Test Case Generation from Mutated Task Models

Ana Barbosa
Universidade do Porto
Faculdade de Engenharia
Departamento de Engenharia Informática
Rua Dr. Roberto Frias, s/n
4200-465 Porto
PORTUGAL
ei05089@fe.up.pt

Ana C. R. Paiva
Universidade do Porto
Faculdade de Engenharia
Departamento de Engenharia Informática
Rua Dr. Roberto Frias, s/n
4200-465 Porto
PORTUGAL
apaiva@fe.up.pt

José Creissac Campos
Departamento de Informática/CCTC
Universidade do Minho
Campus de Gualtar
4710-057 Braga
PORTUGAL
jose.campos@di.uminho.pt

ABSTRACT
This paper describes an approach to the model-based testing of graphical user interfaces from task models. Starting from a task model of the system under test, oracles are generated whose behaviour is compared with the execution of the running system. The use of task models means that the effort of producing the test oracles is reduced. It does also mean, however, that the oracles are confined to the set of expected user behaviours for the system. The paper focuses on solving this problem. It shows how task mutations can be generated automatically, enabling a broader range of user behaviours to be considered. A tool, based on a classification of user errors, generates these mutations. A number of examples illustrate the approach.

Author Keywords
Task models, model based GUI testing

ACM Classification Keywords
H.5.2. Information interfaces and presentation: User Interfaces. D.2.5. Software Engineering: Testing and Debugging.

General Terms
Human Factors, Reliability

INTRODUCTION
Graphical User Interfaces (GUIs) are nowadays the pervasive means of interaction between users and computer systems. Clearly, the quality of the GUI is a determining factor in the decision to use a system or not. At the very least, it will have an impact in the effectiveness, efficiency and satisfaction with which the system is used [1].

GUI quality is a multifaceted problem. Two main aspects can be identified. For the Human-Computer Interaction (HCI) practitioner the focus of analysis is on *Usability*, how the system supports users in achieving their goals (which can range from being productive to having fun, depending on the specific system being considered). For the Software Engineer, the focus of analysis is on the quality of the implementation (from the degree of coverage of requirements, to the maintainability of the code). Clearly there is interplay between these two dimensions. Usability will be a (non-functional) requirement to take into consideration during development, and problems with the implementation (e.g., *bugs* in the code) will create problems to the user, hindering usability.

In a survey of usability evaluation methods, Ivory and Hearst [2] identified 132 methods for usability evaluation, classifying them into five different classes: (User) Testing; Inspection; Inquiry; Analytical Modelling; and Simulation. They concluded that automation of the testing process is greatly unexplored. Automating the testing process is a relevant issue since it will help reduce analysis costs by enabling a more systematic approach to testing.

Another possible division of evaluation methods is between those that require users to use the system, and those that rely on models or simulations of the system for the analysis. In the first case, the costs remain high due to the need for testing sessions with real users of the system to be carried out. Moreover, and given the high costs of user testing, the analysis will not be exhaustive in terms of all the possible interactions between the users and the system. This means that problems with the implementation might remain unnoticed during the analysis. In the second case, an assumption is being made that the implementation will be faithful to the model. This begs the question of how to test the implementation (ideally, without resorting to human users).

The ability to automatically generate, and run, relevant test cases on a target GUI would support the analysis of the implementation while reducing costs. The problem, then, is that while there are several tools for GUI testing, many such tools do not automate the generation of test cases and/or the testing process.

EICS'11, June 13-16, 2011, Pisa, Italy.

Model-based testing methods automate the generation of test cases from a model of the system under test. However, these methods present several difficulties. In the particular case of interactive systems, one such problem is the need to build detailed models of the GUIs [3]. One way to overcome these difficulties is to increase the level of abstraction of the models. In [4], it is shown how task models can be used to achieve this goal of using more abstract models in the model based testing of GUIs. Task models, however, describe the normative operation of a system only. They do not capture the common mistakes that users might make, or alternatives to the expected normative usage.

This paper focuses on user errors, and examines the feasibility of using task models to test GUIs against erroneous user behaviour in a model-based testing setting. It achieves this by building on the existing approach described in [4]. The approach uses ConcurTaskTrees (CTT) [5] as the task modelling notation. This paper proposes an algorithm to carry out changes to the original models (mutants) by introducing typical user errors. For a definition of this algorithm Reason's user errors classification [6] was used. Then several existing applications were analyzed in order to detect patterns in the construction of their task models. The proposed algorithm detects those patterns in the task model and provides a strategy for generating mutants capturing the effect of the different types of errors on them.

To validate the approach, the CMTTool (CTT Model Transformation Tool) was developed. This tool takes a task model and applies the algorithm defined in the approach, thus generating several mutants of the task model for testing purposes. To assess the quality of the generated models, the approach was applied in the model-based testing of a number of GUI applications. The results obtained by analyzing these case studies showed that the approach allowed the detection of faults arising from unexpected behaviours of the users. This shows evidence that the approach supports the inclusion of typical erroneous user behaviour in the automated task models based testing of user interfaces.

In the next four sections, the paper discusses task models and model-based GUI testing (State of the Art); presents the proposed approach and associated tool support (The Approach and the Tool); presents a number of examples of application of the tool to real GUIs (Case Studies); and ends with a discussion of the results and pointers for future and ongoing work (Conclusions and Future Work).

STATE OF THE ART

Task Models

In the context of interactive systems development, a task is an activity that should be performed in order to reach a particular goal. Used as a requirements analysis artefact, task models capture knowledge about the work the system to be developed will be supporting. Used as a design artefact, task models are a representation of the system's interactive layer logic, and describe assumptions about how the user will interact with the device. In any case, task models are usually normative. They describe the *correct* procedures users (should) follow to achieve defined goals in the system.

Several task-modelling languages have been proposed over the years. Some relevant examples include GOMS (Goals, Operators, Methods, Selection Rules) [7], UAN (User Action Notation) [8], TKS (Task Knowledge Structures) [8], or CTT (ConcurTaskTrees) [9]. These are all examples of the family of hierarchical task analysis notations, the most common approach to task analysis. In this style of approach, the task model is a hierarchical decomposition of tasks into sub-tasks that must be carried out to achieve a given goal. For a discussion of alternative approaches see, for example, [7].

GOMS focus is on user behaviour. The actions users perform on the interface, and how they select which actions to use. TKS focus on the knowledge needed to use the system. UAN and CTT describe both user and system actions. UAN defines a language to describe user actions at the level of mouse and keyboard events. Tasks are described in a tabular notation relating user actions to system responses and user interface states. CTT defines a language to describe the temporal relationships between tasks (based on the LOTOS specification language [10]). The notation does not fix the level of abstraction used to model atomic tasks.

CTT has become a popular language for task modelling and analysis, due to its graphical notation, formal semantics and tool support. The TERESA tool [9] supports editing and analysis of CTT models, and a number of features relating to the animation of task models that have proven useful in our work.

Model Based GUI Testing Tools

Model Based Testing (MBT) has been widely investigated for API testing (e.g., [11-12]), and therefore MBT based approaches are more common for API than for GUI testing. However, approaches applying MBT for GUI testing do exist, e.g., Memon's work [13-14], and Paiva's work [15]. They differ in the kind of model they use and in the coverage of the test criteria used to guide the test case generation process. However, both authors have concerns with the effort required to construct the models.

The tool developed by Memon (GUITAR) generates test cases from an Event Flow Graph (EFG) model. In the EFG, a directed edge from one node to another represents an event-flow relationship between two events. Memon tried to diminish the effort in constructing the model by developing a GUI ripping tool to extract EFG from an existing GUI [14].

In his following work [13], Memon generates a sub-graph of the EFG by removing nodes and edges that are not observed in the usage information obtained from the application's real users, and augments it with probabilistic

information in each node (event) that describes the occurrence probability of the event. Test cases are then generated taking into account the probability of the events occurring.

Another problem faced when considering model-based GUI testing related to the mapping between events in the model, and physical actions in the GUI. The GUI Mapping tool developed by Paiva [15] is an extension of the model-based testing tool Spec Explorer, developed by Microsoft Research. The GUI model is written in Spec# with state variables to model the state of the GUI and methods to model the user actions on the GUI. Spec Explorer generates a finite-state machine (FSM) by exploration of the Spec# model and then test cases are generated from the FSM according to coverage criteria like full transition coverage.

To run tests automatically over a GUI some additional (intermediate) code is needed to simulate the user actions on interactive GUI controls. The GUI Mapping Tool generates such code automatically, based on the mapping between model actions and GUI controls where corresponding real actions occur. Although the intermediate code is generated automatically, Paiva agrees that the effort needed for the construction of Spec# GUI models is too high. Similarly to Memon, she also tried to diminish the effort in constructing a model by using a reverse engineering process to extract a preliminary model from an executable GUI. This model is completed afterwards and validated in order to generate test cases. Another attempt to reduce the time spent with GUI model construction was described in [16] where a visual notation (VAN4GUIM) is designed and translated to Spec# automatically. The aim was to have a visual front-end that could hide formalism details from testers. However, a VAN4GUIM model is in an abstract level lower than task models. Another attempt to reduce the effort in constructing GUI models is to increase the level of reuse. In [17], Cunha tried to increase reuse by identifying recurrent GUI behaviour (UI patterns) and defining test strategies for each of those patterns.

Following on from [4], the approach described in this paper addresses the issue of diminishing the effort in the construction of GUI models by increasing the level of abstraction of those models to task models.

THE APPROACH AND THE TOOL

CTT Task Models

A task model in CTT is a tree of nodes, where the goal is at the root of the tree and leaves are atomic tasks. Temporal operators relate adjacent pairs of nodes at the same level in the tree.

The Case Studies section presents several examples of CTT task models. Four different types of tasks can be identified in those examples: interaction tasks () are atomic tasks representing user input to the application; application tasks () are atomic tasks representing application output to the user; user tasks () are atomic tasks representing decision points on the user's part; abstract tasks () are used to structure the model and appear as internal nodes in the tree.

The semantics of the model is defined by the possible traversals of the tree. Tree traversal is done left to right in a depth first fashion, and is governed by the temporal operators relating pairs of nodes (plus two additional operators that are applied to single nodes – see below). A total of eight operators can be used [5]:

- choice operator ([]): T1 [] T2 means that one of T1 and T2 will happen;
- order independency operator (|=|): T1 |=| T2 means that T1 and T2 will happen in any order;
- concurrent operator (|||): T1 ||| T2 means that T1 and T2 will happen concurrently (the operator |[]| is used to express information exchange between the tasks)
- disabling operator ([>): T1 [> T2 means that T2 interrupts T1 (which will not be resumed);
- suspend/resume operator (|>): T1 |> T2 means that T2 suspends T1, but T1 resumes once T2 has finished;
- enabling operator (>>): T1 >> T2 means that T2 happens after T1 is finished ([]>> is used to express information exchange between the tasks);
- iterative operator (*): T1* means task T1 happens repeatedly;
- optional operator ([ ]): [T1] means task T1 might happen or not.

Consider the task model in Figure 15 (the last in the paper). The goal of the task (the root of the tree) is to start the Unit Converter (“Start UnitConverter”). To achieve the goal the user starts by opening the unit converter (“Press OpenUnitConverter”), after which (enabling operator) the system responds by showing it (“Show UnitConverter”). After the unit converter is displayed, repeatedly (iterative operator applied to “AreaConvert”) the user enters digits (“Enter Digit”) and the system responds with displaying results (“Display Results”). Information about the digits entered in “Enter Digit” is passed to “Display results” (enabling operator with information exchange).

As mentioned above, CTT does not define the level of abstraction for writing the atomic tasks in the models. Neither does it constrain how tasks are to be named. In order to automatically generate oracles from the task models, some conventions about how CTT atomic tasks should be named were defined in [4]. More specifically, a set of valid keywords to be used when writing atomic task names was defined. These keywords are also used here and are:

- Start <*task*> — defines the start of a new task (and creates a new namespace);

- Enter <*field*> <*value*> [<*type*>] — the user enters *value* of type *type* in *field* (String is the default type and 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 *window* as a non-modal window;
- ShowM <*window*> — the application opens *window* as a modal window (i.e., it must be closed before the user can interact again with the parent 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.

These keywords were inspired by the Framework for Integrated Tests (FIT) [6]. Describing the process that lead to this specific set of keywords is outside the scope of this paper. The process is described in [4].

Typical mistakes of the user

According to Reason, in [18], the cognitive process of performing tasks is divided into three stages. The first stage consists in planning. During this stage the objective of the task, and the sequence of actions to achieve that objective, the plan, are identified. The second stage consists in storing the plan in memory until it is executed. The third stage involves implementing the plan (implementation of agreed actions).

During this cognitive process errors may arise, associated with each stage. In [18], three types of user errors are identified: slips, lapses and mistakes. The errors of type slips correspond to the implementation stage of the cognitive process, and consist in the wrong execution of an action, e.g., the user performs the sequence of actions in the wrong order. Lapses are errors that occur during storage and consist of the incorrect omission of an action, e.g., the user forgets to perform one action. Mistakes are a type of errors occurring in the planning phase and are the establishment of a wrong plan to achieve the objective, i.e., the plan chosen for achieving the objective is not adequate.

The first two types of errors (slips and lapses) can be represented in the task model by the omission of tasks, changes in the operators, changes in the order of the tasks or combinations of these approaches. The third type of errors (mistakes) can be represented, using the elaboration of different strategies to achieve the objective. Each strategy corresponds to a different task model, thereby checking which is the strategy followed by the application under test.

Methodology

The task-based MBT methodology proposed in this paper takes the above types of errors into consideration and comprises five main steps (Figure 1).

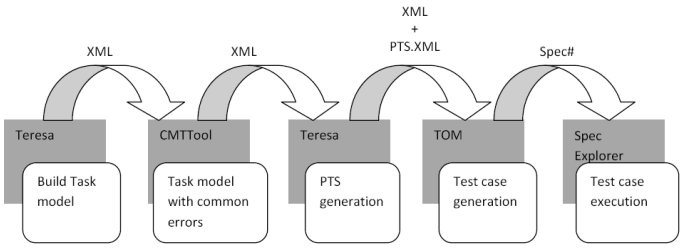

Figure 1: Methodology

The first step corresponds to the design of the task model using the CTT notation. This model is exported to XML file, by using the Teresa tool [19]. The second step is the introduction of typical user errors in the original model producing model mutants. This step is carried out by the CMTTool developed in this work. CMTTool will take the original task model and perform various transformations, constructing new XML files. The mutated models allow testing the GUI against errors, such as slips and lapses. For each mutant, the corresponding finite state machine (PTS – Presentation Task Set) will be generated and exported to XML, using the Teresa tool [19]. In the case of errors of type mistake, several task models will be developed, each corresponding to a different strategy to achieve the goal. In this case, it is not necessary to construct mutants with the CMTTool, jumping directly to the PTS generation step for each of those models. The fourth step generates a Spec# test oracle from the models, mutants, and their PTSs. This generation is automated, using the TOM tool [4, 20]. Finally, Spec Explorer [21] will generate test cases (according to coverage criteria) from the test oracles constructed previously and will execute them.

Transformation algorithm

An algorithm was designed to introduce user errors, such as slips and lapses, in the original task model, i.e., to construct model mutants. This algorithm was designed so that it could work for any task model without depending on the specific GUI that was being modelled.

The first step was to study task models and the CTT operators described above, and define strategies to introduce typical user errors in such models. The goal is not to cover all possible mutations but only those that reflect typical user errors. This has the advantages of focusing the testing activity, helping control scalability problems. Currently, the approach considers leaf tasks only, but it can be extended to consider tasks at other levels in the task tree. The strategies are described in the sequel.

Sequence of tasks

Two interaction tasks, T1 and T2, defined in sequence related by CTT operators such as >> or []>>, have to be executed in the order they are defined. In these situations, it

is interesting to test if the execution order is indeed relevant because, if the GUI allows it, the user may interact with it performing those tasks in the wrong sequence. So, the algorithm generates a mutant with those tasks in an opposite order.

In the case of two tasks related by the operator []>>, e.g., T1[]>>T2, where there is passage of information from T1 to T2, the algorithm changes the order of the tasks and deletes the pre-condition of T2 if it depends on the information passed between them. Otherwise, it would be impossible to generate afterwards test cases with the execution of T2 before executing T1.

When the last task of a sequence of tasks is an application task, the algorithm does not change its order because it will be a task performed by the application as a result of executing the task sequence and will be used to check if the result obtained by executing the tasks is the one expected.

Non-mandatory order

A sequence of tasks with no mandatory order is a set of tasks separated by operator |=|. In this case, what matters is to test whether, in fact, tasks can be performed in any order. So, it would be necessary to test if the result obtained by executing T1>>T2 is the equal to the one obtained by executing T2>>T1. Excluding application tasks, the algorithm generates several mutants with all possible combinations of task orders and replaces the operator |=| by the sequence operator >>.

Optional/mandatory task

Sometimes, the users forget to execute one of the tasks needed to achieve a goal. In a CTT model it is possible to distinguish between optional tasks (within brackets) and mandatory tasks (without brackets). It may be useful to test if optional tasks are effectively optional and if the mandatory tasks are effectively mandatory.

For a sequence of tasks T1>>[T2]>>T3 in which T2 is optional, the algorithm generates four mutants with the following sequence of tasks:

1. T1>>T2>>T3,
2. T1>>T3 to check if the T2 is really optional,
3. T1>>T2 to check if T3 is really mandatory,
4. T2>>T3 to check if T1 is really mandatory.

In the case of a mandatory task, mutants are generated with the task removed. In the case of a sequence of tasks T1>>T2[>T3 where T2 executes until T3 starts, the mutant omitting T2 will be T1>>T3.

If the operator []>> is used between two tasks, e.g., T1>>T2[]>>T3, when the tool generates the mutant T1>>T3 it deletes the pre-condition of T3 if it depends on the information passed by T2 to T3.

Task choice

When a sequence of tasks is separated by the choice operator [], e.g., T1[]T2[]T3, it means that the user can choose to perform one of those tasks. For each set of tasks separated by [] operator, the algorithm generates mutants, keeping one of those tasks and omitting the other ones. The result of executing the test cases generated will say if the set of tasks are really a choice. For the sequence T1[]T2[]T3, the algorithm generates 3 mutants: one with task T1, one with task T2 and another with task T3.

Disabling

T1[>T2 means that task T1 is active until T2 is performed, and that at any time during the execution of T1, T2 can be performed. This can also lead to errors when the user attempts to perform T1 after performing T2. Thus it becomes necessary to test if T1 is really disabled after performing T2. The tool generates a mutant with the following sequence of tasks T1>>T2>>T1', where T1' is a copy of the task T1 to check if the execution of T2 disables T1.

Iterative task

A task followed by *, e.g., T*, means that T can be executed iteratively. When an iterative task has a sequence of subtasks ending with a task of type "Press" and all the other tasks are interaction tasks of the type "Enter", the model may describe a form filling interface. A behaviour that may be useful to test may be to check if between following iterations, the previous inserted information is kept or is throw way. A typical user error is to forget to fill out a required field and fill only that field in the second iteration assuming the all the other information is kept filled. However, not all interfaces record information from one to the following iteration, so, most of the times, the user has to re-fill all fields. One way to simulate these errors is to omit a mandatory task in the first iteration, and perform only that task in the next iteration, thus checking if the information was recorded between iterations.

Test case generation and execution

The tool TOM [4, 20], generates test oracles in Spec# [22] transforming the atomic tasks of the task model in model actions. Afterwards, these oracles are used by the Spec Explorer tool [21] for the generation and execution of tests.

In Spec#, actions can be of several types: controllable, probe, observable and scenario. CTT interaction tasks correspond to controllable actions in Spec#, because they describe user actions. CTT application tasks correspond to probe actions in Spec# because they represent actions that only read the system state without updating it. These tasks allow checking if the application is in the desirable state at a certain time.

For test case generation, the Spec Explorer tool [21] allows the definition of the domain values for the actions' parameters. With this information, Spec Explorer generates a finite state machine (FSM) by exploring the Spec# model and afterwards generates test cases according to a coverage criterion. To execute the test cases, a mapping is needed between actions of the model and methods of an adapter code that will simulate those actions on the GUI under test. After establishing this map, the tests can be executed

automatically by Spec Explorer and the GUI Mapping Tool [15] after which a test report is generated.

CASE STUDIES

Several case studies were conducted over some existing web applications in order to evaluate the approach. Each case study was used to test some of the typical errors of the user, based on the test strategies proposed. The selected applications were:

- an online hotel reservation system (*Online Vip Hotels reservation*);
- the houses' search menu of a real estate agent (*Search houses*);
- the login page of a wiki system (*DokuWiki*);
- a currency converter (*Currency convertor*); and
- unit converter (*online converter*).

Online Vip Hotels reservation

VIP Hotels Group owns a chain of hotels in various regions. Through his address, www.viphotels.com, you can access various features. The online booking functionality allows searching available rooms in hotels of the group for the selected dates and afterwards the booking may be performed. Only the search functionality was tested in this case study. Figure 2 shows the menu of online reservations.

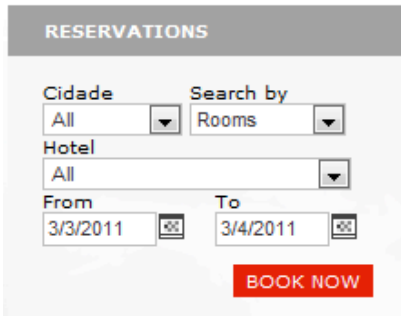

Figure 2: Vip Hotels online reservation.

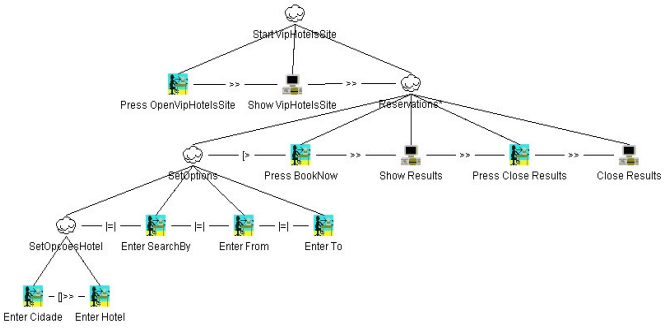

Figure 3: Task model of the Vip Hotels online reservation menu

As can be seen by examining the model in Figure 3, after opening the Vip Hotels group page, the Hotel Reservations menu is immediately available. The menu options can be chosen in any order. However, the city (*Cidade*) must be selected before the hotel. This happens because the list of hotels to choose depends on the chosen city (there is a dependency between those tasks because the first one passes information to the second). After filling in the requested information, it is possible to see, in a new window, the search results (rooms available) by performing the task "Press BookNow". The CMTTool is used to generate mutants according to the predefined algorithm. One of the mutants generated for this case study will be analyzed in the sequel.

A typical user error is to exchange the order of a sequence of tasks. One of the mutants generated exchanged the order of "Enter Cidade" and "Enter Hotel" tasks (Figure 4).

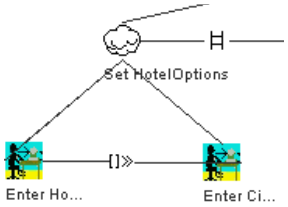

Figure 4: Detail of the mutated model

The TERESA tool generated the PTS and afterwards, the TOM tool generated the Spec# model. Test cases were generated and executed with Spec Explorer. The input values used were:

- City (*Cidade*) = Lisboa;
- Hotel = Vip Inn Berna;
- Search by = Rooms;
- From = 2/09/2010;
- To = 30/09/2010.

The Vip Hotels online reservation allows choosing the hotel first and then the city however, after selecting the hotel and afterwards the city, the value of the hotel changes to its default value which is "All". Then, when performing "Press BookNow" the results obtained are different from the ones obtained by executing the tasks in their original order. To avoid this kind of user error, the interface should be made less flexible, allowing setting the value of the Hotel only after setting the city (*Cidade*) value.

Search houses

The website of Agimoura (www.agimoura.com) has a set of features to buy, sell, search houses, among others. In this case study, only the quick search feature properties were analyzed in greater detail. It is a simple search for which information must be provided, such as: the type of housing, the type of information to configure the search (e.g., County, Place, Price, Reference, Type, etc.) and details about the information selected to configure the search (Figure 5).

Figure 5: Search houses menu.

One typical error that may occur is the case where the user forgets to perform one of the mandatory tasks, e.g., filling in one of the search options (“SetOpcoes”) (Figure 6). To describe this situation, the algorithm generates a mutant with the task “Enter Info” omitted.

When executing the test cases generated by spec Explorer, for the above mutation, it was not possible to press the search button at the end because this button was disabled. So, it is possible to conclude that it is really mandatory to fill in the “Enter Info” field.

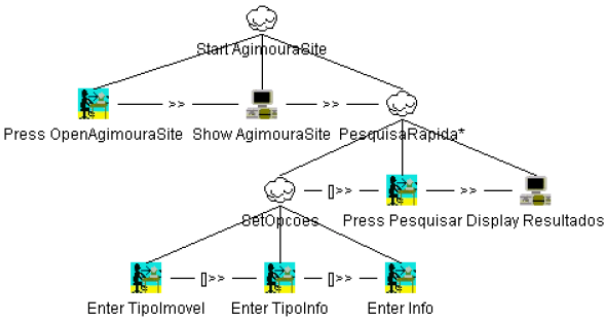

Figure 6: Task model of the Agimoura website

One interesting point to notice here is that, when testing GUIs with mutated models, such as this one, a failing test does not necessarily indicates a problem. It might instead be showing that the user interface prevents erroneous user actions from occurring.

DukuWiki

This case study aimed to test the Login/Password dialog (Figure 7) of the wiki-type DokuWiki.

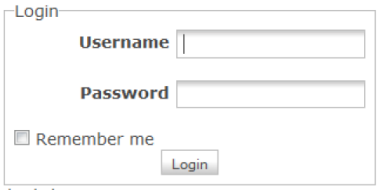

Figure 7: Login of the DokuWiki

As can be seen in the task model of Figure 8, there is an iterative task in the model: “Login”. The goal was to test if, in following iterations, the inserted information is saved. To test this situation, the algorithm generates several mutated models, e.g., Figure 9.

As can be seen in the model of Figure 9, the task "Enter Password" was omitted from the first iteration of the cycle and executed in the second iteration, in order to check if the login information is kept between successive iterations. To perform this check it was necessary to create two types of probe actions. The first one (“Display LoginResults”) checks the failure of the sign-in process after the first iteration. This should happen because the password field is not filled. The second one (“Display LoginResults_Copy”) verifies the success of the sign-in process at the end of the second iteration, indicating whether or not there was recording of information. After executing the generated test cases, it was possible to conclude that the login information is recorded between successive iterations.

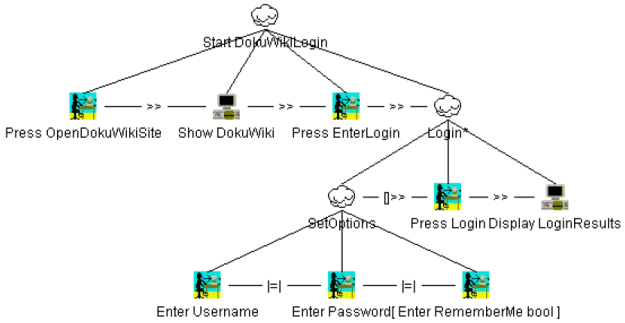

Figure 8: Task model of the login

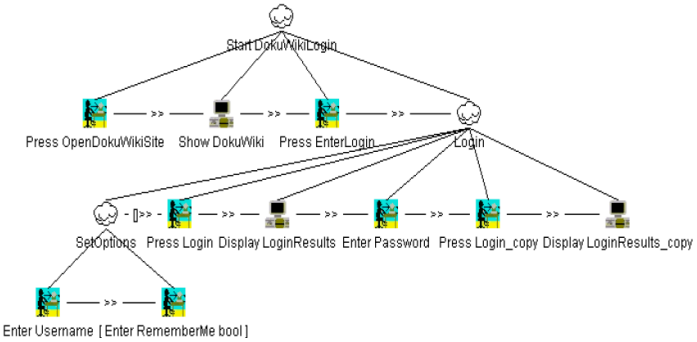

Figure 9: Mutated task model: iterations

Currency converter

The XE Converter is a universal currency converter to convert an amount from one currency to another. This application can be accessed at the following URL: www.xe.com/ucc/. Figure 10 shows the menu of this application.

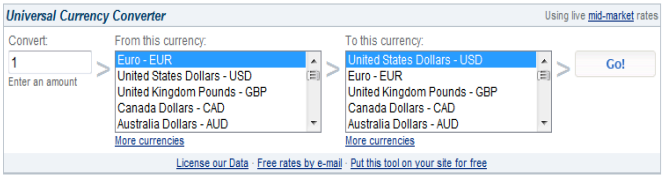

Figure 10: XE converter

As can be seen in the model in Figure 11, after accessing the converter, it is possible to select the conversion options you want (task “SetOptions”). The sub-tasks within “SetOptions” can be performed in any order until “Press Go!” is performed. Thereafter, the previous tasks are no longer active, they are disabled ([> operator), and the result of the conversion is displayed. The algorithm generates several mutants of the model.

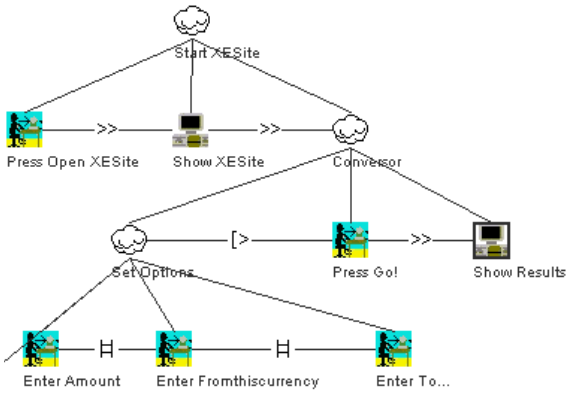

Figure 11: Task model of the XE converter

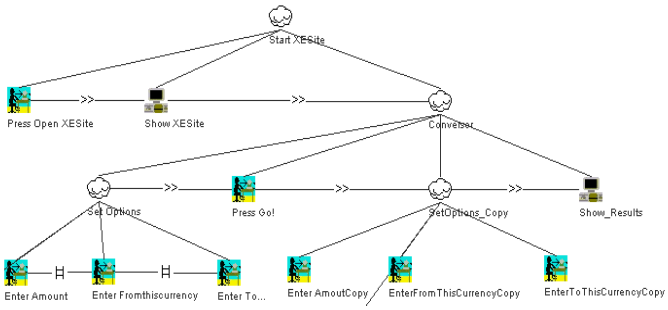

Figure 12: Mutated task model: disabling tasks

The one that checks if the tasks are really disabled is shown in Figure 12. As can be seen in the Figure 12, the task "SetOptions" is repeated (with the name "SetOptions_Copy") after the "Press Go!" task. The input values used to execute the test cases were:

- amount: 1;
- from this currency: Euro - EUR;
- to this currency: United States Dollars - USD.

During the execution of the test cases, it was possible to verify that the tasks within "SetOptions_Copy" were not available.

Online converter

The online converter (see the form depicted in Figure 13) (www.peters1.dk/webtools/conversion.php?sprog=en) was used to test the approach regarding mistakes, i.e., the use of wrong plans to achieve a goal. Two different plans were built: Figure 14 and Figure 15.

The plan described by Figure 14 models entering several digits at the input value field ("Enter Digit"), and at the end "Press Submit" to see the conversion result.

The plan described in Figure 15 models entering one digit at a time, and seeing the result after each input digit without the need to perform a specific action for that. That is, it models a user that is assuming the system will react to each digit press by updating the result values.

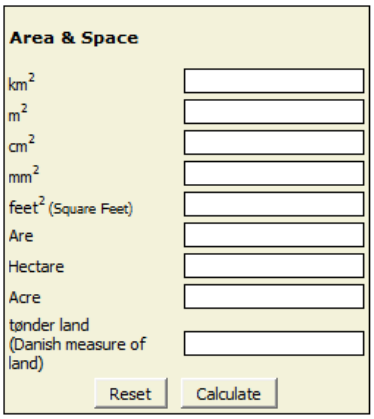

Figure 13: Online converter

Both plans were tested against the online converter application. Results showed that Plan B is not supported by the software application. This happens because the result values are updated only after the Calculate button is pressed, and not after entering each input digit as was expected.

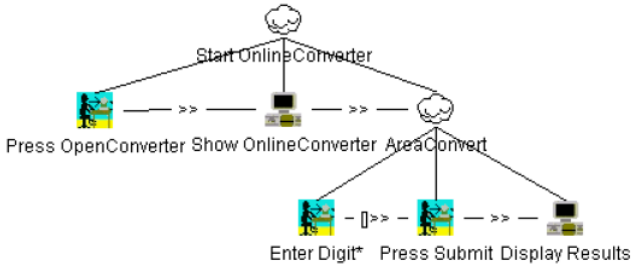

Figure 14: Plan A to convert units

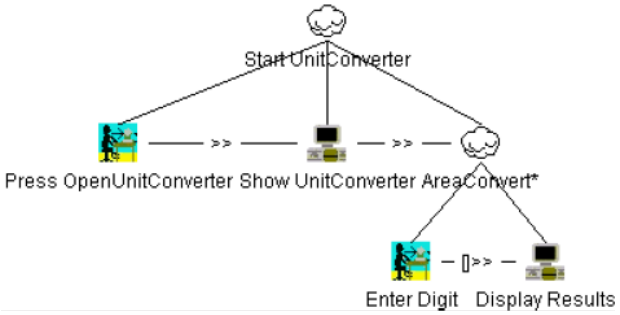

Figure 15: Plan B to convert units

CONCLUSIONS AND FUTURE WORK

Model-based testing automates the generation and execution of test cases from a model of the system under test. However, the need to build detailed models of the systems under test creates a barrier for the adoption of this testing approach. Previous work has looked at the possibility of using task models in the model based testing of GUIs [4]. Task models, however, describe the normative operation of a system only. They do not capture the common errors that users might make, or alternatives to the

expected normative usage they might consider. This was initially addressed by manually introducing changes in the task models (i.e., creating mutations of the task models). This paper builds on this previous work by developing an automated approach to the generation of these mutations.

Using a number of cases studies, the paper shows how it is possible to systematically introduce changes in a task model in order to generate behaviours corresponding to errors users typically make. This mutation process is supported by the CMTTool, which was built specifically to the effect. Together with the TOM tool, it supports a process of model-based testing of GUIs from task models, where the test oracles focus on expected user behaviour, both correct (via the original task model), and erroneous (via the task's mutations). Hence, together the two tools increase the degree of coverage of the more likely user behaviours of the model based testing approach.

The user interfaces used in the case studies were small and targeted at experimenting with different types of tasks. As such, they did not bring into light scalability problems. Investigating scalability is an area for future work. Nevertheless, for large models where this problem may be visible, the approach can always be applied to sub-models of the original one.

The CTT language is currently a popular task modelling language due to its tool support and precise semantics. As such, it should present a low barrier for the use of the approach. One drawback of the current status of the approach however, is the number of different tools that must be brought together for its application. To solve this, we will have to bring all the tools together in a single deliverable. Now that all the pieces are in place, the process should be made easier.

The GUI Mapping tool, in particular, uses proprietary software libraries and cannot be made available. We are working on an alternative tool based on an open library.

While we believe that the mutations being used enable us to capture relevant user interface problems, we do not yet have the data to back this claim. To this end, we plan to carry out a study comparing the results of applying the approach against problems actually felt by users. This will enable us to assess the quality of the mutations by determining the degree of coverage achieved. Once user data is available, the fact that the testing approach is automated means we will be able to experiment with different mutations to determine the best set of mutations, and further explore scalability issues.

At a more technical level, an area where work is being carried out is in automating the generation of the input values to be used during the testing process. To that end, we are studying the possibility of integrating formal models of the application into the task model, in order to be able to rigorously analyse the input data types, partitioning them into equivalence classes.

ACKNOWLEDGMENTS

The authors wish to thank José Luís Silva, the author of the TOM tool, for making the tool available, and for his support with using it.

José Campos acknowledges support from the CROSS project (An Infrastructure for Certification and Re-engineering of Open Source Software), funded by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-010049.

REFERENCES

1. ISO 9241-11. Ergonomic requirements for office work with visual display terminals (VDTs) -- Part 11: Guidance on usability. First Edition ed. 1998: International Organization for Standardization.
2. Ivory, M.Y. and M.A. Hearst, The State of the Art in Automating Usability Evaluation of User Interfaces. ACM Computing Surveys, 2001. 33(4): p. 470-516.
3. Paiva, A.C.R., Automated Specification-Based Testing of Graphical User Interfaces, in Department of Electrical and Computer Engineering. 2007, Engineering Faculty of Porto University (Ph.D thesis): Porto. p. 228.
4. Silva, J.L., J.C. Campos, and A.C.R. Paiva. Model-based user interface testing with Spec Explorer and ConcurTaskTrees. in 2nd International Workshop on Formal Methods for Interactive Systems. 2007. Lancaster, UK.
5. Paternò, F., Model-Based Design and Evaluation of Interactive Applications. 1999, London, UK: Springer-Verlag.
6. Mugridge, R. and W. Cunningham, Fit for Developing Software: Framework for Integrated Tests. 1st Edition ed. 2005: Prentice Hall. 384.
7. Card, S.K., T.P. Moran, and A. Newell, The Psychology of Human-Computer Interaction 1986: Lawrence Erlbaum Associates. 469.
8. Hamilton, F., Predictive evaluation using task knowledge structures, in Conference companion on Human factors in computing systems: common ground. 1996, ACM: Vancouver, British Columbia, Canada. p. 261-262.
9. Mori, G., F. Paternò, and C. Santoro, CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. IEEE Transactions on Software Engineering, 2002. 28(9).
10. Bolognesi, T. and E. Brinksma, Introduction to the ISO specification language LOTOS. Computer Networks

and ISDN Systems - Special Issue: Protocol Specification and Testing, 1987. 14(1).

11. Hartman, A. and K. Nagin. The AGEDIS Tools for Model Based Testing. in ISSTA'04. 2004. Boston, Massachusetts, USA: Springer.
12. Jacky, J., et al., Model-Based Software Testing and Analysis with C#. 2007: Cambridge University Press. 366.
13. Brooks, P.A. and A.M. Memon, Automated GUI testing guided by usage profiles, in Proceedings of the 22nd IEEE international conference on Automated software engineering (ASE'07). 2007, IEEE CS: Washington, DC, USA. p. 333-342.
14. Memon, A., I. Banerjee, and A. Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. in Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03). 2003. Washington, DC, USA: IEEE CS.
15. Paiva, A.C.R., et al. A Model-to-implementation Mapping Tool for Automated Model-based GUI Testing. in Proceedings of the 7th International Conferece on Formal Engineering Methods (ICFEM'05). 2005.
16. Moreira, R.M.L.M. and A.C.R. Paiva, Visual Abstract Notation for GUI Modelling and Testing: VAN4GUIM, in Proceedings of the 3rd International Conference on Software and Data Technologies (ICSOFT'08), J. Cordeiro, et al., Editors. 2008, INSTICC Press: Gaia, Portugal.
17. Cunha, M., et al., PETTool: A Pattern-Based GUI Testing Tool, in 2nd International Conference on Software Technology and Engineering (ICSTE'10). 2010. p. 202-206.
18. Reason, J., Human Error. 1990: Cambridge University Press.
19. Paternò F. Santoro C. Mäntyjärvi J., Mori G., Sansone S., Authoring pervasive multimodal user interfaces, International Journal of Web Engineering and Technology, vol. 4 pp. 235 - 261. Inderscience Enterprises Ltd, 2008.
20. Campos, J.C., J.L. Silva, and A.C.R. Paiva, Task models in the model-based testing of user interfaces. Technical Report, 2009, Universidade do Minho.
21. Veans, M., et al., Model-based testing of object-oriented reactive systems with Spec Explorer, in Formal Methods and testing: an outcome of the FORTEST network. 2008, Springer-Verlag. p. 39-76.
22. Barnett, M., K.R.M. Leino, and W. Schulte. The Spec# Programming System: An Overview. in CASSIS'04 - International workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices. 2004. Marseille.