Systematic Automation of Scenario-Based Testing of User Interfaces

José C. Campos[1], Camille Fayollas[2], Célia Martinie[2], David Navarre[2], Philippe Palanque[2], Miguel Pinto[1]
[1]Universidade do Minho & HASLab/INESC TEC, Braga, Portugal
jose.campos@di.uminho.pt, mcpinto98@gmail.com
[2]ICS-IRIT, University of Toulouse, Toulouse, France
{fayollas, martinie, palanque, navarre}@irit.fr

ABSTRACT
Ensuring the effectiveness factor of usability consists in ensuring that the application allows users to reach their goals and perform their tasks. One of the few means for reaching this goal relies on task analysis and proving the compatibility between the interactive application and its task models. Synergistic execution enables the validation of a system against its task model by co-executing the system and the task model and comparing the behavior of the system against what is prescribed in the model. This allows a tester to explore scenarios in order to detect deviations between the two behaviors. Manual exploration of scenarios does not guarantee a good coverage of the analysis. To address this, we resort to model-based testing (MBT) techniques to automatically generate scenarios for automated synergistic execution. To achieve this, we generate, from the task model, scenarios to be co-executed over the task model and the system. During this generation step we explore the possibility of including considerations about user error in the analysis. The automation of the execution of the scenarios closes the process. We illustrate the approach with an example.

Author Keywords
Interactive systems, task models, model-based testing

ACM Classification Keywords
D.2.2 [Software] Design Tools and Techniques – *User interfaces & Computer-aided software engineering (CASE).*

INTRODUCTION
The adoption of interactive computing systems in safety and mission critical domains is increasing. Airplane cockpits and medical devices are two examples where user interfaces are becoming increasingly computer based. The design of these interfaces, then, must be addressed having in mind that failures might have unacceptable costs. Tools are needed that, as much as possible, support automated analytical analyses of the user interfaces of systems in order to guarantee systematic and repeatable analysis.

In what follows we are particularly interested in analysing the effectiveness of user interfaces (*c.f.,* the definition of usability in the ISO 9241-11 standard [8]). We argue that, when taking all goals of a particular user with a particular system into account, effectiveness is a required (even if not sufficient) condition to achieve efficiency and satisfaction, and hence, usability. We will show how effectiveness can be analyzed analytically.

In order to assess effectiveness, what is needed is a description of the goals and how the user is expected to accomplish them in the system. This information can be captured in a task model. Then, by determining the compatibility of the system (design) with the task model, it becomes possible to assess effectiveness.

The approach presented in [7] enables the interactive checking of the compatibility of a task model with an application by performing co-execution. This approach has the advantage that it enables the exploration of the design, but the fact that the co-execution is performed manually means that the analysis cannot be exhaustive, except for the simplest cases. In order to address this, the co-execution needs to be automated. This implies both support to replay scenarios and the automatic generation of relevant scenarios for co-execution.

To achieve this, in this paper we resort to model-based testing techniques to generate the scenarios for automatic exploration. The proposed approach uses the task models as input to generate both scenarios that comply with the behavior prescribed by the task model, and scenarios that incorporate possible erroneous behavior as deviations from the normative behavior prescribed in the model. By feeding back the scenarios for co-execution, it becomes possible to assess the degree of support of the interactive system to the task model.

By combining these two approaches for the Systematic Automation of Scenario-Based Testing of User Interfaces, this paper presents two major contributions:

1) An approach for ensuring the effectiveness of an interactive application through: i) the automated generation of test campaigns based on scenarios; ii) the automated

EICS'16, June 21-24, 2016, Brussels, Belgium
 ISBN 978-1-4503-4322-0/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2933242.2933256

testing of the application consistency with these scenarios; thus assessing if the application enables the user to achieve its goals.

2) An approach for ensuring task-application compatibility through: i) the automated mutation of scenarios, generating negative test cases; ii) the automated testing of the application consistency with these scenarios; thus assessing if the application enables more behaviors than the one described by the task models (e.g. allowing actions performed due to human error).

The reminder of the paper is structured as follows. The next section provides a quick overview of related work on task-application compatibility and Model-Based Testing of GUIs. The third section describes a stepwise process to ensure both efficiency and task-application compatibility of interactive systems. The fourth section presents the tools that support the proposed process. The fifth section presents the application of the approach on an illustrative example from airplane cockpits. The two last sections conclude this paper, making explicit its benefits and limitations and highlighting future work.

RELATED WORK

Ensuring Task-Application Compatibility

There have been mainly three different alternatives to assess compatibility between task models and interactive applications (i.e., ensuring that the application enables the performing of all the tasks describe in the task models): i) generating the application from the task model, ii) defining a correspondence between a model of the application and the task model and iii) coupling task models and interactive applications.

Generation of Application from a Task Model

Many authors (*e.g.*, the work of Manca et al. in [10]) have followed and refined the work of ADEPT [22], assuming that user interface design should be task centered and that it is possible to generate an interactive application from task models (while adding other ingredients such as UI guidelines for instance). The main claim is that such a generation can be done for different platforms thus reducing the development costs. However, the main drawbacks are that it is difficult to integrate design and craft knowledge in such processes ending up with stereotyped user interfaces far away (in terms of design and interaction techniques) from leading edge applications.

Correspondence at models level

In [16], the author promoted that it was possible to integrate task and system models. Approaches such as the one proposed claim a full integration of system and task models, thus enabling the verification of compatibility between them. However, they require a lot of work to guarantee the consistence between task models and system models (as presented in [14] where such compatibility was assessed through scenarios extracted from the task models and executed on the system model). Another drawback is the high development costs for the construction of the application and interaction models; along with the fact that such approaches are very different from current processes in interactive application development (where Rapid Application Developments toolkits are common practice); thus limiting usually their use to safety critical applications.

Task Model and Interactive Application Coupling

Starting from the drawbacks of the two previous aternatives to assess the compatibility between task model and interactive applications, the authors of [11] proposed an approach for coupling tasks models with an existing interactive application (avoiding the need for an application model). This approach enables, through the instrumentation of the existing application and the use of a synergistic module, to co-execute the application and the task model in order to assess their compatibility. While this approach is resolving many drawbacks from the previous ones (e.g. suppressing the work associated with the creation of system models and enabling the use of such approaches for non safety critical applications), some drawbacks are still remaining. The main one is the fact that the co-execution of task models and interactive applications is done manually, thus it does not guarantee a good coverage of the analysis and it is very time consuming. The approach presented in this paper aims at suppressing this drawback. To this end, the proposed approach builds on the work in [11] and aims at bringing to it the benefits of Model-Based Testing in order to enable the automatization of the compatibility testing between ask models and interactive applications.

Model-Based Testing of GUIs

Model-Based Testing (MBT) [21] is a black-box testing technique that aims to verify if a software implementation of a system complies with its specification (or model), focusing on automated test generation. It allows test engineers to get involved early in the development cycle. The basic idea is to use an abstract model representing the system under test (SUT) to generate test cases. These tests can then be run both on the SUT and on the model (the oracle) and their results compared.

The MBT process starts with the construction of an abstract model of the SUT. From this model test cases are then generated that represent how the system should behave. To decide when enough test cases have been generated, coverage criteria over the model can be used. The result of this phase will be sequences of operations expressed over the model and guaranteeing some specified coverage of the model. These abstract test cases need to be transformed into concrete test cases prior to being executed in the SUT. In the next phase the tests are run. When applying MBT to interactive systems, this typically involves instrumentation of the SUT, as programmatic access to the user interface controls is needed in order to both execute the test case and analyze the output to the user. At the end of the process, an analysis of the results is performed, making sure that they are

consistent with the expected results and highlighting any inconsistencies found.

Memon was among the first to apply MBT to graphical user interfaces [13]. He developed the GUITAR GUI testing framework. GUITAR supports the model-based testing of Java applications' GUIs, from the generation of event-based models from source code, to the generation of test cases in the form of GUI event sequences, through to the execution of these test cases on the Java application. Since then, a considerable number of proposals have been put forward (see [9] for a short review).

Several different directions have been explored. One particular direction of work has been concerned with improving the quality of the test cases; for example, through appropriate coverage criteria, or through the reuse of test strategies. An example of the latter is the work by Paiva et al. on Pattern Based GUI Testing (PBGT) [18], promoting the reuse of test strategies to test common behaviors on Web Applications. The PARADIGM language was developed to ease the modeling of the GUI patterns and support the process. Bowen and Reeves explore the generation of abstract tests from GUI design artifacts [3].

Other authors have explored different alternatives to modeling. In order to alleviate the cost of producing models to be used as oracles, Silva et al. [19] proposed the use of task models as oracles. As task models typically represent correct behavior only, later the use of mutations on the task models to enables tests to cover user error was also explored [1]. Lelli et al. [9], focused not on the cost but on the expressiveness of the models proposing a modeling approach able to deal with advanced multi-event GUIs.

Still, other authors focus on the integration of model-based testing in the UI design and development process. Bowen and Reeves explore the applicability of test-first development (an approach similar to test-based development, but using models of the requirements as the basis for the tests) to GUI development [5].

A SYSTEMATIC APPROACH FOR SCENARIO-BASED TESTING OF INTERACTIVE APPLICATIONS

Bringing together the idea of test case generation from models of how the system should behave (task models in particular), and the idea of co-execution of task models and actual systems against usage scenarios, we propose a semi-automated process to analyze task-application compatibility.

As presented in Figure 1, the process assumes a model-based approach to systems development; more specifically, one in which task models of the proposed systems are developed. Hence, the inputs to the process are the implemented interactive application (the SUT) and its associated task models (to be used as the oracle).

The proposed approach is divided in two phases (see Figure 1): the first phase aims at ensuring the effectiveness of the interactive system; the second phase aims at ensuring the compatibility between the interactive application and its corresponding task models.

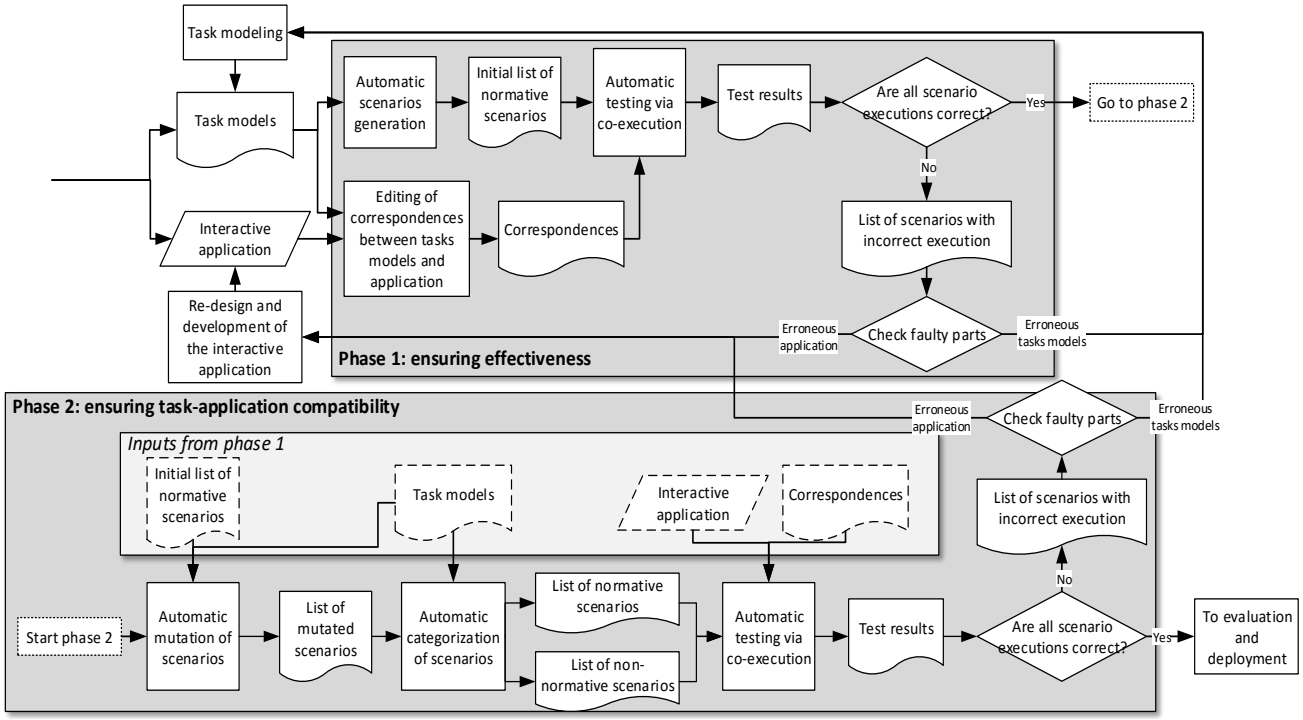

Figure 1. Process for validating the effectiveness of an interactive application and its compatibility with its task model

Phase 1: Ensuring Effectiveness of an Interactive Application

This phase starts with two steps that can be performed concurrently: the scenario generation and the correspondence editing between interactive tasks from the tasks models and their corresponding event sources and renderers in the interactive application.

Scenario generation ("*Automatic scenarios generation*" in Figure 1) leads to the extraction of scenarios from the tasks models ("*Initial list of normative scenarios*" artifact in Figure 1). These scenarios capture concrete sequences of actions to be performed as described by the task model and will be used as test cases.

Since task models are usually employed to capture normative behaviors – *i.e.,* they describe how a system is supposed to be used – the extracted scenarios represent correct user behavior. More than that, they represent how users are expected to use the system.

The scenarios generated from the task models are independent from any particular implementation (they represent *abstract* test cases), so they cannot be directly executed in a particular SUT. Prior to their execution, a mapping between actions in the task models and controls/widgets in the SUT must be defined ("*Editing of correspondences between tasks models and application*" step in Figure 1, explained below). Using this mapping, scenarios are made to represent *concrete* test cases. Once that is done, they can be executed.

The correspondence editing step relies on the approach proposed by [7]. In this step, the developer has to instrument the existing application in order to be able to co-execute it with tasks models at run time. To achieve this goal, the developer has to identify the event sources (list of events related to the different widgets) and renderers (graphical representation of data within widgets) of the application and s/he is then in charge of putting these elements in correspondences with the ones in the tasks models: interactive input tasks may be connected to event sources and interactive output tasks may be connected to renderers. This step is iterative in order to allow the detection of wrong correspondences. This checking can be done by executing (and monitoring the execution) of a set of scenarios representing 100% of the tasks in the task model. This set is usually very limited as valued objects and preconditions do not need to be taken into account.

Once the scenarios are generated and the correspondences edited, an automatic scenario-based testing of the application is done ("*Automatic testing via co-execution*" in phase 1 in Figure 1). This step is achieved through the scenario-driven co-execution of the scenarios and the interactive application where the execution of the system is controlled by a step-by-step execution of scenarios. This step results in a list of scenarios execution results ("*Test results*" artifact in phase 1 frame in Figure 1). A normative scenario execution is considered correct if all of its tasks have been performed successfully on the application. Failure to perform a task can be due to several reasons, as for instance:

- An incompatibility between the value of objects in the interactive task and the system domain value (*e.g.,* one wants to enter the value "3.1", but the widget accepts only integer values);
- An incompatibility between the interactive task and the enabling of widgets (*e.g.*, trying to interact with a disabled widget);
- An incompatibility between the interactive task and the visibility of widgets (*e.g.*, trying to interact with a invisible widget);
- Tasks' preconditions not being met (*i.e.*, the task model defines a precondition which is not met by the application).

The co-execution results must be analysed ("*Are all scenario executions correct*" in phase 1 frame in Figure 1). Test cases (scenarios) generated from the task model represent specific instances of the oracle. Unless the correspondence between the model and the SUT is incorrect, any mismatch (*e.g.,* in the availability, state or value of an interface element) between what is defined in the test case and the SUT can then be considered as SUT errors. Therefore, if the execution of one or more scenarios is not successful, an inconsistency is detected: the application did not allow the completion of one of the tasks specified by the task model; the effectiveness of the application is thus not observed. In this case, the developer has to check whether the error comes from an error within the task model or within the application ("*Check faulty parts*" in phase 1 frame in Figure 1). In the first case, the tasks models need to be amended in order to correct the error (loop back to the step "*Task modeling*" on the left-hand side in Figure 1). In the second case, the application design needs to be amended in order to enable the completion of this task (loop back to the step "*Re-design and development of the interactive application*" on the left-hand side in Figure 1).

When all the scenario executions are correct, the effectiveness of the application is ensured by the fact that the application enables the completion of all the tasks that need to be accomplished by the user; phase 1 is then finished and phase 2 can start ("*Go to phase 2*" in phase 1 frame in Figure 1).

Phase 2: Ensuring Task-Application Compatibility

The inputs to this second phase are of two types:

- First, the same inputs than to phase 1: the tasks models and the interactive application, both of them being corrected by the accomplishment of phase 1 ("*Tasks models*" and "*Interactive application*" in Figure 1);
- Second, coming from the outputs of phase 1: the scenarios that have been generated during that first phase ("*Initial list of normative scenarios*" in Figure 1) and the correspondences between interactive tasks and event sources and renderers ("*Correspondences*" in Figure 1).

Please note that in order to highlight the fact that all of these inputs are coming from the process, we have chosen to represent them a second time, with dotted lines, in Figure 1.

Restricting analysis to the normative scenarios that can be obtained from the task model would weaken the analytic power of the approach. However, considering all possible user behaviors for co-execution would be unfeasible. Hence, to enable the exploration of non-normative behaviors, scenarios are subject to a number of mutations that intend to capture possible user errors as deviations from the norm. The first step of phase 2 consists in generating mutated scenarios ("*Automatic mutation of scenarios*" step in phase 2 frame in Figure 1).

The specific type of mutations to be used is not a prerequisite of the proposed approach and might be influenced by, for example, the application domain. For illustration purposes we follow Reason's [17] classification and consider possible mutations that might be applied on the test cases for the three types of user error: Slips, Lapses and Mistakes. Slips and lapses are skill-based errors where the user's intention is correct but the execution of the action flawed due either to attention (slips) or memory (lapses) failure. Slips might be represented by mutations that change the order of action execution, or the control that is activated. Information about user interface layout will be useful here. To represents lapses, we can introduce mutations that omit or repeat actions. Mistakes are knowledge-based errors. Their impact in the execution of the tasks is more profound as they might imply selecting the wrong strategy (task) to achieve some goal in a particular situation (*e.g.,* due to mode errors). Since in this case the scenarios capture the execution of predefined tasks, they contain no choice steps, nor any information on alternatives. That information is present at the task model only. Hence, mutations to represent mistakes will range from changing the values input by the user, to represent situations where the user chooses the wrong input value for a particular situation, up to replacing whole scenarios, to represent situations where the user chooses the wrong strategy for the goal. While these mutations are by no means exhaustive, they provide a first approach to reason about the impact of user error on the user interface.

Once the mutated scenarios are generated ("*List of mutated scenarios*" in Figure 1), we have to take into account the fact that some of the mutated scenarios might be normative scenarios while the other will be non-normative ones. Therefore, for each mutated scenario, we have to categorize it within these two types, in order to be able to know if their execution on the interactive application must be successful (for the normative ones) or not (for the non-normative ones). This categorization is accomplished through the automatic running of all mutated scenarios on the tasks models ("*Automatic categorization of scenarios*" step in Figure 1). If a scenario can be executed on the tasks models it is normative, otherwise, it is non-normative. This step thus leads to two pools of scenarios: the normative ones and the non-normative ones ("*List of normative scenarios*" and "*List of non-normative scenarios*" in the phase 2 frame in Figure 1).

Once the mutated scenarios have been categorized, the automatic scenario-based testing of the application is performed once again ("*Automatic testing via co-execution*" in phase 2 frame in Figure 1); leading to a list of test cases execution results ("*Test results*" in phase 2 frame in Figure 1).

As for phase 1, the test results need to be analysed ("*Are all scenario executions correct*" in phase 2 frame in Figure 1). In this case the notion of correctness of a scenario execution (a test case) depends on whether it is a normative or a non-normative one. Indeed, unlike for phase 1, mismatches between mutated test cases and the SUT do not necessarily represent an implementation error. In many cases the goal will be that the mutated test case not be accepted by the SUT. To address this distinction, the concept of positive and negative tests must be introduced. Positive tests are those that exercise correct usages of the system. The sequence of actions and the values input are correct and so the SUT should behave according to what is prescribed in the task model. They are typically generated directly from the oracle, but can also result from mutations of the test scenarios that produce normative (acceptable) behaviors. Negative tests represent user errors, either intentional or not, and they enable checking the SUT's error handling and recovery. For negative tests, if the SUT is unable to carry out the test (*e.g.*, an error message is produced, or the execution of the next action is not possible) the test is considered as passed. If the SUT accepts the invalid test, then there may be an implementation error that needs to be investigated. Mutations of test cases can usually seen as negative tests. Therefore, this step leads to a list of scenarios with incorrect execution ("*List of scenarios with incorrect execution*" in phase 2 frame in Figure 1), containing two types of scenarios:

- Normative scenarios that lead to a negative test (their execution on the interactive application have not been successful when they should have been);
- Non-normative scenarios that lead to a positive test (their execution on the interactive application have been successful while they should not have been).

For normative scenarios, as for phase 1, the developer has to check whether the error comes from an error within the task model or within the application ("*Check faulty parts*" in phase 1 frame in Figure 1), leading to the amendment of the task model in the first case or to the amendment of the interactive application in the second case (loop back to the step "*Task modeling*", and loop back to the step "*Re-design and development of the interactive application*" on the left-hand side in Figure 1) and thus contributing to guarantying the effectiveness of the interactive application.

For non-normative scenarios, the "*Check faulty parts*" step consists in analysing the error and deciding if the faulty

behavior should be allowed by the interactive application. In that case, the tasks models must be amended in order to present this behavior within the user tasks (loop back to the "*Task modeling*" step in top left-hand side in Figure 1). Otherwise, the interactive application is embedding a behavior that should not be implemented (*e.g.*, a ATM system allowing the user to take the cash before taking the card, when the task model specifies the inverse order of events). In that case, the application must be amended to suppress this behavior (loop back to the "*Re-design and development of the interactive application*" step in Figure 1).

Once all the mutated scenario executions are correct, the task-application compatibility is ensured through the fact that the application is not allowing more interactions than the ones described in the tasks models.

A TOOL-SUPPORTED PROCESS

The process above can be carried out using a combination of existing tools.

HAMSTERS Task Modeling

HAMSTERS [12] is a tool-supported graphical task modeling notation for representing human activities in a hierarchical and ordered manner. At the higher abstraction level, goals can be decomposed into sub-goals, which can in turn be decomposed into activities. The output of this decomposition is a graphical tree of nodes. Nodes can be tasks or temporal operators.

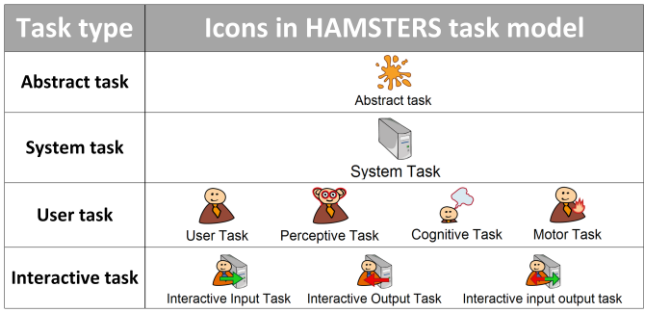

Figure 2. High-level Task Types in HAMSTERS

Tasks can be of several types (see Figure 2) and contain information such as a name, information details, and criticality level. Only the single user high-level task types are presented here but they are further refined. For instance the cognitive tasks can be refined in Analysis and Decision tasks and collaborative activities can be refined in several task types. Temporal operators (presented in Table 1) are used to represent temporal relationships between sub-goals and between activities. Tasks can also be tagged by temporal properties to indicate whether or not they are iterative, optional or both.

The HAMSTERS notation and tool provide support for task-system integration at the tool level by structuring a large number and complex set of tasks, introducing the mechanism of subroutines and generic components, and describing data that is required and manipulated in order to accomplish tasks.

Table 1. Temporal Ordering Operators in HAMSTERS

Operator type	Symbol	Description
Enable	T1>>T2	T2 is executed after T1
Concurrent	T1|||T2	T1 and T2 are executed at the same time
Choice	T1[]T2	T1 is executed OR T2 is executed
Disable	T1[>T2	Execution of T2 interrupts the execution of T1
Suspend-resume	T1|>T2	Execution of T2 interrupts the execution of T1, T1 execution is resumed after T2
Order Independent	T1|=|T2	T1 is executed then T2 OR T2 is executed then T1

Scenario Generation with TOM

For scenario generation we resort to the TOM tool. TOM's goal is to support a task-based model-based testing approach. An initial version was described in [19]. That version was restricted to CTT task models and MSWindows applications. Since then the tool has been re-implemented as a modular framework with the goal of making it more flexible. In this new version, each step of the model-based-testing process is performed by a dedicated module, with the dependencies between modules being restricted to the input and output file formats used by the different modules. The current version of TOM can interface with different task modeling notations, provided a module to translate the task model into its internal presentation is available. It can also generate test cases in different formats.

In this case, in order for the modules to be used, the HAMSTERS task models must be translated to the state machine notation used by TOM to represent oracles. This is done by defining each state in the state machine as the set of possible tasks in the model at a given instant. At the moment this translation is done resorting to the simulation features of the HAMSTERS tool.

The state machine is then traversed to generate test cases. TOM generates both *valid* test cases and *mutated* test cases, thus supporting both phases of the process (Effectiveness insurance and Compatibility insurance). The mutations currently supported by TOM include changing the order of action execution, omitting actions, or changing the input values to be used. Once the test cases have been produced they need to be translated into Hamsters' scenario notation for co-execution.

Scenario-Based Testing of an Application with TOUCAN

For the scenario-based co-execution, we rely on the TOUCAN tool. TOUCAN is a set of modules that extends Netbeans IDE. TOUCAN's architecture follows the synergistic framework that has been presented in [11]. It includes two HAMSTERS modules for task model editing and simulation and modules for connecting and co-executing task models with an interactive application.

Editing of Correspondences between Tasks and Widgets

TOUCAN enables one to define correspondences between interactive tasks and event sources and renderers. This support is achieved through the automatic extraction of

interactive input and output tasks in the HAMSTERS task models and the automatic extraction of event sources and renderers from annotated applications using Java technology. These elements are presented in an editor that enables the user to put them in correspondence. This editor also presents a view of the correspondence coverage, thus allowing one to check the completeness of the defined correspondences. An example of the use of such editor can be found in [7].

Scenario-based Testing

Once the correspondence between interactive tasks and event sources and renderers is completed, the TOUCAN tool provides three different means for the co-execution between the interactive application and its task models:

- *Task-Model driven co-execution*: in this case, the execution of the system is controlled by the task model; when an interactive task (which has been included in the correspondence file) is performed by the HAMSTERS simulator, the corresponding event handler is fired within the interactive application.
- *System driven co-execution*: in this case, the execution of the system is controlled by the user; user actions are linked to the corresponding interactive tasks from the task model and a user action on the interactive application changes the state of the task model simulation.
- *Scenario driven co-execution*: in this case, the execution of the system is controlled by a step-by-step execution of a scenario.

As said previously, we are interested here in scenario driven co-execution. This feature takes as inputs one or several test campaigns (composed of a *list of HAMSTERS scenarios)* and automatically runs all of them, step by step, on the application using the co-execution. The results of this test campaign consist in a report about the successful execution of all scenarios in the test campaign. A scenario execution is considered successful if all of its tasks are completed successfully on the application. On the contrary, a scenario running is considered not successful if one of its tasks execution is not successful. A task execution is not successful in case of an incompatibility between this task and the state of the interactive application.

ILLUSTRATIVE EXAMPLE

This section illustrates the application of the proposed approach on an example that has been extracted from a case study in the avionics application domain. While the example is necessarily small, it represents a specific case of safety and mission critical applications and its features are enough to demonstrate the approach and its capabilities.

a) b)

Figure 3. EFIS control panel (with (b) and without (a) the activation of the weather radar)

Presentation of the FCU Software

In interactive cockpits, the Flight Control Unit (FCU) is a hardware panel composed of several electronic devices (such as buttons, knobs, displays,…). It allows crew members to interact with the Auto-Pilot and to configure flying and navigation displays. The FCU Software is considered as a graphical interactive application for replacing the FCU hardware panel by graphical interfaces. It is composed of two interactive pages:

- EFIS_CP: Electronic Flight Information System Control Panel for configuring piloting and navigation displays.
- AFS_CP: Auto Flight System Control Panel for the setting of the autopilot state and parameters.

For example, this application is displayed on two of the eight cockpit LCD screens in the Airbus A380, one for the Captain and the other for the First Officer. The crew members can interact with the application via the Keyboard and Cursor Control Units which gather in a single hardware component a keyboard and a trackball.

The EFIS Control Panel is depicted in Figure 3 (with and without the activation of the Weather Radar). The left panel is dedicated to the configuration of the Primary Flight Display while the right panel is dedicated to the configuration of the Navigation Display; enabling the display of several navigation information and allowing to choose the display mode and scale.

Task model for the goal

In this paper, we will focus on the different activities that have to be performed to check the weather and verify if thunderstorms are on the flight route of the aircraft.

The HAMSTERS task model corresponding to this activity is presented in Figure 4. This task is divided in two tasks: the

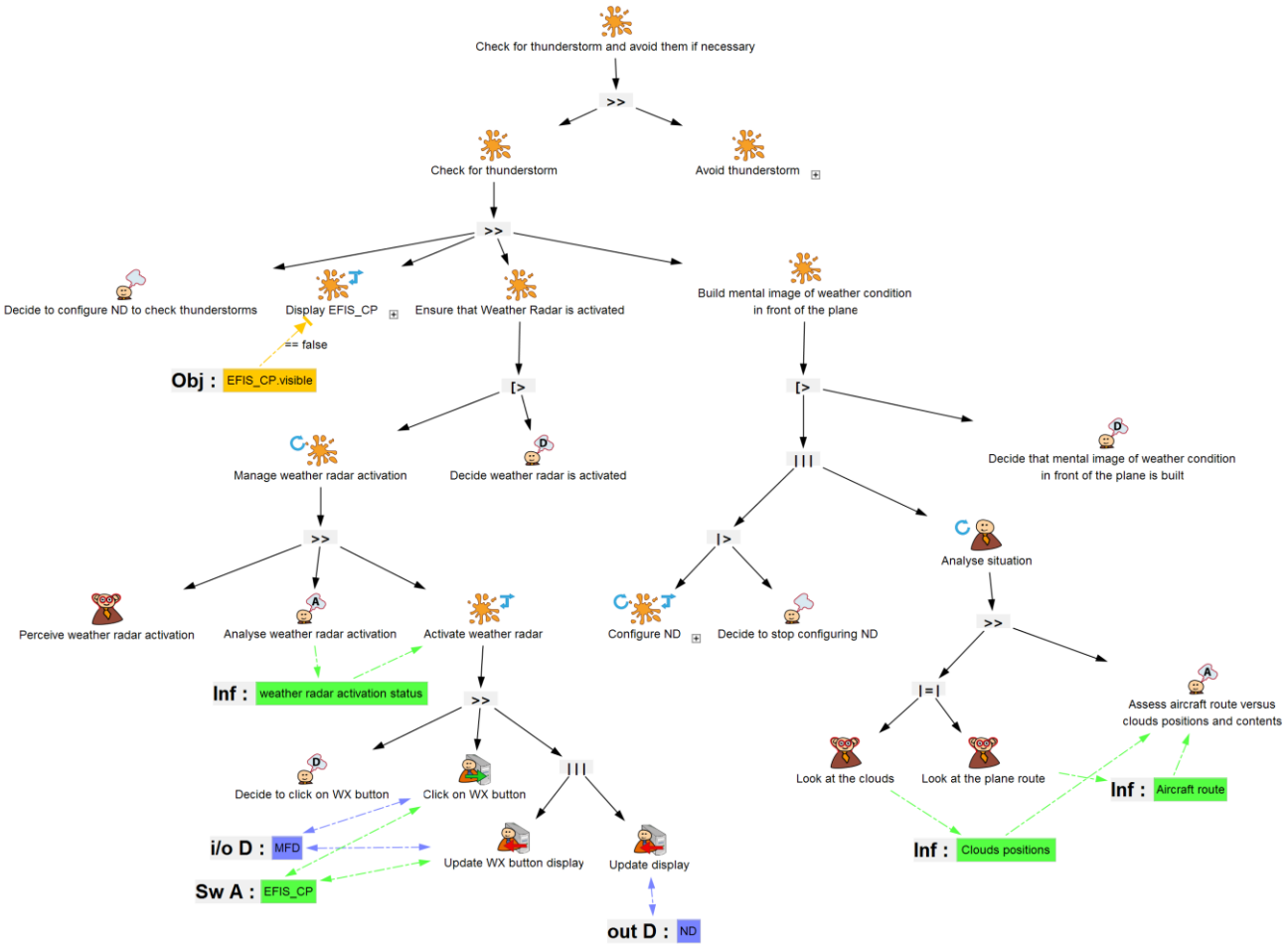

Figure 4. HAMSTERS task model for "Check for thunderstorm and avoid them if necessary" task

first one is to check if a thunderstorm is going to cross the aircraft route (abstract task "*Check for thunderstorm*" in Figure 4) and the second one is to change the aircraft route if necessary (abstract task "*Avoid thunderstorm*" in Figure 4). It is important to note that, to simplify the reading of this task model, we choose to fold some of the tasks; a folded task is indicated by a ⊞ symbol (e.g. abstract task "*Avoid thunderstorm*" in Figure 4).

In order to check if a thunderstorm is going to cross the aircraft route, the pilot must, after displaying the EFIS_CP page if this page was not the one displayed (abstract task "*Display EFIS_CP*" in Figure 4), check if the weather radar is activated (abstract task "*Ensure that Weather Radar is activated*" in Figure 4). Once the Weather Radar is activated, the pilot can analyse the weather condition in front of the plane (abstract task "*Build mental image of weather condition in front of the plane*" in Figure 4) by configuring the Navigation Display (abstract task "*Configure ND*" in Figure 4) while analysing the situation (user task "*Analyse situation*" in Figure 4).

When the pilot decides that s/he has a correct image of the weather condition (user task "*Decide that mental image of weather condition in front of the plane is built*" in Figure 4), s/he must then decide whether the aircraft route is correct or whether it should be modified (abstract task "*Avoid thunderstorm*" in Figure 4).

Scenario Generation

Following the process, a state machine representation of the task model was first generated. This involved using the simulator to explore the model, taking note of the sets of available tasks at each step. Once this was done, TOM was then used to automatically generate test cases (paths over the state machine) and translate them to HAMSTERS executions scenarios. The number of generated test cases depends on the algorithms used. Applying a shortest path algorithm between the start and end of the task generated 1176 test cases. The export feature was added to TOM in order to support the approach. Figure 5 presents (an excerpt of) a generated execution scenario.

Similarly, mutated test cases could be generated and translated into HAMSTERS scenarios. TOM has two modes

of operations regarding mutations. The tool can be used in random mode, in which cases mutations are randomly introduced in the test cases, or specific mutations can be selected for application.

Scenario-Based Testing of the Interactive Application

Once the task model and the scenario are loaded into the tool, execution proceeds autonomously. In this case, the scenario from Figure 5 was completed successfully, meaning the application supports the execution of that particular variation of the task execution (each scenario capture a possibility of carrying out the task).

To illustrate the situation of a failed test, we can consider that user interface mode changes and dynamic function allocation are two aspects that can interfere with how a user expects to use a system. They can lead to erroneous interpretation of the behavior of the system and/or automation surprises. Regarding the GUI they will affect how the system responds to user actions, but also what user actions are possible at any given moment. For illustration purposes, we changed the application to disable the WX button so that no interaction could be performed anymore on this button. The new running of the test campaign containing the scenario presented in Figure 5 leads to the results presented in Figure 6. The figure depicts a screenshot of the test campaign, using TOUCAN, showing the user interface on the right and the co-execution panel on the left. This panel is further divided in two parts. The left part shows the list of scenario present in the test campaign. Each scenario is associated with a green symbol (✔) if its running has been successful, or with a red symbol (⛔) if its running has failed. In that case, the concerned scenario execution ("*Scenario 3*") has failed.

The right part shows the task list of the selected scenario (here "Scenario 3"). The successful tasks are highlighted in green. If a task is not successful, it is highlighted in red, the co-execution then stops and the tasks that have not been executed are highlighted in grey. It can be seen that *"Scenario 3"* failed due to the fact that interactive input task "*Click on WX button*" cannot be performed on the application.

DISCUSSION

We can identify two contributions of the work reported in this paper: a stepwise process for ensuring the effectiveness of an application by analyzing task-application compatibility; an instantiation of that process with a concrete set of tools. The process assumes a model-based approach to interactive systems development, assuming task models will be available. Variations on this generic process can be envisaged. For example, for approaches based on state machines representations of the user interface (e.g., [1, 3, 19]) the generation of the test cases could be done directly from those state machines, although the notion of normative and non-normative behavior provided by task models would be lost.

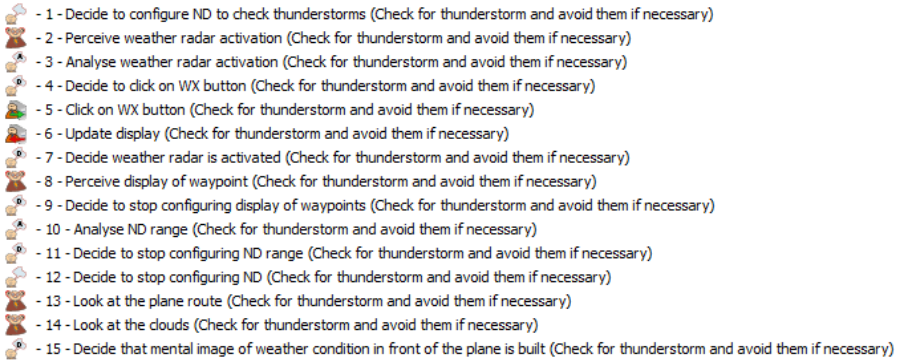

Figure 5. Extract of one of the generated scenarios

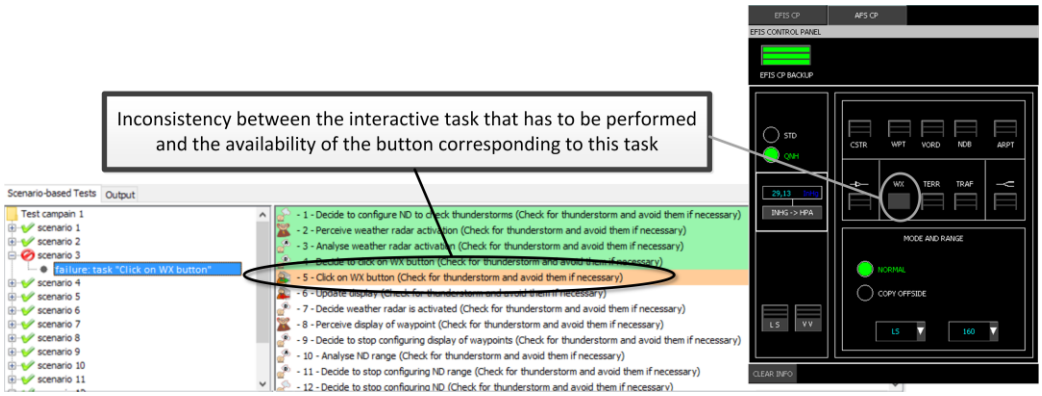

Figure 6. Results of the test campaign while the WX button has been disabled in the FCUS application

The proposed instantiation supports the semi-automated analysis of Java applications against their task models, expressed in HAMSTERS, in a manner that would be unfeasible manually. The tool set used, however, inevitably presents restrictions both in terms of the supported technology, and their support to the process.

Regarding the former, the main restriction is the co-execution component. TOUCAN currently supports Java applications. However, the concepts remain the same with any other technology.

Regarding the latter, at the moment, both the generation of (mutated) scenarios, as well as the execution of test campaigns composed of several scenarios are automated. However, some steps still needs to be performed manually that might represent bottlenecks. One is bridging from HAMSTERS to TOM; i.e., the generation of the state machine representation of the task model. A viable solution to automate this step seems to be to automate the execution of the simulator, so that it will automatically explore all possible tasks, taking note of the available tasks at each step of the process. The information thus gathered will then be exported as a state chart model. This automation would enable to complete the automation of the process, leading to the ability to deal with more complex task models. Another approach to investigate this issue could be to build upon the work that has been done with CTT [14].

Regarding the scenario generation phase, the main manual step is the analysis of test results. Given the high number of test cases that can be automatically generated and tested, this task can grow rapidly. One solution to this problem is to improve the quality of the generated test cases. This can be done by exploring adequate coverage criteria for non-mutated test cases, in particular whether information from the task model might be used to define coverage criteria, and by improving the quality of the mutations, thus also improving the coverage of the test cases.

One relevant aspect that needs to be addressed when considering an approach such as the one proposed here is how to deal with false positive and false negative results. In this regard, the approach has two main potential sources of problems. One is the task model itself. If the model is incorrect, test cases will not represent the intended usage of the system. It should be noted that the model is an input to the process, so it is assumed the model is correct. In any case, negative results will prompt analysis of the test cases and SUT helping in correcting not only the SUT but also the task model (via the test cases generated from it). False positives are harder to identify as they represent a silent failure. Another is the correspondence between model and SUT. Here, failures will typically correspond to failures in the co-execution, making them easier to identify. Additionally, tool support further reduces the opportunity for such errors.

CONCLUSION

This article presented a stepwise process for ensuring the effectiveness of an application by analyzing task-application compatibility. The proposed approach builds on a synergistic approach, enabling the coupling of task models and interactive applications, and brings to it the benefits of MBT in order to automate the scenario-based testing of interactive application, thus ensuring a less expensive (and less time consuming) test phase to check the consistency between task models and interactive applications, guaranteeing at the same time better test coverage. The application of a proposed instantiation of this process on an example from aircraft cockpits has been presented.

The proposed approach aims to be generic. The tool set used to illustrate it, however, inevitably presents restrictions. These relate to both the technology that might be used for applications development, and the support given to the steps the process. Current limitations have been discussed and opportunities for further work identified. These range from automating steps that are at the moment done manually, such as the generation of states machines from task models, to improving the generation of mutated scenarios. Currently the mutation strategies used in TOM are rather simple. One potential advantage of using HAMSTERS, is the fact that task models can be enriched with information about the objects being manipulated and the errors that might be expectable from the users at each step in the interaction [6]. Using this information will enable a more powerful exploration of variations on the prescribed user behavior, thus improving the quality of the test suites being generated.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments on the original version of this paper.

José Campos acknowledges support from project "NORTE-01-0145-FEDER-000016", financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

REFERENCES

1. Appert, C. and Beaudouin-Lafon, M. SwingStates: adding state machines to Java and the Swing toolkit. *Software: Practice and Experience 38*, 11 (2008), 1149-1182.
2. Barbosa, A., Paiva, A. and Campos, J.C. Test case generation from mutated task models. *Proc. ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2011)*, ACM Press (2011), 175-184.
3. Blanch, R. and Beaudouin-Lafon, M. Programming rich interactions using the hierarchical state machine toolkit. *Proc. Working Conference on Advanced Visual Interfaces (AVI '06)*. ACM Press (2006), 51-58.
4. Bowen, J. and Reeves, S. UI-design driven model-based testing. *Innovations in Systems and Software Engineering, 9*, 3 (2013), 201-215.

5. Bowen J. and Reeves S. UI-driven test-first development of interactive systems. *Proc. 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2011)*. ACM Press (2011), 165-174.
6. Fahssi, R., Martinie, C. and Palanque P. Enhanced Task Modelling for Systematic Identification and Explicit Representation of Human Errors. *Proc. IFIP TC13 INTERACT 2015*, vol. 9299 of Lecture Notes in Computer Science, Springer (2015), 192-212.
7. Fayollas, C., Martinie, C., Navarre, D. and Palanque, P. A Generic Approach for Assessing Compatibility Between Task Descriptions and Interactive Systems: Application to the Effectiveness of a Flight Control Unit. *i-com 14*, 3 (2015), 170–191.
8. ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) - Part 11: Guidance on usability (1998).
9. Lelli, V., Blouin, A., Baudry, B. and Coulon, F. On Model-Based Testing Advanced GUIs. *Proc. 2015 IEEE 8th Intl. Conf. Software Testing, Verification and Validation Workshops (ICSTW), 11th Workshop on Advances in Model Based Testing (A-MOST)*, IEEE (2015).
10. Manca, M., Paternò, F., Santoro, C. and Spano, L. D. Generation of multi-device adaptive multimodal web applications. *Proc. Mobile Web Information Systems (MobiWIS 2013)*, vol. 8093 of Lecture Notes in Computer Science, Springer (2013), 218-232.
11. Martinie, C., Navarre, D., Palanque, P. and Fayollas, C. A generic tool-supported framework for coupling task models and interactive applications. *Proc. 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS 2015)*. ACM Press (2015), 244-253.
12. Martinie, C., Palanque, P. and Winckler, M. Structuring and Composition Mechanism to Address Scalability Issues in Task Models. *Proc. IFIP TC13 INTERACT 2011*, vol. 6948 of Lecture Notes in Computer Science, Springer (2011), 589-609.
13. Memon, A.M. A Comprehensive Framework For Testing Graphical User Interfaces. *PhD thesis*, University of Pittsburgh, 2001.
14. Mori, G., Paternò, F. and Santoro C. CTTE: Support for Developing and Analyzing Task Models for Interactive System Design. *IEEE Trans. Software Eng. 28*, 8 (2002), 797-813.
15. Navarre, D., Palanque, P., Paternò, F., Santoro, C. and Bastide, R. A Tool Suite for Integrating Task and System Models through Scenarios. *Proc. DSV-IS 2001*, vol. 2220 of Lecture Notes in Computer Science, Springer (2001), 88-113.
16. Palanque, P., Bastide, R. and Sengès, V. Validating interactive system design through the verification of formal task and system models. *Proc. IFIP TC2/WG2.7 Work. Conf. on Eng. for Human-Computer Interaction (EHCI 1995)*, Chapman & Hall (1995), 189-212
17. Reason, J. *Human error*. Cambridge University Press, 1990.
18. Rodrigo, M., Moreira, L.M. and Paiva, A. PBGT tool: an integrated modeling and testing environment for pattern-based GUI testing. *Proc. 29th ACM/IEEE Intl. Conf. on Automated Software Engineering (ASE '14)*, ACM (2014), 863-866.
19. Rossignol, V. SCADE Display® for the Design of Airborne and Ground-Based Radar Human-Machine Interfaces (HMIs). *Infowaves 11*, 4 (2014).
20. Silva, J.L., Campos, J.C. and Paiva A. Model-based user interface testing with Spec Explorer and ConcurTaskTrees. *Electronic Notes in Theoretical Computer Science 208*, (2008), 77-93.
21. Utting, M. and Legeard, B. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.
22. Wilson, S., Johnson, P., Kelly, C., Cunningham, J. and Markopoulos, P. Beyond hacking: A model based approach to user interface design, *People and computers VIII, Proc. HCI 93*, Cambridge University Press, BCS HCI (1993), 217-231.