured as follows: sections 2 and 3 introduce model-based testing in general, and model-based testing in particular of user interfaces with Spec Explorer; section 4 introduces CTT task models, and section 5 describes how they can be used as oracles in a model-based testing setting; section 6 describes an example of use, and section 7 discusses the results achieved and lessons learnt; the paper ends with conclusions and ideas for future work in section 8.

2 Software testing and model-based software testing

Software testing is a dynamic technique which aims to increase confidence in the correctness of the software, i.e., check if the software does what is expected. There are many approaches to software testing [11] but effective testing is difficult to achieve and has to do with process systematization, automation, tester's expertise, and a bit of luck. Model-based testing increases software testing systematization and automation. It compares the state and behaviour of a software product against its model.

The model used can vary in formality: formal, semi-formal or informal. Formal executable models can be used as test oracles determining whether or not the results of a test case execution are correct. Test cases are sequences of possible actions with input parameters and results expected. They can be generated automatically from the model. They are executed both over the model and over the software implementation, and the results obtained are compared. Whenever there is an inconsistency between results expected and results obtained a potential problem has been found.

There are several examples of model-based testing tools for software API (Application Programming Interface) testing (for example, [5,3]). Generally speaking, these tools have to deal with two main problems: controlling the *state explosion problem*, and *bridging the gap* between the models and software implementations. The state explosion problem has to do with the number of states of a software system. This number may be so large that it may be impractical to analyse it completely. So, test case generation has to be controlled appropriately to generate test cases of manageable size while still guaranteeing adequate testing. To bridge the gap between models and implementations, concrete test cases must be constructed from abstract test cases in order to be executed over a software implementation. This means that a mapping between model and implementation must be created, so as to run related actions at both levels (implementation and model) in a "lock-step" mode, comparing the results obtained after each step.

Spec Explorer [3] is a model-based testing tool from Microsoft Research. Software models can be constructed in AsmL or Spec# [1]. Test cases are generated in two steps: a Finite State Machine (FSM) is constructed from an exploration process of the software model, then a traversal engine is used to unwind the resulting FSM to produce behavioural tests that fulfil the coverage criteria (e.g., full transition coverage; shortest path; random walk). Spec Explorer has techniques to deal with the state explosion problem such as state filters; additional pre-conditions; restriction of the domains; equivalence classes; stop conditions; scenarios and on-the-fly exploration. For API testing, specifications should be constructed in such a way that it is possible to map abstract actions with concrete actions within a software DLL for test execution.

3 Model-based UI testing with Spec Explorer

Testing graphical user interfaces (GUIs) poses additional challenges either when compared to API testing, or when one wishes to automate the testing process. These challenges arise due to time constraints, test case explosion problems, the need to combine testing techniques, and test automation difficulties [11]. Spec Explorer can also be used to test GUIs but considerable effort is required, and three challenges must be faced:

- There is the need to write adaptor code to simulate user actions over a GUI so that the implementation can be tested without actual user intervention.
- There is the need to map those concrete actions from abstract actions for test execution.
- There is the need to develop models of the user interface at an adequate level of abstraction, ideally capturing possible user actions over the GUI and the effect of those actions.

The above issues are addressed by Paiva in [11]. She has developed several extensions to the model-based testing environment Spec Explorer to foster its application to GUI testing. Besides constructing the "GUI Mapping Tool" which automates the mapping between the GUI model and its implementation, and automatically generates the adaptor code [12], she also proposed guidelines to model GUIs in Spec#, and developed another tool to avoid test case explosion taking into account the hierarchical structure of GUIs [13].

The level of abstraction of the GUI models can vary, either modelling atomic user actions (like clicking on a button); or composite actions constructed as sequences of atomic actions (e.g., "drag and drop"); or modelling high level GUI constructs (like a GUI navigation map); or modelling scenarios which describe how the user should interact with the GUI to achieve a goal. The level of abstraction of the GUI models used in [11] was set to the level of atomic user actions. While the approach was successful in testing and identifying bugs on currently available systems, it was found that the construction of the models required too much work. Techniques such as specification reuse were used but, even so, the construction of such models required substantial work and resulted in models with around 40% of the lines of its implementation. Additionally, the use of Spec# as the GUI modelling language means that user interface developers would have to learn yet another language, and one which works at a level of abstraction that may be too concrete.

Task models are common for modelling GUIs. Task models may be used to describe typical usage scenarios in a higher level of abstraction. The goal of this research work is to make the GUI models used for model-based testing more abstract (when compared to, for example, [11]), and diminish the effort in their construction. We propose to achieve this by adopting task models as oracles.

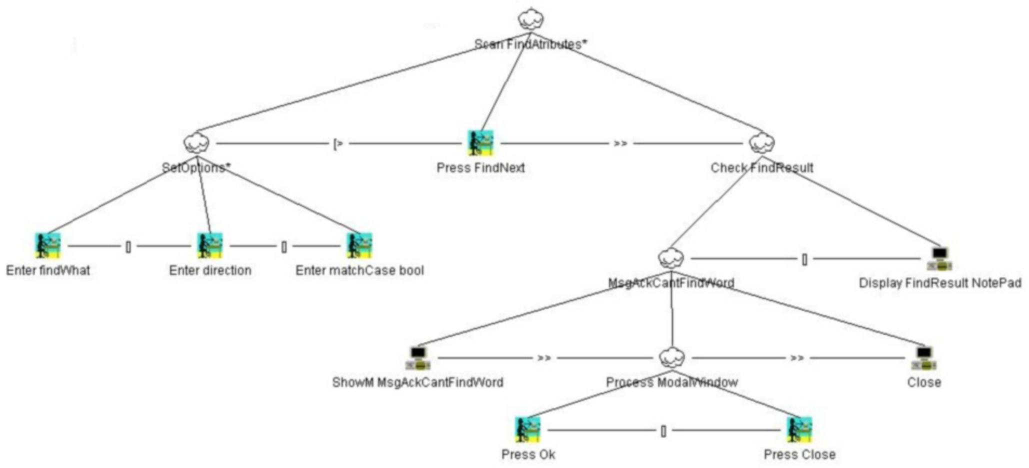


Fig. 1. CTT Model

4 CTT Task Models

The basic idea of task modelling is to identify useful user-device interaction abstractions highlighting the main issues that should be considered when designing effective interactive applications. A task is an activity that should be performed in order to reach a particular goal. We use task models as the basis for our work because they are a representation of the logic of the interactive layer of the applications, and they describe assumptions about how the user will interact with the device. Considering that we are interested in testing the behaviour of systems' user interfaces, task models seem a promising model against which to test. This will be further discussed in section 7.

Having decided to investigate how task models can be used as oracles in a model-based testing context, we must now choose a task modelling notation. We have chosen the ConcurTaskTrees (CTT) notation. This choice was based on a number of factors: CTT belongs to the family of hierarchical task analysis notations, the most common approach to task analysis; CTT itself is becoming a popular language for task modelling and analysis; and, finally, and most important of all, CTT has tool support. In fact, the TERESA tool[9] supports editing and analysis of CTT models, and a number of features relating to the animation of task models that will be useful in our work.

In figure 1, an extract of the task model which will be used as example in section 6 is presented. The task model is a hierarchical decomposition of the sub-tasks that must be carried out to achieve a given goal. The goal is at the root of the tree. Leaves represent the actual interaction between the user and the device. Four different types of tasks are possible:

- interaction tasks () represent user input to the application;
- application tasks () represent application output to the user;
- user tasks () represent decision points on the user's part;

- while interaction, application and user tasks must appear as leaves in the model, abstract tasks (☁) are used to structure the model and appear as internal nodes in the tree.

The problem of identifying user and application tasks is simplified through this classification.

The particular sub-tree presented in figure 1 belongs to the task model for finding text in a text editor that will be used in section 6. The semantics of the model is defined by the possible traversals of the tree. Tree traversal proceeds left to right in a depth first fashion, and is governed by operators applied to tasks or pairs of tasks at the same level. We now introduce the most relevant operators by describing the dialogue induced by the tree in the figure. First of all, we note that the top task (*ScanFindAttributes*) is iterative (iterative task operator: $*$), meaning that it can be repeatedly executed. To execute it, *SetOptions* must be executed (itself also an iterative task). At this point the user can choose between three possible alternatives (choice operator: []): *Enter findWhat*, *Enter direction*, or *Enter matchCase bool*. At any point in the execution of *SetOptions* the user can choose to execute task *Press FindNext* (disabling operator: [>), thus terminating the previous task. After *Press FindNext*, task *Check FindResult* becomes possible (enabling operator: >>). Dialogue continues by going down the *Check FindResult* sub-tree.

The description above does not include all CTT operators. Additional relevant operators in the present context are the optional task operator (expression "[T]" means that task T is optional), and the suspend resume operator (expression "T1 |> T2" means that T2 interrupts T1, but T1 will continue from where it was interrupted after T2 is finished). For a complete description of the language see [14].

5 Using CTT models as oracles

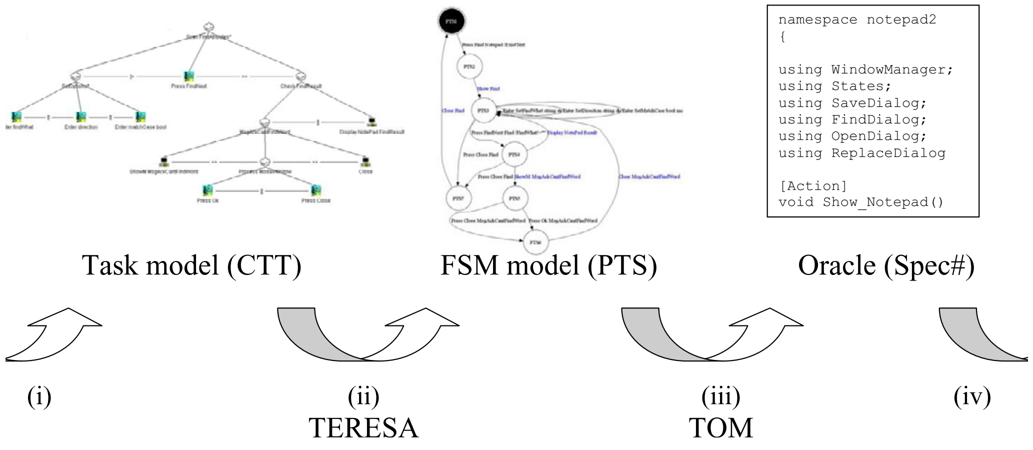
In order to automatically generate oracles from the task models, we must establish some conventions about how CTT is to be used. In this section, we describe both these conventions and the process used to generate the Spec# oracles from the CTT models.

In short the process is the following (see figure 2):

(i) task models are either developed according to our conventions, or adapted to them from already existing ones — these modelling conventions are described later in the section. This should imply minimal work if a task model has already been developed during the design process, alternatively building the task model from scratch will amount to capturing high level user interface requirements for the system to be tested;

(ii) finite state machine representations of the task models are automatically generated — this is done by resorting to the TERESA tool;

(iii) a Spec# oracle is generated from the finite state machine — this is done by a



```
namespace notepad2
{

using WindowManager;
using States;
using SaveDialog;
using FindDialog;
using OpenDialog;
using ReplaceDialog

[Action]
void Show_Notepad()
```

Fig. 2. The process

tool that we are developing (TOM – Task to Oracle Mapping tool) [4], the tool takes the finite state model and information obtained directly from the task model;

(iv) the testing framework from [11] is used to test the system against the oracle — this might involve some refinement of the oracle in order to introduce relevant information that was not present in the task model (in that case, we will refer to the oracle in step (4) as the *preliminary oracle*, and to the result of the refinement step as the *refined oracle*).

Each of these phases is further detailed below.

5.1 *Writing/Refining the CTT models*

In order to automatically generate test oracles from CTT models we must further refine the type of tasks that are available. That is because the oracles will be expressed at a low level of abstraction and we want to include as much information as possible in the task model in a transparent manner (for example, regarding issues of window management). We decided to make use of the keywords from the Framework for Integrated Tests (FIT) [10], and structure the CTT models around them.

The main objective of FIT is to help design and communicate test cases through concrete examples. Notations to specify the test cases have been developed, as well as a framework to help automate API testing procedures (hence, not immediately applicable to GUI testing).

One specific type of test is workflow validation. A workflow consists of a sequence of method invocations and checks on system responses. For specifying the workflow, four basic types of actions are offered by FIT:

- Start $<$ *class* $>$ — indicates the class which will be tested (thus setting the context for the following steps in the workflow);

[4] Not to be confused with the TOM (Task Oriented Modelling) project [16].

- Enter < *process* > < *argument* > — indicates the invocation of a method with one argument;
- Press < *process* > — indicates the invocation of a method without arguments;
- Check < *process* > < *value* > — verifies if the method call returns the given value.

In order to use FIT in the context of CTT models, we extended it by adding actions to deal with windows (e.g., closing and opening) and adding parameters to some of the existing ones. These extensions are described bellow:

- Because FIT does not have the notion of a window, we had to introduce keywords to deal with them (*Show*, *ShowM*, and *Close*).
- Because in a GUI it is possible for users to enter values of different types, we introduced an optional argument to *Enter* where the modeller may specify the type of the value expected.
- By default *Press* keywords are understood in the context of the currently active window, an additional argument was added to keyword *Press* for situations where we want to specify the action of pressing a button in a window other than the currently active one.
- The *Check* keyword was substituted by the *Display* keyword, indicating that some result is presented in the screen.

For modularity reasons, we organize the model into namespaces. Each Start keyword creates a new namespace, and tasks are interpreted in the context of a namespace.

The resulting set of actions is the following (arguments between square brackets are optional):

- Start < *task* > — creates a task that represent a new namespace;
- Enter < *field* > < *value* > [< *type* >] — the user enters *value* of type *type* in *field* (if the type is String, it can be omitted);
- Press < *button* > [< *window* >] — the user presses *button* in *window*, if the window is not specified the current window is assumed;
- Show < *window* > — the application opens a non-modal window;
- ShowM < *window* > — the application opens a modal window;
- Display < *value* > < *window* > — the application displays *value* in *window*;
- Close [< *window* >] — the application closes *window*, if the window is not specified then the current window is assumed.

With these extended FIT keywords it becomes simpler to express task models that can later be mapped to adaptor code.

In addition to the structuring of task names just described, the approach makes extensive use of the facilities provided by the TERESA modelling tool in terms of pre-conditions definition, and use of description fields to attach additional information to the tasks (for example, to define the effects of atomic tasks in the state of the model so that pre-conditions might later be calculated).

5.2 Presentation Tasks Sets and Finite State Machines

To use a task model as an oracle during model-based testing, we must be able to animate the model, in order to compare its behaviour with that of the actual application. From CTT task models we can generate task sets that represent which tasks are enabled at each particular point in the interaction. These are called Presentation Task Sets (PTS). PTS are automatically derived by TERESA through analysis of the formal semantics of the CTT temporal operators [9].

The PTS generation feature will be very useful for implementation of our tool, because from the PTS we are able to deduce a Finite State Machine (FSM) that represents the possible transitions that the user can generate when interacting with the interface. States correspond to the PTS, transitions correspond to the tasks that can be executed from each state.

5.3 Spec# model generation

From the FSM represented by the PTS we must now create a Spec# model suitable for model-based testing with Spec Explorer. Spec# [2] extends C# with specification constructs like pre-conditions, post-conditions, invariants, and non-null types. A model written in Spec# describes a possibly infinite state transition system. States are modelled by state variables; transitions are modelled by methods annotated as *Action*. Actions can have pre-conditions, written as *requires* clauses, that define the states in which they are enabled.

There are different kinds of actions:

- *controllable* — to describe actions that are controlled by the user of the modelled system;
- *probe* — to describe actions that read the state of the system and do not perform updates; these actions are invoked by the test harness in every state where they are enabled to check whether the model and the implementation have the same characteristics in a given state;
- *observable* — to describe the spontaneous execution of asynchronous actions in the GUI under test possibly caused by some internal thread; and
- *scenario* — to describe sequences of sub-actions.

Action methods can be used to model complex user actions (e.g., enter a string into a field, issue a command, load contents from files, etc.) and describe its effect on the overall state of the system.

A tool (TOM) is being developed that performs the generation of the Spec# model from the task model. The tool starts from the information in the PTS. PTS generation, however, does not take into account all the additional semantic information that has been included in the task model (pre-conditions, system state, etc.). Additionally, modal sub-dialogues, introduced with the *ShowM* keyword, are something that TERESA is not aware of, and we must perform the adaptation with our tool. To solve this, the tool accesses the actual CTT model. This is so that the tool might be able to extract all the necessary information form the CTT model that is missing in the PTS.

The models that are generated follow the structure defined in [11]. Providing a full description of these models is out of the scope of the paper. In short, there is one module or class to describe each window within the GUI under test; state variables to describe the state of the windows and their controls; and methods annotated as *controllable* to describe the behaviour and control the GUI and methods annotated as *probe* to observe the state of the system and check the effect of the actions performed.

A simple window manager model is used to deal with the application windows being placed on and removed from the screen. Window manager behaviour, at the CTT level, is described by *Show* and *Close* actions, which are translated to the appropriate method calls in Spec# (e.g., *WindowManager.AddWindow(...)*). Additionally, for *Press*, *Enter*, and *Display* tasks, the pre-condition which checks if the current window is enabled is automatically inserted into the generated Spec# model.

5.4 Refining and using the oracle

To perform conformance tests with Spec Explorer, a mapping between the model actions and implementation methods must be provided. For this, some glue code might be needed to map the data and method calls in both directions.

The GUI Mapping Tool [12] assists the user in relating the model actions ("logical" actions) to "physical" actions on "physical" GUI objects. A major difficulty solved by the tool is the identification of the GUI physical objects that the model actions refer to. The adaptor code is automatically generated from high-level mapping information and methods of this code are automatically bound to related modelled actions of the specification.

An additional step that must be performed is the definition of the domains of the input parameters (the possible values each parameter might take). Spec Explorer does not provide support for the definition of these domains. At the moment this step has to be done manually. However, it would be possible to have that information incorporated in the CTT model, and have TOM extract it and process it automatically.

Defining the domains for the parameters is a crucial point in the testing process. A random definition of the domains may result in a FSM that does not have relevant properties from the testing point of view. Generating test cases from a FSM like this will not be very useful because it will not test some relevant properties or, in the worst case, it will not test anything useful. A careful choice of domains will help test situations where the user input erroneous values, or values that are at the boundaries of the acceptable. These are typical situations that can cause problems in an implementation.

The task model identifies the main functionalities of the application under test. It describes the requirements of the application and can be used for requirement coverage analysis. Applying structural coverage analysis on the task models, we determine which domains to associate with the variables in order to achieve full coverage.

Once the oracle is complete, Spec Explorer will generate a FSM from the Spec# model and then will generate test cases from this FSM. The coverage criteria can be set to:

- Full Transition Coverage: the test suite generated covers all transitions of the FSM. In addition, this algorithm can be configured so as to generate test segments/paths that whenever possible return to the initial state, and can also be pruned so as to generate test segments bound by a given number of transitions;
- Shortest Path: the test suite generated is the shortest path (sequence of transitions) that reaches a specified goal state;
- Random Walk: generates a test suite with a single sequence of invocations. At each state, one of the outgoing transitions is randomly selected.

The construction of the Spec# model may seem a step back in the overall process; however, this step is required because we may need to refine the automatically generated Spec# model in order to test the GUI. The specification of auxiliary functions, such as finding a word within a text, will be provided at this stage, as typically they will not be part of the CTT model. This refinement of the preliminary Spec# model will produce the final oracle for testing.

After the steps above, test cases can be finally generated and executed and inconsistencies between the specification and the implementation are reported using Spec Explorer and its extensions [11]. These inconsistencies will occur every time probe actions produce different results for the oracle and the system in the same state.

6 An example

In this section we demonstrate, with an example, how the tool that is being developed is able to generate Spec# oracles automatically from CTT models which respect the conventions defined in the previous section. The application being used to demonstrate the approach is the Windows Notepad editor. Due to space constraints, we concentrate the analysis on the *Find* functionality of the Notepad.

The section is divided into four subsections, each one describing one phase of the process applied to generate the oracle.

6.1 The task model

Since we were dealing with a third party application, we started the process by developing a task model from scratch. This was done based on previous experience of using the tool, and on extended testing of all possible interactions with the tool. TERESA was used to write the model.

Seven sub-goals for the editor were identified. These consisted of the normal editing of text (which was not further refined), and six operations: replacing text, finding text, saving the file with a different name, saving the file with the current name, opening a file, and creating a new file.

As stated, we will restrict the analysis to the find sub-goal. The part of the model representing the interaction with the find dialogue window has already been presented in figure 1.

Although not visible in the figure, the model has been enriched with pre-conditions to express constraints on the availability of certain actions. These pre-conditions are expressed as properties of the atomic tasks in the model. Two such tasks are *Press Find* and *Press FindNext*. In the first case, text in the Notepad must exist. Only if the pre-condition is true, is it possible to press the Find button in the main window of the Notepad. In the second case, the text field holding the text to be searched must not be empty. That is represented by the expression "`FindWhatG != empty`" where the `FindWhatG` variable contains the text that was previously inserted in the text field. These pre-conditions are necessary to generate a useful Spec# oracle.

In order for the pre-conditions to be effective, the model must also be enriched with information regarding the state of the application, and the effect of the different actions in that state. This was done through annotations in the description field of the task properties in TERESA.

For the definition of the state, the top level goal and appropriate sub-goals were annotated with variable declarations. Variables declared at the top level included the string to be searched ("`string FindWhatG = empty`"), and the text in the editor window ("`string TextG = empty`"), among others. At the *start FindDialog* task a variable representing the direction of the search was declared ("`string DirectionG = -Down-`"). Manipulation of the state is done at the leaves of the task tree, and a simple syntax has been developed to allow for this. Hence, task *Enter direction* was annotated with the expression "`DirectionG = direction`" to express the fact that it affects the direction of search. All the expression in the model will be further refined in following steps (see section 6.4).

Note that we could have opted for including less detail in the model at this stage, leaving that to the oracle refinement step. The balance between how much information to include in the task model, and how much to leave for the oracle refinement stage, is one that must be decided by the team/person carrying out the development/testing.

6.2 Generating the Finite State Machine

Having developed the task model, the next step is to generate the FSM representation of that model. This step is fully automated and executed in the TERESA tool (by generating the PTS).

Some problems were identified in this step that relate to how TERESA handles some of the operators. In fact, it was found that the tool does not always correctly generate the PTS for disabling ([>) and suspend/resume (|>). Since access to the sources of the tool was not possible, we are currently planning to correct these

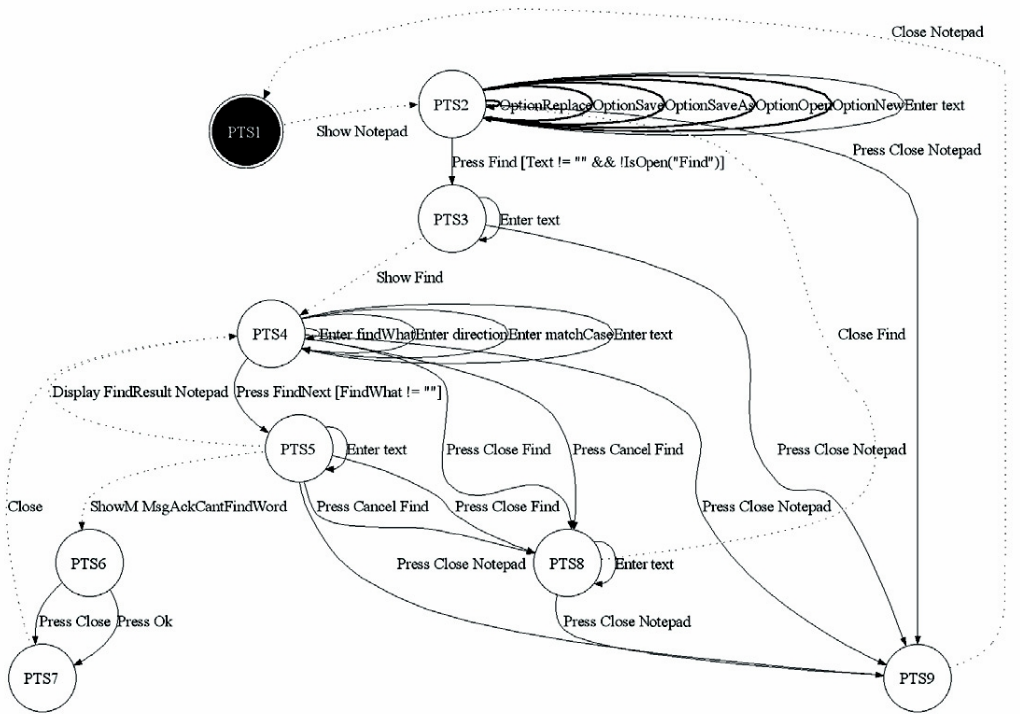


Fig. 3. FSM of model

aspects in TOM.

6.3 Generating the preliminary Spec# oracle

This, again, is a fully automated step. In this case, performed by our tool (TOM). With the PTS generated by TERESA, and the original CTT model, TOM was able to create a richer version of the original FSM (expressed with the PTS) representing the intended behaviour of the model. This FSM is expressed in Spec#, and becomes the preliminary version of the oracle.

In figure 3 we present a graphical representation of the FSM created by the tool to produce the Spec# oracle. This representation describes the find goal only. The black state corresponds to the initial and final state. Normal arcs represent "Interaction Tasks" which are translated to *controllable* actions in Spec#. Dotted arcs represent "Application Tasks" which are translated to *probe* actions in Spec#, and will check if the effect of the user actions on the GUI is the one expected. Bold arcs represent abstract (not refined) tasks in the original model (remember that we are only considering the find goal) which will not be translated to Spec#.

6.4 Using the oracle

In order to use the generated oracle, some final adjustment had to be made due to the need to add more semantics to the model. When the CTT model does not describe the complete behaviour of the GUI under test, the tester as an opportunity to refine manually the preliminary generated Spec# oracle. For example, the tester may opt to maintain the CTT model in a higher level of abstraction and refine later the generated Spec# model to describe, for instance, the behaviour of finding a word in a text. The above adjustments took only half a person hour.

Once the oracle was ready, test cases were generated automatically and executed over the GUI under test using the framework in [11]. For this particular sub-set of the task model Notepad was found to provide adequate support. This was to be expected both because Notepad is a stable commercial tool, and because we were only testing a small subset of its functionality. However, there is no reason to believe that errors already detected by larger models of the Notepad [12] would not be found by this approach when the functionality taken into account in the task model is extended.

When compared to the types of models used in [12], we saved time in the GUI modelling activity (just half of the time was needed) and were able to approximate the modelling notation to the ones testers like better (they usually prefer graphical notations to textual ones).

7 Discussion

At the start of the project, the vision was that of developing task models that would fully capture all possible interactions with the system. Several authors have looked into generating interactive systems from their task models (see, for example, TERESA [9] or Mobi-D [15]), so this would be the counterpart: testing the systems against the task models. The analysis would be a complement to traditional analytic approaches to usability analysis, enabling an analysis of whether the final implementation obeys the envisaged (and analysed) design.

As we started developing such models it become apparent that the task language had not been designed to support the level of detail that would be needed. Modelling dialogues involving wizards and modal windows created considerable problems. In fact, task models are supposed to capture how users should interact with a system to achieve some goal, not the design of the system itself. Hence, a separation must be made between task models (developed from the user's perspective), and dialogue models (developed from the system's perspective).

Task models capture idealised user behaviour. In practice, users may achieve goals in different ways and therefore oracles based on these idealised behaviours may fail to test the range of possibilities. Dialogue models capture dialogue control within the application (i.e., all possible behaviour of its GUI). The original Spec# oracles in [11] are more akin to dialogue models than task models, since they captured all possible system behaviours.

Hence, we have found limitations generating the test oracles directly from task models. There are two particular areas where problems might arise: testing specific implementation details, and testing behaviours outside the task model.

Regarding the first, testing specific implementation details becomes less straight forward. If a specific dialogue is implemented with a wizard, testing the wizard becomes problematic since that information is not present in the task model. This is an issue that needs further analysis, but the envisaged solution is to annotate the task model with information about specific implementation patterns used. In fact, this has already been done in the case of modal windows.

Regarding the second, testing behaviours outside the task model will not be possible since the oracle being used will not be able to cope with them. However, because the testing process is mostly automated, once the testing set up has been created it is easy to envisage a scenario where different variations of the task model are used to test the system against. Hence, by applying appropriate transformations, we will be able to test the system against possible user erroneous behaviour. For example, we might alter the task model to allow situations were the user performs post-completion errors [7].

In theory the above idea can be extended to the point were we are testing the device against oracles that capture relevant usability related issues. In that context, the analysis will no longer be limited to analysing the quality of the implementation against a given pattern of quality (the oracle), but will also enable some analysis of the usability of the system.

Despite the limitations, the results described here indicate that task models can be a viable solution for generating test oracles. What must be take into account is that these oracles will naturally test “normal” use of the application only. By normal we mean those behaviours that are expected of the typical users, and against which the application was designed. Hence, these oracles will be useful in a first stage of testing where the goal is to test if the application behaves as expected under typical conditions.

In a second phase, we will be interested in testing resilience (the capacity of the GUI to deal with user errors, and/or unexpected behaviours). For that, one possibility is to introduce changes into the task model to allow for those anomalous or boundary behaviours to be present in the task model, as discussed above. Alternatively a more dialogue control based oracle must be used.

8 Conclusion and Future work

Model-based testing can have a role in guaranteeing the quality and usability of interactive systems. While it does not enable a true assessment of the users’ response to the system, it can help identify implementation related problems that are not detected with other model based analysis techniques. This is achieved in a mostly automated process, and without the cost of a full blown usability test. The technique will be relevant in a forward engineering context, as well as in maintenance and redesign contexts.

Model-based testing works by comparing the execution of a running system against a model (or oracle) of what the system should behave like. Since the testing phase is automated, most of the cost of applying the technique goes into developing the model, and mapping it into the existing implementation. For the technique to be successful in the area of interactive system design, adequate models to use as oracles must be found. These models should already be in use in the development of the system.

In this paper we have shown that task models, despite limitations, can be used to generate oracles economically in an interactive systems model based testing context. Tool support is being developed that enables the automated generation of oracles from task models. An example of use has briefly been described, and the advantages and shortcomings of the approach have been discussed.

In the future we plan to extend the approach to better cope with different user interface patterns without having to expand the task models excessively. We also plan to look into the generation of test cases from the task models as a complement the the work developed thus far.

Acknowledgement

The authors wish to thank João P. Faria for suggesting the use of FIT, and Michael D. Harrison for his helpful comments during the preparation of the paper.

This work was partially carried out while José Campos was a visiting research fellow at Newcastle University. He acknowledges with thanks the support of UK EPSRC grant EP/F01404X/1.

References

[1] Barnett, M., R. DeLine, B. Jacobs, M. Fhndrich, K. R. M. Leino, W. Schulte and H. Venter, *The spec# programming system: Challenges and directions*, in: *VSTTE2005 – Verified Software: Theories, Tools, Experiments*, ETH, Zurich, Switzerland, 2005, uRL: http://vstte.ethz.ch/program.html.

[2] Barnett, M., K. R. M. Leino and W. Schulte, *The Spec# programming system: An overview*, in: G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Lecture Notes in Computer Science **3362** (2005), pp. 49–69.

[3] Campbell, C., W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann and M. Veanes, *Model-based testing of object-oriented reactive systems with spec explorer*, Technical Report MSR-TR-2005-59, Microsoft Research (2005).

[4] d'Ausbourg, B., G. Durrieu and P. Roché, *Deriving a formal model of an interactive system from its UIL description in order to verify and to test its behaviour*, in: F. Bodart and J. Vanderdonckt, editors, *Design, Specification and Verification of Interactive Systems '96*, Springer Computer Science, Springer-Verlag/Wien, 1996 pp. 105–122.

[5] Hartman, A. and K. Nagin, *The AGEDIS tools for model based testing*, in: G. Rothermel, editor, *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis* (2004), pp. 129–132.

[6] Horrocks, I., "Constructing the User Interface with Statecharts," Addison-Wesley, Harlow, England, 1999.

[7] Li, S. Y. W., A. Blandford, P. Cairns and R. M. Young, *Post-completion errors in problem solving*, in: B. G. Bara, L. Barsalou and M. Bucciarelli, editors, *CogSci 2005: XXVII Annual Conference of the Cognitive Science Society* (2005), pp. 1278–1283.

[8] Memon, A. M., M. E. Pollack and M. L. Soffa, *Automated test oracles for GUIs*, in: D. S. Rosenblum, editor, *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering* (2000), pp. 30–39.

[9] Mori, G., F. Paternó and C. Santoro, *Design and development of multi-device user interfaces through multiple logical descriptions*, IEEE Transactions on Software Engineering **30** (2004), pp. 507–520.

[10] Mugridge, R. and W. Cunningham, "Fit for Developing Software: Framework for Integrated Tests," Prentice-Hall, 2005.

[11] P., A. C. R. P., "Automated Specification-Based Testing of Graphical User Interfaces," Ph.D. thesis, Engineering Faculty of Porto University, Department of Electrical and Computer Engineering (2007). URL http://www.fe.up.pt/~apaiva/PhD/PhDGUITesting.pdf

[12] Paiva, A. C., J. P. Faria, N. Tillmann and R. M. Vidal, *A model-to-implementation mapping tool for automated model-based GUI testing*, in: K.-K. Lau and R. Banach, editors, *Formal Methods and Software Engineering*, Lecture Notes in Computer Science **3785** (2005), pp. 450–464.

[13] Paiva, A. C., N. Tillmann, J. P. Faria and R. M. Vidal, *Modeling and testing hierarchical guis*, in: *Proceedings of the 12th International Workshop on Abstract State Machines, ASM'05*, 2005, pp. 329–344, uRL: http://www.univ-paris12.fr/lacl/dima/asm05/asm05-contents.html.

[14] Paternò, F., *ConcurTaskTrees: An engineered notation for task models*, in: D. Diaper and N. Stanton, editors, *The Handbook of Task Analysis for Human-Computer Interaction*, LEA, 2003 pp. 483–503.

[15] Puerta, A. and J. Eisenstein, *Towards a general computational framework for model-based interface development systems*, in: *Proceedings ACM IUI99* (1999), pp. 171–178.

[16] Warren, C., *The task oriented modelling (TOM) approach to the development of real-time safety-critical systems*, in: S. Ashlund, K. Mullet, A. Henderson, E. Hollnagel and T. White, editors, *INTERACT '93 and CHI '93 conference companion* (1993), pp. 167–168.