A "More Intelligent" Test Case Generation Approach through Task Models Manipulation

CAMILLE FAYOLLAS, CELIA MARTINIE, DAVID NAVARRE, and PHILIPPE PALANQUE, Interactive Critical Systems, University Toulouse III
JOSÉ C. CAMPOS, MARCELO GONÇALVES, and MIGUEL PINTO, HASLab/INESC TEC & Department of Informatics/University of Minho

Ensuring that an interactive application allows users to perform their activities and reach their goals is critical to the overall usability of the interactive application. Indeed, the effectiveness factor of usability directly refers to this capability. Assessing effectiveness is a real challenge for usability testing as usability tests only cover a very limited number of tasks and activities. This paper proposes an approach towards automated testing of effectiveness of interactive applications. To this end we resort to two main elements: an exhaustive description of users' activities and goals using task models, and the generation of scenarios (from the task models) to be tested over the application. However, the number of scenarios can be very high (beyond the computing capabilities of machines) and we might end up testing multiple similar scenarios. In order to overcome these problems, we propose strategies based on task models manipulations (e.g., manipulating task nodes, operator nodes, information ...) resulting in a more intelligent test case generation approach. For each strategy, we investigate its relevance (both in terms of test case generation and in terms of validity compared to the original task models) and we illustrate it with a small example. Finally, the proposed strategies are applied on a real-size case study demonstrating their relevance and validity to test interactive applications.

CCS Concepts: • **Human-centered computing → HCI design and evaluation methods**; *Graphical user interfaces*; *HCI theory, concepts and models*; • **Software and its engineering → Software testing and debugging**; *Model-driven software engineering*;

Additional Key Words and Phrases: Model-Based testing, Task modeling

ACM Reference format:
Camille Fayollas, Celia Martinie, David Navarre, Philippe Palanque, José C. Campos, Marcelo Gonçalves, and Miguel Pinto. 2017. A "More Intelligent" Test Case Generation Approach through Task Models Manipulation. *Proc. ACM Hum.-Comput. Interact.* 1, 1, Article 8 (2017), 20 pages.
https://doi.org/0000001.0000001

1 INTRODUCTION

Usability testing is typically a craft process involving high-level expertise evaluators involved in very repetitive testing tasks with multiple end users. According to the ISO 9241 standard [1] usability is decomposed into three factors: efficiency, effectiveness and satisfaction. Efficiency can be (partly) assessed in a predictive way exploiting high-level models such as GOMS [14], KLM [6] or Fitts' law [9] related research contributions [11]. Satisfaction received a lot of attention with the user experience wave, and questionnaires such as SUS [7] or AttrakDiff [12] provide efficient ways of assessing users' satisfaction. Effectiveness corresponds to the capability of the interactive application to allow users to reach their goals and to perform their activities. Assessing effectiveness requires assessing (in an exhaustive way) that every goal is reachable and that each activity can be performed on the application. This assessment is not compatible with the craft and manual process of usability evaluation and thus requires computational support to automate that otherwise cumbersome assessment. The activity of effectiveness testing is very similar to the one of software testing. Indeed, software testing aims at covering all the possible execution cases to detect (and possibly remove) faults added by developers [24]. In that domain, exhaustive testing can only be achieved using automated techniques. Interactive applications testing brings additional issues as demonstrated in [21] and exploiting models during tests is the only way to deal with them. Indeed, model-based testing [34] supports the automated generation, execution and evaluation of test cases. This has been shown, for example, in [5, 26]. In the later, the TOM framework for the model-based testing of Graphical User Interfaces (GUIs) is used to generate execution scenarios to be automatically executed by CIRCUS [8] a tool supporting task modeling and analysis in HAMSTERS [19].

Except for simple (or even trivial) user interfaces, however, generating test cases to cover all possible sequences of user actions with the system is not feasible [26]. As this "brute force" approach is not feasible, it is important to find ways of reducing the number of test cases to be checked. Even doing so, the number of relevant test cases can be huge and testing them all time and resources consuming. Hence, before discarding a test case, a trade-off must be reached between its quality and the cost of applying it. An example criterion could be interactive application function coverage. With this criterion, a test suite covering all the functions offered by the interactive application would be favored. Similarly, another criterion could be test case size minimization. With this criterion, a test suite limiting the number of visits to the same state of the interactive application would be favored. However, these criteria offer a system oriented perspective to the test cases selection and thus do not explicitly take into account how the users will use the system. While this is acceptable in the area of software testing, this does not stand when considering interactive applications and especially when their effectiveness has to be assessed.

This paper addresses this problem and proposes strategies for a so-called "more intelligent" test cases generation approach, combining "brute force" avoidance and usage-centered selection of test cases. The goal is to guarantee that a specified subset of all possible interactions between user and system (as defined in a task model) can be fully covered by the testing process. With this in mind, the main contribution of the paper is a set of task manipulation strategies that can be used to introduce simplifications into a task model. These manipulation strategies were defined taking into particular consideration a specific application area (interactive cockpits), and are applied on the most recent version of the task model notation HAMSTERS [18]. However, most of them would be applicable to other domains and other task model notations, as most of the concepts in HAMSTERS come from agreed knowledge in the domain of task modeling.

The paper is structured as follows: first related work is presented in order to put the work into context. We then analyze alternatives for controlling test cases generation. Our strategies, based on task model's manipulation, are presented, followed by their application on a case study from the avionics domain. A discussion section presents the advantages and the limitations of the strategies presented and outlines a process for organizing them. Finally, the last section concludes the paper highlighting directions for future work.

2 BACKGROUND AND RELATED WORK

This section provides background information about model-based testing for graphical user interfaces and presents related work on scenario-based testing of interactive applications.

2.1 Model-based Testing of Graphical User Interfaces

Model-Based Testing (MBT) is a black-box testing technique based on a model of the System Under Test (SUT). The purpose of the technique is to evaluate a system's implementation against its specification. The basic idea is to use an abstract model that represents the behavior of the SUT to generate test cases. Then, after or during the execution of the test cases, the model is used as an oracle. The discovery of system errors is achieved by comparing the results of the tests' execution against those prescribed by the oracle. A distinctive advantage of MBT is that a high level of automation can be achieved. Test case generation, test case execution and the comparison of the oracle's prediction with actual results are all tasks that can be automated.

The application of MBT to GUIs enables the automation of user interface testing from a reliability perspective (cf. [13]). GUITAR [20] was probably the first project developed in this area. The goal was to develop a model-based GUI testing framework, supporting the automation and integration of different tools and techniques used in various phases of GUI testing. The framework has evolved into a extensible plugin-based architecture supporting automated test case generation and execution, as well as platform-specific customizations, the addition of new algorithms as plugins, and support for integration into third-party test harnesses and quality assurance workflows [26].

The challenges placed by the application of MBT to GUIs range from which modeling approaches to use, and how to obtain the needed models, to how to define an adequate test suite. A number of authors have explored the application of reverse engineering techniques to deduce models from implemented systems in support for test cases generation [3, 10, 23, 28, 31], while others have proposed deriving the test cases from design artifacts [4, 5, 32]. Others, still, are concerned with the limitations posed by the use of event-based models of systems, which are not adequate to test multi-touch and gesture-based interactions [16], and explore the use of specification languages able to deal with advanced multi-event GUIs.

Because fully testing a GUI is rarely feasible for interactive computing systems of reasonable size, a balance must be reached between the amount of coverage provided by the test cases, and the cost effectiveness of the test suite. The concept of Test Patterns has been used to represent a set of test strategies that simplify the testing of specific GUI features. This approach aims at promoting the reuse of test strategies to test typical behaviors and has been applied both to web [22] and mobile [23] applications, but it is only able to test the aspects of the user interface that are covered by the pattern language in use. Silva et al. [32] use ConcurTaskTree (CTT) [29] task models as oracle in order to both reduce the cost of producing the oracle and focus the test cases on the normative uses of the system. The drawback, however, is that the test suite can become too restrictive as users might deviate from that normative behavior. The work was extended in [5] by including the notion of test cases mutations, in order to take into account the possibility of deviations from the prescribed behavior (more specifically, user errors).

2.2 Background

The work presented in this paper refines and extends the approach that has been presented in [5]. That approach exploited task models to produce scenarios, that is, flows of execution of a given task model as proposed by [25]. Scenario-based testing of software applications received attention in software engineering. For instance, [17] proposes the use of scenario-based test cases from explicit representation of user needs or more precisely user safety needs as presented in [33]. [2] proposes the production of scenarios from requirements to limit tests to the most purposeful ones. However none of them considers scenarios in connection with descriptions of users' goals and tasks.

This section first details the scenario-based testing of interactive application presented in [5]. Then, it quickly introduces the HAMSTERS task modeling notation [18]. Finally, by highlighting the limitations of the approach (tightly related to the expressiveness of the HAMSTERS notation), it defines the focus of this paper.

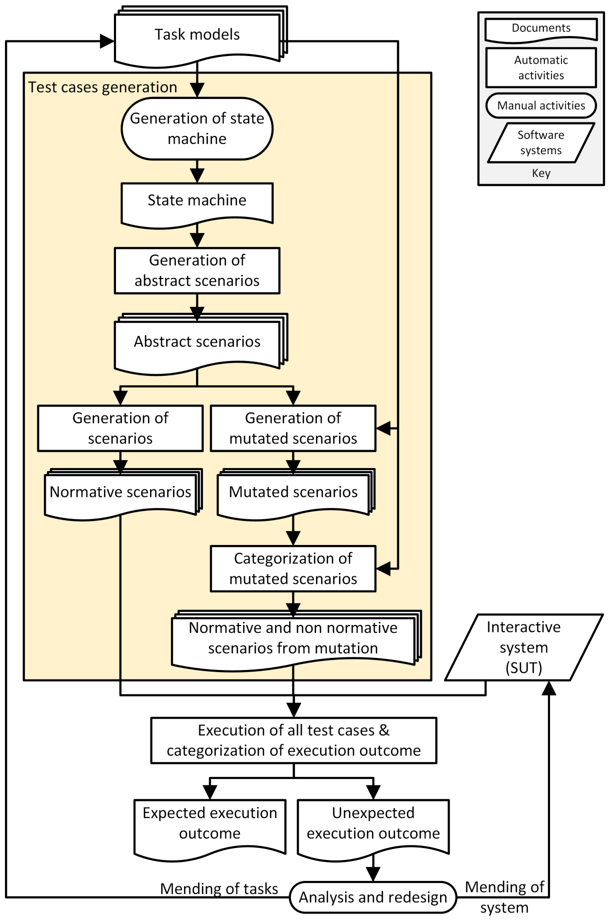

Task models
Test cases generation
Generation of state machine
State machine
Generation of abstract scenarios
Abstract scenarios
Generation of scenarios
Generation of mutated scenarios
Normative scenarios
Mutated scenarios
Categorization of mutated scenarios
Normative and non normative scenarios from mutation
Interactive system (SUT)
Execution of all test cases & categorization of execution outcome
Expected execution outcome
Unexpected execution outcome
Mending of tasks
Analysis and redesign
Mending of system
Documents
Automatic activities
Manual activities
Software systems
Key

Fig. 1. Scenario-based testing of interactive application process

2.2.1 Scenario-Based Testing of Interactive Applications. The tool supported approach presented in [5] (see Figure 1) assumes a model-based approach to systems development; more specifically, one in which task models of the proposed systems are developed. Starting from the task models, a state machine is manually generated. A task model defines the possible sequences of user and system actions to achieve a goal. At each instant a set of actions is allowed. Executing an action, changes the set of allowed actions, as defined by the task model hierarchical structure and operators. The set of task available at a given moment is called a Presentation Task Set (PTS - cf. [30]).

A finite state machine capturing the behavior expressed in a task model can be created by considering the PTS as states in the state machine and labeling the transitions with the tasks. The algorithm starts by calculating the initial PTS of the model (which constitutes the initial state) and then, for each task in the PTS, calculates the PTS/state resulting from its execution. The algorithm is applied to all PTS until all tasks in all PTS have been exercised. This state machine is used to generate a list of abstract scenarios that will be used to automatically generate the test cases.

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

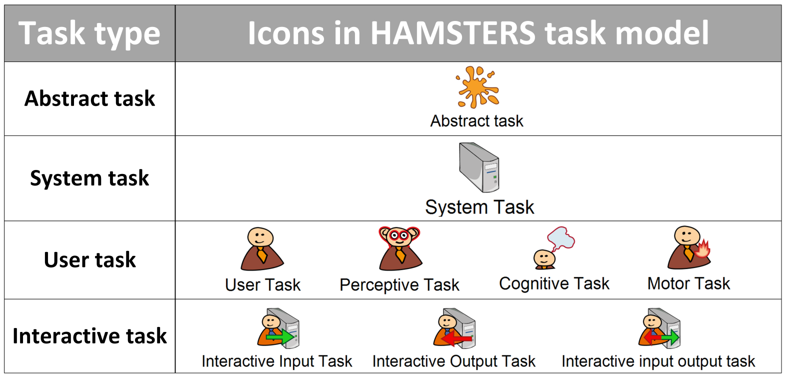
The test cases can be: normative or mutated scenarios. Normative scenarios capture concrete sequences of action as described by the task model. Mutated scenarios intent to capture possible user errors as deviations from the norm (described by the task model). These mutated scenarios however might be normative or not and have to be categorized. This is done through their automatic running on the task model: if a mutated scenario can be executed on the task model, it is categorized as normative, otherwise as non-normative. Together, normative scenarios and non-normative scenarios form the set of test cases. While the mutation of test cases helps increase the coverage provided by a test cases set, the focus here is on normative behaviors and mutations will not be further addressed.

These test cases are automatically executed on the system through the scenario-driven co-execution of the task model and the interactive system (the SUT). This step leads to i) a list of scenarios with expected execution outcome and ii) a list of scenarios with unexpected execution outcome. The execution outcome of a normative scenario is expected to be successful (meaning that all of its tasks have to be performed successfully on the system); on the contrary, the execution outcome of a non-normative scenario is expected to be unsuccessful. Finally, the scenarios with an unexpected execution outcome are analyzed: the developer has to check whether the error comes from an error within the task model or within the application. In the first case, the task model needs to be amended in order to correct the error (“Mending of tasks” loop back to the task models 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 (“Mending of system” loop back to the interactive system on the right-hand side in Figure 1).

2.2.2 HAMSTERS Notation. HAMSTERS [19] 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.

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 into Analysis and Decision tasks, and collaborative activities can be refined into 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.

2.2.3 Limitations of the Original Approach. The main limitation of the approach just described is that it takes for granted the fact that it is possible to generate the exhaustive list of scenarios. In complex systems, testing based on covering all control flow or all data flow paths is intractable [2]. In this particular approach, problems in



Task type	Icons in HAMSTERS task model			
Abstract task	Abstract task			
System task	System Task			
User task	User Task	Perceptive Task	Cognitive Task	Motor Task
Interactive task	Interactive Input Task	Interactive Output Task	Interactive input output task	

Fig. 2. High-level Task Types in HAMSTERS

test case generation arise due to the complexity of the dialogue between user and system. This complexity if manifested in the task model, which, in turn, reflects on the state machine used for test case generation. In order to guide the test case generation process, it is first necessary to understand this complexity.

Sources of complexity stem from the interplay between a number of HAMSTERS operators and the fact that behaviors must be explicitly represented in the state machine derived from the task model:

- *Iteration.* Cycles in the state machine reduce the choice of algorithms (e.g., for state traversal or state machine reduction) that can be applied as several such algorithms require acyclic *graphs.*
- *Order independence.* The state machine must allow all permutations of the sub-tasks of a order independence operator, this will originate $n!$ paths, where n is the number of sub-tasks of the operator.
- *Concurrency.* In this case, the permutations must be considered at the level of each sub-tasks' activities, taking into account that they must preserve the individual behavior of each sub-task in isolation; as sub-tasks grow in complexity, the concurrency operator rapidly increases the complexity of the generated state machine.
- *Interruptions.* To model interruptions the state machine must provide the interruption sub-task in all steps of the interruptible sub-tasks, thus contributing to increase the number of transitions in the model; interruptions are routinely used, as a modeling pattern, as a means to stop iterations.
- *Suspend-resume.* Suspend-resume operators extend interruptions with the need to return to the point where the interruption happens; for that to be modeled, the interrupting task must be repeated has many times as the number of atomic tasks in the interrupted sub-task.
- *Data space.* When generating the HAMSTERS scenarios, concrete values must be provided for the input values, thus each path over the state machine will generate many scenarios.

3 ALTERNATIVES FOR CONTROLLING TEST CASE GENERATION

Having identified the need to reduce the number of test cases generated, while ensuring a good coverage of the most likely usages of the system, this section now discusses alternatives to having a smarter test case generation process. In particular, it first presents the possibility to deal with this issue at the level of the state machine, and then at the level of the task model. Finally, it presents the approach that we propose in this paper.

3.1 At the Level of the State Machine

The typical approach for reducing the number of test cases generated is to define coverage criteria and then produce the minimal set of test cases that comply with those criteria. Memon [21] argues that traditional structural criteria (based on code coverage), are not appropriate for GUI testing and that GUI event-based criteria should be used instead. Several criteria based on event coverage are proposed.

Since interactive tasks in a task model can be mapped to GUI events, we could imagine defining a coverage criterion requiring every interactive task to be executed at least once. The problem, however, is that it is not immediate how to choose a specific visiting order with such criterion. What this means is that it is not easy to control which scenarios will be generated when traversing the state machine.

The above problem is relevant because not all behaviors will be equally likely on the user interface, and ideally the testing process should give priority to those behaviors the user is more likely to exhibit. For example, while the task model might define as valid any order of filling in a login form, a user is more likely to fill in the user name first and only then the password. If resources are limited, priority should be given to testing the first possibility.

3.2 At the Level of the Task Model

The analysis of which scenarios are more likely, is more easily done at the task model level. The hierarchical decomposition of tasks/goals into sub-tasks can be used to more easily judge which behavior to consider. This leads to the consideration that the control over which scenarios to generate should be done at this level. Additionally this allows creating a smaller state machine representation of the model, since only the relevant scenarios will be considered, which is advantageous in terms of the performance of the scenarios generation process.

The goal, then, is to generate a state machine that exhibits only a subset of the behavior of the task model, as this will simplify the generation of more relevant scenarios. Two options can be considered for this. A first possibility is to change the PTS-based state machine generation algorithm to take into consideration the likelihood of each specific task execution alternative. The algorithm would have to be configured, for example, by using annotations on the task model to determine which parts of the model would be represented on the generated state machine. A simpler alternative is to directly manipulate the task model to restrict allowed behaviors to those that are deemed more likely to happen. The generation algorithm can remain the same, and the decisions made are more easily visualized in the intermediate task model thus generated. It should be stressed that this, intermediate, modified task model is purely instrumental and not a normative model of the system’s tasks.

3.3 Our Approach

The two alternatives above address the same problem, albeit from different starting points. The second alternative has the advantage that the generation algorithm is kept simple, while allowing a more clear visualization (through the manipulated task model) of which assumptions about user behavior are being made. Hence, in what follows, the control of scenarios generation through the introduction of changes in the task model is the preferred approach. Nevertheless, in specific cases, the first approach may also be used.

4 CONTROLLING TEST CASE GENERATION THROUGH TASK MODEL MANIPULATION

This section presents a catalog of strategies for task model manipulation, targeted at guiding the generation of test scenarios to those more relevant for the system under analysis. The strategies to manipulate the task model can be categorized in one of four types, depending on what type of modification is applied:

- *Removal of non-interaction tasks.* These strategies use the fact that only interaction related tasks are relevant for the automated execution of test scenarios to simplify the task model.

- *Modifying the task attributes.* These include putting a upper bound on the number of times an iterative task can be repeated.
- *Modifying the operators.* These include replacing operators that lead to complexity in the generated state machine (such as independence or concurrency operators) by enabling operators to simplify the dialogue structure.
- *Limiting the data space.* These include defining equivalence classes for the data to input. The problem of generating input values is a well studied topic with its specific set of techniques (e.g., [27]) and will not be addressed further here.

As it will be discussed below, applying these strategies must be done taking into consideration what the likely usages of the system are. This makes applying the proposed process compatible with an evidence based software engineering approach [15].

4.1 Removal of Non-Interaction Tasks

In practice, only interaction tasks are relevant for the execution of the scenarios on the system. Non-interaction related tasks (e.g., system tasks or cognition related tasks), are relevant for task analysis and for assessing possible usability related problems. Here however, while some level of usability related reasoning can be achieved (such as reasoning about the effectiveness of the user interface wrt. the tasks that should be supported), we are focusing mainly on the reliability of the user interface implementation. When concrete test scenarios are generated, only interaction tasks are considered, and all non-interaction related tasks present in the abstract test case are disregarded. This happens because the MBT process captures the interaction between user and SUT, but not the user's cognitive processes nor the system's internal processes. It is relevant because it opens the possibility of simplifying the models by removing some, or all, of the non-interaction tasks from the model, prior to the generation of the state machine.

Two approaches can be considered for this: aggregating non-interaction tasks or complete removal of non-interaction tasks. These approaches are described below. It is relevant to note that, while the examples in this paper do not present system tasks, the ideas presented herein are also applicable to these tasks (as they are non-interaction tasks).

4.1.1 *Aggregation of Non-Interaction Tasks.*

The first, more conservative approach, is to simplify the model by abstracting sequences of non-interaction related tasks into single tasks. The changes to the model are small in this case as the non-interaction tasks are kept in the model, albeit at a higher level of abstraction. Consider, for example, the sub-task tree for task "Ensure display of waypoint" in Figure 3. In this situation, the two tasks "Perceive ND image" and "Analyse if waypoints are displayed" are replaced by a single task (e.g., an abstract user task named "Perceive ND image and Analyse if waypoints are displayed").

4.1.2 *Complete Removal of Non-Interaction Tasks.*

A second, more aggressive, alternative is to filter all non-interaction related tasks out of the model. This will imply structural changes to the task model. Consider, again, the sub-task "Ensure display of waypoint" in Figure 3. If tasks "Perceive ND image" and "Analyse if waypoints are displayed" are removed, then it makes sense to aggregate "Ensure display of waypoints" and "Activate display of waypoints" abstract tasks and replace them by a single task. Changes might also be necessary to ensure that the set of possible behaviours, when only interaction tasks are considered, is not changed.

4.2 Modifying the Task Attributes: Limiting the Number of Iterations

In this strategy, we define a maximum number of iterations for iterative tasks, by defining an upper bound on all relevant iterative tasks in the task model. This information can be used in one or two ways. The first is to generate a state machine that structurally allows for the maximum number of iterations only. In practice this

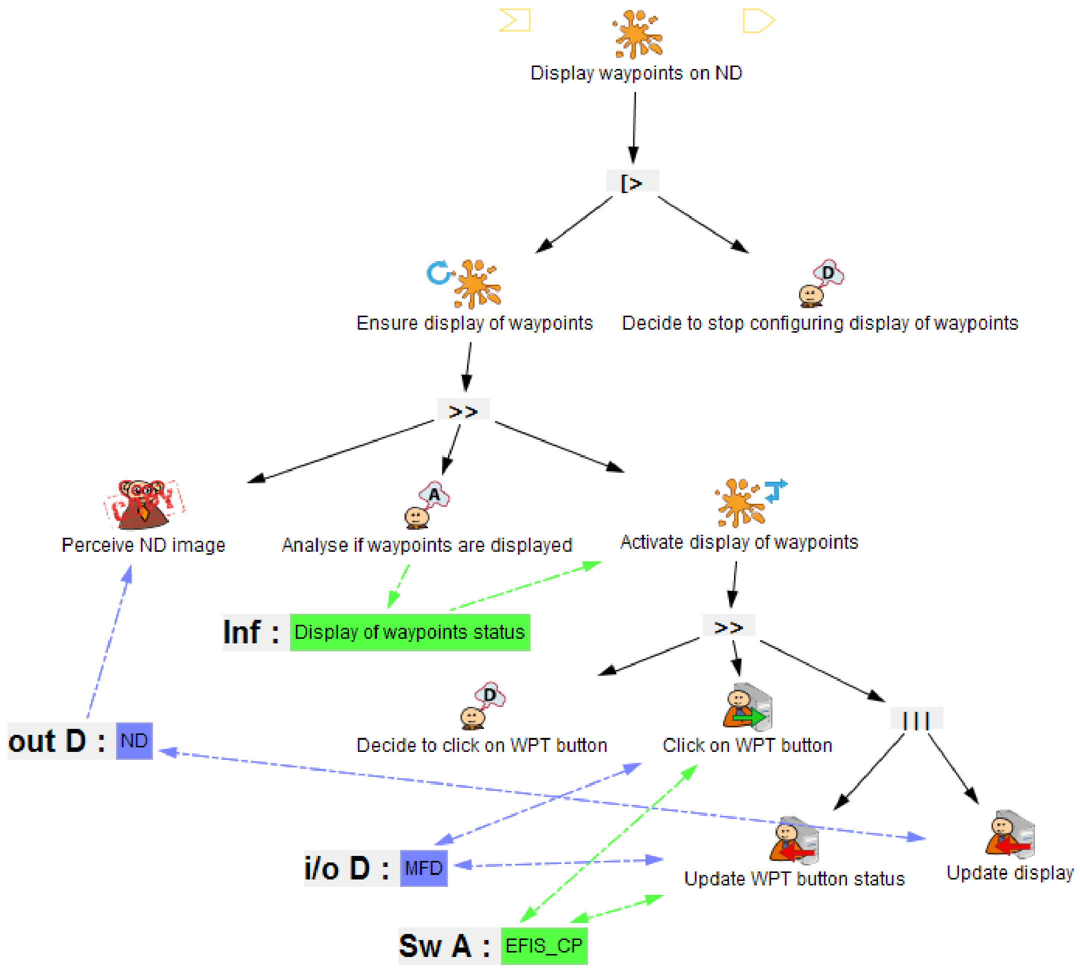

Display waypoints on ND
[>
Ensure display of waypoints
Decide to stop configuring display of waypoints
>>
Perceive ND image
Analyse if waypoints are displayed
Activate display of waypoints
Inf : Display of waypoints status
>>
Decide to click on WPT button
Click on WPT button
|||
out D : ND
Update WPT button status
Update display
i/o D : MFD
Sw A : EFIS_CP

Fig. 3. "Display waypoints on ND" task model

means unfolding the cycles in the state machine up to the defined upper bound. The advantage of this approach is that the state machine becomes acyclic. The drawback is that the state machine will increase in size.

An alternative is to use the cycles' upper bounds to control the traversal algorithm. Hence, if a cycle can only be executed once, we instruct the algorithm to allow only two visits to the relevant starting state (one happens at the start of the cycle, the other at the end). If information is included in the states regarding how many times they can be visited, the algorithm can be constructed such that different bounds can be used for each iterative task. If not, a general bound can be used which will be applied to all cycles. In general, for an upper bound of n, the starting state for the cycle will be visited $n + 1$ times.

The rationale behind this manipulation is that although a task might be repeated many times, in practice there is a maximum number of iterations that it is reasonable to expect. Additionally, from a testing perspective, it typically will not be relevant to repeat the task a large number of times. To judge what a reasonable number of

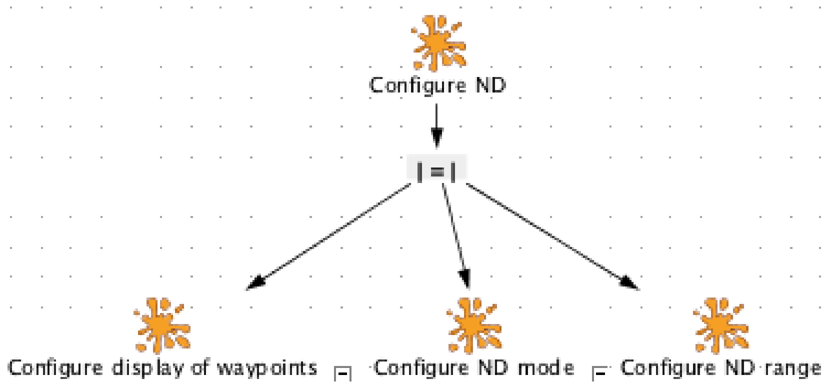


Fig. 4. Task model illustrating the use of an order independence operator

iterations is, evidence obtained either from requirements analysis or from observation or records of system use can be used.

4.3 Modifying the Operators

Four operators are candidates to being replaced, as they increase the complexity of the generated state machine: order-independent, concurrent, interruption and suspend-resume operators.

4.3.1 Replace Order-Independent Operators. Order independence between sub-tasks is a major contributor to the state explosion in the state machine generated from a task model. Consider the simple example in Figure 4, although the task model has only three tasks at the leaves, the resulting state machine has 15 transitions (see Figure 5). This happens because we must consider all the permutations of the three tasks' execution. Hence for 3 tasks we must have 6 (3!) paths in the state machine.

To avoid this, in this strategy, we define a concrete order for the order independent tasks. This is achieved by replacing order independence operators with enabling operators. The most appropriate sequencing of sub-tasks must be determined and the task model updated accordingly.

The rationale behind this strategy is that, as already discussed, while the execution order for the task might not be relevant, typically users will tend to carry it out in a specific order and, in many cases, that order is consistent across users. This happens, e.g., in aircraft cockpits where pilots have well defined operations procedures (either prescribed or observed in practice).

4.3.2 Replace Concurrent Operators. Concurrent tasks add an additional level of complexity to order independence. In particular, if the concurrent sub-tasks are further decomposed. In order independence, a sub-task, once started, would be executed until completion, before another sub-task could be executed. With concurrency, the state machine must consider all possible permutations of the sub-tasks' activities.

In the strategy to address this issue, we again define a concrete order to concurrent tasks by replacing order concurrency operators with enabling operators. As above, the most appropriate sequencing of sub-tasks must be determined and the task model updated accordingly.

The rationale behind this strategy is the same as for order independence.

4.3.3 Replace Interruption Operators. The interruption operator adds an additional transition to all states of the interrupted sub-task (see Figures 6 and 7). In this strategy, we replace interruption operators with enabling

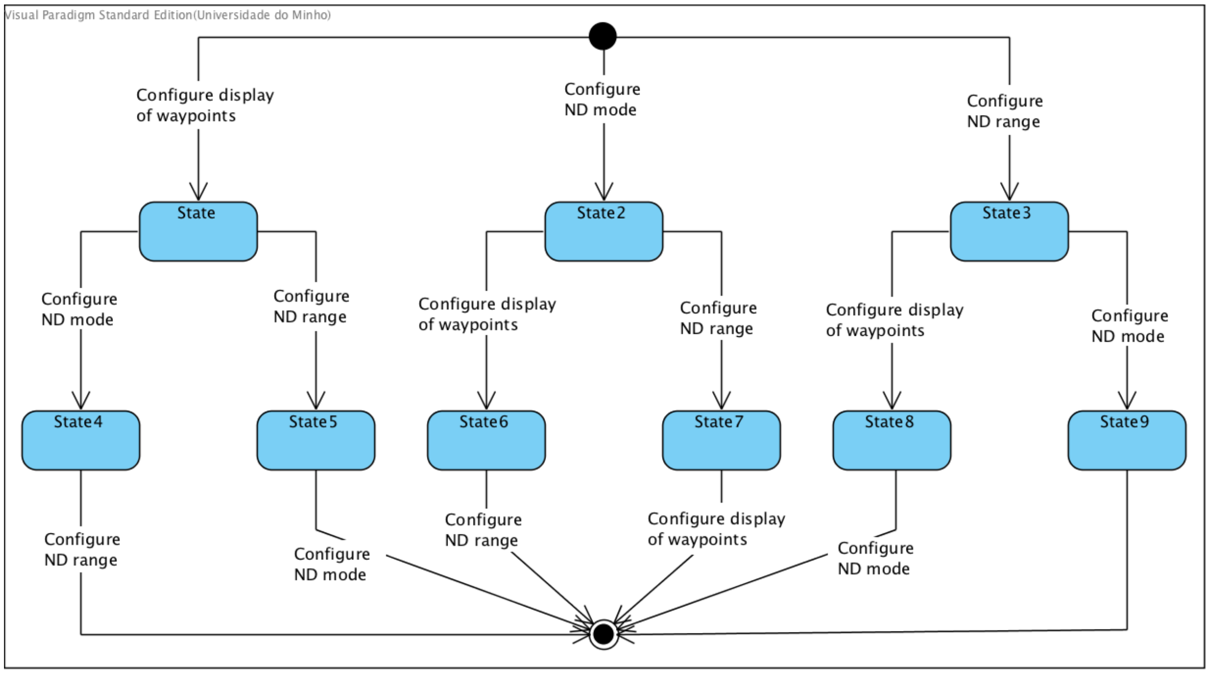


Fig. 5. Finite State Machine corresponding to the task model presented in Figure 4

operators. As before, the viability and approach to removing the interruption operator must take into consideration the concrete task and usage scenarios.

The presence of interruption operators in a HAMSTER task model can have one of two origins. Interruption operators are routinely used as a modeling pattern to control iterative behavior. This is the case of the model in Figure 6, where the task "Decide to stop configuring display of waypoints" is used to signal the end of the iteration over “Manage display of waypoints”. These tasks can be removed once the number of iterations of the iterative task is limited, using the Limit number of iterations strategy. The need to model the decision to stop becomes unnecessary if we also assume that tasks will always be repeated the maximum number of times.

Interruptions might, of course, also be due to the nature of the task. In that case, simply removing the operator might not be feasible. The option is to choose the most representative interruption conditions and include only those in the model. This amounts to defining equivalence classes w.r.t. interruptions and model only one concrete interruption per equivalence class. Once this is done, the interruption operator can be removed by adding the interrupting sub-task as an alternative in the location of the model where the interruption should be considered.

4.3.4 Replace Suspend-Resume Operators. This strategy is similar to the one for interruption operators just discussed. As in that case, representative instances of suspend-resume behaviors must be selected for testing. The main difference is that the suspend-resume operators are then removed by adding the suspend sub-tasks as optional sub-tasks and not as alternatives. This guarantees that the task might be executed without terminating the suspended tasks, and that these tasks will resume their course once the suspending sub-task ends.

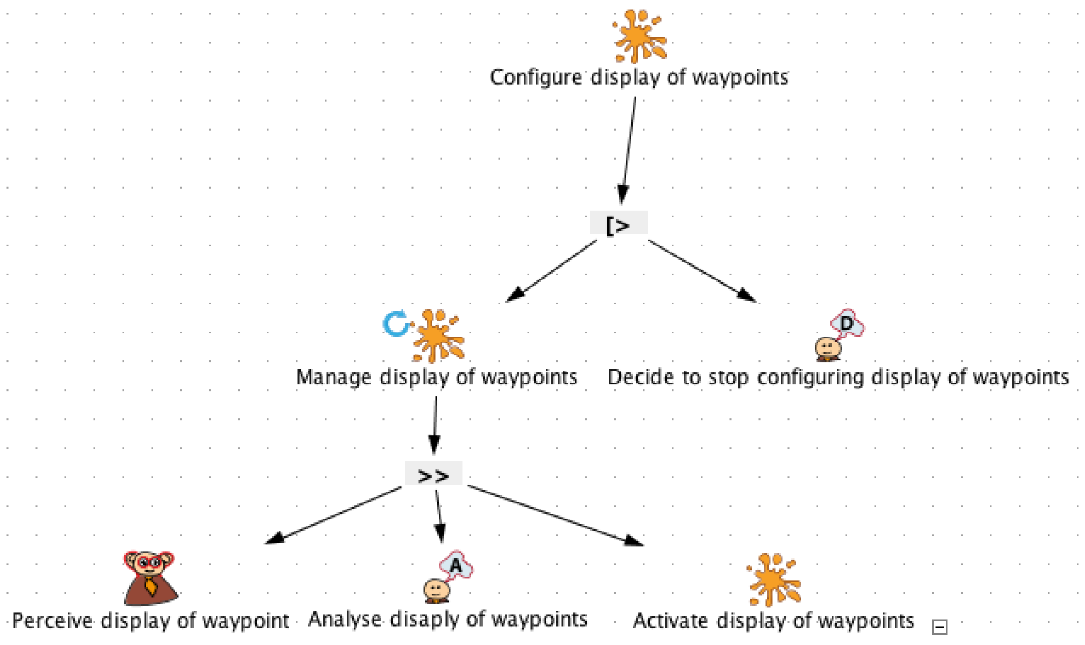

Configure display of waypoints
[>
Manage display of waypoints
Decide to stop configuring display of waypoints
>>
Perceive display of waypoint
Analyse disaply of waypoints
Activate display of waypoints

Fig. 6. Task model illustrating the use of an interruption operators

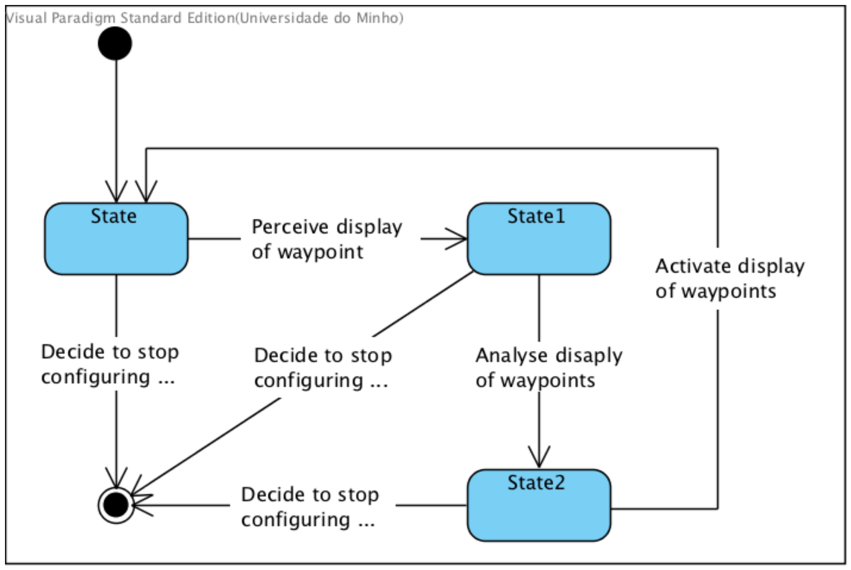

Visual Paradigm Standard Edition(Universidade do Minho)
State
Perceive display of waypoint
State1
Activate display of waypoints
Decide to stop configuring ...
Decide to stop configuring ...
Analyse disaply of waypoints
Decide to stop configuring ...
State2

Fig. 7. Finite State Machine corresponding to the task model presented in Figure 6

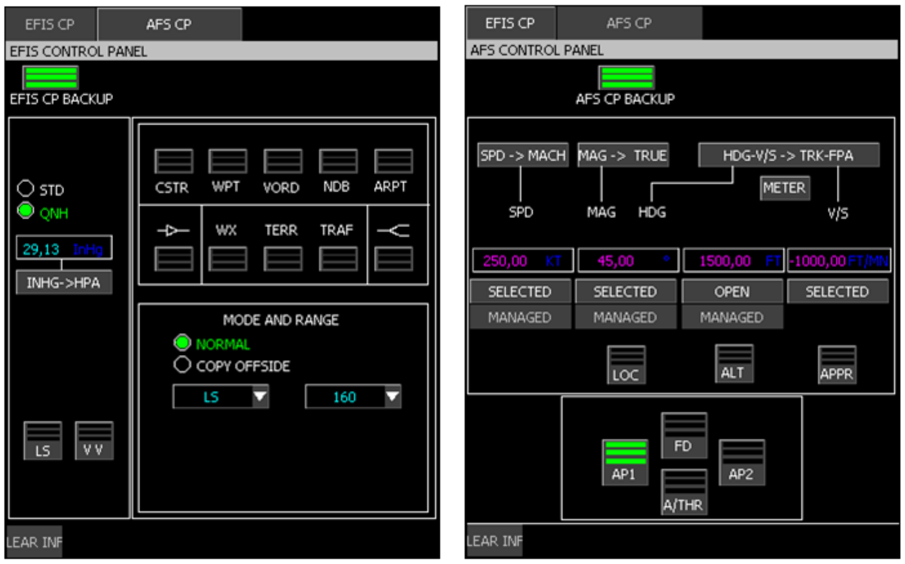


Fig. 8. The EFIS_CP (left side) and the AFS_CP (right side) pages composing the Flight Control Unit Software interactive application

5 APPLICATION ON A CASE STUDY

In this section we discuss the impact of applying the above strategies to an example: the task models of the Flight Control Unit Software (FCUS) system. First, details about the case study and its task modeling are provided. Then the results of applying the different strategies to the FCUS task model are described and discussed.

All test were run on a 2012 MacBook Pro laptop with an i7 processor and 8GB of RAM. CIRCUS was used to edit the task models and TOM (implemented in Java using JDK8) to generate the test cases. The algorithm for generating state machines from task models was manually applied and the result double checked by the task model authors.

5.1 The Flight Control Unit Software

This section first introduces the Flight Control Unit Software application and then describes the task modeling for the "Check for thunderstorms and avoid them if necessary" task that can be performed using this application.

5.1.1 Informal Presentation of the Flight Control Unit 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 (that are displayed in Figure 8):

- 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.

One of the activities that the pilot has to perform using the FCUS interactive application is to check the weather condition and verify if thunderstorms are on the flight route of the aircraft. This task can be done using the EFIS_CP to bring up the navigation display (ND) where whether radar information and the aircraft flight route are shown. If thunderstorms happen to be on the aircraft route, the pilots can avoid them by modifying (using the AFS_CP) the aircraft heading or its flight level. The task modeling for this goal is presented in next section.

5.1.2 Task Modeling. As presented above, the "Check for thunderstorms and avoid them if necessary" task is divided in two subtasks. First, the pilot has to check if the aircraft route is crossing any thunderstorms, and second, the pilot has to change the aircraft route (if necessary) in order to avoid thunderstorms.

Due to space limitations, we describe in this paper the "Check for thunderstorms" subtask only. Its HAMSTERS task model is presented in Figure 9. In order to accomplish this task, the pilot has first to decide to configure the navigation display ("Decide to configure ND to check for thunderstorms" cognitive user task in Figure 9). Then the pilot displays the EFIS_CP page if needed ("Display EFIS_CP" optional abstract task), and verifies that the weather radar is activated ("Ensure that weather radar is activated" abstract task). Finally, the pilot builds a mental model of the weather condition in front of the aircraft ("Build mental model of weather condition in front of the plane" abstract task). This last task consists of concurrently configuring the ND ("Configure ND" abstract task) and examining the map ("Examine map abstract" task). For configuring the ND, the pilot first displays the waypoints if needed ("Display waypoints on ND" optional abstract task, detailed in Figure 3). Then, after analyzing the ND image, s/he sets, in an order independent way, a new ND mode and a new ND range, if needed.

5.2 Applying the Strategies

The state machine generated from the full task model contained 92 states and 261 transitions. Generating scenarios directly from this task model proved unfeasible: even limiting the number of iterations to 1 (by setting up the algorithm to transverse every edge only once and each vertex at most twice), the Java Virtual Machine (JVM) run out of memory (due to exceeding the garbage collection overhead limit) after having generated over 368 thousand paths. Increasing the maximum memory allocation pool for the JVM to 4GB allowed the generation of more paths (over 740 thousand) but the JVM still run out of memory before the process ended.

In order to illustrate the use of the strategies proposed above, the following process will be followed: application of each relevant strategy, which results in the modification of the task model, and presentation of the results of its application in terms of the resulting state machines' size and number of generated test cases (where feasible). Presentation of the resulting state machines (similar to the ones presented in Figures 7 and 5) is omitted, due to space constraints. More specifically, we present, in the following subsections, first the cumulative application of the different operator modification strategies on the case study, and then the application of the non-interaction task removal strategies. The modifying task attributes strategy has been applied in all cases: for all of them, the number of iterations was initially restricted to one and then progressively lifted if feasible.

5.2.1 Modifying Operators on the Full Task Model. This section presents the cumulative application of the different operator modification strategies on the case study:

(1) *Replacing order independent operators.* First an execution order was defined for order independent tasks. This had, for instance, an impact on the last (rightmost) operator of the "Configure ND" task sub-tree in Figure 9. In this case, the decided order was that "Set ND mode" would always be done before "Set ND range". Over the full model, the change resulted in eliminating 21 states and 69 transitions from the state machine, which then featured 71 states and 192 transitions. However, the number of possible scenarios still exceeded what was feasible to generate.

(2) *Replacing concurrent operators.* The next step was to define an order to concurrent tasks. This was relevant, for example, to the "Build mental model of weather condition in front of the plane" abstract task in Figure 9.

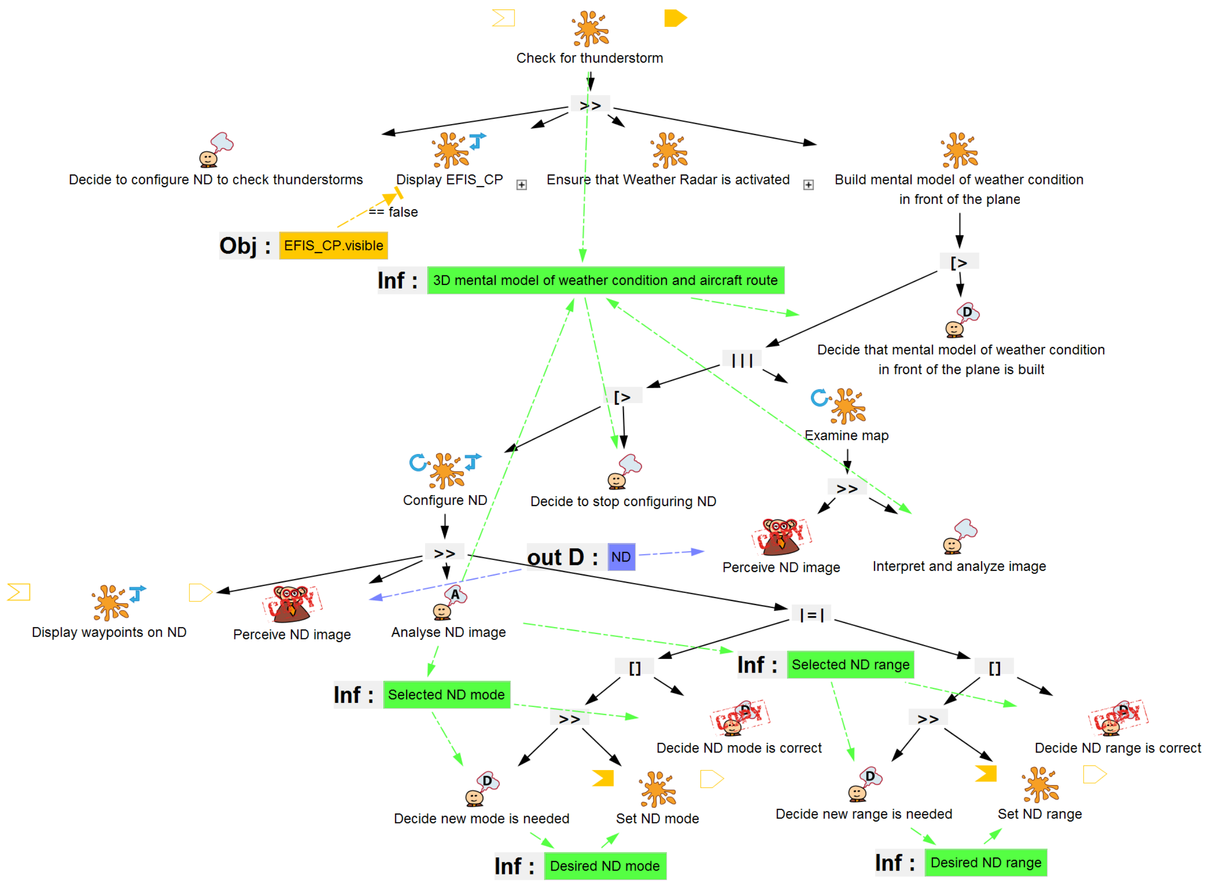

Check for thunderstorm
>>
Decide to configure ND to check thunderstorms
Display EFIS_CP
Ensure that Weather Radar is activated
Build mental model of weather condition in front of the plane
== false
Obj : EFIS_CP.visible
Inf : 3D mental model of weather condition and aircraft route
[>
Decide that mental model of weather condition in front of the plane is built
|||
[>
Examine map
Configure ND
Decide to stop configuring ND
>>
Perceive ND image
Interpret and analyze image
>>
out D : ND
Display waypoints on ND
Perceive ND image
Analyse ND image
|=|
[]
Inf : Selected ND range
[]
Inf : Selected ND mode
>>
Decide ND mode is correct
>>
Decide ND range is correct
Decide new mode is needed
Set ND mode
Decide new range is needed
Set ND range
Inf : Desired ND mode
Inf : Desired ND range

Fig. 9. Task model for the "check for thunderstorm" task

In that case, the decided order was that the "Configure ND" abstract task would be done before the "Examine map" abstract task. The resulting changes allowed for a further simplification of the state machine which now consisted of 67 states and 176 transitions. With this version of the model it become possible to generate scenarios if iterations were disallowed (by restricting the maximum number of visits to each vertex to 1). A total of 322608 unique paths were generated over the state machine. However, in the HAMSTERS notation, iterative tasks must always be executed at least once (unless they are also optional), so this configuration is in fact not valid. If iterations were allowed, even if only once, again the process could not be completed.

(3) *Replacing interruption operators.* The final step was to eliminate the interruption tasks (as no suspend-resume operator is present within the studied task model). As all uses of interruption were instances of the iterative modeling pattern, iterative tasks were considered to always execute their maximum allowed number of times. The impact of the change resulted in the removal of tasks such as "Decide to stop configuring ND". While this did not reduce the number of states, the number of transitions was reduced to 98. Hence, this proved to be the most effective strategy in terms of reducing the number of transitions in the states machine.

Overall, the accumulated effect of the changes was to reduce the state machine from 92 states and 261 transitions to 67 states and 98 transitions. With this machine, the tool was able to finish the generation process. With the maximum number of iterations restricted to one, the total number of paths generated was 27840. If a maximum of 2 iterations was allowed, the total number of paths raised to 240480.

5.2.2 Removing of Non-Interaction Tasks. This section presents the application of the two strategies for removing the non-interaction tasks:

(1) *Aggregating non-interaction tasks.* Aggregating non-interaction tasks considerably reduced the size of the original model to 70 states and 202 transitions. Applying the node strategies above allowed a further reduction to 51 states and 82 transitions. With this model, it was possible to generate 9984 and 48000 paths, for the 1 iteration and 2 iterations limits, respectively.

(2) *Complete removing of non-interaction tasks.* The final experiment was the complete removal of the non-interaction tasks from the model. This proved to be a non-trivial process, as it can considerably affect the structure of the task model.
Figure 10 depicts the task model for the "Check for thunderstorms" task without any non-interaction tasks. In this figure, we can see that the structure of the model is completely changed. The more important changes concern the "Ensure that weather radar is activated" and the "Build mental model of weather condition in front of the plane" abstract tasks in Figure 9: we can see that in Figure 10, these two tasks are replaced by the "Activate weather radar" and "Configure ND" abstract tasks. This is due to the fact that the original tasks (in Figure 9) contain a large number of non-interaction tasks. In the specific case of the "Build mental model of weather condition in front of the plane" abstract task, if we remove all the non-interaction tasks, the "Examine map" sub-task is deleted, as are all of the decision tasks leading to interruptions (e.g., "Decide to stop configuring ND" cognitive user task). When all these tasks have been removed, the "Build mental model of weather condition in front of the plane" abstract tasks in Figure 9 only contains the "Configure ND" abstract task. That's why, in Figure 10, the "Configure ND" task replaces the "Build mental model of weather condition in front of the plane" task of Figure 9.
In fact, the activity of completely removing non-interaction tasks also implied that the algorithm for generating the states machine had to be adjusted. As already mentioned, the interruption operator is routinely used to terminate iterations. Typically these iterative behaviors are terminated by a choice from the user *not to continue* the interaction (i.e., there is not necessarily an interaction action to indicate termination, but rather the user will engage in another sub-task or end the task). In these cases, once the non-interaction tasks are removed, there is no action in the model to indicate termination and, if the iteration happens at the end of the task, the algorithm is not able to generate an end state. since this state is needed for the generation of tests cases, the algorithm had to be adjusted to include an artificial end state. In terms of machine size reduction this proved the most effective approach as the generated states machine became 28 states and 59 transitions. With this machine, a total of 11264 and 63960 paths, for the 1 and 2 iterations limits, were generated.
The node strategies became of limited use here due to the structural changes that removing the non-interactive tasks caused in the model. Even so, applying them further educed the states machine to 24 states and 51 transitions. With this simplified version, the number of generated test cases decreased to 1584 and 9750 for for the 1 and 2 iterations limits. With three iterations, 90288 paths were generated.

6 DISCUSSION

The examples above show the feasibility of applying the defined strategies for reducing the number of test cases generated. What has not been addressed is the quality of the resulting evaluation (i.e. of the generated test cases set). Ultimately, the quality of the test cases is dependent on two aspects: the quality of the strategies per

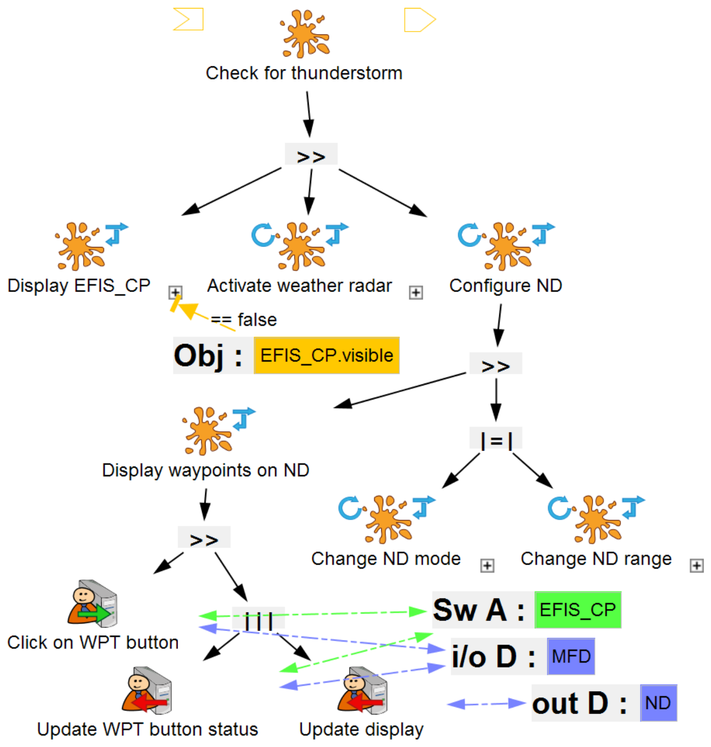


Fig. 10. Task model for the “check for thunderstorm” task without non-interaction tasks

se, and the concrete choices made when applying them. While the simplifications introduced by the strategies reduce the set of behaviors that are possible, thus the number of test cases that are generated, we argue that the approach guarantees, or at least makes it possible to guarantee, that the analysis will focus on the relevant interactions with the user interface under test. This happens because the task manipulations that are applied to the task model clearly identify which behaviors are being removed from the model (hence from the analysis), thus supporting the tester in guiding the testing process to those aspects of the interaction that are deemed more relevant. Additionally, once generation of all test cases becomes feasible, the process guarantees full coverage (up to some number of iterations of iterative tasks) of these relevant interactions.

Different combinations of the strategies were explored in the previous section. While the more aggressive non-interaction tasks removal strategies is the one that presents better results, it is the most complex to apply requiring considerable structural changes to the task model, which makes it more error prone. Hence, the best approach is to start the process with the more conservative non-interaction tasks aggregation strategy. Thus, the proposed process for applying the strategies that we presented is the following:

(1) Start by applying non-interaction tasks aggregation to the task model;
(2) Apply the concurrency and order-independence operators replacement strategies to the resulting model;
(3) Apply the interruption and suspend-resume replacement strategies;
(4) Run the algorithm starting with a low number of iterations and increasing as needed/feasible.
(5) If the algorithm is not able to ensure full coverage of the resulting model, apply the non-interactive tasks removal strategy.

Following this process it becomes possible to achieve full coverage of a clearly identified subset of the original task model. Defining this subset inevitably requires knowledge of the domain (e.g. about operation procedures and/or the operation of similar systems). This is the type of knowledge a tester should have in the first place. The goal of the approach is to help capture that knowledge in order to automate the testing process, making it more exhaustive and repeatable. In the present case TOM was used, but arguably the approach could be integrated into other testing frameworks (e.g. GUITAR).

Once abstract test cases have been generated, they can be reused on any system implementation. However, when the task model is updated, test cases must be regenerated. This means that a small change in the task model implies a complete repetition of the process. Avoiding this requires an incremental approach to test cases generation. If the difference between the previous and new models is determined, then only test cases for that part of the model that has changed need to be regenerated. Test cases not covering that part of the model can be reused.

While this strategies, and process, proved useful in the example above, some cautionary notes should be considered regarding the validity of the results. First, to gain confidence on the adequacy of the strategies proposed, more and more extensive case studies need to be carried out. It is particularly relevant that these cover different application domains. Indeed, the strategies were developed in the specific domain of interactive cockpits, so there is a risk that the approach is biased to this domain. We feel they are reasonably generic, but whether other domains' tasks might have characteristics that require additional of different strategies is still to be determined. Second, these further case studies need to take data input into consideration. At this point of the study, this aspect has not been addressed, and its effect on the process needs to be assessed. Finally, the strategies were developed using the HAMSTERS task modeling notation. It should be possible to adapt them to other hierarchical task modeling notations, for example CTT [29]. However, particularly for the strategies that have a structural impact on the models or exploit detailed data models, the effect of the difference in approaches to represent operators between the two task modeling notations must be analyzed.

7 CONCLUSION AND FUTURE WORK

Model-based testing supports the automation of GUI testing, but it is still the case that, in practice, only a subset of all possible interactions can be generated and tested. This paper was concerned with how to define a "more intelligent" test cases generation approach, so that test cases generation and execution is made possible by reducing their number, and an usage-centered selection of test-cases is favored.

To deal with this issue, the main contributions of this paper are: first, a set of task model manipulation strategies, and second, a process to apply them, which can be used to introduce simplifications into a HAMSTERS task model. These simplifications restrict the possible behaviors of the model to those deemed more relevant. By doing this, it becomes possible to ensure full coverage of the simplified task model. The advantage is that it becomes clear, during the tasks manipulation process, which behaviors are being considered inside the analysis, and which behaviors are left out of the analysis.

The paper opens a number of further research lines. First, the test case generation process must be extended to include input data. Once that is done, the strategies need to be revised to consider whether adjustments need to be made or new data specific strategies need to be included in the approach. Second, an incremental approach needs to be developed as this will further reduce the cost of the testing approach. A possible approach is to generate state machines for both task models and determine their difference, using this information to generate test cases for those parts that are different from the original. In order to make the paths complete (traversing the state machine from start to end), they can be complemented with the shortest path from the start state up to the point where the differences start. An alternative is to perform the difference on the task models. Determining the best approach is a topic for future work. Finally, while the generation of test cases from the state machine, and the co-execution of the corresponding scenarios against a SUT, are currently automated, tool support for two other aspects of the approach needs to be developed. One the one hand, the generation of the state machine from the task model. It is envisaged that this step can be fully automated. On the other hand, the manipulation of the task models. While this can already be done in CIRCUS, it would be useful to have a dedicated tool (or CIRCUS plugin) to support the application of the strategies.

ACKNOWLEDGMENTS

Work at HASLab was carried out in the context of 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] 1998. *ISO 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs) - Part 11 : Guidance on usability* (1 ed.). Technical Report. International Organization for Standardization, Geneva.

[2] Thomas A. Alspaugh, Debra J. Richardson, and Thomas A. Standish. 2005. Scenarios, State Machines and Purpose-driven Testing. In *Proceedings of the Fourth International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM '05)*. ACM, New York, NY, USA, 1–5. https://doi.org/10.1145/1082983.1083185

[3] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. 2011. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW '11)*. IEEE Computer Society, Washington, DC, USA, 252–261. https://doi.org/10.1109/ICSTW.2011.77

[4] Judy Bowen and Steve Reeves. 2011. UI-driven Test-first Development of Interactive Systems. In *Proceedings of the 3rd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '11)*. ACM, New York, NY, USA, 165–174. https://doi.org/10.1145/1996461.1996515

[5] José C. Campos, Camille Fayollas, Célia Martinie, David Navarre, Philippe Palanque, and Miguel Pinto. 2016. Systematic Automation of Scenario-based Testing of User Interfaces. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '16)*. ACM, New York, NY, USA, 138–148. https://doi.org/10.1145/2933242.2948735

[6] Stuart K. Card, Thomas P. Moran, and Allen Newell. 1980. The Keystroke-level Model for User Performance Time with Interactive Systems. *Commun. ACM* 23, 7 (July 1980), 396–410. https://doi.org/10.1145/358886.358895

[7] John P. Chin, Virginia A. Diehl, and Kent L. Norman. 1988. Development of an Instrument Measuring User Satisfaction of the Human-computer Interface. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '88)*. ACM, New York, NY, USA, 213–218. https://doi.org/10.1145/57167.57203

[8] Camille Fayollas, Célia Martinie, David Navarre, and Philippe Palanque. 2016. Engineering Mixed-criticality Interactive Applications. In *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '16)*. ACM, New York, NY, USA, 108–119. https://doi.org/10.1145/2933242.2933258

[9] P. M. Fitts. 1954. The information capacity of the human motor system in controlling the amplitude of movement. *Journal of Experimental PSychology* 74 (1954), 381–391.

[10] Andy Gimblett and Harold Thimbleby. 2010. User Interface Model Discovery: Towards a Generic Approach. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '10)*. ACM, New York, NY, USA, 145–154. https://doi.org/10.1145/1822018.1822041

[11] Yves Guiard. 2009. The Problem of Consistency in the Design of Fitts' Law Experiments: Consider Either Target Distance and Width or Movement Form and Scale. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09)*. ACM, New York, NY, USA, 1809–1818. https://doi.org/10.1145/1518701.1518980

[12] Marc Hassenzahl. 2004. Funology. Kluwer Academic Publishers, Norwell, MA, USA, Chapter The Thing and I: Understanding the Relationship Between User and Product, 31–42. http://dl.acm.org/citation.cfm?id=1139008.1139015

[13] ISO/IEC. 2010. *ISO/IEC 25010 - Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models*. Technical Report.

[14] Bonnie E. John and David E. Kieras. 1996. Using GOMS for User Interface Design and Evaluation: Which Technique? *ACM Trans. Comput.-Hum. Interact.* 3, 4 (Dec. 1996), 287–319. https://doi.org/10.1145/235833.236050

[15] Barbara A. Kitchenham, Tore Dyba, and Magne Jorgensen. 2004. Evidence-Based Software Engineering. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*. IEEE Computer Society, Washington, DC, USA, 273–281. http://dl.acm.org/citation.cfm?id=998675.999432

[16] Valéria Lelli, Arnaud Blouin, Benoit Baudry, and Fabien Coulon. 2015. On Model-Based Testing Advanced GUIs. In *Proceedings of the 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW) – Workshop on Advances in Model Based Testing (A-MOST 2015)*. Graz, Austria. https://doi.org/10.1109/ICSTW.2015.7107403

[17] Jane Li, Laurie Wilson, Stuart Stapleton, and Patrick Cregan. 2006. Design of an Advanced Telemedicine System for Emergency Care. In *Proceedings of the 18th Australia Conference on Computer-Human Interaction: Design: Activities, Artefacts and Environments (OZCHI '06)*. ACM, New York, NY, USA, 413–416. https://doi.org/10.1145/1228175.1228261

[18] Célia Martinie, David Navarre, Philippe Palanque, and Camille Fayollas. 2015. A Generic Tool-supported Framework for Coupling Task Models and Interactive Applications. In *Proceedings of the 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '15)*. ACM, New York, NY, USA, 244–253. https://doi.org/10.1145/2774225.2774845

[19] Célia Martinie, Philippe Palanque, and Marco Winckler. 2011. *Structuring and Composition Mechanisms to Address Scalability Issues in Task Models*. Springer Berlin Heidelberg, Berlin, Heidelberg, 589–609. https://doi.org/10.1007/978-3-642-23765-2_40

[20] Atif M. Memon. 2001. *A comprehensive framework for testing graphical user interfaces*. Ph.D. University of Pittsburgh.

[21] Atif M. Memon, Mary Lou Soffa, and Martha E. Pollack. 2001. Coverage Criteria for GUI Testing. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-9)*. ACM, New York, NY, USA, 256–267. https://doi.org/10.1145/503209.503244

[22] Rodrigo M.L.M. Moreira and Ana C.R. Paiva. 2014. PBGT Tool: An Integrated Modeling and Testing Environment for Pattern-based GUI Testing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, USA, 863–866. https://doi.org/10.1145/2642937.2648618

[23] Inês Coimbra Morgado. 2016. *Automated Pattern-Based Testing of Mobile Applications*. Ph.D. Faculdade de Engenharia da Universidade do Porto.

[24] Glenford J. Myers and Corey Sandler. 2004. *The Art of Software Testing*. John Wiley & Sons.

[25] David Navarre, Philippe A. Palanque, Fabio Paternò, Carmen Santoro, and Rémi Bastide. 2001. A Tool Suite for Integrating Task and System Models Through Scenarios. In *Proceedings of the 8th International Workshop on Interactive Systems: Design, Specification, and Verification-Revised Papers (DSV-IS '01)*. Springer-Verlag, London, UK, UK, 88–113. http://dl.acm.org/citation.cfm?id=646166.680841

[26] Bao N. Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: an innovative tool for automated testing of GUI-driven software. *Automated Software Engineering* 21, 1 (2014), 65–105. https://doi.org/10.1007/s10515-013-0128-9

[27] Simeon Ntafos. 1998. On Random and Partition Testing. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '98)*. ACM, New York, NY, USA, 42–48. https://doi.org/10.1145/271771.271785

[28] Ana C. R. Paiva, João C. P. Faria, and Pedro M. C. Mendes. 2008. *Reverse Engineered Formal Models for GUI Testing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 218–233. https://doi.org/10.1007/978-3-540-79707-4_16

[29] Fabio Paternò. 2004. ConcurTaskTrees: An Engineered Notation for Task Models. In *The Handbook of Task Analysis for Human-Computer Interaction*, Dan Diaper and Neville Stanton (Eds.). Lawrence Erlbaum Associates, Chapter 24, 483–501.

[30] Carmen Santoro. 2005. *A task model-based approach for design and evaluation of innovative user interfaces*. Ph.D. Presses universitaires de Louvain.

[31] J.C. Silva, C. Silva, R. Goncalo, J. Saraiva, and J.C. Campos. 2010. The GUISurfer tool: towards a language independent approach to reverse engineering GUI code. In *Proceedings of the 2nd ACM SIGCHI Symposium on Engineering interactive computing systems*. ACM, 181–186. http://doi.acm.org/10.1145/1822018.1822045 ISBN: 978-1-4503-0083-4.

[32] J. L. Silva, J. C. Campos, and A. Paiva. 2008. Model-based user interface testing with Spec Explorer and ConcurTaskTrees. *Electronic Notes in Theoretical Computer Science* 208 (2008), 77–93. http://dx.doi.org/10.1016/j.entcs.2008.03.108

[33] Wan-Hui Tseng and Chin-Feng Fan. 2013. Systematic Scenario Test Case Generation for Nuclear Safety Systems. *Inf. Softw. Technol.* 55, 2 (Feb. 2013), 344–356. https://doi.org/10.1016/j.infsof.2012.08.016

[34] Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-based Testing Approaches. *Software: Testing, Verification and Reliability* 22, 5 (Aug. 2012), 297–312. https://doi.org/10.1002/stvr.456

Received January 2017; revised May 2017; accepted May 2017