# Utilizing User Interface Models for Automated Instantiation and Execution of System Tests

Benedikt Hauptmann
Technische Universität München, Germany
benedikt.hauptmann@in.tum.de

Maximilian Junker
Technische Universität München, Germany
maximilian.junker@in.tum.de

## ABSTRACT

Scripts for automated system tests often contain technical knowledge about the user interface (UI). This makes test scripts brittle and hard to maintain which leads to high maintenance costs. As a consequence, automation of system tests is often abandoned.

We present a model-driven approach that separates UI knowledge from test scripts. Tests are defined on a higher level, abstracting from UI usage. During test instantiation, abstract tests are enriched with UI information and executed against the system. We demonstrate the application of our approach to graphical UIs (GUIs) such as rich clients and web applications. To show the feasibility, we present a prototypical implementation testing the open-source application Bugzilla.

## Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.9 [**Software Engineering**]: Management—*Software quality assurance*

## General Terms

Verification, Design

## Keywords

Testing, User Interface, Model-Based

## 1. INTRODUCTION

Testing is a central activity for quality assurance. The system under test (SUT) is executed with the intention to find errors as well as to gain confidence that it works as intended.

*System tests* execute the whole system to check whether it fulfills its functional requirements. The focus is on the observable input/output behavior rather than on the structure or the internal state of the system or on timing aspects [1]. For systems with user interfaces (UI) this usually means that system tests are executed against the UI. The execution of UI-based tests is easy to perform by humans. Their brainpower, experience and intuition facilitate them to interpret high level descriptions (e.g., *activate the human resource module*) without the need of detailed UI specific information.

If system tests have to be executed repeatedly, for example for regression testing, *test automation* can be very efficient [7, 10, 3]. To automate system tests, UI specific information has to be included in the *test scripts*. As such information may change when the software evolves (e.g., a button is moved to a different dialog), automatically executable test scripts tend to be fragile and need to be maintained often [2, 22]. This causes considerable costs and the decision whether and when tests should be automated highly depends on the maintenance effort for the test scripts [7, 6]. The following problems motivate an efficient and flexible way of test instantiation and execution for systems with UIs.

*1. Maintenance of tests:* I the case of changes to the UI, test scripts tend to be fragile because of the mixture of functional (e.g., providing a certain input value) and technical aspects (e.g., clicking on a specific button). This makes it difficult to adapt test scripts when the SUT changes.

*2. Reuse between tests:* The same functionality is usually tested with different inputs or in different variants. Furthermore, different tests may use the same parts of the user interface. A main obstacle for maintainable tests is redundancy, or put differently, the low level of information reuse within tests. There are several approaches addressing those issues (e.g., keyword-driven testing) [7]. However, using these approaches, tests still contain much technical information and the potential for reuse is not exploited.

We propose a model-driven approach to separate tests from UI related information. Tests are defined on the functional level, abstracting from UI interactions. During test execution, abstract tests are enriched with UI information and are executed against the system.

The remainder of this paper is structured as follows. In Section 2, related work is reviewed. In Section 3, we present our model-based approach to abstract UIs in testing. In Section 4, we show how our approach can be applied to graphical UIs by introducing a suitable meta-model. In Section 5, we present a prototypical implementation showing models for the open-source application Bugzilla. In Section 6, we discuss our approach based on a changing scenario. In Section 7, we conclude and give an outlook on future works.


*ETSE*'11, July 17, 2011, Toronto, ON, Canada

## 2. RELATED WORK

Improving maintainability of test scripts has been discussed repeatedly. Two popular concepts are *data-driven testing (DDT)* and *keyword-driven testing (KDT)* [7] which both raise the abstraction level of test scripts to enhance the level of reuse. In the latter, to write abstract tests, action words are used which are mapped to test script snippets. Our approach is similar to KDT but models the relevant parts of the UI instead of defining executable scripts. With this, we expect simpler maintenance and better reuse.

The open-source testing framework Tellurium[1] uses UI models to reduce the effort of creating test scripts snippets in KDT. However, several UI concepts are mixed in one model. Our approach aims to provide one comprehensive UI model, containing all aspects of UIs on dedicated abstraction levels. By this separation, we expect that the UI model can be performed easier.

In [9], tool-support is used to ease the maintenance of test scripts for GUIs. Differences between GUIs are automatically analyzed and the affected parts of the test scripts are detected. However, this approach still focuses on low level test scripts and does not exploit the full potential for test reuse.

*Model-based testing (MBT)* is concerned with testing software using models. However, the focus of most works is on test case generation, which we do not target. To generate test cases for GUIs, many approaches exist [14]. For example, [20, 15] use variants of finite state machines (FSMs) which are very detailed and complex to create and maintain. In our work, we want to keep the effort for the test designer as low as possible by reducing the necessary models to just that information required for test execution.

Although it is acknowledged that the gap between abstract test cases and the system needs to be bridged [23], test instantiation in MBT is not covered in detail in most works. Most MBT approaches in literature are applied to systems with simple UIs, for example, embedded systems or chip cards. Mostly, there is a direct mapping between abstract actions and the code that implements that action. This is true, for example, for approaches that build on SpecExplorer [24, 15, 17]. Katara et al. [13] implement a layered approach for test case execution.

In [8] a model-driven approach which stepwise enriches tests with UI information is presented. However, since they focus only on infotainment systems of cars, they are limited in their field of application.

We already presented the idea of separating tests based on functional and UI related concerns in [11]. In the current paper, we continue this work and show how this basic idea can be applied using the example of graphical UIs. We adapt concepts from model-based UI development and present a meta-model designed for the needs of test instantiation (Section 4). Furthermore, we show the feasibility of the approach by giving a concrete example testing an open-source application (Section 5).

## 3. OVERVIEW

To analyze the mentioned problems, we make use of a conceptual UI model introduced in [25] (see Figure 1). The model separates an interactive system into two logical parts: the *application* and a *mediator*. The application realizes the desired functionality and is independent from any UI related concepts. The mediator is the intermediary between the functionality (realized by the application) and the user. Furthermore, the mediator forms the interface at the system boundary. It is built to be used by a certain type of user and is therefore optimized for it. A system may also have several mediators for several types of users (e.g., a GUI and a web service interface).

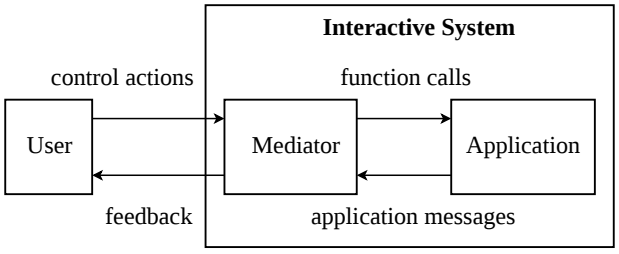


**Figure 1: Conceptual UI model (based on [25]).**

For example, a telephone answering device typically has a UI containing a display and several buttons to be used in a physical way. The same machine may also be used remotely by voice control. Both of the UIs (mediators) will trigger the same functionality (the application), even though they are used in different ways.

In the sense of this model, an automated test is a user too. However, the mediator is typically not optimized to be used by a test. An easy solution would be to extend the system with another mediator, optimized for automated testing (e.g., a special testing interface). This is in conflict with the paradigm of system testing which is to test the complete system in a black box way (i.e., including the mediator).

Since we do not want to adapt an SUT for automated testing, we have to find a way to ease automated communication with the SUT.

Our approach aims to separate tests into functional and mediator specific concepts to improve their reuse and make them insusceptible to changes to the system's UI. We split tests into two artifacts. (1) Pure test logic (the actual *test cases*) which has no dependency to any information of the UI, and (2) the pure *UI knowledge* which is necessary to execute the tests (see *Test Cases* and *UI Model* in Figure 2). For the latter, we suggest a descriptive model. To execute these UI independent test cases, we reconstruct the necessary information by using a generic *test adapter* which instantiates test cases using the UI information stored in the UI model for a given UI (see *Test Adapter* in Figure 2).

### 3.1 Test Modeling

Considering UI specific issues during test case creation can be very distracting. By ignoring these concerns, the test engineer can focus on creating good tests. Furthermore, since appropriate test case specification techniques can be used, the test engineer does not need to have programming skills. We envision a description technique that largely uses natural language. However, as the test cases will focus on inputs and outputs, tabular notations may be suitable for certain contexts as well. We expect that a test case specification language which ignores UI related information helps the test engineer to be more efficient and to create better tests.

[1] `http://code.google.com/p/aost/`

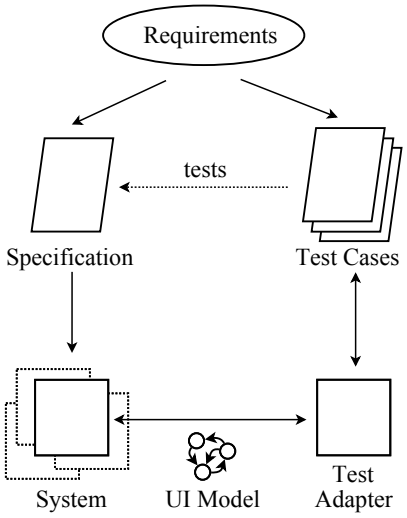


**Figure 2: Overview of the approach.**

Tests without UI specific information can be reused for different variants of an application implementing the same functionality. This is especially relevant in the context of software product lines where UI independent tests can be reused for different products of the same product line. Furthermore, if a system contains several UIs (for example a web interface and a rich client), tests can be reused for all of them.

Considering software evolution, many slightly different versions of the same system will exist. Even though the system is modified over time, most of the functionality will remain constant. UI independent tests will be valid if the UI changes.

## 3.2 UI Modeling

By keeping all UI related information in a dedicated model which is separate from tests, competences between test creation and test instantiation can be allocated more explicitly. Whereas a test engineer can focus on creating test cases, a test execution engineer is responsible for creating and maintaining the UI model.

If tests have to be performed for several instances or versions of a system, instead of copying and adapting every single test, just a corresponding UI model has to be created and the tests can be reused.

To support maintenance of tests, we suggest dedicated, descriptive models to store UI related information. Compared to conventional test scripts where functional test logic and UI information are mixed, having a single artifact where all UI related data are kept centrally makes changes to the UI easier to handle.

## 3.3 Test Case Instantiation and Execution

The transition from test cases to executable test scripts is called *test instantiation* [18]. Abstract test cases are *concretized* to concrete interactions with the system and the responses of the system are *abstracted* back to the level of the test cases and compared with the expected results. Generally, two different approaches exist. Test cases are either *transformed* into executable test scripts by adding the necessary information of the system's interface, or an *adapter* is built which interprets the test cases and dynamically performs the necessary actions on the system interface during test execution.

Test scripts are created before test execution, for example, using a capture/replay tool. They therefore have limited options to react dynamically on the system's behavior during test execution. Before the execution of a test has been started, every single detail of the behavior of the UI has to be foreseen. A test adapter, however, can react dynamically on the system's behavior during execution and therefore does not need to know every response of the UI in advance. For example, after an action has been triggered on the UI, a test script needs to exactly know if this action has led to a dialog switch and if yes, which the new active dialog is. A test adapter, however, can detect a potential dialog transition during execution and can calculate the necessary UI interactions to perform the next commands of the test case on the fly. Therefore, using test adapters, less information of the dynamic behavior of the UI is necessary for the test execution which again leads to smaller UI models and reduced maintenance effort.

In our approach, a generic test adapter interprets test cases and loads the necessary UI information from a given UI model. Using this adapter, test cases can be reused for different UIs by switching the UI model for the corresponding UI.

# 4. MODELING AND ADAPTING GUIS

The approach introduced so far is not limited to graphical UIs, but can also be applied to other forms of UIs like textual or voice UIs. However, in the following sections, we will focus on how to create models and adapters for graphical interfaces like in web or desktop applications. We introduce a multi-layered architecture for test adapters as well as a suitable meta-model.

To handle the complexity of sophisticated UIs, we divide the test instantiation into several parts. We propose a proceeding which is oriented towards multi-layered *communication abstraction* [19], as in network communication stacks (e.g., the ISO/OSI reference model). We use several layers, stepwise abstracting from the usage of the UI. Every layer reduces typical UI concepts like concrete widgets, dialogs or navigation, and provides a more abstract view of the system's interface (see *System Interface*, $\textit{System Interface}_{layer1}$, ... in Figure 3). These virtual, more abstract SUT interfaces are the base for the next layers which will reduce further UI concepts. This is repeated until all UI related concepts are removed and a virtual, completely UI independent SUT interface has been created. Test cases are defined on this, most abstract layer using the interactions provided by this UI independent SUT interface. To execute tests, on each layer, models hold the extracted UI information. The adapter uses these models to concretize tests respectively abstract the system's response from one layer to another and finally executes the tests against the actual SUT (see ..., $\textit{Test}_{layer2}$, $\textit{Test}_{layer1}$, *Test* in Figure 3).

## 4.1 System

As topmost layer, we introduce the *Abstract System Model* which will form the abstracted interface for testing of the SUT. It consists of the *abstract system interactions* a user can perform with the system. An abstract system interac-

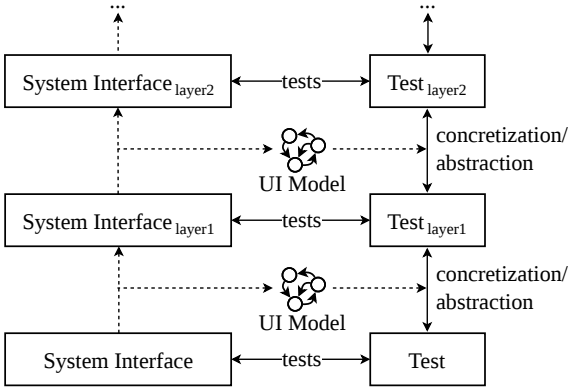


**Figure 3: Multi-layered communication abstraction.**

tion can be an *input*, *output*, or *action* and is independent from any concrete UI implementation. In our approach, the abstract system model has two roles. From the perspective of test creation, it forms the interface of the SUT against which tests are created (see left of Figure 4). Secondly, it forms the starting point for the creation of the UI model.

## 4.2 GUI Modeling

In this section, we focus on how to model graphical UIs as they exist on desktop PCs or web applications. For the UI model, we have the following goals in mind:

*Ease of creation:* UI model creation should not hinder the testing process. Therefore UI models have to be easy and quick to create. We address this by providing a meta-model for GUIs which works as a framework for the creation of UI models and is the base for tool support.

*Partial and incremental modeling:* Only the parts of the UI which are relevant for testing should have to be modeled. Parts of the UI which are not affected during testing are not relevant and therefore should not have to be modeled. Only necessary UI information has to be modeled (see Section 3.3). If new tests are created, the existing UI model can be expanded with the further needed parts of the UI.

*Flexible to changes to the GUI:* Changes to the GUI should be easy to reproduce in the UI model. We addressed this by reducing redundancy in the UI model. Every single piece of information is stored only once. All connections within the model are realized using references. Furthermore, we designed our meta-model based on typical concepts of GUIs. If the UI changes, it is easy to find the according parts in the model and perform the changes.

To create suitable UI models, we first give an overview on Model-Based UI Development, a research area of software engineering which aims to create UIs based on UI models. After that, inspired by MBUID, we introduce our own UI meta-model optimized for test instantiation.

### 4.2.1 Model-Based UI Development (MBUID)

*Model-Based User Interface Development (MBUID)* [21, 5, 12] is a set of concepts and models from the research area of human-computer interaction (HCI) and software engineering which aims to design and develop UIs based on models. MBUID introduces a forward engineering approach to stepwise transform UI models by systematically enriching them with design and layout information.

Typically, the starting points of an MBUID are *Task Models* and *Domain Models* which are independent from UI design and layout specific concepts. Task Models describe the tasks a user can perform with a software system. A common notation is *ConcurTaskTree* [16] which represents tasks in terms of abstract human-computer interactions (input, output, action triggered, ...). Tasks can be hierarchically decomposed into subtasks and connected with temporal relationships (e.g., two tasks are concurrent or one task enables another task). Domain models are not limited to UI design, but are also used in other fields of software engineering. They describe the structure of the application (e.g., in terms of a UML class diagram).

Task and domain models are transformed into an *Abstract User Interface (AUI)*, dealing with layout concepts such as dialogs, navigation and abstract widgets. These abstract widgets are highly abstract UI elements, for example, abstract buttons or text field elements. AUIs already define the basic layout of the dialogs but are still independent from any implementation specific concept like the target platform or the GUI framework.

The abstract layout defined in the AUI is transferred to concepts of the target platform. This so called *Concrete User Interface (CUI)* contains implementation details necessary to automatically generate the UI.

Borrowing from MBUID, we present a meta-model to create UI models for GUIs. It addresses, but is not limited to, web and desktop applications. Performing minor adaptations, this meta-model also fits to other graphical, dialog-based UIs like they occur in mobile or multi-touch applications. The meta-model contains three sub models which build upon each other. Each sub model enriches the previous sub model by adding further UI related concepts (see Figure 4).

Compared to MBUID, we are not interested in modeling every single detail of the GUI. For test instantiation, the focus is on controlling an already existing GUI. Therefore, our models differ from the ones from MBUID. For example, our most abstract model, the system model, just deals with the system's abstract interactions and does not focus on their relationships as task models in MBUID do. Furthermore, since we are interested in an understandable, easy to create, and maintainable tracing from abstract system interactions to GUI widgets, we cut the AUI into two parts, the cluster and the abstract page model.

### 4.2.2 Cluster Model

The *cluster model* enriches the system model with the concept of *clusters*. Inspired by [4], clusters group abstract system interactions (see *Interaction* in Figure 4) which are visible to the user at the same time. Those clusters form the basis for the presentation units of the later UI. Furthermore, the interactive dialog structure is modeled as *Navigations* referencing all clusters which are navigable by a user (or test). As concept of reuse, we added a second reference to other clusters modeling *includes*. They represent sub-clusters like tabs or side bars which can be activated, deactivated and reused among clusters.

Although an abstract system interaction exists just once in the abstract system model, it can appear in several clusters in a UI. Therefore, there is a $1 : n$ relationship between abstract system interactions and clusters.

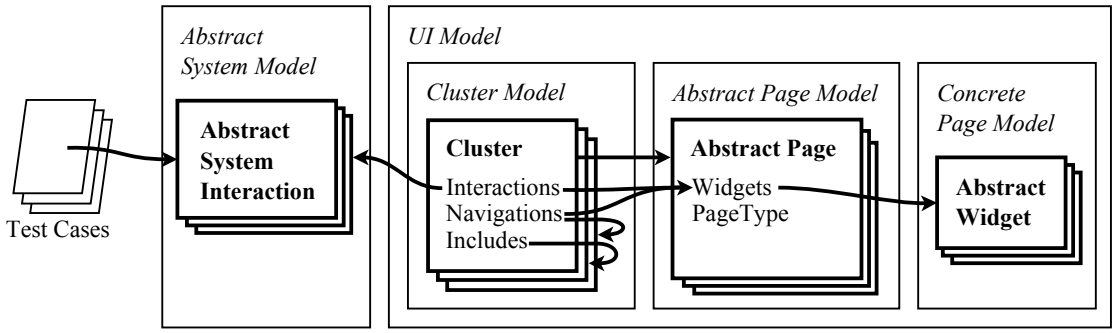


**Figure 4: Meta-model for graphical UIs (GUIs) (arrows represent references to other elements).**

#### 4.2.3 Abstract Page Model

Looking at modern GUIs, there is often no 1 : 1 relationship between abstract system interactions as we modeled them in the abstract system model and widgets at the UI. Very often, the same interaction is represented multiple times on the same dialog. For example, in web applications, headers and footers containing the same content are a very common layout pattern. But also the other way round occurs frequently. Several interactions can be realized using a single GUI widget. Date inputs, for example, can be seen as functional disjunctive inputs for day, month and year. On the implementation level however, they can be realized using a single calendar widget. Furthermore, there are different ways a dialog may be implemented, for example as a window or as view in a rich client application. We reflect those possibilities via a parameter *PageType*.

We extend the GUI model to an *Abstract Page Model* containing *Abstract Pages*. Every abstract page maps to one cluster from the cluster model and lists *Abstract Widgets* from the concrete UI. Between abstract system interactions referenced by the cluster model and abstract widgets from the abstract pages, there is an $n : m$ relationship.

#### 4.2.4 Concrete Page Model

The *Concrete Page Model*, the last sub-model, bridges the gap between the abstract widgets and the actual UI. Abstract widgets are mapped to concrete instances of UI widgets in the actual UI, for example, a certain text input field on the main window. This mapping should be based on a UI platform specific test framework.

## 5. PROOF OF CONCEPT

To demonstrate the feasibility of our approach, we created a prototypical implementation in Java. As test object, we chose the open-source application Bugzilla[2], a web-based general-purpose bug tracking system. It is a grown system and many old versions are still available with which we can simulate an evolving system. As system specification, we used the official Bugzilla manual[3]. Even though the manual contains many GUI related details, the functional concepts are clearly recognizable.

In the rest of this section, we will show how our approach can be applied and which artifacts have to be created. We will focus on the function *reporting a new bug* as it is described in section 5.6.1 of the Bugzilla manual. The user has to click on the link *new* of the navigation panel. After selecting a product, several attributes of the new bug have to be set (e.g., the product's component, a summary, a description, ...). At the end, the user has to confirm the new bug by clicking on a *submit* button.

### 5.1 Tests and the Abstract System Model

Analyzing the Bugzilla manual, abstract interactions can be elicited by searching for data exchanged between the user and the system as well as actions triggered by the user. Table 1 shows the part of the *Abstract System Model* necessary to report a new bug.

**Table 1: The abstract system model for Bugzilla**

| | Abstract System Interactions |
|---|---|
| Inputs: | Product, Component, Summary, Description, Search String, Select Result, ... |
| Outputs: | Product, Component, Summary, Description, ... |
| Actions: | Submit Bug, Search, ... |

Using the abstract system model, test cases for the bug reporting function can be created by referencing the elicited abstract system interactions. Table 2 shows a simple test in table notation. Every row represents a system interaction having a type (input, output or action), an interaction and value which has to be applied. This table is interpreted by our adapter (introduced in Section 5.3) which will execute the test line by line.

### 5.2 GUI Modeling

To reduce unnecessary GUI modeling, we perform a reverse engineering approach. We execute test cases manually and model all parts of the GUI which have been used during the execution. Using this bottom up procedure, we have to model just the parts of the UI that are necessary for the execution of the tests.

#### 5.2.1 Concrete Page Model

For the concrete page model, we make use of the GUI automation framework Selenium[4]. We chose Selenium as it is open-source and freely available. Since our approach is

[2] `http://www.bugzilla.org`
[3] `http://www.bugzilla.org/docs/3.6/`
[4] `http://www.seleniumhq.org`

**Table 2: An example test case for Bugzilla.**

| Type | Abstr. Inter. | Value |
|---|---|---|
| | ... | |
| Set Input | Product | `'Test Product'` |
| Set Input | Component | `'Test Component'` |
| Set Input | Summary | `'Test Bug'` |
| Set Input | Description | `'This is a Test Bug'` |
| | ... | |
| Perform Action | Submit Bug | — |
| Set Input | Search String | `'Test Bug'` |
| Perform Action | Search | — |
| Set Input | Select Result | `1` |
| | ... | |
| Expect Output | Product | `'Test Product'` |
| Expect Output | Component | `'Test Component'` |
| Expect Output | Summary | `'Test Bug'` |
| Expect Output | Description | `'This is a Test Bug'` |
| | ... | |

independent of the GUI technology used other automation tool will work as well. Selenium provides a capture/replay tool recording repeatable test scripts, as well as a programming framework for several programming languages (including Java) to manually write tests. Using this programming framework, we wrote a reusable set of wrapper classes abstracting from the usage of all required HMTL widget types. With this wrapper classes, the concrete page model is represented by a mapping from abstract widgets to HTML widget types and a Selenium specific identification attribute (*Selenium Target*). Table 3 shows a subset of the HTML widgets used in the test introduced before (Table 2).

**Table 3: The concrete page model for Bugzilla.**

| Abstract Widget | HTML Widget | Selenium Target |
|---|---|---|
| Product Selection | Link List | `'//table'` |
| Component Selection | Selection | `'id=component'` |
| Summary Input | Input Field | `'id=short_desc'` |
| Description Input | Input Field | `'id=comment'` |
| Submit Bug Button | Button | `'id=commit'` |
| Component Text | Text | `'//div[2]/form/table/tbody/tr/td/table/tbody/tr[4]/td[2]'` |
| | ... | |

### 5.2.2 Abstract Page Model

In the abstract page model, we group the abstract widgets from the concrete page model in pages. To reduce redundant modeling, our meta-model allows reusing recurring UI parts by including other clusters (Section 4.2.2). For example, each page at the Bugzilla web application contains the same navigation links (header and footer). To avoid modeling these links at every single page, we create a (fictive) page *Navigation* containing all common widgets of the used pages. This fictive page is included in other pages in the cluster model. Table 4 shows a subset of the abstract page model for the test introduced in Table 2. We omit the *PageType* parameter as in this case all pages have the same type.

**Table 4: The abstract page model for Bugzilla.**

| Abstract Page | Abstract Widgets |
|---|---|
| Home | — |
| Navigation | Home Link, New Link, Search Link |
| Select Product | Product Selection |
| New Bug | Component Selection, Summary Input, Description Input, Submit Button, ... |
| | ... |

### 5.2.3 Cluster Model

So far, the concrete and abstract page model abstracted the Bugzilla web application to pages containing abstract widgets. In the cluster model, these concepts are brought together with the abstract system interactions from the abstract system model (see Figure 4). The cluster model contains the following information: (1) The navigation between and inclusion of clusters, (2) the abstract system interactions represented by a cluster, and (3) the mapping between abstract system interactions and abstract widgets.

Figure 5 shows the navigation and inclusion relationship of the example introduced before. The dotted edges represent cluster inclusions. The cluster *Navigation*, for example, contains the navigation area which exists on every page and therefore is included in every cluster. Each solid edge represents a navigation link between pages. The labels on the edges bind navigation to an abstract widget. For example, by using the abstract widget *Home Link*, a navigation to the cluster home can be performed.

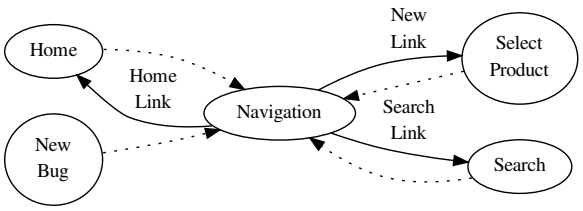


**Figure 5: The navigation between clusters of Bugzilla.**

All abstract widgets have one of the following functions. They can either be directly mapped to an abstract system interaction from the abstract system model (input, output, or action), or they are auxiliary interactions to control the GUI (e.g., a navigation link). Since the latter is UI dependent, it is not part of the abstract system interface.

Table 5 and 6 show the abstract system interactions and their mapping to the abstract page model for the example from Figure 5.

## 5.3 The Adapter/Execution

Our adapter is able to interpret tests using a given UI model. It stores the cluster that represents the current state of the UI (a specified start cluster at the beginning of a test). To perform an abstract interaction on the UI our adapter executes the following steps. (1) If the current active cluster does not contain the needed abstract interaction, the adapter searches a cluster which offers the desired interaction. A breadth first search is performed in the model, starting at the current cluster and following the navigation

**Table 5: The first part of the cluster model for Bugzilla. It shows the mapping to the abstract pages.**

| Cluster | Abstract Page |
|---|---|
| Home | Home Page |
| Navigation | Navigation Page |
| Select Product | Product Page |
| New Bug | New Bug Page |
| Search | Search Page |
| ... | |

**Table 6: The second part of the cluster model for Bugzilla. It shows the mapping between interactions and abstract widgets.**

| Cluster | Abstr. Inter. | Abstract Widget |
|---|---|---|
| Select Product | Product | Product Selection |
| New Bug | Component | Component Selection |
| | Summary | Summary Input |
| | Description | Description Input |
| | Submit Bug | Submit Button |
| | ... | |

edges. If a path has been found, the GUI is navigated to the cluster providing the desired abstract system interaction and the interaction can be handled. (2) The current active cluster now contains the needed abstract system interaction. The UI model is used to find the necessary widgets and the interaction is performed.

In many cases, performed actions lead to page transitions. Since these transitions depend on the current state of the system, the target of this navigation cannot be foreseen. To overcome this (from the viewpoint of the adapter) nondeterminism, we synchronize the state of the adapter before every interaction. We extend every cluster with a special output which identifies the pages distinctly (e.g., the title of the HTML document).

Using our prototypical implementation, a test execution leads to one of the following three verdicts. (1) *Pass:* the test has been executed successfully and the visible behavior was as expected. (2) *Fail:* the test has been executed successfully but the visible behavior was not as expected. This is because outputs had different values as expected. (3) *Inconclusive:* the test has not been executed successfully because the adapter could not perform all interactions. This can have two reasons. Either the system's GUI behaved unexpectedly (maybe because a page did not load or the application crashed) or the UI model has not been correct or detailed enough to perform the test.

We now assume we want to execute the test-case given in Table 2. Starting from the cluster *Home*, the first interaction is the selection of the product. The cluster *Home* does not contain such an interaction. Searching along the navigation links (using the included *Navigation* cluster) the adapter finds that *Product Selection* provides the desired interaction. To reach this cluster the adapter executes the navigation link *New* which maps to a HTML link. Now the adapter sets the input to the abstract interaction *Product* which maps to selecting a link that matches the input value. The remaining steps are performed similarly.

# 6. DISCUSSION

In this work, we suggested a solution to ease maintainability and enhance the level of reuse of tests. Using our model-driven approach, modifications of the UI reduce to adapting the UI model. To give an example, using our meta-model for GUIs (see Section 4), a relocation of a widget to a different dialog would lead to the following changes in the UI model: (1) In the concrete page model, the relocated *abstract widget* has to be adapted. The concrete changes depend on the utilized test framework. In our prototype, the selenium identification attribute (*Selenium Target* see Table 3) has to be adapted. (2) The affected *abstract pages* have to be updated (Table 4). At the initial abstract page, the widget has to be removed and at the abstract page which represents the new host, it has to be added. (3) Finally, the interactions of the affected *clusters* as well as their mappings to the abstract page model have to be updated (Table 5 and 6). Since the test cases do not contain UI related information, they do not have to be adapted. They can be reused for the changed UI without touching them.

This short example shows that changes to the UI may be handled by changing only the UI model. We believe that with suitable tool support, which is feasible due to our explicit meta-model, even major changes to the UI can be performed efficiently. The presented proof-of-concept, however, is too small to make a statement about the maintainability. We therefore want to apply our approach to a larger system in near future to validate our claims.

# 7. CONCLUSIONS AND OUTLOOK

Automated test instantiation and execution for systems with UIs is a very challenging task. Even small changes to the UI can lead to a high maintenance effort of test scripts, causing considerable costs. This paper provides the following three contributions addressing this problem.

Firstly, we present a model-based approach to make functional tests easier to create and more flexible to changes of a system's UI. We suggest separating tests into two types of artifacts, the actual test cases containing just functional logic of the tests and a dedicated UI model containing all necessary information to instantiate the tests. Given this UI model, a test adapter executes tests against a UI (see Section 3).

Secondly, we introduce a meta-model to build UI models for GUIs like desktop, mobile or web applications (see Section 4). To reduce the modeling effort, the meta-model is designed to support partial modeling for a certain set of tests. Furthermore, the meta-model can be incrementally extended for further tests.

Thirdly, we show the feasibility of our approach by presenting a prototypical implementation. We create a test for the open-source application Bugzilla as well as a UI model sufficient to execute the tests. Furthermore, we present the conceptual architecture of an adapter, executing these test cases (see Section 5).

This work describes the principles of model-based test instantiation and execution for UI based systems. The functional principle has been proven using a prototypical implementation. However, to bring our approach into practice, the meta-model has to be adapted to the concrete UI of the target application.

Furthermore, in future works, we aim to develop tool support to create and maintain UI models. A capture/replay-like tool might be appropriate to determine the necessary UI knowledge during a manual execution of a test.

## ACKNOWLEDGMENTS

The authors thank Elmar Juergens, Sebastian Eder, Lars Heinemann, Georg Hackenberg and Philipp Neubeck from Technische Universität München for their support and insightful comments.

## REFERENCES

[1] IEEE Standard Computer Dictionary. A Compilation of IEEE Standard Computer Glossaries. *IEEE Std 610*, 1991.

[2] S. Berner, R. Weber, and R. K. Keller. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering (ICSE '05)*, 2005.

[3] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *Future of Software Engineering (FOSE '07)*, 2007.

[4] G. Botterweck. A model-driven approach to the engineering of multiple user interfaces. In *Proceedings of the Workshop on Model-driven development of advanced user interfaces (MDDAUI '06)*, 2006.

[5] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, N. Souchon, L. Bouillon, M. Florins, and J. Vanderdonckt. Plasticity of user interfaces: A revised reference framework. In *Proceedings of the First International Workshop on Task Models and Diagrams for User Interface Design*, 2002.

[6] E. Dustin, J. Rashka, and J. Paul. *Automated software testing: introduction, management, and performance.* Addison-Wesley, 1999.

[7] M. Fewster and D. Graham. *Software test automation: effective use of test execution tools.* Addison-Wesley, 1999.

[8] H. Grandy and S. Benz. Specification based testing of automotive human machine interfaces. In *GI Jahrestagung*, 2009.

[9] M. Grechanik, Q. Xie, and C. Fu. Maintaining and evolving gui-directed test scripts. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*, 2009.

[10] M. J. Harrold and A. Orso. Retesting software during development and maintenance. In *Proceedings of the Frontiers of Software Maintenance (FoSM '08)*, 2008.

[11] B. Hauptmann. Model-based test instantiation for applications with user interfaces. In *Proc. Doctoral Symposium at the International Conference on Product Focused Software Development and Process Improvement (PROFES '11)*, 2011.

[12] H. Hussmann, G. Meixner, and D. Zuehlke, editors. *Model-Driven Development of Advanced User Interfaces.* Springer, 2011.

[13] M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, and M. Satama. Towards deploying model-based testing with a domain-specific modeling approach. In *Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*, 2006.

[14] A. M. Memon and B. N. Nguyen. Advances in automated model-based system testing of software applications with a GUI front-end. In M. V. Zelkowitz, editor, *Advances in Computers*, volume 80. 2010.

[15] A. C. R. Paiva, N. Tillmann, J. C. P. Faria, and R. F. A. M. Vidal. Modeling and testing hierarchical GUIs. In *Proceedings of the 12th International Workshop on Abstract State Machines*, 2005.

[16] F. Paternò, C. Mancini, and S. Meniconi. Concurtasktrees: A diagrammatic notation for specifying task models. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction (INTERACT '97)*, 1997.

[17] A. Pimenta. *Automated Specification Based Testing of Graphical User Interfaces.* PhD thesis, Engineering Faculty of Porto University, Department of Electrical and Computer Engineering, 2006.

[18] W. Prenninger, M. El-Ramly, and M. Horstmann. Case studies. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science.* Springer, 2005.

[19] W. Prenninger and A. Pretschner. Abstractions for model-based testing. *Electron. Notes Theor. Comput. Sci.*, 116, January 2005.

[20] R. K. Shehady and D. P. Siewiorek. A method to automate user interface testing using variable finite state machines. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS '97)*, 1997.

[21] P. A. Szekely. Retrospective and challenges for model-based interface development. In *Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces (CADUI'96)*, 1996.

[22] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann, 2006.

[23] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. Technical report, The University of Waikato, April 2006.

[24] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. *Formal methods and testing*, 2008.

[25] S. Winter. *Modellbasierte Analyse von Nutzerschnittstellen.* Dissertation, Technische Universität München, München, 2009.