Proceedings
Joint Research Workshop

10th Systems Testing and Validation Workshop (STV15)

1st International Workshop on User Interface Test Automation (INTUITEST 2015)

Edited by

Pekka Aho
Tanja Vos
Juan Garbajosa
Jørgen Bøegh
Axel Rennoch

19th October 2015
Sophia Antipolis, France

Online version

http://publica.fraunhofer.de/dokumente/N-360101.html
urn:nbn:de:0011-n-3601016

Contacts

Mr. Axel Rennoch
Fraunhofer Institute for Open Communication Systems FOKUS
email: axel.rennoch@fokus.fraunhofer.de

Mr. Pekka Aho
VTT Technical Research Centre of Finland
email: Pekka.Aho@vtt.fi

Preface

In the development process the testing and validation activities are important steps for quality assurance of any product or system. The System Testing and Validation Workshop (STV) is a series of events initiated in the year 2002 and seeks to provide answers to the many open issues related to testing and validation. In 2015 STV is held as a joint research workshop with the first International Workshop on User Interface Test Automation (INTUITEST) and is co-located with the 3rd User Conference on Advanced Automated Testing (UCAAT)

We like to thank all members of the program committees for reviewing the submitted papers, the European Telecommunications Standards Institute (ETSI) for hosting the joint research workshop and in particular Mrs. Nathalie Guinet and Mrs. Emmanuelle Chaulot-Talmon for their support in the organization of the event.

Sophia Antipolis, 19th October 2015

Pekka Aho, Tanja Vos, Juan Garbajosa, Jørgen Bøegh, Axel Rennoch

Workshop homepages

STV: https://www.fokus.fraunhofer.de/go/stv15
INTUITEST: https://staq.dsic.upv.es/INTUITEST/2015/

Workshop Committees

STV:

Steering Commitee

Jørgen Bøegh (Lemvig, Denmark)
Juan Garbajosa (Technical University of Madrid, Spain)
Axel Rennoch (Fraunhofer Fokus, Germany)

Programme Committee Members:

Jorgen Boegh (Lemvig, Denmark)
Ana Cavalli (IT-Sudparis, France)
Juan Garbajosa (Technical University of Madrid, Spain)
Stephane Maag (IT-Sudparis, France)
Axel Rennoch (Fraunhofer Fokus, Germany)
György Réthy (Ericsson, Hungary)
Juha Röning (University of Oulu, Finland)
Theo Vassiliou (Testing Technologies, Germany)

INTUITEST:

General Chair
Pekka Aho (VTT Technical Research Centre of Finland, Finland)

Programme Chair:
Tanja Vos (Universidad Politecnica de Valencia, Spain)

Programme Committee Members:

Emil Alegroth (Chalmers University of Technology, Sweden)
Domenico Amalfitano (University of Naples Federico II, Italy)
Mokhtar Beldjehem (University of Ottawa, Canada)
José Creissac Campos (Universidade do Minho, Portugal)
Valentin Dallmeier (Saarland University, Germany)
Anna Rita Fasolino (University of Naples Federico II, Italy)
Jorge Francisco Cutigi (Instituto Federal de São Paulo, IFSP - Campus São Carlos.)
Teemu Kanstren (VTT Technical Research Centre of Finland, Finland)
Peter M. Kruse (Berner & Mattner, Germany)
Leonardo Mariani (University of Milano Bicocca, Italy)
Atif Memon (University of Maryland, USA)
Rafael Oliveira (University of Sao Paulo, Brazil)
Ana Paiva (Faculty of Engineering of the University of Porto, Portugal)
Wishnu Prasetya (Universiteit Utrecht, Netherlands)
Matias Suarez (F-Secure Ltd)
Andreas Zeller (Saarland University, Germany)

Agenda

STV

A Process for Model Transformation Testing

Teemu Kanstrén*, Marsha Chechik†, Juha-Pekka Tolvanen‡
*VTT, Oulu, Finland
email:teemu.kanstren@vtt.fi
†University of Toronto, Canada
email: chechik@cs.toronto.edu
‡MetaCase, Jyväskylä, Finland
email: jpt@metacase.com

***Abstract*—This paper describes a process for testing model transformations. The process is based on systematic analysis of the transformation rules and the related metamodels. These are used to identify the relevant parts of the metamodel to test the transformation, to define coverage criteria, and to define test oracles (checks) as invariants on what the transformation output should hold in relation to the input model. Tests are then created (generated) to produce suitable input models to fulfill the coverage criteria, and to check that the invariants hold in the transformation outputs for the different types of inputs generated. A case study of testing a transformation from the EAST-ADL specification for the automotive industry is presented.**

I. Introduction

Domain-Specific Languages (DSL) and model-driven engineering are increasingly applied in different contexts. In these techniques, models at a suitable abstraction level are used to describe the developed system. One of the core elements in this is the model transformation, translating the input model into another format and possibly to a different abstraction level.The result of the transformation can be code, models, configuration files or other artifacts. The correctness of the transformation is crucial as it will impact every input model using that transformation.

Testing model transformations is challenging as they are typically created as part of specialized modeling environments, using their own programming languages and often with limited support for specific testing features such as extensive test generation. The potential input space is also typically huge and difficult to cover to a sufficient extent in manual testing. We describe how we have addressed these issues in our work, using a systematic process to describe input models to be generated, define coverage criteria, apply test generators to achieve this criteria and to define test oracles for the generated test cases.

Testing traditional software systems typically involves producing various inputs, defining the matching expected output, and the impact on the System Under Test (SUT) state, which can in turn influence how further input is turned into output. Model transformation testing is slightly different in that the input is given typically as a single input model, and the observable effects are only the produced the output artifact(s). In many cases there is no visible state to test from the external black-box viewpoint. This puts more focus on forming the input model and the output model for the test cases and less on the concept of state.

Model transformations have a set of specific properties that can be used to create more effective test solutions. The input and output are typically described by metamodels, explicitly defining their valid structure. The operators applied in the transformation rules are also typically more constrained than in using general purpose programming languages. We make use of these special properties and constraints of the model transformations in our test process. While various aspects of model transformation testing have been applied in previous research, they typically address isolated phases or activities. We present an overall approach to practical transformation testing and describe its application in a realistic case study in the automotive domain.

We use the transformation rules and metamodels to define the forms of input to create, to define the coverage criteria to achieve sufficient testing, and to define checks to verify the correctness of the transformation. We describe how we turn this information into concrete test cases to execute and verify the transformation. We demonstrate the approach by applying it to a model transformation from EAST-ADL specification language to a Simulink model. EAST-ADL is an Architecture Description Language (ADL) for the automotive domain. While we have produced large scale models for different parts of the specification, our case study focuses on a specific subset to illustrate the process.

The rest of the paper is structured as follows. In Section II we describe related work. In Section III we present our process for testing model transformations. In Section IV we apply this to a transformation from the EAST-ADL specification. Section V discusses the results, followed by conclusions.

II. Related Work

In [1], transformation testing is described as consisting of three activities: *generating test data*, *defining test adequacy criteria*, and *constructing an oracle*. In [17], these are called phases and the fourth phase of *test execution* is added. We see transformation testing as being composed of several activities performed iteratively in different phases. The works described in [1] and [17] are related to different parts of these activities. These related works describe different phases of a possible process, with a limited view on how these tie together and can be applied in practice. In this paper, we present our approach for an overall, practical model transformation testing process and illustrate it with a concrete and realistic example. The rest of this section present an overview of related works in different areas of such processes, from which we have applied and adapted different parts to our approach.

A central term in the transformation testing literature is the Effective Metamodel (EM). This is the part of the input metamodel relevant for the transformation [6]. It includes the parts of the input that the transformation is expected to operate on, and excludes the parts that the transformation does not operate on. The EM can be identified by analyzing the transformation rules to see what model elements they are associated with. When static analysis tools for the given transformation platform are available, those can be used to automate this analysis for existing code [16]. We use the EM as one input for our test process.

What [1] describes as *test adequacy criteria*, we more generally term as the test coverage requirements. Different ways to define coverage requirements for transformation testing have been presented, and we use these different forms of coverage as part of our process.

For white-box transformation testing, similar to traditional white-box testing, we can use code-coverage measures of the transformation code itself [8]. For black-box testing, we can define a set of coverage criteria based on the EM. In [6] and [7], EM coverage is defined as combinations of EM classes, their associations and attributes. Combinations of these in different input models should then be covered in testing the transformation. Different coverage criteria for the combinations can be defined, such as each range must appear in one input model, or all combinations of all ranges must appear in the combined set of input models [7]. In general software testing terminology this is typically called category-partitioning.

Generic coverage measures such as those based on transformation rules or the EM can be used generally for any transformation but can require very large test suites. Knowledge based partitioning on the other hand is based on the expert providing specific sets of combinations that need to be covered [2]. This requires more effort from the domain expert but is better at capturing the important needs of the domain in a concise way. In Section IV we demonstrate how our approach can effectively cover the general level EM based coverage requirements, and how these can be fine-tuned with domain-specific definitions.

Finally, coverage can also be analyzed using mutation analysis [16]. In this case, the transformation is mutated (small behavior modifying changes made to the rules) and when all mutants are covered by the test cases the test suite is considered adequate.

Covering these coverage criteria requires a significant test set, making manual testing expensive. As transformations are typically well defined (with metamodels and specific transformation rules), and executed fast (no user interface) they are well suited for automated test generation. Different approaches to generating such test sets have been taken. For example, constraint solvers [3] and randomization based test generators [18] have been used. We apply a method described in [10] which uses randomization based algorithms to optimize the test set for a wide variety of coverage criteria, providing means to address effectively the different types of coverage requirements.

To evaluate the correctness of any test execution, a test oracle is needed. A test oracle is a procedure that evaluates whether a given test case should pass or fail. A single test case may have several different test oracles to assess different properties. Different levels of such test oracles have been defined for transformations. A simple and generic approach is to start with checking that the transformation output conforms to the output metamodel [2]. For making more detailed comparisons, several rules are presented in [14], such as finding matching and missing elements in test input models vs transformation output. From these, four different outcomes are suggested: *element is correctly/incorrectly mapped*, *transformation is incomplete* and *comparison is incomplete* [14].

Some additional oracles are proposed in [15]. These include pattern matching, contracts, and extensions of the comparison operators. Six examples are given. When a *reference transformation* exists, it can be used for comparison, such as running several refactoring tools on the same input and comparing their results [4], [18]. When possible, *Inverse transformation* can be applied to see if the result given back is the original input. Again, an example is to run two opposite refactorings in a sequence [4], [18]. An *expected output model* can be provided by an expert for a given input model. A *generic contract* is provided by a test expert and should always hold when comparing the input model to the output model, for example, matching input elements to output elements. *Specific asserts* are checks tailored by the test expert for a given input model (test case specific). *Model snippets* define partial input and output (snippets) for a test case.

We use different types of oracles for different purposes in our test process. Metamodel validation is used as a generic test oracle for all tests. Expected output models are used for manually defined tests, which are used to cover basic coverage requirements. Contracts (invariants) between the input and output are used as part of generated tests (e.g., matching model elements).

An approach for transformation testing called *Tracts* is presented in [9]. The user specifies a set of input metamodel, output metamodel, and input-output metamodel mapping constraints called *Tracts* for what needs to be tested and evaluated. The effort is described as often matching re-implementing the transformation or more [9]. We use a black-box model-based testing approach describing the relevant input and the test oracles for evaluating the produced output. In our experience, this lets us focus on parts of interest and focus the effort in more intuitive way for a tester. However, another approach such as *Tracts* could be used as part of the process as well.

III. Process

We describe our process for testing model transformations in three different parts:

1) Characterizing the domain
2) Defining checks (test oracles)
3) Creating and executing test cases

These are illustrated in Fig. 1. Characterizing the domain refers to identifying the important elements in the transformation to be tested. Following this, we define test oracles to verify the transformation as checks over properties in the transformation input vs. transformation output. Creating and executing test cases refers to turning the understanding of the input and output and their relations into concrete test cases that can be executed to evaluate the transformation.

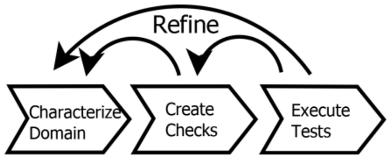


Fig. 1: Process flow

This process is a learning and improvement process where the three activities are iterated in collaboration with different stakeholders (e.g., testers, developers, domain experts). The loop terminates when a suitable level of confidence on the transformation quality has been achieved for involved parties, respecting the available resources and observed criticality of the test target. It re-starts when this confidence no longer holds. The following subsections characterize these three activities in more detail.

A. Characterizing the domain

In characterizing the domain, we identify the effective metamodel (EM) and define the coverage (test adequacy) criteria. We see this as generally a black-box activity, making use of the white-box information where available (e.g., tools discussed in Section II). However, simply using a white-box analysis tool to identify the EM from the transformation implementation is not sufficient. This alone cannot tell if the rules are correct, if some are missing or if some should not have been implemented at all. For this reason, we find it important to always perform an analysis of the requirements against the EM.

In our experience, and similar to testing in general, performing a black-box analysis based on specifications and discussing this with other stakeholders also helps identify ambiguities and misunderstandings. This can be augmented with white-box analysis tools where possible.

We have developed a set of questions to guide the transformation analysis and discussion, and to identify the EM. These questions are intended to help systematically go through the transformation metamodels and rules as a basis for the later phases. The questions are:

- Why does the transformation do what it does?
- For each input (meta)model element:
  - Is it relevant (in scope) for the transformation?
  - How is it (or should be) affected by the transformation?
  - What should be produced for it in the output?
  - Does it have any other impact on the transformation?

The first question helps raise discussion about the purpose of the transformation, why testing it is important, and what areas should the testing focus on.

The four questions related to each model element help define a set of (expected) high-level transformation rules, the EM, and test coverage criteria. Identifying the EM allows us to define the different forms of input models to create for testing, which should include various combinations of the elements in the EM. The EM, as metamodels in general, consist of model elements (e.g., classes, attributes, ports, roles [13]) and their associations. As our tests cannot possibly cover all possible combinations in any non-trivial meta-model, we need to limit this set by defining the coverage criteria more strictly.

We have used categorizations of 0,1,N (N = 2 or more) as coverage goals for the number of different types of model elements and associations as the primary coverage goal. We illustrate this with a hypothetical model having two elements: *A* and *B*. For this, the set of input models should have at least one model with 0 of *A*, one with 1 of *A* and one with N of *A*, and the same for *B*. Several such goals can be covered by a single input model, for example, a single model that has 0 *A* and 2 *B*. We use the combinations of these as the secondary coverage goal. For example, one input model with 0 *A* and 2 *B* and another with 1 *A* and 2 *B*. The coverage criteria can be further weighted by the importance of specific model elements. For example, informal input model elements (e.g. comments) may be given lower weight than functional elements. Works presented in Section II can be used to assist in producing more complex coverage requirements as needed.

A specific part to consider is the input metamodel not part of the EM. While the transformation should correctly handle the EM and nothing more, we do not know if it correctly handles the nothing more part until we test it. A transformation viewed as a black-box may break if there are elements included in the input that are not part of our EM. However, such elements do not need to be covered in extensive detail and can be given lower coverage priority. When white-box transformation analysis tools are available, they can also be used to provide assurance that only the EM has impact on the transformation implementation and execution.

Once we have defined the rules to form relevant test input, and the coverage criteria, we can use them to define checks.

B. Define Checks (Test Oracles)

What use *checks* to refer to the test oracles that make assertions about the produced output with regards to the provided input. They can be implemented in different ways.

The output metamodel can be used to verify that the transformation always produces valid output conforming to the output metamodel. This generic check can be applied for all test cases and for all transformations when the output metamodel is available. More specific checks need to be defined on a per-transformation basis.

As discussed in Section II, one potential test oracle is to use reference results where the expected output for a specific input is manually defined. We have used previously designed production input models and their associated, transformed and manually checked, output models as such references. However, manual testing typically does not scale to high level of assurance over complex input combinations.

To scale higher, we have applied test generation techniques using model-based testing (MBT) tools. In these cases, the generator produces large sets of input model variants covering the potential input space as defined by our coverage criteria. The checks are defined as invariants that should hold over all outputs the transformation produces for these inputs.

The process we have followed to create the checks consists of the following steps:

- Define a check at least for each transformation rule.
- Combine checks for the same elements.
- Map the checks to the concrete output.
- Review with domain expert.
- Repeat as needed.

We start with the transformation rules identified when characterizing the domain. We look at each rule and consider what its effect is on the output, and define a check to evaluate that property holds when the element is present in the input model. We repeat this for each transformation rule identified. In the end, we go through the checks and combine the ones related to the same output element.

For example, several rules may produce new output elements of type A, each from a different input model element. We first create checks for each of these to find the matching number of A in the output. These checks would fail as the overall number of A is actually impacted by several rules each generating more A separately. To address this, we go through the checks, combine all that impact A and use the sum as the overall check for A. Other combination operations may also be required, such as merging several if the output property should be overwritten in case of duplicates. This is further illustrated in Section IV with concrete examples.

Depending on what level the analysis is performed, we may need to separately map the checks to the concrete output. For example, we may define a check that an association of A to B in the input model is represented correctly in the output. Concretely this may require checking each side of the association (A and B) in the output separately, due to properties of the output model.

Finally, the checks, similar to the domain characterization need to be reviewed with stakeholders and iterated as needed. At any time it is also possible to go back to refine the domain characterization as we learn more.

C. Create and Execute Test Cases

Test cases are created by combining the information collected in characterizing the domain and defining checks. They formalize the acquired understanding and provide a means to validate it through test case execution. They are based on the structure of the input (EM) and the defined test coverage (adequacy) criteria.

We use different types of techniques to cover different parts of the coverage requirements. We use reference inputs and outputs where available as a starting point. We add coverage for large-scale variation with a MBT test generator. When needed, we apply the generator separately to focus on specific aspects, such as providing more extensive coverage for more complex parts. We compose such smaller (partial) test models to higher-level test models. This helps to keep the complexity of the test generators under control.

In our experience, issues with the transformation are found in all stages of the process. Biggest issues are typically discovered before any tests are executed as ambiguities, misunderstandings and missing requirements during the test modeling process. Manually created tests and reference models are good at finding the most obvious issues in the implementation. Large scale, automatically generated tests are best at discovering issues with complex interactions and combinations of different elements.

The information learned from this phase can again lead to refinement of the characterization of the domain as well as to refining the test oracles (checks).

IV. Case Study: EAST-ADL

In this section, we present an example of applying the process for testing a transformation from an architectural description written using a DSL based on EAST-ADL in the MetaEdit+ tool to a Simulink model.

Automotive manufacturers use architecture description languages, like EAST-ADL, to specify vehicle architecture at a functional level. This is used to describe all types of system parts such as software, electric and electronic functionality. Typically EAST-ADL is used to describe the static architecture and Simulink is used to describe the behaviour of different components. To facilitate working on these different parts with the different tools, MetaEdit+ provides three different transformations. One for transforming the EAST-ADL model from MetaEdit+ to Simulink, one for transforming the Simulink model back to EAST-ADL, and one for checking that the EAST-ADL and Simulink models have the same elements (to support co-evolution).

We use the first one that transforms EAST-ADL model to Simulink model to illustrate our testing process. As an example model we use the Anti-Lock Braking (ABS) system. This EAST-ADL model is illustrated in Fig. 2, showing a brake pedal, a brake controller, and four wheels with a brake torque controlled by the brake controller. This model itself is hierarchically part of a set of larger models, but we focus on this smaller part to illustrate the concept.

Fig. 3 shows the Simulink visualization of the same model after the transformation. The transformation aims to preserve as much as possible of the model information to make switching between tools as seamless as possible. MetaEdit+ allows several users to work on a single model simultaneously,

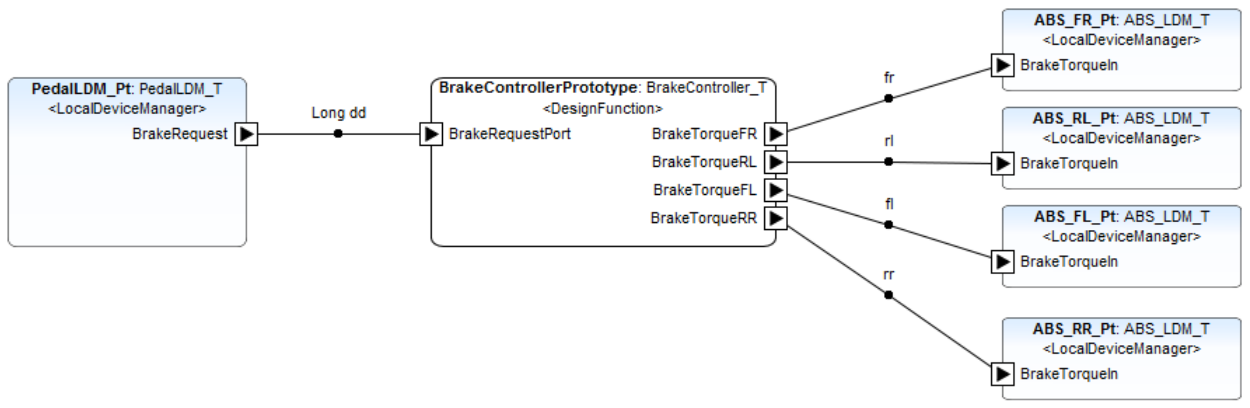


Fig. 2: ABS Model in MetaEdit+

which is typical to the architecture definition phase (EAST-ADL use). Simulink on the other hand only supports one-at-a-time editing, which is more typical for the behavior modeling part, focusing on particular module details but not the overall functionality of the vehicle. As these tool, languages and environments need to be used both to work on the same models, these transformations are needed.

Listing 1 shows a partial output for the EAST-ADL to Simulink transformation for the model visualized in Fig. 3 by Simulink. This listing is used as basis for discussing the transformation in the following subsections.

Listing 1: Output (Simulink) model snippet

```
Model {
  Name  "BBW_FDA"
  Description  "Example model"
  Version  1.0
  System {
    Name  "BBW_FDA"
    Location  [10, 100, 700, 500]
    Block {
      BlockType  ModelReference
      Name  "ABS_FR_Pt"
      SID  "1"
      Ports  [1,0]
      ModelNameDialog "ABS_LDM_T"
      ModelReferenceVersion "1.0"
      List {
                ListType  InputPortNames
                port0   "BrakeTorqueIn"  }
      List {
                ListType  OutputPortNames  }
      CopyOfModelName "ABS_LDM_T"
    }
    Block {
      BlockType  ModelReference
      Name  "ABS_RL_Pt"
      SID  "2"
      Ports  [1,0]
      ModelNameDialog "ABS_LDM_T"
      ModelReferenceVersion "1.0"
      List {
                ListType  InputPortNames
                port0   "BrakeTorqueIn"  }
      List {
                ListType  OutputPortNames  }
    }
    Line {
      Name  "fl"
      SrcBlock  "BrakeCtrl_Pt"
      SrcPort  3
      DstBlock  "ABS_FL_Pt"
      DstPort  1
    }
  }
}
```

A. Characterizing the domain

The overall metamodel for EAST-ADL is described in detail in the EAST-ADL specifications (Section 6) [5]. Identifying the parts of this metamodel relevant for the transformation gives us the EM. The two main elements in the EM identified are related to the reference block part of the EAST-ADL metamodel:

- DesignFunctionType (DFT): Defines a type for a DFP.
- DesignFunctionPrototype (DFP): Representes an occurrence (instance) of a DFT that types it.

Fig. 4 illustrates this part. A DFT defines the type of a DFP and a set of ports to connect to other DFT instances (DFP's). It may also define a composition of several DFP as part of it. The ABS model in Fig. 2 concretely illustrates this.

In Fig. 2, the boxes are DFP instances, with an associated DFT. For example, *BrakeControllerPrototype* is a DFP, and it has a DFT of *BrakeController_T*. The four blocks on the right side are the wheels. They are all separate DFP instances but share the same DFT type *ABS_LDM_T*. In Fig. 2, the *BrakeControllerPrototype* DFP has four ports each linking to a different DFP's (of type *ABS_LDM_T*). We use the term *block* to refer to the components (boxes) in Fig. 2, and the term *line* to the connection lines between them.

In order to identify the transformation rules for the following steps of the process, we had several iterations of discussions and reviews between the tester, the developer and the domain expert. The tester also had access to the modelling and tranformation tool environment (MetaEdit+ with the models). This was used by the tester to clarify details and to refine questions about the transformation between the stakeholder meetings. A large set of existing reference models existed for the transformation, so in this case the process focused on producing a test generator to cover the EM variation at larger scale. Close to 70 rules were identified for the transformation.

We describe here the rules for transforming the *DesignFunctionPrototype* (DFP), focusing on a subset sufficient to illustrate the process. At the highest level, a *Block* element is generated in the output (Simulink model) for each DFP in the input (MetaEdit+) model. This *Block* is then used as the container for properties of the DFP in the (Simulink) output model. A subset of these DFP transformation rules is (*add*

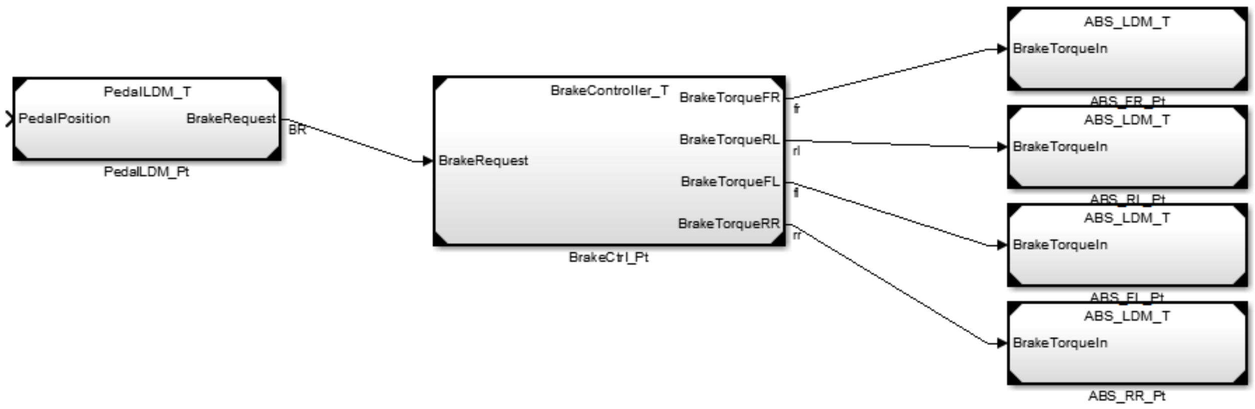


Fig. 3: ABS Model in Simulink

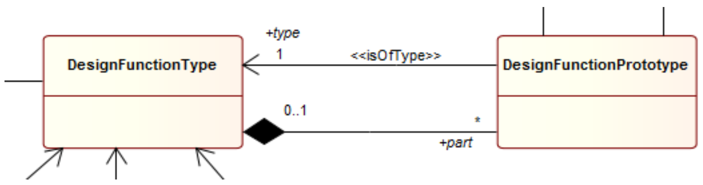


Fig. 4: Effective Metamodel (EM) for the reference block part of the transformation.

refers to adding attributes to a *Block*):

- Add *BlockType* with static value *ModelReference*
- Add *Name*, with value of *Short name* from DFP.
- Add *SID*, with increasing unique integer.
- If *Description* is defined for DFP, add *Description* with value of *Description* from DFP.
- Add *AttributeFormatString*, with value of *Name* from DFP.
- Add *ModelNameDialog*, with value of *FunctionName* from DFP.
- Add *ModelReferenceVersion* with static value *1.0*
- Add *List* and inside it
  - Add *ListType* with static value *InputPortNames*
  - For each *InFlowPort* in DFP
    - Add *port X* where X is an increasing integer, with value of *Short name* from *InFlowPort*
  - For each *InPowerPort* in DFP, do the same as for *InFlowPort*
  - For each *ServerPort* in DFP, do the same as for *InFlowPort*
- Add *List* and inside it
  - Add *ListType* with static value *outputPortNames*
  - For each *OutFlowPort* in DFP,
    - add *port X* where X is increasing integer, with value of *Short name* from *OutFlowPort*
  - For each *OutPowerPort* in DFP, do the same as for *OutFlowPort*
  - For each *ClientPort* in DFP, do the same as for *OutFlowPort*

TABLE I: Example coverage categories

Model Element	**Categories**
DFP	1,2,N
InFlowPorts in DFP	0,1,N
OutFlowPorts in DFP	0,1,N
InPowerPorts in DFP	0,1,N
OutPowerPorts in DFP	0,1,N
ServerPorts in DFP	0,1,N
ClientPorts in DFP	0,1,N
Description	0,1

These and other rules were identified by going through the EM following the process in Section III. In a similar way, we also looked at the elements of the EM and their properties to set the coverage goals. A subset of these is shown in Table I, Where N refers to 2 or more (in the first row 3 or more). Combinations for these were defined as described in Section III-A. Besides these, the EM contains associations (lines) and various types of additional properties. These are not shown here to keep the example concise but were also included in the input models and coverage criteria.

Besides these criteria related to single model elements, we defined criteria for their various combinations. This refers to having variants of the categories in the same model. For example, one combination to cover by an input model would be 1 *InFlowPorts*, 1 *OutFlowPorts*, N DFP instances and an association between the two flow ports. Combinations used include variants of:

- port type count and DFP counts,
- connections types, connection numbers, and number of different types of ports

The number of combinations for the options we defined comes to 300. Covering such combinations is by far too large a task to handle by manual test creation. While we could define much bigger sets and use more complex models, our goal is not to evaluate the specific test generation algorithm but rather to illustrate the process and its application.

The *Description* element is an example of an item that is not used for anything other than informational purposes in the input and output models. Thus it was given less weight in the coverage criteria.

Beyond covering different combinations for the elements and properties of the EM, we also analyzed the specifications to define specific coverage criteria of interest. These include traditional test targets where errors are most likely to exist, such as boundary conditions or invalid parameters types. Following are some examples of these:

- Connection from same port type to same port type (e.g., *InFlowPort* to *InFlowPort*)
- Several connections from/to a single port
- Overlapping DFP names
- DFP without type (DFT) defined

While we can create tests manually or via generation to cover these, we also need to consider how actual models by users would be created and under what constraints. The actual models are intended to be created using the DSL defined in MetaEdit+. The DSL defines a set of constraints for what types of models can be created, even if they could otherwise be fed to the transformation engine.

In this case, the DSL defines a number of constraints including forbidding connections between different kind of ports, allowing a connection to be attached to only one port at each end, not allowing adding a connection in a DFP without its DFT having defined a port, and requiring DFP and DFT names to be unique in their context.

However, not all of our corner cases are enforced in the DSL, and were defined after very interesting stakeholder discussions. For example, multiple connections from a single input port are allowed by EAST-ADL but not in Simulink. As such, these still need to be considered by the transformation and the tests created. Another example is missing DFT definition for a DFP that are allowed and only produce a warning at modeling time.

A property that generally needs to be considered in testing is input formatting. EAST-ADL defines a set of strict formatting rules for what types of names and values are allowed in the model. As these are enforced by the DSL in MetaEdit+, we also follow these rules in our tests and do not test invalid input formatting. In another context, we might be interested to also test these DSL constraints but due to resource limitations we chose to trust the DSL with these.

B. Defining Checks

For the manually created and reference input models, the test oracles used are the reference output models. For the generated test cases, the problem of providing a test oracle is more interesting. To define the test oracles for them, we analyzed the identified transformation rules identified in Section IV-A using the process described in Section III-B. These checks include (SL refers to the Simulink models produced by the transformation):

1) The output model conforms to the output metamodel.
2) Number of *Input Ports* in SL is sum of input model ports of type *InFlowPort*, *InPowerPort*, and *ServerPort*.
3) Number of *Output Ports* in SL is sum of input model ports of type *OutFlowPort*, *OutPowerPort*, and *ClientPort*.
4) Number of lines in SL matches those in input.
5) All SID values in output are unique.
6) Dynamic attribute values are copied as expected.
7) Required static attribute values are found for relevant model elements.

Checking that the output model conforms to the output metamodel ensures that the transformation always produces a structurally valid output. Once we have assurance that the output model is structurally valid, we check it for the defined invariants (checks 2-7 above). The checks for the number of input and output ports are examples of combining checks for several rules into one. For example, *InFlowPort*, *InPowerPort*, and *ServerPort* are all combined into one check for *Input Ports* as all three input ports produce same types of outputs but as a result of different rules.

These checks in general are intended to be more abstract than the implementation. For example, we check that each SID is unique, not that they are in an increasing order (even though they are implemented sequentially). Partly this is because this is the true requirement (not the sequential order but uniqueness). In practice it also easier to check for no duplicates, and shows how we do not wish to re-implement the transformation for test oracle checking. Instead, we want to see that their structure is valid (check 1) and the important invariants always hold (checks 2-7) in a cost-effective manner.

Finally, reviews of the checks vs rules were performed together with the domain expert to evaluate that the relevant properties had been included, there were no extra properties and that the checks that were defined were correct. This was iterated several times.

C. Create and Execute Test Cases

We used the existing input models as references for the initial test set to create test cases for the transformation. To generate large scale test suites, we used our MBT generator (available at [11]) described in [10]. Like many test generators, it was originally designed to create state-based tests for reactive systems. In such cases, the SUT is provided with inputs in small steps (e.g., login, add item to cart, checkout) and checks are performed at step granularity (e.g., login success, load correct page, check cart for correct items).

TABLE II: Example rules and actions

Rules	Actions
Always allowed	Create a DFT
Always allowed	Create a DFP
[Unlinked DFP's $>$ 0 and DFT's $>$ 0]	Link DFP to DFT
[DFT's $>$ 0]	Add *InFlowPort* for DFT
[DFT's $>$ 0]	Add *OutFlowPort* for DFT
[DFP without Description exists]	Add Description to DFP

To produce a test case for a transformation, we do not consider a test case as executing and checking such small steps. Rather we need to first produce a complete input model, execute the transformation on it, and then perform all the checks as a single operation on the output model. Thus we had to adapt the concept of our MBT generation to transformations.

Our generator notation represents the tests to be generated as a set of actions and rules for creating steps in a test case. The actions implement the test steps and the rules specify when each action is allowed. The generator maintains state to enable creating more complex rules and actions. These are implemented in a *test model*, which the test generator executes in different ways using a given algorithm (e.g., weighted random choice). This is described in more detail in [10]. To avoid confusion between different models (MBT test models and transformation models), we refer to this in the following as just the test generator.

To adapt this to transformation testing, we used the rules and actions to describe steps of building the input model. We used generator state to reflect the input model being generated. In our view, such a generator represents the process a human expert would use to build a set of input models. As the generated input model structure is stored in the generator state, we can also use this state to define the checks for the invariants that should hold over the input.

Using the transformation rules defined in Section IV-A, we defined a set of rules and actions for the test generator to produce EAST-ADL input models. Table II illustrates a subset of these. The complete set was created to cover the overall EM and the identified transformation rules. Due to space constraints we do not show all of them here, but the complete generator is available at [12]. The test generator executes these actions in different sequences to generate input model variants. The generator state maintains a list of the created DFP and DFT instances, their attributes, DFP links to DFT, and port connections between the DFP's.

Executing one action produces a model element for the output model, which updates the generator state. For example, *Create a DFT* adds a new DFT to the generator state. This state can then be used by the rules to enable or disable other actions. For example, [DFT's $>$ 0] in Table II checks if the generator state (current input model being generated) has a DFT defined. If it does, the actions to add an *InFlowPort* or an *OutFlowPort* to a DFT are enabled and can be taken by the generator.

Listing 2 shows an example generator trace for building a test case. Each line represents a step invoked by the generator to build a single test input model. First, two DFP instances and their descriptions are created (steps 1-4). Following this, a DFT is created, a DFP is linked to it, and five ports are created for the DFT (steps 5-11). A second DFT is then created and three more ports are created (steps 12-16). These could be for either of the two DFT in the generator state. Finally, the unlinked DFP is also linked to a DFT, forming a potential DFT pair for the generator to create port connections (step 17). The generator then creates two flow connections (18-20).

Listing 2: Example generator sequence

```
1. CREATEDFP
2. ADDDESCRIPTION
3. CREATEDFP
4. ADDDESCRIPTION
5. CREATEDFT
6. LINKDFPTODFT
7. ADDSERVER
8. ADDINFLOW
9. ADDOUTPOWER
10. ADDSERVER
11. ADDCLIENT
12. CREATEDFT
13. ADDOUTFLOW
14. ADDINPOWER
15. ADDOUTPOWER
16. ADDINFLOW
17. LINKDFPTODFT
18. CREATEFLOWLINK
19. ADDOUTPOWER
20. CREATEFLOWLINK
```

This trace does not directly show which DFT is linked to which DFP or which DFT the ports are generated for. The actual input model the generator built while executing this trace is shown in Listing 3. This listing is an approximation of the actual generated MetaEdit+ XML format, to provide a more concise and easy to read example. The created DFT are shown in the beginning, the DFP instances and their links to the DFT next, and finally the set of connections between the DFP instances.

Here we also see how each DFP is given a unique DFT and how the created ports are in this case distributed over the DFT. Other tests have different elements, properties and links, as well as different generation sequences, all together contributing to the overall test suite fulfilling all the coverage combinations we requested. This is the model built piece by piece by the steps (actions) taken by the generator in Listing 2.

Listing 3: Input Model for example sequence

```
<GXL>
  <TYPE ID="1" NAME="DFT1">
    <PORTS>
      <INFLOW NAME="InFlow1" ID="1"/>
      <OUTFLOW NAME="OutFlow1" ID="1"/>
      <INPOWER NAME="InPower1" ID="1"/>
      <OUTPOWER NAME="OutPower1" ID="1"/>
      <OUTPOWER NAME="OutPower2" ID="2"/>
      <CLIENT NAME="Client1" ID="1"/>
      <SERVER NAME="Server1" ID="1"/>
      <SERVER NAME="Server2" ID="2"/>
    </PORTS>
  </TYPE>
  <TYPE ID="2" NAME="DFT2">
    <PORTS>
      <INFLOW NAME="InFlow1" ID="1"/>
      <OUTPOWER NAME="OutPower1" ID="1"/>
    </PORTS>
  </TYPE>
  <PROTO DESCRIPTION="Description1" TYPE="DFT2" NAME="DFT1" ID="1"/>
  <PROTO DESCRIPTION="Description2" TYPE="DFT1" NAME="DFP2" ID="2"/>
  <CONNECTIONS>
```

```
    <CONNECTION>
      <SRC DFP_NAME="DFP1" DFP_ID="1" PORT_NAME="InFlow1" PORT_ID="1"/>
      <DST DFP_NAME="DFP2" DFP_ID="2" PORT_NAME="OutFlow1" PORT_ID="1"/>
    </CONNECTION>
  </CONNECTIONS>
</GXL>
```

Similarly, the set of checks generated for this same model are shown in Listing 4. These are again shown in an abstracted format, whereas in an actual environment they were mapped to a concrete scripting language (we generated executable Python scripts in this case). However, the information required to execute the checks is all shown here. We check that the output matches the metamodel. We check that all SID are unique and that each block has one. We generate a check for each DFP in the input that there is a matching number of blocks, and that one of them has the properties of each DFP in the input model for that test case. All these have been generated from the generator state that was built when producing the input model shown in Listing 3.

Listing 4: Example checks

```
metamodel validates
all SID are unique
BLOCKS = 2
TYPES = 2
TYPE1.NAME = DFT1
TYPE2.NAME = DFT2
BLOCK1.NAME = DFP1
BLOCK1.INPORT_COUNT = 1
BLOCK1.OUTPORT_COUNT = 1
BLOCK1.DESCRIPTION = Description1
BLOCK2.NAME = DFP2
BLOCK2.INPORT_COUNT = 4
BLOCK2.OUTPORT_COUNT = 4
BLOCK2.DESCRIPTION = Description2
CONNECTIONS=1
CONNECTION1=DFP1.1 − DFP2.1
```

To execute the test cases with the inputs and checks, we provide the generated input model to MetaEdit+ transformation engine, run the transformation and perform the checks on the output.

The test generation algorithms and coverage optimizers we use are described in more detail in [10]. For space reasons, we do not repeat them here. To briefly summarize, a large set of variants are generated and optimization algorithms are used to pick ones that best fulfill the defined coverage criteria.

To define the coverage criteria, we also use the mechanisms described in [10] specifically applied for transformation testing. The criteria were defined before in Section IV-A and illustrated in Table I. We use a hierarchy of coverage criteria, where we start with single model elements and their properties. For example, we represent the number of DFP in the input model as *DFP(X)*, X being the category identifier from Table I. Other elements are represented similarly, such as *DFP-InFlow(X)*. A single input model can satisfy several coverage requirements, such as *DFP(1)* and *DFP-InFlow(2)*.

Element combinations are represented similarly as pairs of different types. For example, *DFP-InFlow1+DFP-OutFlowN*, refers to having a DFP with one *InFlowPort* and N *OutFlowPorts*, and *InFlow0+InFlowN* refers to having to different DFP instances in the input model, one with no InFlowPorts and one with N of them. different DFP instances in the input model, one with X InFlowPorts and the other with Y OutFlowPorts. Given such coverage definitions, the test generator optimizes the set of generated input models to achieve high coverage over these.

The generator achieves full coverage to the criteria we defined in about 25 tests. As discussed before, this is the set of basic model elements and sets of variant combinations (about 300 combinations). This relatively small set of tests for relatively complex coverage criteria is possible due to the generator combining large sets of criteria is a single test (input model), and having capability to track their fulfillment across large sets and optimize for it.

Where required, the generation can also be guided towards specific areas if we find those areas especially important. For example, by using generator scenarios that define the parts of input to focus on in the overall model (by slicing the test model). As model transformation are also generally not very heavy to execute (when invoked directly), they can also be executed in large numbers fast, which makes them well suited for automated test generation and execution approaches.

The results for executing the generated tests, our results are similar to those for other systems. Many ambiguities and different interpretations are revealed during a systematic modeling process. This is often seen already as a major contribution. Beyond this, large scale generation and test coverage is in our experience very good at finding bugs in more complex interactions and corner cases.

V. Discussion

While we have described the overall process as consisting of iterative phases, this also applies at a more fine grained level. We have found it best to build small parts of the test model, focus on specific aspects, and iterate from these with additional parts, including coverage criteria and test oracles.

In terms of coverage optimization, our example is not the most complex we have seen. In very complex scenarios, the optimizer may not achieve 100% coverage of the possible combinations. However, in our experience it finds a very large number of them, and the ones generated have provided us with good results in including the important combinations to find potential issues. In [10] we evaluated the optimization algorithm also with tens of thousands of options, showing ability to achieve good coverage. A specific aspect to note is that the coverage criteria and combinations can be complex to achieve together with complex input test generation. This is one part of what makes the transformation testing an interesting and challenging area.

In the case of the EAST-ADL transformation using MetaEdit+ we have been able to use the limitations enforced by the modeling environment to limit the generated input models to mainly contain valid input models. This is true when we can assume the input models come from a trusted source and are used by responsible experts. When the transformation input models come from external sources, and can be created with any type of tool, we cannot assume they match particular metamodel or DSL constraints. In such cases, we should also consider invalid inputs more broadly.

Based on our experience, we have identified a number of different types of issues that can be wrong in the transformation input, and how these may be addressed in different modeling environments:

1) DSL constraints prevent creating illegal inputs
2) DSL warns about illegal input but allows creating it and running the transformation
3) DSL allows creating illegal input but prevents running the transformation
4) output is illegal but target tool (TT) opens it (showing errors)
5) output is illegal and TT does not even open it
6) output is illegal and TT does not recognize it
7) input is legal based on source metamodel but not on target metamodel

The first three of these describe how the modelling environment can manage the different forms of invalid input. Depending on how far these are implemented in the DSL, we can ignore related invalid inputs in our testing as the related constraints are already enforced. Options 4-6 are related to the target tool for which the transformation output is intended. These require a deeper understanding also for the target tool and its notations.

In the last four cases, the target tool will show different levels of errors for the invalid output models. Typically we have ruled out such cases from our testing as they are not valid. However, identifying such cases can be difficult and commonly not all this information is known to all parties. In our experience, this is something we learn over time in performing the testing process and executing the tests.

The last one is in our experience the most tricky one, as it requires detailed understanding of both the input metamodel and associated tooling as well as the output metamodel and associated tooling. For example, EAST-ADL allows multiple connections from a single input port to multiple output ports, but blocks targeted in Simulink forbids these.

The case study we presented in this paper is based on the MetaEdit+ environment and its transformation languages. We have also implemented this same process in the Eclipse ATL transformation environment. While the details are different, we have found the approach to be applicable across the different transformation environments. Differences include the more rule-oriented transformation definition language in ATL and a model navigator-oriented language in MetaEdit+.

The more distinct rules in ATL make it easier to perform white-box style analysis and to show more specific transformation rules to the test engineer. The more generic approach of MetaEdit+ makes it easier to build more powerful generators and to re-use elements across different subparts of the transformations. In both cases, the input metamodel provides a good starting point for providing test modeling assistance through our process.

The case studies we have done in this area have concerned performing the process manually with the help of the test generation tools. For wider adoption, we see it would be useful to implement support for the process in the actual modeling environments. This would include providing the user with means to go through the elements of the input and output metamodels and their properties, to use these to define coverage criteria and test oracles, and to integrate a test generator to generate tests from these.

VI. Conclusions

In this paper we described a process we have applied for testing model transformations. An attempt was made to keep it generic and allow using any available tools and techniques to be used to apply it. An example was then presented of a transformation from an EAST-ADL specification to a Simulink model. Depending on the support provided by the transformation framework used, the amount of effort required can vary. In the future we hope to see this type of support integrated as part of the modeling and transformation environments to make it more cost-effective and easier to apply for domain experts.

References

[1] B. Baudry, S. Ghosh, F. Fleurey, R. France, Y. Le Traon and J.-M. Mottu, Barriers to Systematic Model Transformation Testing, Communications of the ACM, vol. 53, no. 6, pp. 139-143, 2010.

[2] E. Brottier, F. Fleurey, J. Steel, B. Baudry and Y. Le Traon, Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool, Int'l. Symposium on Software Reliability Eng. (ISSRE), 2006.

[3] F. Büttner, M. Egea and J. Cabot, On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers, in Model Driven Engineering Languages and Systems (MODELS), 2012.

[4] B. Daniel, D. Dig, K. Garcia and D. Marinov, Automated Testing of Refactoring Engines, in ESEC-FSE, 2007.

[5] EAST-ADL Association, *EAST-ADL Domain Model v2.1.12*, http://www.east-adl.info/Specification.html, referenced April 2015.

[6] F. Fleurey, J. Steel and B. Baudry, Validation in Model-Driven Engineering: Testing Model Transformations, in 1st Int'l. Conf. on Model, Design, and Validation, 2004.

[7] F. Fleurey, B. Baudry, P.-A. Muller and Y. Le Traon, Qualifying Input Test Data for Model Transformations, Journal of Software and Systems Modelling, vol. 8, no. 2, pp. 185-203, 2009.

[8] C. A. Gonz, ATLTest: A White-Box Test Generation Approach for ATL Transformations, in Model Driven Engineering Languages and Systems (MODELS), 2012.

[9] M. Gogolla and A. Vallecillo, Tractable model transformation testing, in Proc. of ECMFA11, ser. LNCS, vol. 6698. Springer, 2011, pp. 221236.

[10] T. Kanstrén and M. Chechik, A Comparison of Three Black-Box Optimization Approaches for Model-Based Testing, in 5th Int'l. Workshop on Automating Test Case Design, Selection and Evaluation (ATSE), 2014.

[11] T. Kanstrén, *OSMO Tester Model-Based Testing Tool*, https://github.com/mukatee/osmo, referenced April 2015.

[12] T. Kanstrén, *EAST-ADL to Simulink Example Test Generator*, https://github.com/mukatee/dsm-mbt-example, referenced April 2015.

[13] H. Kern, A. Hummel and S. Kuhne, *Towards a Comparative Analysis of Meta-Metamodels*, The 11th Workshop on Domain-Specific Modeling (DSM), 2011.

[14] D. S. Kolovos, R. F. Paige and F. A. C. Polack, Model comparison: A Foundation for Model Composition and Model Transformation Testing, in Workshop on Global Integrated Model Management (GaMMa), 2006.

[15] J. Mottu, B. Baudry and Y. Le Traon, Model Transformation Testing: Oracle Issue, in ICST workshops, 2008.

[16] M. Mottu, S. Sen, M. Tisi and J. Cabot, Static Analysis of Model Transformations for Effective Test Generation, in 23rd Int'l. Symposium on Software Reliability Engineering (ISSRE), 2012.

[17] G. M. K. Selim, J. R. Cordy and J. Dingel, Model Transformation Testing: The State of the Art, in Workshop on the Analysis on Model Transformations (AMT), 2012.

[18] G. Soares, R. Gheyi and T. Massoni, Automated Behavioral Testing of Refactoring Engines, IEEE Transactions on Software Engineering, vol. 39, no. 2, pp. 147-162, 2013.

Approaches to automated test implementation in model-driven test automation architectures

Marc-Florian Wendland
Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin
+49 30 3463-395
marc-florian.wendland@fokus.fraunhofer.de

Abstract:
Automated test execution, in particular represented by keyword-driven testing, is long established in industry and research. Automated test design, better known as model-based testing or test generation, was the subject of intensive research in the last ten to fifteen years. Scarce work was done in the area of systematic approaches to automated test implementation, though. Test implementation summarizes the activities that derive executable test cases from high-level test case specifications. As such, the test implementation assumes an important role within the entire test process. Depending on the level of abstraction applied for test design and the desired test execution method (automated, manual), the test implementation activities vary from simple in-mind translations for manual test execution to formally specified mapping rules for automated test execution.
In this paper, we identify, specify and compare different approaches to automated test implementation that act as a mediator between automated test design and automated test execution. We will, in particular, emphasize the prerequisites of each approach and the circumstances under which these approaches positively affect the overall level of test automation in an otherwise fully automated test design and execution environment.

Introduction

In the recent years many approaches and methods to automated test design, better known as model-based testing (MBT), have been described and published in numerous scientific and industrial work. MBT has been a very active research field, though, within the last ten to fifteen years. The increased interest in model-based approaches to test generation could indeed be related to the release of UML 2 back in those days and the promising idea of OMG's model-driven architecture (MDA) [14]. Whereas automated test execution – first and foremost the idea of keyword-driven testing (KDT) to be mentioned – was already adopted by the industry back in the late 1990s [6], MBT allowed further increasing the overall degree of automation in software testing by automating the test design activities. Between the test design and test execution phase, the test implementation activities take place. The ISTQB [10] assigns a dedicated phase in its fundamental test process for the test implementation activities. ISO 29119 [4] instead merges test design and test implementation into a single phase. The underlying task and purposes of the test implementation activities, however, remain the same: concretization of high-level test case specifications obtained from the test design phase into technical test cases that can be executed by a test execution system. Test implementation can, thus, be classified as a transformation either automated or manually performed.

In a test automation architecture that is based on MBT for automated test design and KDT for automated test execution the result of the test implementation activities are keyword test cases that use keywords organized in libraries, and an adaptation layer that provides the implementation of the utilized keywords. The test execution system is then responsible to run the test cases against the system under test (SUT). The efficiency of the test implementation activities are, thus, vital for the overall efficiency of the test automation architecture. To obtain the best efficiency, the test implementation activities should be performed in an automated manner as well. Interestingly, there is almost no work available that systematically analyse and distinguish automated approaches to test implementations in an otherwise automated test architecture. Our work describes different approaches to automated test implementation in such a MBT-/KDT-based test automation architecture.

The contributions of this paper are:

- Analysis of abstraction levels and their dependencies in a model-driven test automation architecture based on MBT and KDT
- Definition of different approaches to automated test implementation that are based on abstraction levels of test models and model transformations
- A comparison of these approaches regarding their prerequisites, complexity, advantages respectively disadvantages.

The reminder of this paper is structured as follows: Section 2 summarizes the related work in the field of systematic approaches to test implementation automation. Section 3 depicts our understanding of abstraction levels usually found in a test automation architecture that is realized by MBT and KDT. Section 4 builds upon our understanding of such a test automation architecture we have faced so far and describes the different approaches to automated test implementation. Section 5 finally summarized this paper and provides a view on future work in that area.

Related Work

The related work that deals with a systematic classification and comparison of automated test implementation approaches is limited. To the best of our knowledge, there is no work available that systematically address different approaches to automated test implementation based on (test) models. Model-based ([7], [8], [2], [1]) and keyword-driven ([6], [4] part 5, [12]) testing has been discussed, partly also the combination of the two [13]. However, none of this work addressed or compared the test implementation approaches applied dedicatedly. Our work, in contrast, systematically distinguishes different approaches to test implementation and, thus, helps test engineers make decision, respectively foresee potential impacts of their decision that influences the architecture of model-driven test automation solutions.

Abstractions in test automation architectures

A test automation architecture describes a framework that facilitates the automated execution of different activities in a test process. Activities that can be automated usually refer, among others, to test design, test implementation, test execution, test planning and test execution scheduling or test evaluation activities. A reference framework for a generic test automation architecture was describes by the ISTQB [9]. It states, and we agree, that automation is almost always based on abstraction. Figure 1 shows a condensed view on that reference framework including the ideal abstraction level of different artifacts produced or used in such a framework. Ideal means that the actual abstraction level of those artifacts may vary in different realizations of such a framework, depending, among others, on the approach for automated test implementation. Moreover, Figure 1 depicts also the different phases or activities and their outcomes in an MBT- and KDT-based test automation architecture.

The SUT resides on the lowest level of abstractions. It represents a concrete implementation of the *system requirements specification* (SRS), which is located on the highest level of abstraction. The analysis of the test basis from a tester's point leads to a test model that realizes the SUT's expected behaviour and serves as the input for an automated derivation of test cases. It contains the *logical actions* that can be performed on the SUT. Logical means that it is not yet clear, how a particular feature is or was realized in the SUT. The corresponding test cases are, thus, called *logical test cases*, capturing the aspects of the SUT that shall be tested, not how these aspects are tested technically. Hence, the test model and its resulting test cases reside on the same level of abstraction. Afterwards the test implementation phase turns the logical test cases into two different parts: *technical test cases* and the *adaptation layer*. Technical test cases are machine-readable artifacts of the test execution system. They can be as sophisticated as a TTCN-3 module or as simple as an XML file. The test execution system must be able to execute the technical test cases. The adaptation layer is an essential part of every KDT approach. It comprises both a library of known keywords that can be used to build technical test cases (upper interfaces, abstraction level 2), but also an implementation of these keyword libraries so that that the test execution is eventually able to execute the keywords against the SUT (lower interfaces, abstraction level 0). The adaptation layer is, thus, kind of a transformation that turns the technical test cases into invocations against the actual interfaces of the SUT at the lowest abstraction level. Technical test cases together with the adaptation layer form the *executable test cases*.

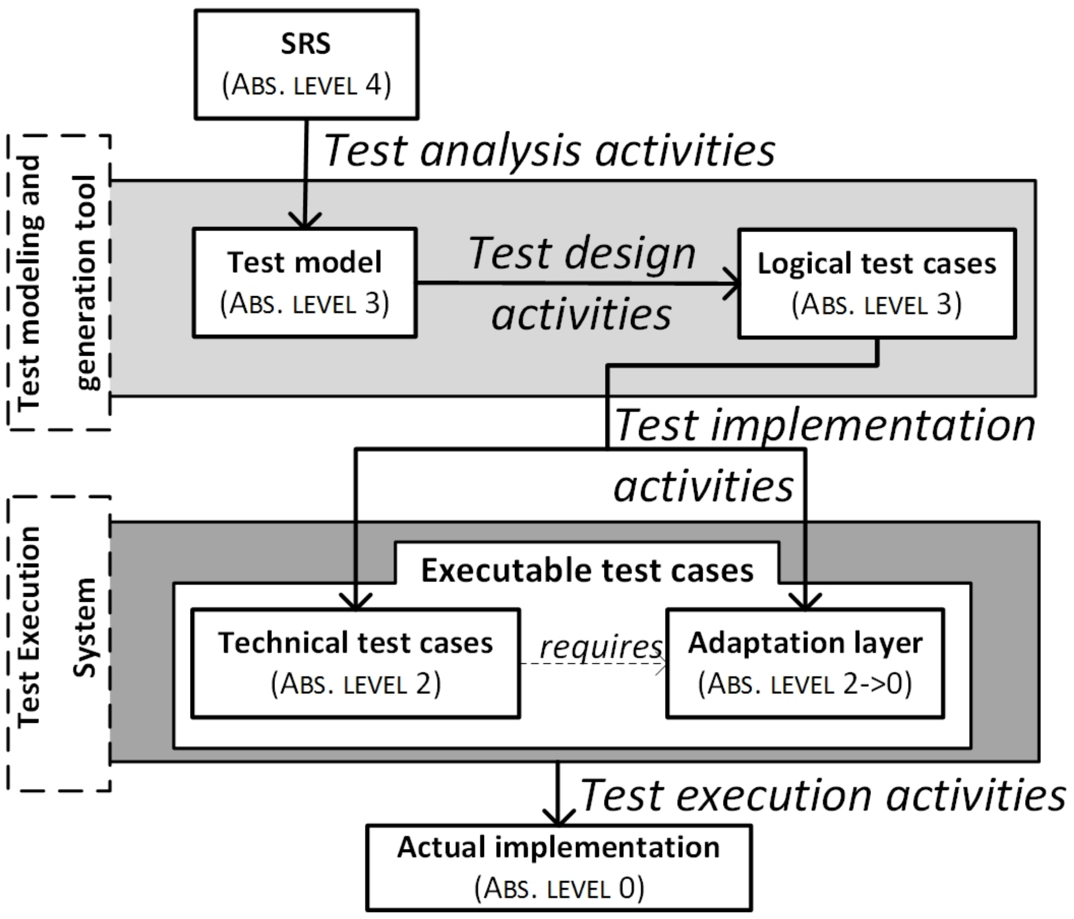


Figure 1. Abstraction levels in a test automation architecture

Approaches to automated test implementation

The adaptation layer heavily influences the approaches to automated test implementation. First and foremost, the existence (or non-existence) of an adaptation layer has to be clarified. If there is already a working adaptation layer (or supposed to be, e.g., obtained from a commercial vendor), the test implementation approach should consider it for being immediately executable of the technical test cases. This means, in turn, that the degree of freedom with respect to the realization of the approach is limited. In the end, the approach has to produce technical test cases that fit with the adaptation layer. If there is no adaptation layer available nor supposed to be available in near future, the restrictions on the test implementation approach are less strict. In such cases, the adaptation layer might be implemented manually or completely derived from the test model layer.

We distinguish at least three different approaches to automated test implementation: *Top-down*, *bottom-up*, and *meet-in-the-middle* (see Figure 2). We reduced this figure to the essence of what is needed for the understanding of our work. Since the test model and the logical test cases reside on the same level of abstraction, they are merged into the concept *test model layer*. The SUT is not shown at all because the test implementation approach has no direct influence on the SUT but on the technical test cases and the adaptation layer. The test implementation approaches are differentiated in greater detail in the following subsections regarding the following characteristics: Description, prerequisites, complexity, pros, cons, recommendation, and examples.

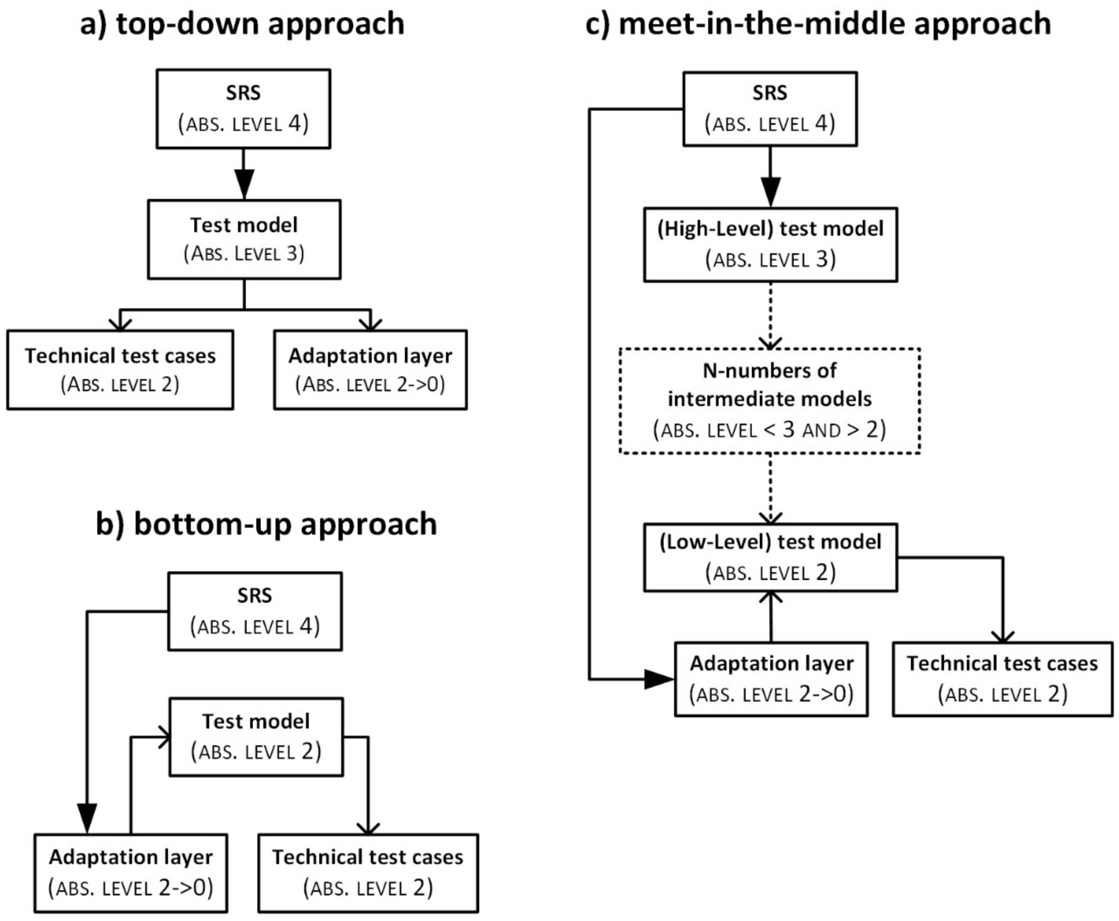

a) top-down approach
SRS (ABS. LEVEL 4)
Test model (ABS. LEVEL 3)
Technical test cases (ABS. LEVEL 2)
Adaptation layer (ABS. LEVEL 2->0)
b) bottom-up approach
SRS (ABS. LEVEL 4)
Test model (ABS. LEVEL 2)
Adaptation layer (ABS. LEVEL 2->0)
Technical test cases (ABS. LEVEL 2)
c) meet-in-the-middle approach
SRS (ABS. LEVEL 4)
(High-Level) test model (ABS. LEVEL 3)
N-numbers of intermediate models (ABS. LEVEL < 3 AND > 2)
(Low-Level) test model (ABS. LEVEL 2)
Adaptation layer (ABS. LEVEL 2->0)
Technical test cases (ABS. LEVEL 2)

Figure 2. Approaches to automated test design

Top-Down Approach

Description. In the top-down approach (see Figure 2a) the test model layer assumes a dominant position. Neither the test execution system nor the adaptation layer do not have an impact on the test model layer, which is based on the logical actions described in the test basis. The abstraction level of the test model layer need to be harmonized with the interface and type system of the SUT. This approach is very often applied and described in scientific work, due to its simplicity with respect to the required test environment.

Prerequisites. This approach has minimal constraints on how to actually executed the test cases. Neither does it presume a certain adaptation layer, nor does it enforce a certain test execution system. The modeling language to constitute the test model layer should at least be capable of describing the logical actions of the SUT as a behavioural specification (e.g. state machines, transition system). If the adaptation layer shall be generated from the test model layer, too, the modeling language must, in addition, allow specifying the SUT interfaces and its type system. The UML Testing Profile (UTP) [3] is a good example for a modeling language that meet both prerequisites.

Complexity. The test implementation transformation consists basically of a model-to-text-transformation that generates the technical test cases out of the test model layer. This transformation is usually rather straightforward for no other impacts have to be considered. If, however, the adaptation layer shall be automatically generated, too, this approach becomes more sophisticated. The modeling language would have to be capable to express three abstraction levels (test model, technical test cases and SUT) as well as a mechanism to link test definitions (i.e., interface and type definitions) on different layers on abstraction and a dedicated mapping among these layers. Automated generation of adaptation layers, however, is not consider as the main scenario for this paper.

Pro. Since the test model layer reflects the logical actions of the SUT obtained from the test basis, the automated design and implementation can be performed even before the SUT is even available. In addition, this approach does not influence the abstraction level of the test model. Any abstraction level that is appropriate for the test analysts can be targeted. Finally, this approach grants the highest degree of freedom regarding which test execution system or (programming) language to use for test execution. There does not even have to be a dedicated test execution system utilized. A plain programming language (or even executable modeling language) is sufficient to realize this approach.

Con. The biggest disadvantage of the top-down approach is the lack of reusability of the test implementation transformation. The transformation was designed to tie the test model layer on a certain abstraction level directly to the SUT. Thus, any other SUT implementation or abstraction level would require re-work of the test implementation transformation. Moreover, if the adaptation layer shall be generated from the test model layer, the transformation would have a high complexity. If the adaptation layer shall be implemented manually, it would be resource-consuming.

Recommendation. This approach is often used in research projects and for early proof-of-concepts due to its simplicity regarding the integration of test design and test execution and the minimal constraints on the employed test automation solution. This approach can be beneficial nonetheless, when no suitable adaptation layer is given for the SUT implementation and reuse of the test adaptation layer for other test automation architectures is not required or intended.

Example. In the EU project REMICS, this approach was successfully applied [5]. The UTP test model was translated into TTCN-3 scripts and the according adaptation layer was implemented manually.

Bottom-Up Approach

Description. The bottom-up approach (see Figure 2b) constitute the counterpart of the top-down approach. The adaption layer assume a dominant position, thus, the interfaces and type system of the adaptation layer is pulled up into the test model layer and directly influences the design of the test model and the logical test cases. The test model layer and the upper interfaces of the test adaptation layer reside on the same abstraction level.

Prerequisites. By definition, this approach demands the existence of an adaptation layer and test execution system. The test modeling language need to offer concepts to describe the technical test cases and the interfaces and type system of the adaptation layer, but not of the SUT.

Complexity. The test implementation transformation usually consists of both a re-engineering transformation (adaptation to test model layer) and a transformation from logical to technical test cases (model-to-model or model-to-text transformation, depending on the format of the technical test cases). The transformation of test cases is usually straightforward, for the logical test cases are located on the same abstraction level as the technical test cases. The type system and interface definitions of the adaptation layer can be automatically pulled up to the test model level in many cases.

Pro. The biggest advantage is that the generated test cases are guaranteed to being immediately executable because of the dominant position of the adaptation layer. Whenever changes to the SUT have to be incorporated by the adaption layer, the test model layer can be (in many cases) automatically updated and test cases re-generated. The complexity of the transformation of test cases is rather low.

Con. The inherent disadvantage of this approach is that the test model layer is depending on the adaptation layer. The test model layer does not (necessarily) constitute the most suitable abstraction level of the logical actions, but the one obtained from the adaptation layer. This is reflected in the deviation of the ideal abstraction level of the test model layer in Figure 1 (i.e., abstraction level 3) and the effective one in this approach in Figure 2 (i.e., abstraction level 2). If the adaptation layer is poorly abstracted (i.e. the interfaces and type system contain too low-level implementation details), the test model would suffer from the same poor abstraction. This can result in test models that are difficult to write, read, maintain or understand for test analysts.

Recommendation. The application of the bottom-up approach shall be used when the adaptation layer is located on an appropriate abstraction level that efficiently abstracts from the actual implementation.

Examples. In the EU project MIDAS [11] the bottom-up approach was realized. In short, the application of the bottom-up approach was successful, because the abstraction level of the WSDL-based adaptation layer was appropriate for the test analysts to build test models and generate test cases from it. The test model layer was automatically re-engineered from the adaptation layer using an MDA-inspired process.

Meet-in-the-middle Approach

Description. The meet-in-the-middle approach (see Figure 2)) builds upon the top-down and the bottom-up approach. It leverages either advantages and mitigates either shortcomings by splitting the test model layer into a high-level and a low-level test models. The high-level test model is derived from the test basis on an appropriate abstraction level for the test analysts (similar to the top-down approach). In addition, the adaptation layer is re-engineered into the low-level test model that facilitates automated test execution (similar to the bottom-up approach).

Prerequisites. This approach requires the existence of both a capable test basis and an existing (or anticipated) adaptation layer and test execution system. In between the high-level and the low-level test model a number of model-to-model-transformations (at least one) are required that gradually refines the logical test cases into technical test cases, respectively executable ones. Furthermore, the test modeling language must offer concepts to describe both abstraction levels. This approach requires by definition a model-to-model transformation.

Complexity. There is an inherent higher complexity in the meet-in-the-middle approach compared to the former to approaches. There at least two rather independent test model layers that need to be integrated with each other. Opposed to what we have said so far, the meet-in-the-middle approach requires by definition a model-to-model transformation. The more intermediate models are involved, the more complex (but flexible) becomes the automated test implementation.

Pro. First, the combined advantages of the top-down and bottom-up approaches guarantees an appropriate abstraction level for test analyst (efficient automated test design) and ensure being immediately executable based on the existing adaptation layer (efficient automated test execution). This approach can be realized even when no adaptation layer is available yet, because of the separation of high-level and low-level test models. This fosters early testing. Another benefit of the layered test models is that even the low-level actions (i.e., the actions contained in the low-level test model) can be composed into a more complex ones. Such macro-like actions can be utilized as library to encapsulate complex communication with the SUT in a single action without losing the guarantee of being executable thanks to the model-to-model transformations in between any test model.

Con. The biggest (and most probably sole) disadvantage is the higher and initial complexity of the test implementation approach. It is, in fact, the only approach that inherently requires model-to-model transformations or, even more sophisticated, a complete MDA framework including traces among each test model level. The realization of such a test automation architecture will be quite costly when applied the first time.

Recommendation. The approach shall be used when the abstraction level of the adaptation layer is not feasible for the test model and the project has a long duration with a most likely evolving SUT. If an MDA framework shall be applied, a meta-modeling technology shall be used that natively supports the idea of the MDA including superior facilities for implementing model-to-model transformations. The high complexity of this approach is payed off by several benefits, though.

Example. In MIDAS [11], the application of the data fuzzing techniques has been realized on this approach. High-level data fuzzing strategies have been defined and applied on the high-level test model, agnostic of any (test) programming language. Finally, a model-to-model transformation was developed that resolved these strategies into concrete TTCN-3 external libraries and invocations thereof.

Conclusion

In this paper, we have identified different approaches to automated test implementation as a link between automated test design (i.e., test generation) and automated test execution. We have compared the approaches with respect to their prerequisites, complexity, pros and cons. The assessment of the complexity of each approach was based on abstractions levels. These abstraction levels have been discussed in the context of a generic test automation architecture inspired by the ISTQB [9].

We have argued that the top-down approach is the most intuitive, but least feasible approach for rigor industrial application due to the lack of flexibility. Due to its isolated nature, this approach is predestined for (academic) proof-of-concepts without the need of considering existing environments. Apart from these settings, we do not recommend to utilize the top-down approach. The bottom-up approach is built upon an automated test execution environment and pulls the adaptation layer into the test model layer. This guarantees that the generated test cases are immediately executable since the actions in the test model reflect the logical abstraction level of the adaptation layer. On the downside, this could easily lead to test models with an inappropriate, yet too low, abstraction level. The meet-in-the-middle approach mitigates the shortcomings of either approach by introducing at least two test models that are located on different levels of abstraction. As a result, test analyst are able to create test models that are easy to maintain, to read and write, and the technical test analyst has to face less abstraction for realizing the executable test cases. In between the high-level and low-level test models any number of test models on with lower, respectively, higher abstraction level can reside. Among those abstraction levels, transformation need to be employed to actually *meet in the middle*.

Future work will in particular address a systematic and empirical evaluation of the bottom-up and meet-in-the-middle approach. We think that these two approaches have the highest potential for being adopted by the industry in order to transition from automated text execution to test automated test design. We are going to analyse this transitions in greater detail in future. The idea is to define a process framework that guide test engineers along their path from automated test execution towards automated test design by relying on one of the two approaches for automated test implementation.

References

[1] Prenninger W., and Pretschner, A., Abstractions for Model-Based Testing, in International Workshop on Test and Analysis of Component Based Systems (TACoS 2004)..

[2] Pretschner, A. and Philipps, J., Methodological Issues in Model-Based Testing. in Model-Based Testing of Reactive Systems. Springer, 2004.

[3] Object Management Group (OMG): UML Testing Profile. URL: http://www.omg.org/spec/UTP

[4] International Organisation for Standardisation (ISO): ISO/IEC 29119, Software Testing Standard, http://www.softwaretestingstandard.org

[5] Wendland, Marc-Florian et al., Model-based testing in legacy software modernization: an experience report, in Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to testing Automation (JAMAICA 2013). JAMAICA'13, July 15, 2013, Lugano, Switzerland.

[6] Foster, M. and Graham, D., Software Test Automation. Addison-Wesley Professionals, 1999. ISBN: 978-0201331400.

[7] Utting, M.; Pretschner, A., Legeard, B.: A Taxonomy of Model-Based Testing. ISSN 1170-487X, 2006. http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf.

[8] Utting, Mark; Legeard, Bruno, Practical Model-based testing – A Tools Approach, Elsevier, 2007.

[9] International Software Testing Qualifications Board (ISTQB): Certified Tester Expert Level Syllabus – Test Automation - Engineering. Version 2014.

[10] International Software Testing Qualifications Board (ISTQB): Certified Tester Foundation Level Syllabus. Version 2012.

[11] Wendland, Marc-Florian; Schneider, Martin; and Hoffmann, Andreas. A model-driven approach to test automation for SOA systems. *Accepted for STTT Special Issue Model-based Testing on the Cloud.* Springer, Heidelberg, 2015.

[12] Takala, Tommi; Maunumaa, Mika; and Katara, Mika. An Adapter Framework for Keyword-Driven Testing. In: Proceedings of the Ninth International Conference on Quality Software (QSIC) 2009, IEEE Computer Society, 2009.

[13] Pajunen, Tuomas; Takala, Tommi; and Katara, Mika. Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework. Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST) 2012, Germany.

[14] Object Management Group (OMG). MDA Guide rev. 2.0, URL: http://www.omg.org/cgi-bin/doc?ormsc/14-06-01.pdf, last visit: 16th September, 2015.

Service functional test automation

Lom Messan Hillah, Ariele-Paolo Maesano
Laboratoire d'Informatique de Paris VI, Sorbonne Universités, UPMC
lom-messan.hillah@lip6.fr, ariele.maesano@lip6.fr
Fabio De Rosa, Libero Maesano
Simple Engineering France SARL
fabio.de-rosa@simple-eng.com, libero.maesano@simple-eng.com
Marco Lettere, Riccardo Fontanelli
Dedalus S.p.A
marco.lettere@dedalus.eu, riccardo.fontanelli@dedalus.eu

Abstract:
This paper presents the automation of the functional test of services (black-box testing) and services architectures (grey-box testing) that has been developed by the MIDAS project and is accessible on the MIDAS SaaS. In particular, the paper illustrates the solutions of tough functional test automation problems such as: (i) the configuration of the automated test execution system against large and complex services architectures, (ii) the constraint-based test input generation, (iii) the specification-based test oracle generation, (iv) the intelligent dynamic scheduling of test cases, (v) the intelligent reactive planning of test campaigns. The paper describes the usage of the MIDAS prototype for the functional test of an operational distributed application in the domain of healthcare.

Introduction

Services are everywhere. They are involved in services architectures built of service components that: (i) expose *service APIs*, (ii) interact through service protocols (REST/XML, REST/JSON, SOAP…) and (iii) are deployed independently of each other. The SOA approach has been used for fifteen years to let distributed vertical applications cooperate. More recently, systems have exposed service APIs for interaction with mobile apps. Presently, the internal structure of applications, once monolithic, is going to be designed as a micro-services architecture [9] that is particularly well adapted for cloud deployment. Services are loosely coupled, allowing agility of design, development, integration (continuous integration - CI), delivery (continuous delivery - CD) and deployment.
The **Calabria Cephalalgic Network** (CCN) [3] is a multi-owner distributed application that supports the *headache integrated care processes*,

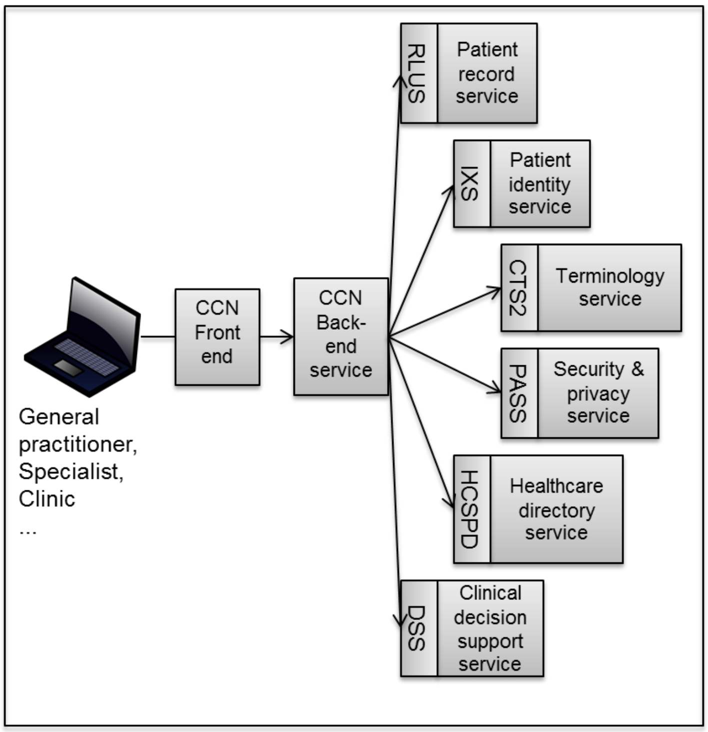

General practitioner, Specialist, Clinic ...
CCN Front end
CCN Back-end service
RLUS
Patient record service
IXS
Patient identity service
CTS2
Terminology service
PASS
Security & privacy service
HCSPD
Healthcare directory service
DSS
Clinical decision support service

Figure 1. Calabria Cephalalgic Network.

effectively coordinating different care settings (general practitioners, specialists, clinics, labs...) in a patient-centred vision. The application is designed as a services architecture (Figure 1), is operational today and its components' services are physically deployed in different data centres. The service APIs are compliant with the HL7/OMG HSSP international standards (RLUS, IXS, CTS2...) [13].

Dedalus [12], a company specialised in healthcare systems, is in charge of the provision within the CCN of the Patient record, the Patient identity and the Terminology services. CCN service-oriented architecture allows Dedalus to put in place a modular integration process with one separate source code repository and one separate build per service component.

Actually, the service integration process is a full testing process, constituted of all the testing activities: functional, security, fault tolerance and performance test. In order to improve agility and time-to-market, these activities shall be organised in an optimized manner. The service integration and delivery process pattern that is becoming popular is the *CD pipeline* [15], in which the testing activities are placed as *stages* between the *service build* formation and its deployment in the production environment. The transition from a stage to the next is permitted only whether the stage tests *pass*, otherwise the sequence is interrupted and restarts with the check-in of the updated code. An example of service CD pipeline is sketched in Figure 2.

The test tasks in each stage and the chosen sequence of the test stages can and should maximise the effectiveness (the *fault exposing potential* and the *troubleshooting efficacy*) and the efficiency (the *fault detection rate*) of the testing tasks. Test effectiveness and efficiency are important even for completely automated stages - to say nothing about manual ones - that can be heavyweight and can slow the entire process.

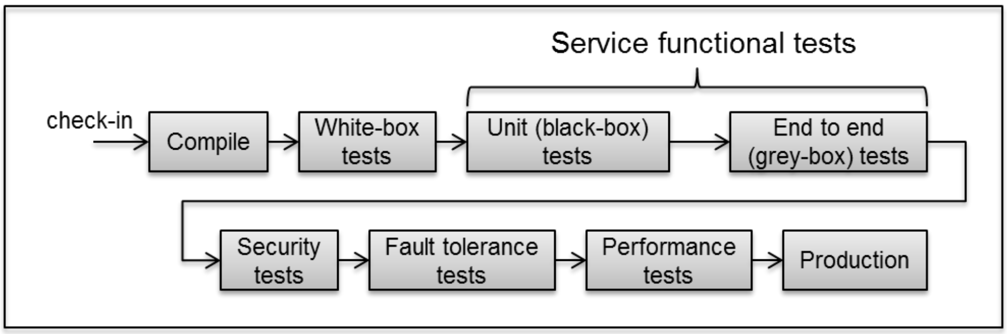


Figure 2. Service integration process as a pipeline.

In the CD pipeline sketched in Figure 2, the successfully constituted build is firstly submitted to acceptance white-box tests. All the subsequent test stages target different aspects of the service external behaviour and are independent of the service implementation technology. The subsequent two stages are about functional test and are detailed in the section '**Automated functional test**'. The security tests follow - they can be effective and efficient only whether the service build passes the functional tests. The last two stages are about quality of service: the fault tolerance tests challenge the *resilience* of the service implementation in the face of failures of the underlying computing resources or the unavailability of the services it interacts with. Lastly, the performance tests concern mainly the service invocation and provision *latency*. The CD pipeline can be more or less automated. A single *CD pipeline stage* can be fully automated whether: (i) its internal tasks can be fully automated and produces automatically a meaningful report and (ii) the automated tasks can be invoked through APIs by the CD server (for instance Jenkins [14]).

This paper reports a solution of automation of service functional test. The section '**Related work**' gives the motivation for doing research on the topic and a short review of the state of the art. The section '**Automated functional test**' presents the prototype developed within the MIDAS project and provided as-a-service by the MIDAS SaaS [16]. Dedalus has incorporated the functional test automation services in its integration process: this experience is presented in the '**Prototype usage in an operational environment**' section. The '**Conclusion**' discusses major advantages and drawbacks of the new solution and outlines future work.

Related work

Service test and, in particular, end-to-end test of complex services architectures is difficult, knowledge intensive, hard to manage and

expensive in terms of labour effort, hardware/software equipment and time-to-market. Since the inception of the service orientation, service testing automation has been a critical challenge for researches and practitioners [2] [1] [11]. In particular, tasks such as: (i) automated optimised generation of test inputs [2], (ii) automated generation of test oracles [1], and (iii) optimised management of test suite for different test activities - such as first testing, re-testing, regression testing [11], has not yet found automation solutions that can be applied to real complex services architecture such as those that are implemented in healthcare [3].
Model-based testing (MBT) utilises formal models (structural, functional and behavioural) of the services architecture under test to undertake the automation of the testing tasks [5]. The "first-generation" MBT research is essentially focused on test input generation. More recently, formal methods, especially SAT/SMT-based techniques have been leveraged [6] that allow the exhaustive exploration of the system execution traces, and efficient test input generation satisfying constraints (formal properties expressed in temporal logic). Jehan and colleagues [6] use a constraint solver to compute the expected inputs for each particular execution of the business process as extracted from the control flow graph.
The MIDAS approach to the prioritization of test cases [11] is entirely original [8]: it is based on the usage of probabilistic graphical models [10] [7] in order to dynamically choose the next test case to run on the basis of the preceding verdicts. Moreover, the scheduler is able to establish a dynamic relationship between test case prioritization and the generation of new test cases, by supplying on the fly to the generator evidence-driven directives based on the preceding verdicts.

Automated functional test

Test environments

The service functional test automation is illustrated through an example of a simplified services architecture related to the CCN application (Figure 3). In order to provide its service, eHealth service consumes the Patient record, the Patient identity and the Terminology services. These services that are

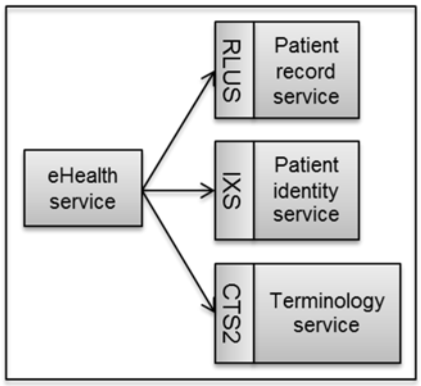


Figure 3. Services architecture under test.

not consumers of other services are called *terminal services*.

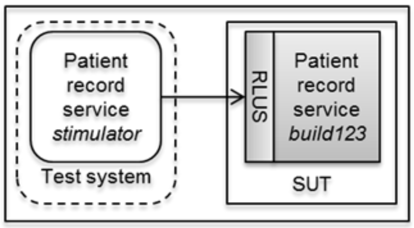


Figure 4. Test environment for terminal service unit test.

Unit test stage

The unit test stage includes the following tasks: (i) produce test inputs (stimuli), (ii) produce test oracles (expected outcomes), (iii) deploy and initialise the build in an appropriate environment (service under test - SUT), (iv) configure and generate the test system, (v) bind the test system with the SUT, (vi) run test cases (transmit stimuli, collect and log outcomes), (vii) arbitrate test outcomes against test oracles, (viii) schedule test case runs (dynamic scheduling), (ix) plan test campaigns (reactive planning) and (x) report test campaigns. For every terminal service, the unit test environment architecture is similar to that sketched in Figure 4.

For non-terminal services, such as the eHealth service, the typical unit test environment is depicted in Figure 5. The test tasks involved in the stage are the same as those for terminal services, but in the test system are generated, in addition to the *stimulator*, three *mocks* that “virtualise” the downstream services. The binding sub-task enables the mocks receipt the requests of the eHealth service, and send back the canned responses. The test system must be able to evaluate against the oracles that the requests that are issued target the appropriate services, are in time, are in the exact sequence and are the right ones.

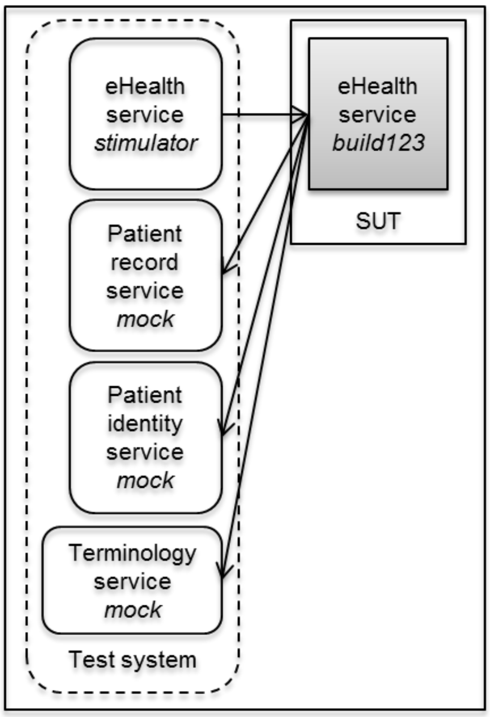


Figure 5. Test environment for non-terminal service unit test.

End-to-end test stage

The services architecture under test (SAUT) distributed environment is deployed with the lastest release builds of the downstream services. In the test system are generated the *interceptors* that catch the exchanges forth and back between the eHealth service and the downstream services (Figure 6). This test environment is put in place in the end-to-end stages of the CD pipelines of all the services involved in the SAUT, including the terminal services. An interesting point is that the end-to-end tests can highlight functional failures of any of the SAUT services - not only of the service of the pipeline

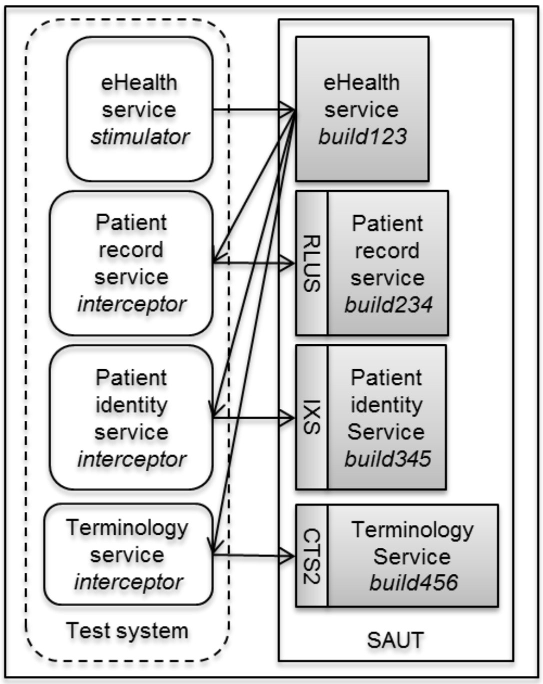

eHealth service *stimulator*
Patient record service *interceptor*
Patient identity service *interceptor*
Terminology service *interceptor*
Test system
eHealth service *build123*
RLUS
Patient record service *build234*
IXS
Patient identity Service *build345*
CTS2
Terminology Service *build456*
SAUT

Figure 6. Test environment for end to end test.

in which the stage is accomplished - and so eventually reveal *service tight coupling* - when a change in one service produces an unexpected failure in another service.

End-to-end testing of multi-owner services architecture requires *collaborative testing projects* that involve all the service owners and that explore systematically the cooperation scenarios between all the services. Systematic end-to-end testing campaigns are mandatory for first testing of new distributed applications, but are also recommended as regular activities of re-testing and regression testing. A collaborative testing project involving all the owners of the CCN application services is in progress.

Test automation methods

The MIDAS functional test prototype brings automation solutions (*test automation methods*) for the most critical test tasks: (i) configuration of the test system against distributed services architectures, (ii) test case (input/oracle) generation based on constraint propagation and symbolic execution, (iii) intelligent dynamic test case prioritisation and scheduling, (iv) intelligent reactive planning of test campaign with on-the-fly, evidence-based generation of new test cases. These test automation methods are provided as services by the MIDAS SaaS.

Automated configuration of the test system

The structure of the test system (stimulators, mocks, interceptors) is automatically generated from the SAUT model and the test configuration model. The former model is represented through an XML document depicting the *actual components* of the SAUT and the *actual wires* between them – interaction links that are typed by *service specifications* (e.g. WSDL documents). The latter model is obtained from the former model: (i) by adding *virtual components* (stimulators, mocks) and the corresponding *virtual wires* to actual components and (ii) by designating the *actual wires to be observed* (interceptors).

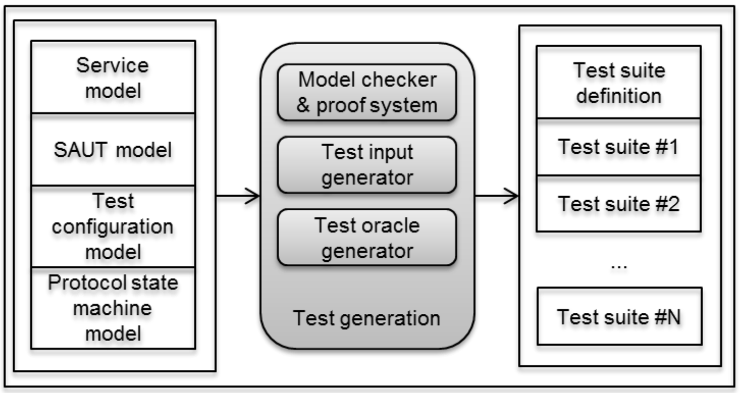


Figure 7. Automated generation of test cases.

Automated generation of test cases
Each SAUT component is equipped with a *protocol state machine* (PSM), modelled as a Harel state-chart [4], that represents the *interaction states* of the component and the *transitions* triggered by received messages (*events*), filtered by conditions (*guards*) and producing *effects* described as data-flow transfer functions. The service component PSMs are represented through standard SCXML documents [17] and the conditions and transfer functions are expressed in Javascript.
Test cases (inputs and oracles) are generated from the set of models (Figure 7). The test cases generation process relies on model-checking the PSM models using TLA+ [18], a well-known formal specification language based on temporal logic. TLA+ is backed by the TLC model checker to exhaustively check correctness properties across all possible executions of the system and by the TLAPS proof system that relies on SMT (Satisfiability-Modulo Theory) solvers for checking TLA+ proofs. The PSMs and the generation parameters are translated into a TLA+ companion algorithm language (PlusCal) that is afterwards compiled into TLA+. Through assertions, *execution traces* of the system that match some criteria - for instance where messages of some specific types, or containing some specific values, are exchanged - are requested to the proof system. Input data are then extracted from the execution traces and fed to the SCXML engine, which executes the PSMs for the scenarios triggered by the input data and produces the related oracles.

Automated dynamic scheduling of test runs
Automated dynamic scheduling takes places in the MIDAS test system that is equipped with automated execution and arbitration of test cases (Figure 8). In this context, the scheduler is able to choose the next test case to run on the basis of the past test verdicts. The cycle schedule/execute/arbitrate continues until there are no more test cases to run or some halting condition is met. The objectives of dynamic scheduling are (i) precocious detection of failures and (ii) localisation of faulty elements (troubleshooting).

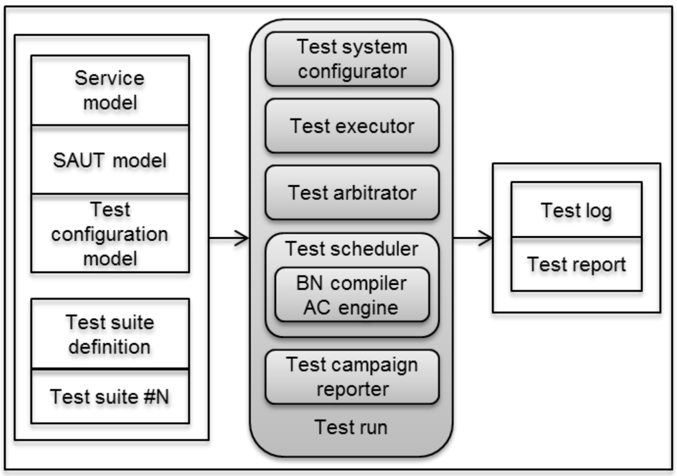


Figure 8. Automated scheduled execution of test cases.

The dynamic scheduler builds a Bayesian Network (BN) model [10] from (i) the SAUT model, (ii) the test suite and (iii) user's beliefs on the SAUT. The BN is compiled into an Arithmetic Circuit (AC) [7]. At each test run the verdicts are inserted as *evidences* in the AC and the subsequent inference calculates a *fitness* probability for each remaining test case that, combined with a scheduling policy (e.g. *max-fitness*, *max-entropy*...), allows the scheduler to choose the next test case.

Automated reactive planning of test campaigns
The idea behind a fully automated workflow for functional testing is to use the scheduler to drive not only the choice among a set of existing test cases but also the generation of new test cases. The test campaign starts with a minimal test suite and, on the basis of evidences (verdicts) brought from the past test runs, the scheduler calculates the degree of ignorance (Shannon entropy) on SAUT elements and recommends the generation of test cases whose execution would diminish this ignorance. This feature is operational and its usage in test campaigns is in progress.

Prototype usage in an operational environment

Dedalus currently utilises a home-made framework for service unit testing that has already significantly shrunk the effort of manually producing and executing test cases and test suites. The major limitations of this solution can be labelled as: (i) “test case overhead”, (ii) “unit testing only”, (iii) “lack of planning and scheduling”, (iv) “manageability”. The “test case overhead” issue relates to the necessity of creating a huge amount of test cases since the services to be tested (such as RLUS) are specified as *generic* and the payload structure varies according to the instantiation of the service. In addition, typical content transferred in the healthcare domain is made of very complex data structures with several thousands of atomic data types. The automated

generation of test cases brought by the MIDAS prototype reduces dramatically the effort that was formerly dedicated to test case handwriting. Moreover, the home-made testing framework is able to support only service unit testing. End-to-end test of service compositions with MIDAS requires only the drafting of the appropriate SAUT, test configuration and PSM models.
With the aforementioned huge amount of test cases, the optimisation of the test campaigns is a must. The home-made test framework doesn’t have any support for test cases prioritization and test case generation optimization. MIDAS intelligent scheduler and reactive planning facility propose solutions to the optimisation problem that are technically operational and whose evaluation is in progress.
Last but not least, with the home-made framework every change in the deployed SAUT (IP addresses, ports, URIs, parametrizations) requires a significant effort of reconfiguration by hands of every individual test case, practically preventing any continuous integration approach. With the MIDAS prototype, the SAUT models, the test configuration models, the PSMs and the generated test suites are independent of the SAUT physical locations that are indicated as configuration parameters to be instantiated at test run time.

Conclusion

The collection of functional test automation methods of the MIDAS prototype covers all the service functional test tasks, including the most “intelligent” and knowledge-based ones. Furthermore, the test automation methods are provided as *services*, allowing the MIDAS SaaS user both to invoke them individually and to easily combine them in service integration and delivery processes directed by CI/CD servers. These methods are actually integrated as services by a MIDAS partner (Dedalus) in its specific integration and delivery process of healthcare distributed applications and services architectures. Experiences for assessing and mastering advanced features such as dynamic scheduling for re-testing and regression testing and evidence-based test case generation are in progress.
Current drawbacks of the MIDAS prototype are manageability and usability issues and are the matters of future work: (i) taking into account REST/JSON service testing; (ii) automated check of the alignment of the SAUT deployment with the SAUT model; (iii) simplifying the specification of the test configuration; (iv) better handling of passive oracles (generated from incomplete specifications).

References

1. Barr, E., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5).
2. Bozkurt, M., Harman, M., & Hassoun, Y. (2010). Testing web services: A survey. Department of Computer Science, King's College London, Tech. Rep. TR-10-01.
3. Conforti, D., Groccia, M. C., Corasaniti, B., Guido, R., & Iannacchero, R. (2014). EHMTI-0172."Calabria cephalalgic network": innovative services and systems for the integrated clinical management of headache patients. *The journal of headache and pain*, 15(1), 1-1.
4. Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8 (3), 231-274.
5. Hierons, R. M., Bogdanov, K., Bowen, J. P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A. J. H., Vilkomir, S., Woodward, M. R., & Zedan, H. (2009). Using formal specifications to support testing. *ACM Comput. Surv.*, *41* (2), 1-76.
6. Jehan, S., Pill, I., & Wotawa, F. (2013, May). Functional SOA testing based on constraints. In *Proceedings of the 8th International Workshop on Automation of Software Test* (pp. 33-39). IEEE Press.
7. Maesano, A. P. (2015). *Bayesian dynamic scheduling for service composition testing*. Ph.D. Dissertation, University Pierre et Marie Curie, Paris.
8. Namin, A. S., & Sridharan, M. (2010). Bayesian reasoning for software testing. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, (pp. 349-354). New York, NY, USA: ACM.
9. Newman, S. (2015). *Building Microservices*. O'Reilly Media, Inc.
10. Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
11. Yoo, S., & Harman, M. (2012). Regression testing minimization, selection and prioritisation: a survey. *Softw. Test. Verif. Reliab.*, *22* (2), 67-120.
12. http://www.dedalus.eu/
13. https://hssp.wikispaces.com/
14. https://jenkins-ci.org/
15. http://martinfowler.com/bliki/DeploymentPipeline.html
16. http://www.midas-project.eu
17. http://www.w3.org/TR/scxml/
18. http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html

Security Testing of WSDL-based Web Services with Fuzzing

Martin A. Schneider, Leon Bornemann
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
martin.schneider@fokus.fraunhofer.de

Abstract:
Our today's lives are relying more and more on services that are connected through the Internet. This degree of interconnection of services will increase due to developments such as the Internet of Things and Cyber-Physical Systems. Since the kind of data processed by such services is also getting sensitive, e.g. health data, the secure operating of services is of significant importance.
In this paper, we present a holistic black-box security testing approach for WSDL-based web services covering test identification, test case generation, verdict arbitration and scheduling of security test cases we are developing within the MIDAS project. Furthermore, we present a draft idea of an approach that combines different fuzzing techniques.

1. Introduction

The increasing degree of interconnection of devices due to developments such as Cyber-Physical Systems and the Internet of Things also increases the demand for the secure operation of connected devices. Since the kind of data processed by such services is also getting sensitive, e.g. health data, this demand is increasing further. Due to events such the revealing how intelligence services are exploiting vulnerabilities, this topic is getting more attendance as well.
In this paper, we present an approach that is based on complementary fuzz testing techniques how web services can be tested for vulnerabilities. This includes (traditional) data fuzzing, an approach we call behavioural fuzzing and a novel combination of both of these techniques. We have implemented these techniques within the MIDAS projects where we will also perform an

evaluation based on two case studies, one from the Logistics domain and one from the eHealth domain.
The remainder of this paper is organized as follows: Section 2 presents related work with respect to fuzz testing. In the following Section 3, we introduce how we fuzz web services and what information we use for this purpose. It presents also an idea for a novel approach combining data and behavioural fuzzing. The conclusions, ongoing and future work within the MIDAS project are presented in Section 4.

2. Related Work

Fuzz testing or fuzzing is a security testing approach that aims at finding zero-day-vulnerabilities by stimulating the system under test (SUT) with invalid or unexpected input data [1,2]. Fuzzing is testing for missing or faulty input validation mechanism by generating corresponding input data based on the interface specification [2] (black-box approach) or the source code (white-box approach) [4]. When such input data is processed instead of being rejected, this may lead to an undefined state and to security-relevant problems [2]. Famous examples for such problems are buffer overflow and SQL injection. In case of a buffer overflow vulnerability, the SUT does not check the length of the input data and thus, the memory may be corrupted be very long input data. In case of an SQL injection, user input is used within a database request in an insecure way. This allows an attacker to alter the semantic of the database request in order to bypass an authentication mechanism or get access to restricted information. In case of a buffer overflow vulnerability, constraints for valid input data are neglected while in case of an SQL injection, SQL syntax elements are not expected as input, leading in both cases to a vulnerability requiring input validation and input sanitizing mechanisms in order to eliminate them.
However, usually several constraints has to be checked in order to decide whether an input date is valid or not. Therefore, the first random-based fuzzers [3] are not very efficient because they generate usually totally invalid input data that violate several constraints for valid input data. Hence, such data is likely being rejected because several validation mechanisms must fail in order to accept it [2]. The focus of so-called smart fuzzers is on generating semi-valid input data that is mostly valid and invalid only in small portions. For this purpose, they are employing a specification of valid input data, e.g. a model (see Figure 1) [2,5].

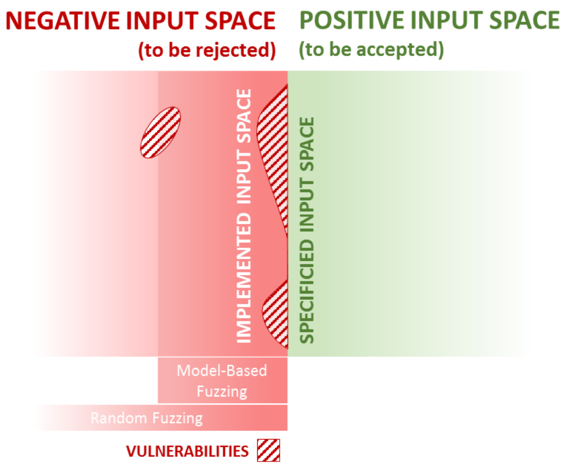


Figure 1: Target of random-based and model-based fuzzing

Basically, there are two types of fuzzing: Traditional data fuzzing is generating invalid input data as described above in order to find faulty or missing input validation and input sanitizing mechanisms. In contrast to this, behavioural fuzzing is generating invalid message sequences in order to find faults in the state machine of the SUT or interdependencies between invalid messages and input validation [5]. Although there are some basic approaches for behavioural fuzzing, e.g. in [6,7,8], a first elaborated approach was presented in [9] that proposes behavioural fuzzing operators that mutate existing valid message sequences in form of UML sequence diagrams.

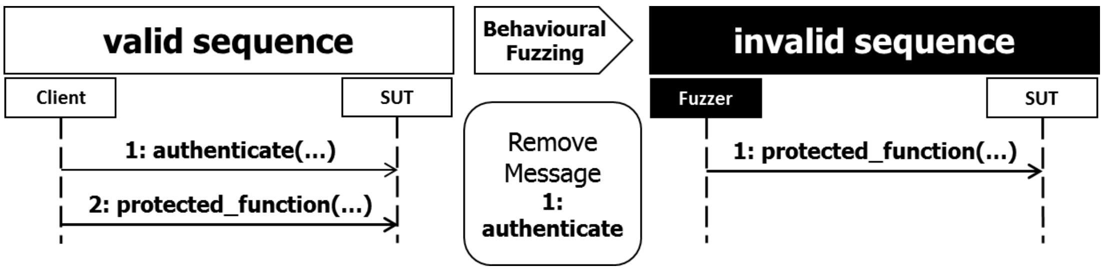


Figure 2: Illustration of test generation for behavioural fuzzing. A valid message sequence (left-hand) is mutated by a behavioural fuzzing operator 'Remove Message' that is applied to an element in order to generate an invalid message sequence (right-hand).

Such behavioural fuzzing operators work on one hand on message level, by removing, repeating, or inserting messages, and on the other hand on control structure level by altering loop boundaries or conditions of alternative branches. The presented approach [9] has the advantage that existing functional test cases or traces can be reused for security testing.

3. Fuzzing of Web Services

Our approach of fuzzing web services presented in this paper is based on data fuzzing and behavioural fuzzing techniques as well. It is based on UML environmental models that contain on one hand structural descriptions, i.e. the type system of the SUT as well as the interface description, and on the other hand behavioural descriptions, i.e. sequence diagrams representing valid interactions with the SUT, e.g. functional test cases.
A data format commonly used to the describe the interface of a web service is the Web Service Description Language (WSDL) format [10]. It contains descriptions of the messages that can be exchanged with a web service as well as the type system in the form of XML schema [11]. Within MIDAS, a tool was developed that automatically imports the message types and the type system from WSDL files and XML schema documents and thus, reduced the effort for creating a test model. By employing other testing techniques, such as usage-based testing, a nearly-complete model can be generated from recorded usage profiles and used as a starting point for security testing as described in [12].

3.1 Data Fuzzing

The approach of data fuzzing is employing a library of existing fuzzing heuristics from the (formerly) Open Source fuzzing tools Peach [13] and Sulley [14], called Fuzzino [15]. It provides a set of basic fuzzing heuristics for different kinds of primitive types. In order to make it feasible to create fuzz test data for real-world web services, we extended it with the capability to create fuzz test data for complex, i.e. structured data types. This allows fuzzing a data structure itself, e.g. by removing, repeating or inserting fields, as well as its enclosed values.
Since the model we use as starting point for fuzz test case generation also provides all the type information from the WSDL file, we also make use of the accompanying type constraints, e.g. minimal and maximal length of a string, regular expressions describing a pattern of a valid string, minimal and maximal value for a number.
Fuzzino uses this information to generate fuzz test data, i.e. invalid or unexpected values. An extended version of Fuzzino is able to generate test data from grammars and transforms a regular expression to a grammar. Thus, it is able to generate fuzz test data from regular expressions by mutating the equivalent grammar in several ways. For instance, it makes a

required element optional in order to generate some test data that do not contain the required element.
As mentioned above, we are using a model-based approach for test generation. Therefore, all the test cases including test data are enclosed within the model. However, fuzzing techniques usually generate a huge number of test data for a certain message argument. Therefore, the corresponding fuzz test cases only differ in this message argument whereas the rest of test case remains the very same. Providing all the test data in separate test cases would blow up the model with only little additional information. Therefore, we get rid of all the fuzz test data by using so-called test strategies. These test strategies indicate which kind of fuzzing heuristic shall be employed for by a fuzz test data generator, i.e. Fuzzino. This approach reduces the size of the model that contains only abstract test cases and test strategies (abstract in that way the fuzz test data is not contained while all the required information in order to generate the actual fuzz test cases is provided). This approach is presented in [16].

Verdict arbitration. For functional testing, the test verdict is usually determined by evaluating the response of the SUT within the functional test case. For security testing, this is quite different. Since we are looking for faults that lead to unintentional processing of invalid input data or message sequences, the response of a stimulus carrying invalid input data or an invalid message is not valid in order to determine whether a security-relevant fault was found or not.
Example: A web service is interacting with a database by SQL queries. If such SQL queries carry user data, the syntax and thus, the semantic of an SQL query may be altered. If this is well done, the SQL query may still be valid in terms of the SQL syntax. Thus, it will be successfully executed while having a totally different semantic. Thus, existing records in the database may be changed, deleted, or added (or other effects). Since the altered SQL query is syntactically correct and therefore, successfully executed, the web service won’t detect any error and respond with the information that the request was successfully processed.
A (partial) solution to this problem for black-box security testing is valid case instrumentation [2]. This means to execute a security test case that stimulates the SUT with malicious inputs and ignore the responses of the SUT. No verdict is determined within the security test case. Instead, after each security test case, a functional test case is executed in order to determine the verdict for the security test case. The functional test case checks whether the SUT is still alive and whether it is working functionally correct. This has a few implications for the selected functional test case. On one hand, it has to be ensured that the functional test case triggers (at best) all

the functionality of the SUT that may be negatively impacted by the security test case previously executed. The more functionality is tested, the lesser is the number of false negatives. On the other hand, the functional test case should not be dependent on a certain state of the SUT that may be changed by a security test case even if no bug was triggered. Therefore, relying on a certain state of the SUT may lead to false positives.
Therefore, functional test cases have to be carefully selected. We introduced the capability to determine one or several functional test cases appropriate for verdict arbitration together with each testing strategy.

Scheduling. However, fuzzing techniques still generate far more test cases than can be executed. Therefore, different kinds of risk-based testing approaches are used for test case identification, selection, and prioritization. In contrast to many approaches that require manual analysis, such as fault tree analysis [17], failure mode and effect analysis [18], and the CORAS approach [19], we propose an automated approach that takes advantage of an already shown correlation between interface complexity and error proneness. Since fuzzing is a negative testing approach, we propose a complexity metric for the negative input space that measures the boundaries of the negative input space.
The works of Cataldo [20] and Bandi [21] form the basis for a negative input space complexity suitable for security testing. Cataldo [20] showed that there is a correlation between interface complexity and error proneness. His approach employed the metrics interface size and operation argument complexity used by Bandi et al. [21]. Operation argument complexity is dependent of the type of the operation's arguments. A constant value is assigned to each type. The operation argument complexity is determined by the sum of the complexity of each argument's type [21]. The interface size is defined as the product of the number of parameters and the sum of their sizes (operation argument complexity).
Since there is a correlation between interface complexity, operation argument complexity and error proneness, we suppose this correlation holds true for security-relevant errors as well. We would like to exploit this correlation for prioritization of security test cases generated by using data fuzzing techniques. Our presumption is: The more constraints apply for an input date of a certain type, the higher is the chance that one of the corresponding validation mechanisms is faulty.
The negative input space of a certain type is determined by the boundaries of the positive input space comprising all valid values. The boundaries between the positive and negative input space are specified by the type constraints valid input data. Therefore, the negative input space metric is expressed with respect to these constraints. We schedule security test cases

based on the hypothesis that a high negative input space complexity is an indicator for a higher risk of a faulty implementation of an input validation mechanism.
Since the metric is based on the constraints for valid input data, we have to investigate the different kinds and the structure of them and how they may be assessed by the metrics. The metric counts the number of constraints for the valid input space and determines the number of dependencies between different parts of a data type.
The calculated score is determined by the number of boundaries, for example 2 for length restricted strings, one for the lower bound (if it is at least 1) and one for the upper bound. In order to determine the complexity score of a regular expression, we resolve predefined character classes and evaluate the different number of character ranges as well as quantifiers.
An example for dependencies are calendar dates where the maximum valid number for the day depends on the month (usually 31 but 28 and 30 for February, April, and so on) and the year (29 for February in case of a leap year). This metric is used by scheduling the data fuzz test cases with respect to the negative input space complexity of the message argument's type for that fuzz test data shall be generated. Based on the hypothesis, this metric leads to a prioritization of test cases that may reveal faulty implementations of input validation mechanism earlier and may reduce the number of test cases if a threshold value for the metric score is used. This would enable to omit those test cases whose complexity score based on the message argument's type is below the selected threshold. If this threshold is carefully selected, the test effort may be reduced where the risk for missing security-relevant bugs is minimized. The metric is described in more detail in [23].

3.2 A Combined Fuzzing Approach

In addition to the isolated usage of data fuzzing and behavioural fuzzing techniques, a combined approach is possible. The combination of both techniques means that a single test case contains both invalid input data and invalid message arguments. Such an approach may help to reveal vulnerabilities with less effort than with an isolated approach.
Considering the vulnerability in the Apache web server revealed by Kitagawa [6]. A malformed HTTP request with a repeated 'Host' message bypasses the input validation mechanism for the message argument, i.e. the virtual host name, and may lead to a buffer overflow. However, the buffer overflow is only revealed if a large number of host message repetitions is

chosen. This may require a large number of test cases with different numbers of repetitions of the host message when a host name with a usually short, valid length is used. A combination of the behavioural fuzzing approach that repeats the host message with data fuzzing techniques that may generate an extremely large host name would reveal this vulnerability earlier because the buffer overflow may be achieved by the first repeated host message.

Generally, two approaches for the combination of data and behavioural fuzzing techniques are possible. On one hand, as discussed in this example, an invalid message may carry invalid or unexpected data. Therefore, both techniques are applied to the same element, behavioural fuzzing is applied to a message and data fuzzing is applied to arguments of the very same message. On the other hand, both fuzzing techniques may be applied to different elements within the very same test case. An invalid message argument may be used for one message and another message may be behavioural fuzzed. This would make sense if the message relies on a fuzzed input data of a previous message.

4. Conclusion, Ongoing and Future Work

We presented a holistic, black-box security testing approach for WSDL-based web services employing different fuzzing techniques, i.e. data fuzzing generating invalid input data, behavioural fuzzing generating invalid input message sequences, and a combination of behavioural fuzzing and data fuzzing generating test cases with invalid messages and invalid input data. We also presented a metric for the negative input space complexity serving as a prioritization mechanism for scheduling of security test cases.

Within the MIDAS European research project [22], we are currently building a test platform on the cloud for testing of service-oriented architectures. As part of this project, we are implementing the presented techniques. Within the project, we are working closely together with industrial partners from the Logistics and the eHealth domain. We will utilize their implementations for the evaluation purposes.

Based on the results, we will improve the presented approaches; in particular, we will see how well the combined approach of data and behavioural fuzzing performs currently implemented in a naïve version. Based on the results, we will work on an improved version of this approach that will take into account data flows.

Acknowledgments. This work is partially funded by the EU FP7 project MIDAS (no. 316853) and RASEN (no. 318786).

References

1. Bekrar, S., Bekrar, C., Groz, R., Mounier, L. (2011, March). Finding software vulnerabilities by smart fuzzing. In Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on (pp. 427-430). IEEE.
2. Takanen, A., DeMott, J., Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House, Boston (2008)
3. Miller, B. P., Fredriksen, L., & So, B. (1990). An empirical study of the reliability of UNIX utilities. Communications of the ACM, 33(12), 32-44.
4. Godefroid, P.; de Halleux, P.; Nori, A.V.; Rajamani, S.K.; Schulte, W.; Tillmann, N.; Levin, M.Y., "Automating Software Testing Using Program Analysis," in *Software, IEEE* , vol.25, no.5, pp.30-37, Sept.-Oct. 2008
5. Kaksonen, R., Laakso, M., & Takanen, A. (2001). Software security assessment through specification mutations and fault injection. In Communications and Multimedia Security Issues of the New Century (pp. 173-183). Springer US.
6. Kitagawa, T., Hanaoka, M., Kono, K.: AspFuzz: A State-aware Protocol Fuzzer based on Application-layer Protocols. In: IEEE Symposium on Computers and Communications, pp.202-208 (2010)
7. Takanen, A., DeMott, J., Miller, C.: Software Security Assessment through Specification Mutations and Fault Injection. In: Communications and Multimedia Security Issues of the New Century, Series: IFIP Advances in Information and Communication Technology, Vol. 64, Steinmetz, Ralf; Dittmann, Jana; Steinebach, Martin (Eds.) (2001)
8. Becker, S., Abdelnur, H., State, R., Engel, T.: An Autonomic Testing Framework for IPv6 Configuration Protocols. In: Mechanisms for Autonomous Management of Networks and Services. AIMS'10: Mechanisms for Autonomous Management of Networks and Services, Zurich, Switzerland (2010)
9. Schneider, M., Grossmann, J., Tcholtchev, N., Schieferdecker, I., Pietschker, A.: Behavioral Fuzzing Operators for UML Sequence Diagrams. In: 7th Workshop on System Analysis and Modelling 2012 (SAMWkshp 2012), ser. LNCS, O. Haugen, R. Reed, and R. Gotzhein, Eds., vol. 7744. Springer, 2013, pp. 88–104

10. Web Service Description Language (WSDL) 1.1, W3C (2001). URL: http://www.w3.org/TR/wsdl
11. XML Schema 1.1, W3C (2004). URL: www.w3.org/XML/Schema
12. Schneider, Herbold, S., Wendland, M.-F., Grabowski, J.: Improving Security Testing With Usage-Based Fuzzing. To appear in: Risk assessment and risk-driven testing 2015 (RISK 2012), ser. LNCS, J. Großmann, F. Seehusen, M. Felderer, M.-F.Wendland, Eds., Springer, 2015
13. Eddington, M. (2011). Peach fuzzing platform. *Peach Fuzzer*. URL: http://www.peachfuzzer.com/
14. Amini, P. A.: Sulley: Fuzzing framework (2007). URL: https://github.com/OpenRCE/sulley
15. Fuzzino at Github: https://github.com/fraunhoferfokus/Fuzzino
16. Großmann, J., Schneider, M., Viehmann, J., & Wendland, M. F. (2014). Combining risk analysis and security testing. In: Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications (pp. 322-336). Springer Berlin Heidelberg.
17. Tanaka, H., Fan, L. T., Lai, F. S., & Toguchi, K. (1983). Fault-tree analysis by fuzzy probability. Reliability, IEEE Transactions on, 32(5), 453-457.
18. Stamatis, D. H. (2003). Failure mode and effect analysis: FMEA from theory to execution. ASQ Quality Press.
19. Lund, M. S., Solhaug, B., & Stølen, K. (2010). Model-driven risk analysis: the CORAS approach. Springer Science & Business Media.
20. Cataldo, M., De Souza, C. R., Bentolila, D. L., Miranda, T. C., & Nambiar, S. (2010). The impact of interface complexity on failures: an empirical analysis and implications for tool design. School of Computer Science, Carnegie Mellon University, Tech. Rep.
21. Bandi, R. K., Vaishnavi, V. K., & Turk, D. E. (2003). Predicting maintenance performance using object-oriented design complexity metrics. Software Engineering, IEEE Transactions on, 29(1), 77-87.
22. EC FP7 MIDAS Project. www.midas-project.eu (2012-2015), FP7-318786
23. Schneider, M. A., Wendland, M.-F., Hoffmann, A.: A Negative Input Space Complexity Metric as Selection Criterion for Fuzz Testing. To appear in: 27th IFIP WG 6.1 International Conference ICTSS 2015 Proceeding, ser. LNCS, K. El-Fakih, G. Barlas, N. Yevtushenko, Eds., vol. 9447. Springer, 2015, pp. 1–6

INTUITEST

A Fully Automated Approach for Debugging GUI Applications

Ethar Elsaka
Department of Computer Science
University of Maryland
ethar.elsaka@gmail.com

Atif Memon
Department of Computer Science
University of Maryland
atif@cs.umd.edu

Abstract:
Recent studies have shown the increasing cost of software testing and debugging, with software testing cost contributing up to 25% of total project costs, and software debugging consuming up to 50% of programmers time. To date, software testing and debugging processes are largely manual, requiring significant involvement of human developers and testers to carry out those tasks. Despite attempts to build automated debugging solutions, existing approaches fall short of full automation, and hence are hardly used in practice. In this paper, we propose a unified framework for fully automated GUI testing and debugging. We develop algorithms and methods that leverage the process of automated software testing and extend it to achieve automated debugging as well. Our automated debugging framework mutually utilizes both program dynamic and static information in order to explain and identify root causes of software bugs. We present several use cases showing the effectiveness of our approach.

Introduction

Software debugging—isolation and correction of programmatic errors—is often considered as one of the most time consuming phases of software development [1], [2], [3]. The debugging process starts after a problem, typically a program failure, e.g., a crash or incorrect output, has been encountered either during testing or usage [4]. A developer uses a mix of manual analysis, program understanding, and tools to isolate the source of the problem, and then fix it [5].

By far, the most common way of debugging a program is by using a manual iterative debugging technique in conjunction with a debugging tool [6], [7]. During iterative debugging, the developer analyzes one or more executions (test cases) of a program, iteratively building up knowledge about the program state surrounding the problem area. In an effort to narrow down the problem areas of the code, the developer may remove some parts of original test case and check if the problem still exists. Once the test case has been sufficiently pared down, the developer may examine program states—values of variables—and track down the source of the problem. This can be done by using debugging hooks, supplied by a debugger, into the program or simply adding a few *print* statements, which output the values of variables at certain points of program execution. As can be imagined, this is a slow and resource intensive process.

The manual debugging remains the most commonly used approach. This is not surpri-sing given that most research efforts to automate debugging remain confusing, difficult to use, and often have no tool support on various common programming platforms. For example, program slicing based techniques [8], [9] require intimate knowledge of static and dynamic analysis, and require (often unavailable) slicing tools; program spectrum-based techniques [10], [11], [12] require the availability of both passed and failed test cases that are *similar* with respect to program spectra distance criteria; ranking based approaches [13], [14], [15], [16] output, for manual examination, a ranked list of pro-gram statements, where ranking is an indicator of *suspiciousness* of the statement towards the failure; and input reduction techniques such as delta debugging [12] requi-re a *correct* execution of the program to accompany the faulty execution. Many of these artifacts are not usually available in practice.

In this paper, we present *Debugging using Sequence Covers*, a framework for *fully automatic debugging*. Our aim is to *democratize* automated debugging by giving deve-lopers a technique that is easy to understand, uses off-the-shelf code instrumenters, and is supported by tools that can be run on any platform. A key enabler of our appro-ach includes wider adoption of automated testing technology, which gives us access to multiple failing test cases that share an underlying bug.

At a high level, our approach captures automated test case execution traces using code instrumenters. We separate traces into groups according to the source of the error. Then, for each group, we extract common subsequences from the execution traces using a novel and efficient common subsequences extraction algorithm that is adapted for processing code traces. After extracting common subsequences, we back-trace the common subsequences using static code analysis to find out the source of the error, and report it to the developer. We show multiple case studies of our appro-ach applied to bugs in applications of varying sizes. We show the easy applicability of our framework to those applications, and show the final output produced by it to ex-plain the bug to the developers.

RELATED WORK

Many research studies were performed to automate the process of debugging and fault localization to some degree. One of the earlier works in automating the debugging process is program slicing. Program slicing identifies all the statements that can affect a variable in a program either statically [8], or dynamically [9]. The main problem of the static and dynamic slicing is that they consume a large amount memory and processing time because they require building the dependency and call graph of the whole program. Furthermore, the size of the slice can still be fairly large even with the dynamic analysis. Furthermore, there are no widely available tools that are used for slicing.

Renieris and Reiss [10] propose 3 fault localization methods set union, set intersection, and nearest neighbor. In the set union method, it computes the set difference between the program spectra of a failed test and the union spectra of a set of successful tests. The set intersection method excludes the code that is executed by all the successful tests but not by the failed test. Lastly, the nearest neighbor contrasts a failed test with another successful test which is most similar to the failed one in terms of the “distance” between them. Also, one of the most popular methods that are based on the program spectrum is Tarantula [11]. It computes the suspiciousness of each statement according to its contribution in the failed and passed test cases. The problem of all these techniques is that they provide a list of ranked statements. Each statement is considered individually with no relation to other statements, so the reason for the failure is unclear. Furthermore, the suggestions are incorrect in most cases. Also, with these lists, a lot of information is still missing like the program state, and the dependency between the statements, which still requires a lot of manual work on the developer's side.

Delta debugging [13] is one of the most popular approaches that are based on the program state. It reduces the causes of failures to a small set of variables by contrasting program states at different points in the program between the execution of a successful test and a failed one using their memory graphs. Furthermore, Zeller [14] proposes a method to identify the locations and times where the cause of failure changes from one variable to another. The main problem with this method is that it requires a large amount of memory in order to save the program states. Furthermore, it may require multiple runs to narrow down the suspicious variables and transition locations. Finally, the output may not contain the source of the error and the developer still needs to navigate through the source code.

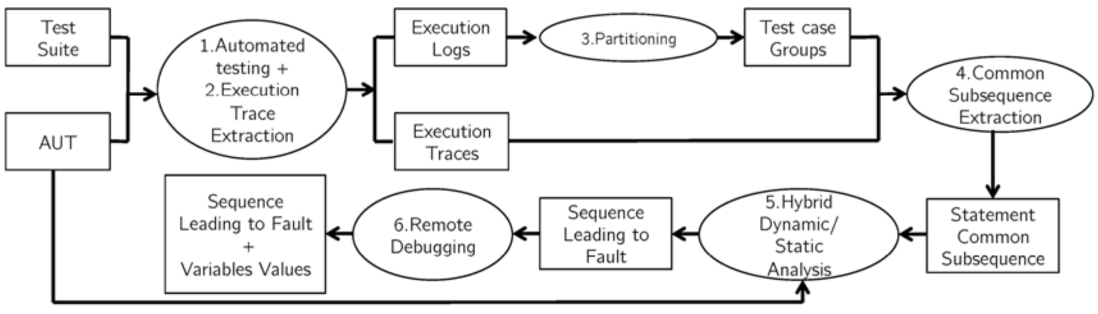

Test Suite
AUT
1.Automated testing + 2.Execution Trace Extraction
Execution Logs
Execution Traces
3.Partitioning
Test case Groups
4.Common Subsequence Extraction
Statement Common Subsequence
5.Hybrid Dynamic/ Static Analysis
Sequence Leading to Fault
6.Remote Debugging
Sequence Leading to Fault + Variables Values

Fig. 1: Automated Debugging Framework

Automated Debugging Framework

Now we present *Disqover*, our automated debugging framework. *Disqover* takes as an input the test suite and the source code of the application under test (AUT) and outputs the detected faults with their recommended code subsequence that leads to the source of the fault. Since in this paper we deal with graphical user interface (GUI) applications, the first module of our framework is a module is capable of extracting GUI information from GUI applications, generating and replaying test cases for GUI applications, but this part can be replaced by any other module that is capable of generating and executing non-GUI applications test cases.

Disqover consists of 6 modules as can be seen in Figure 1:

1. The Automated Testing Module, which is responsible of extracting GUI information, generating and running the test cases. It takes as input the application source code and outputs the test cases execution logs. These logs show the output of the test case, i.e., whether it passed or failed.
2. The Execution Trace Extraction Module, which is responsible of extracting the test cases execution traces. These traces present the order of the statements that are touched during the execution of the test cases.
3. Partitioning Module, which is responsible of grouping the test cases according to the type and the location of the errors caught by the test cases execution. It takes as an input the test cases execution logs and outputs test case groups. Each group has the test cases that are failed for the same error (exception) type at the same location in the source code. In addition, it outputs an additional group for all the passed test cases.
4. Common Subsequence Extraction Module, which is responsible of extracting two kinds of information: common event subsequence and common statement subsequence. In the first one, the algorithm is applied to the test cases and outputs the common event subsequence. In the second one, for each event in the common event subsequence, the algorithm is applied to the statements that are touched during event execution to output the common statement subsequence.
5. Hybrid Dynamic/Static Analysis Module, which analyzes the code mutually along with the common statement subsequences to provide the dependency of the line that throws the error from the common subsequence. It takes as input the common subsequence statements for all the common events together and outputs a final subsequence. This subsequence explains the fault since it contains only the lines that affect the line that throws the exception.
6. Remote Debugging Module, which provides the values of the variables that included in the subsequence that explains the fault.

Case Studies

In this section we evaluate our framework by discussing two case studies of two errors in two different applications. The applications are Crossword Sage, and ArgoUML. The application sizes vary from thousands of lines of code to hundreds of thousands of lines of code. Throughout the case studies, we show concrete examples of our framework's capability to find and identify root causes of bugs, and present them to the developer in a self-explained manner. We also show the final output of the framework for each error, along with the number of lines to inspect in that output. Below, we describe our applications.

Crossword Sage [18] is a tool for creating professional-looking crosswords with powerful word suggestion capabilities. It can be used to build, load, and save crosswords. It can suggest words for adding to the crosswords, and allows the crosswords builder to give clues for them. Furthermore, in addition to building crosswords, it allows users to load pre-built crosswords and solve them. Crossword Sage project consists of 3072 lines of code, 34 classes and 238 methods.

ArgoUML [19] is an open source UML modeling tool. It includes support for Structural and Behavioral UML diagrams. It has been used for the analysis and design of object oriented software systems. Also, It can run on any Java platform and is available in ten languages. The ArgoUML project consists of 152513 lines of code, 1787 classes, and 13117 methods.

Case Study 1: Crossword Sage

In order for the user to create a new crossword puzzle, he/she needs to click on the File menu and choose the New Crossword menu item. Then, the application asks the user to input the size of the puzzle through a dialog box. When the user inputs a numeric number between 2 and 20, the application creates an empty grid to allow the user to start building his/her crossword puzzle.

Normally, if the user enters a non-numeric value as the size of the puzzle, an error dialog box should appear warning the user about the wrong input format and asks the user to enter another input value. However, in this application when the user enters a non numeric value in the dialog box, the application crashes with a NumberFormatException as can be seen in Figure 2.

```
java.lang.NumberFormatException: For input string: ""
at java.lang.NumberFormatException.forInputString
(NumberFormatException.java:48)
at java.lang.Integer.parseInt(Integer.java:470)
at java.lang.Integer.parseInt(Integer.java:499)
at crosswordsage.Grid.<init>(Grid.java:33)
at crosswordsage.CrosswordCompiler.<init>
(CrosswordCompiler.java:40)
at crosswordsage.MainScreen.showCrosswordBuilder
(MainScreen.java:226)
at crosswordsage.MainScreen.access$3(MainScreen.java:216)
at crosswordsage.MainScreen$MenuListene
.actionPerformed(MainScreen.java:423)
```

Fig. 2: Crossword Sage NumberFormatException

```
1 private void showCrosswordBuilder()
2 String reply = JOptionPane.showInputDialog(null,"Please
enter grid size (2-20)...", null);
3 cc = new CrosswordCompiler(reply, reply);

4 public CrosswordCompiler(String width, String height)
5 cw = new Crossword(width, height);

6 public Crossword(String width, String height)
7 isEditable = true;
8 this.width = width;
9 this.height = height;
10 words = new ArrayList();

11 public CrosswordCompiler(String width, String height)
12 grid = new Grid(cw);

13 void Grid(Crossword cw)
14 setLayout(new GridLayout(Integer.parseInt(cw.getHeight()),
Integer.parseInt(cw.getWidth())));
```

Fig. 3: Sequence Explaining Fault for Crossword Sage

Now, we discuss how using *Disqover*, the developer can get the concise sequence of statements explaining the error as shown in Figure 3. We list the steps performed by *Disqover* below. All of those steps are performed automatically.

Step 1: The process starts by generating and running the application test suite using the automated testing module. The output of this step is 347 test cases and their execution logs.

Step 2: At the same time, the test case execution traces are extracted during the test cases execution.

Step 3: Then, the test cases that reveal the NumberFormatException are grouped together using the partitioning module. This step detects 41 test cases that fail because of the exception that is shown in Figure 2.

Step 4: Next, the common subsequence algorithm is applied to execution trace of the 41 test cases. This step returns the common events that cause the NumberFormatException, which are File → New Crossword → Cancel.

Step 5: Then, for each event in the common events, the common subsequence algorithm is applied again for the event code to get the common statement subsequence.

Step 6: Then, the hybrid dynamic/static analysis module gets the dependency of the line that throws the exception. The output of this module is shown in Figure 3.

Step 7: Finally, the remote debugger module is applied to the final output to get the variable values of each assignment statement.

In order for the developer to find the source of the error using our proposed approach, only the following activities will take place:

1. The last line in the sequnece is line 14 (setLayout(new GridLayout(Integer.parseInt(cw.getHeight()), Integer.parseInt(cw.getWidth())));). This line is the line that throws the NumberFormatException. From this line, the developer can conclude that this exception results from applying the Integer.parseInt() function to a non-numeric value.
2. This non-numeric value may be assigned to either the height or the width variables of the cw object (because the Integer.parseInt() appears twice in the line).
3. Now, the developer can go backwards in the subsequence and see that the cw object comes from the method parameter as shown in line 13 (public Grid(Crossword cw)).
4. Going backward, there is a line in the sequence that creates new object from the Grid class and passes the Crossword object as a parameter as can be seen at the line 12 (grid = new Grid(cw)).
5. By going backwards further, the developer can see that the Crossword object cw is created at the line 5 (cw = new Crossword(width,height);) and the width and the height are passed as parameters.
6. These width and height variables are passed to the function through the Crossword- Compiler constructor arguments at line 4(public CrosswordCompiler(String width, String height)).
7. Finally, by going backwards at crosswordsage.MainScreen class, the developer can see that these parameters are passed as arguments when creating a new instance of cross- wordsage.CrosswordCompiler at line 3 (cc = new CrosswordCompiler(reply, reply)) and these arguments are both initialized by the variable reply which takes string values in line 2 (String reply = jOptionPane.showInputDialog(null, ”Please enter grid size (2 - 20)...”, null);).

As we can see, the developer needs only to inspect 6 lines to find the root cause of the bug. Those 6 lines are self-contained, and do not require prior knowledge of the code, as the problem can be seen by just inspecting those lines.

Case Study 2: ArgoUML

When the user exports the graphics using the “Export All Graphics” menu item, and saves them to a file, if the user enters a directory location that does not exist on disk, the application throws a FileNotFoundException as can be seen in Figure 4, and exits the Save dialog without notifying the user of the problem. The error is thrown when the application is actually trying to save the file, while it is originated when the user chooses the improper directory.

```
java.io.FileNotFoundException: /crash/crash/ClassDiagram.png
(No such file or directory)
at java.io.FileOutputStream.open(Native Method)
at java.io.FileOutputStream.<init>(FileOutputStream.java:179)
at java.io.FileOutputStream.<init>(FileOutputStream.java:131)
at org.argouml.uml.ui.ActionSaveAllGraphics.saveGraphicsToFile
(ActionSaveAllGraphics.java:230)
at org.argouml.uml.ui.ActionSaveAllGraphics.trySaveDiagram
(ActionSaveAllGraphics.java:161)
at org.argouml.uml.ui.ActionSaveAllGraphics.trySave
(ActionSaveAllGraphics.java:130)
at org.argouml.uml.ui.ActionSaveAllGraphics.trySave
(ActionSaveAllGraphics.java:106)
at org.argouml.uml.ui.ActionSaveAllGraphics.actionPerformed
(ActionSaveAllGraphics.java:98)
```

Fig. 4: ArgoUML FileNotFoundException

```
1 public void actionPerformed( ActionEvent ae )
2 trySave( false );

3 public boolean trySave(boolean canOverwrite)
4 return trySave(canOverwrite, null);

5 public boolean trySave(boolean canOverwrite, File
directory)
6 Project p = ProjectManager.getManager().getCurrentProject();
7 File saveDir = (directory != null) ? directory :
getSaveDir(p);
8 for (ArgoDiagram d : p.getDiagramList())
9 okSoFar = trySaveDiagram(d, saveDir);

10 protected boolean trySaveDiagram(Object target, File
saveDir)
11 File theFile = new File(saveDir, defaultName + "." +
SaveGraphicsManager.getInstance().getDefaultSuffix());
12 SaveGraphicsAction cmd = SaveGraphicsManager.getInstance().
getSaveActionBySuffix(SaveGraphicsManager.getInstance()
.getDefaultSuffix());
13 boolean result = saveGraphicsToFile(theFile, cmd);
```

Fig. 5: Sequence Explaining Fault for ArgoUML

The output of *Disqover* after being applied to this exception is shown in Figure 5. To obtain that output, *Disqover*, performs all the following steps automatically.

Step 1: The process starts by applying the automated testing module, which generates and runs 6317 test cases.

Step 2: At the same time, the trace execution extraction module finds out that the average number of lines per test case trace is 221795 lines. This large number of lines makes the manual debugging impractical.

Step 3: From the 6317 test cases, the partitioning module finds out that only 122 test cases reveal the FileNotFoundException exception that is shown in Figure 4.

Step 4: Now, after applying the common subsequence algorithm on the 122 failed test cases, it detects that the common events that cause the exception are File → Export All Graphics... → Save As: → Save.

Step 5: Then, for each event in the common events, the common subsequence algorithm is applied again for the event code to get the common statement subsequence. This step reduces the number of lines that need to be inspected to 234 lines.

Step 6: Then, the hybrid dynamic/static analysis module gets the final common statement subsequence. The number of lines to be inspected is reduced again to be 31 lines. We show a relevant subset of those lines in Figure 5.

Step 7: Finally, the remote debugging module gets the variable values of each assignment statement in the final sequence.

In order for the developer to find the source of the error using our proposed approach, only the following activities will take place:

1. The last line in the sequence is line 15 (fo = new FileOutputStream(theFile)). This line is the line that throws the FileNotFoundException. From this line, the developer can conclude that this exception results from an attempt to output stream to a file "theFile" and this file does not exist.
2. Now, the developer can go backwards in the subsequence and see that the ``theFile" variable comes from the method parameter as shown in line 14 (private boolean saveGraphicsToFile(File theFile, SaveGraphicsAction cmd)).
3. Going backward, there is a line in the sequence that calls the saveGraphicsToFile function as can be seen at the line 13 (boolean result = saveGraphicsToFile(theFile, cmd)).
4. Since the developer is investigating the variable "theFile", we can see that this variable is defined at line 11 (File theFile = new File(saveDir, defaultName + "." + SaveGraphicsManager.getInstance().getDefaultSuffix())).
5. This line uses a "saveDir" variable that is passed as the method parameter as can be seen at line 10 (protected boolean trySaveDiagram(Object target, File saveDir)).
6. By going backwards further, the developer can see that the function "trySaveDiagram" is called at line 9 (okSoFar = trySaveDiagram(d, saveDir)).
7. Finally, by going backwards at "trySave" function, the developer can see that the "saveDir" variable is set at line 7 (File saveDir = (directory != null) ? directory : getSaveDir(p)) to non existing location "/crash/crash".

As we can see, the total number of lines that are needed to be inspected are only 7.

References

[1] I. Vessey, “Expertise in debugging computer programs: A process analysis,” International Journal of Man-Machine Studies, vol. 23, no. 5, pp. 459–494, 1985.
[2] G. J. Myers, The art of software testing (2. ed.). Wiley, 2004.
[3] “Cambridge University study states software bugs cost economy $312billion per year,” http://undo-software.com/press- releases/cambridge-university-study-states-software-bugs-cost- economy-312billion-per-year/.
[4] D. Lo, H. Cheng, J. Han, S. Khoo, and C. Sun, “Classification of software behaviors for failure detection: a discriminative pattern mining approach,” in Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Paris, France, June 28 - July 1, 2009, 2009, pp. 557–566.
[5] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?” in Proceedings of the 20th International Sym- posium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011, 2011, pp. 199–209.
[6] “GDB: The GNU Project Debugger,” http://www.gnu.org/software/gdb/, 2006.

[7] “jdb - The Java Debugger,”
http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html, 1993.
[8] M. Weiser, “Program slicing,” in ICSE, 1981, pp. 439–449.
[9] X. Zhang, R. Gupta, and Y. Zhang, “Precise dynamic slicing algo-
rithms,” in ICSE, 2003, pp. 319–329.
[10] M. Renieris and S. P. Reiss, “Fault localization with nearest neighbor
queries.” in ASE. IEEE Computer Society, 2003, pp. 30–39.
[11] J. A. Jones, M. J. Harrold, and J. T. Stasko, “Visualization of test
information to assist fault localization,” in ICSE, 2002, pp. 467–477.
[12] A. Zeller, “Automated debugging: Are we close,” IEEE Computer, vol. 34, no. 11, pp. 26–31, 2001. [Online]. Available:
http://doi.ieeecomputersociety.org/10.1109/2.963440
[13] ——, “Isolating cause-effect chains from computer programs,” in Pro- ceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, Novem- ber 18-22, 2002, 2002, pp. 1–10.
[14] H. Cleve and A. Zeller, “Locating causes of program failures,” in ICSE, 2005, pp. 342–351.
[15] B. N. Nguyen, B. Robbins, I. Banerjee, and A. M. Memon, “GUITAR: an innovative tool for automated testing of gui-driven software,” Autom. Softw. Eng., vol. 21, no. 1, pp. 65–105, 2014.
[16] “Cobertura (A code coverage utility for Java),” http://cobertura.github.io/cobertura/, 2001.
[17] R. Valle eRai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundare- san, “Soot - a java bytecode optimization framework,” in Proceedings of the 1999 conferen-ce of the Centre for Advanced Studies on Collabo- rative Research, November 8-11, 1999, Mississauga, Ontario, Canada, 1999, p. 13.
[18] “Crossword Sage,” http://sourceforge.net/projects/crosswordsage/.
[19] “ArgoUML,” http://argouml.tigris.org/.

Toward testing multiple User Interface versions

Nelson Mariano Leite Neto, Julien Lenormand, Lydie du Bousquet, Sophie Dupuy-Chessa
Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
CNRS, LIG, F-38000 Grenoble, France
{lydie.du-bousquet, sophie.dupuy}@imag.fr

Abstract:
More and more software systems are susceptible to be used in different contexts. Specific user interfaces are thus developed to take into account the execution platform, the environment and the user. The multiplication of user interfaces increases the testing task, although the core application remains the same. In this article, we explore a solution to automate testing in presence of multiple user interfaces designed for the same application (e.g. web-based, mobile, …). It consists of expressing abstract test scenarios in a high-level language, and then to apply concretization rules specific to each UI version to generate executable tests.

1. Introduction

With the rise of mobile devices such as notebooks, smartphones and tablets, software systems are susceptible to be used everywhere and in different contexts. A context of use involves three factors: the platform (i.e. the type of device), the environment (e.g. the level of brightness) and the user (e.g. with different levels of expertise) [1]. These factors affect the interaction between the user and the system. That is why different User Interfaces (UI) can be proposed in order to fit the user's needs. Systems with adaptable UIs are built to dynamically propose the relevant UI with respect to their interpretation of the contextual situation. In these conditions, developing a set of relevant UIs can become as complex as developing the core of the system [2].

This article is concerned with the problem of asserting quality while developing different UIs in parallel for the same system (e.g., in order to propose adaptable UIs). Quality is achieved during the development and often evaluated by testing [2, 3, 4]. Testing becomes more and more expensive and automation can be a key to reduce this cost. Creation and maintenance of test scripts have to be taken into account to make test automation cost-effective [4].

To be cost effective, our proposition is to factorize the testing process as much as possible. Our starting point is a set of different UIs for the same system. We aim at automatically generating executable test scripts for each UI from a ***single*** description. To do that, we express test scenarios in an abstract way, and concretizing them for each UI version using specific translation rules. We think that this approach has several advantages. An abstract test scenario is easier to write and maintain than different executable test scripts. It is also easier to make evolve a set of test scripts

This paper is structured as follows. First, the related work is presented. Then, we introduce an illustrative example, using three web-based UIs and one mobile UI. Next, we detail the approach. The last section concludes and draws some perspectives.

2. Related Work

Our work concerns the problem of automating validation of multiple UI for the same application that are developed to fit different contexts of use. Executable test scripts are specific to each UI since they have to match the widget and navigation specificities of each version. Our solution aims at factorizing the effort of testing required for the different versions. In this section, we explore some approaches that have been proposed to automate UI testing with factorization point of view.

UI test scripts can be manually written and then *automatically executed* in some testing tools such as Abbot tool[1] or Selenium WebDriver[2]. The oracle is implemented as assertions in the code of the scripts. The automation relies mainly on the execution part.
Writing scripts is a laborious task. To ease it, “capture and replay” tools can be used to *record* user’s interactions with the UI. The recorded interactions can then be replayed. Many tools propose this feature, both for web-based or mobile application testing [5]. Oracle can rely on visual inspection during re-execution of the captured scenarios, image comparison or manual added assertions [6, 7].
Direct scripting and capture and replay approaches provide no direct factorization possibilities for test generation. Each interface has to be analyzed separately.

Model-based approaches have also been proposed to *automate test generation* for UIs [8, 9, 10, 11, 12, 13, 14, 15, 16]. They offer more

[1] http://abbot.sourceforge.net/
[2] http://www.seleniumhq.org/

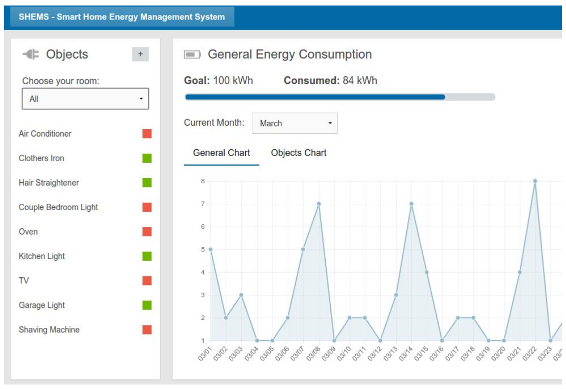
possibilities of factorization, especially when the model is built manually during the development process. But such a construction is quite difficult to carry out since it may require a high-level expertise [13]. Moreover, it is often difficult to maintain the equivalence of a model and the implementation during the application evolutions.
To deal with this problem, different authors propose *to extract automatically* the model from the existing interfaces [8, 9, 10, 12]. This type of approach is less adapted to our needs, since each interface has to be analyzed separately. However, being able to extract a specific model from each interface can then allow checking automatically the equivalence of interfaces [17].
When tests are generated from an abstract model, mapping from the model to the code has to be expressed in order to produce executable tests [14]. In [11], authors use a keyword machine to transform abstract test cases into executable ones.

No related work directly addresses the problem of generating executable test scripts for each UI version from a single description. However the idea of transforming abstract test cases into executable ones can be of interest to factorize. In this article, our high-level scenarios that are common to all interfaces were produced manually. But a model-based approach to generate them should be possible if a model is available. The next section describes small example of application.

3. Illustrative example

The illustrative example used in this work is a prototype for a smart home energy management system. It allows users to control and monitor the energy consumption in a home from different devices.

We have developed four UIs for this system: a mobile application for Android (named “mobile”), a web interface for desktop browser (named “web0”), and two web interfaces for mobile browsers (one with a menu page named “web1”, the other with a menu bar, named “web2”). The web-based versions are implemented in HTML5, JavaScript and JQuery. The mobile version is developed in Java 1.8. All versions have the same features. The differences between the different web versions are of two sorts. First widgets are different. Second navigation among pages is different.



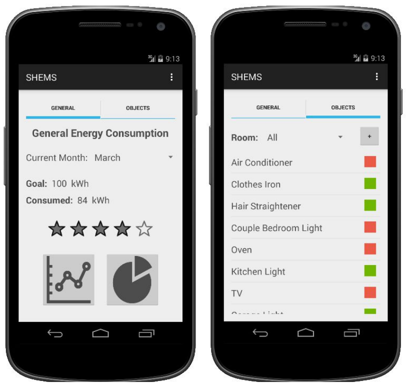


Figure 1: goal and filter features on the web0 interface

Figure 2: goal and filter features on the mobile interface

In this paper, we focus on three features.

- Goal: the user can check information about the general energy consumption per month. He can access the goal and actual consumption in kWh for the chosen month (Fig. 1 and 2).
- Filter: the user has access to a list of all the objects in the house, having the possibility to filter them per room. An object means any component that can be controlled and whose energy consumption can be registered, such as lights and electronic devices (Fig. 1 and 2).
- Comparator: the user can choose two objects and compare, side by side, their energy consumption charts (Fig. 3).

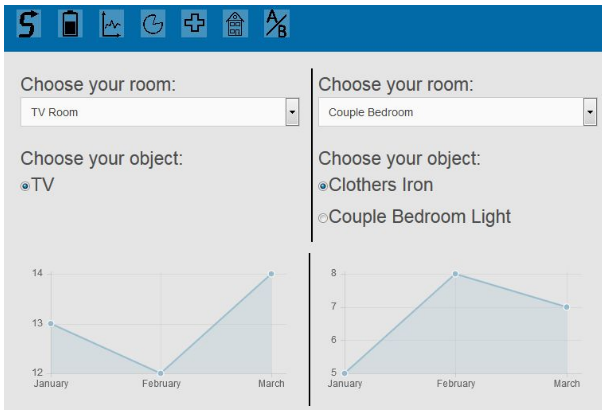


Figure 3: compare feature on the web2 interface

Test scripts for web versions are executed in Selenium [20], a tool following a capture and playback approach for web-based application. The Selendroid framework completes the Selenium environment for the mobile version[3].

[3] http://selendroid.io/

4. Approach

4.1 Principles

As said previously, the four UI versions share the same features but have different widgets and navigation paths. For this reason, to test them, it is necessary to write four specific executable test script sets. To avoid this tedious work, we propose the approach illustrated Fig. 4.

Abstract test “scenarios” are expressed in a high-level language. These scenarios are common to all the UI versions. A scenario is composed of a sequence of abstract instructions. A set of translation rules is used to transform the abstract test scenario into executable test scripts. The rules explicitly associate executable code to abstract descriptions. The translation rules are specific to each interface and defined manually. A translation rule simply rewrite an abstract instruction into an executable one, taking into account the implementation specificities. Translation rules can be the same for different UIs if they share the same widgets. A tool is used to translate the abstract scenario into executable test scripts.

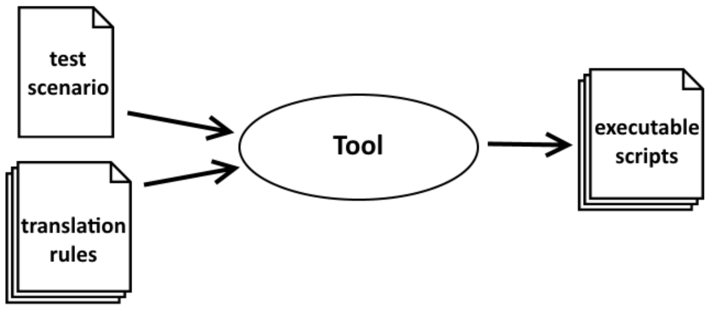


Figure 4: our approach to test script generation

4.2 Principles put into Practice

To express the high level scenarios, we use an existing language called TSLT (Test Schema Language for Tobias [18]). It is a textual language that contains several types of constructs allowing the definition of complex system scenarios. It is used as input of a testing tool called Tobias [18], which is responsible of the translation into the executable scripts.

Tobias is a test generator based on combinatorial testing. Combinatorial testing performs combinations of selected input parameter values for given operations and given states. Tobias adapts this principle to the generation of operation call sequences. It allows exploring systematically a large set of behavior sequences from a single abstract description, called scenario. An on-line version of Tobias is available at http://tobias.liglab.fr/

In our use of Tobias for UI testing, we start by expressing the abstract scenarios. Then we identify the executable code corresponding to each abstract instruction. This correspondance is expressed by a translation rule in TSLT. For the moment, this is carried out manually.

For instance, Listing 1 shows an abstract scenario in TSLT, designed to check that the displayed values are the expected ones for each goal. It consists of a sequence of three abstract instructions that allows to (1) navigate to the appropriate view (@goToGoal), (2) choose a month (@selectMonth) and (3) check that the displayed value is the expected one (@verifyValues).
Instruction “Integer month = [1-3]” indicates to Tobias to repeat the sequence for the first three months (combinatorial approach). Listings 2 to 5 show the four specific translations rules for “@goToGoal” abstract instruction.

```
group testMonthValue[us=true] {
   Integer month = [1-3];
   @goToGoal;
   @selectMonth;
   @verifyValues;
}
```

Listing 1: Abstract scenario of the Goal test case

```
group goToGoal[us=false] {
   // does nothing
}
```

Listing 2: Translation rule of @goToGoal for Mobile

```
group goToGoal[us=false] {
   driver.get(siteAddress);
}
```

Listing 3: Translation rule of @goToGoal for Web0

```
group goToGoal[us=false] {
   driver.get(siteAddress);
   WebElement goalButton = driv-
er.findElement(By.xpath("/html/body/div/section/ul/li[1]/div/a"));
   goalButton.click();
}
```

Listing 4: Translation rule of @goToGoal for Web1

```
group goToGoal[us=false] {
   WebElement goalButton = driver.findElement(By.id("menu_goal"));
   goalButton.click();
}
```

Listing 5: Translation rule of @goToGoal for Web2

From these TSLT rules, Tobias is able to translate the abstract scenarios into executable scripts. The executable scripts are executed in JUnit with Selenium or Selendroid frameworks.

For testing the three features of our illustrative example, six scenarios were designed, using 11 abstract instructions. Scenarios were translated into 21 executable test scripts in JUnit. The difference between the number of abstract scenarios and the number of executable tests is due to the combinatorial nature of Tobias. For example, the test case shown in Listing 1 is translated into three JUnit tests, each one corresponding to a different value for the month (1, 2, 3). By only changing “Integer month = [1-3]” into “Integer month = [1-12]” it is possible to generate the 12 test cases necessary to check all the months. It can also be changed into “Integer month = [0-13];” and then generate robustness tests.

As said previously, it is important to have test suites easy to maintain. The size of the description is one factor that impacts the cost of maintenance. Table 1 shows the number of lines written for the abstract scenarios with the translation rules for each feature. It also displays the number of line of code for the executable test scripts. Without the approach, those executable test scripts should have been written by hand. It can be observed that the number of lines to write has been at least halved.
This diminution of code between the abstract scenario and the executable test case is also due to the fact that the JUnit syntax is not described in the abstract scenarios nor in the translation rules. Tobias tool automatically generate the JUnit packaging.

Feature tested	Implementation	Mobile	Web0	Web1	Web2
Goal	Test scripts	161	143	151	147
	Abstract scenario	76	67	69	68
Filter	Test scripts	529	350	402	369
	Abstract scenario	117	88	93	92
Compare	Test scripts	231	235	235	227
	Abstract scenario	65	60	59	58

Table 1: Number of lines for the abstract scenarios and generated tests

Feature	Rule	Mobile	Web0	Web1	Web2
Goal	@goToGoal	0	0	3	2
	@selectMonth	5	3	3	3
	@verifyValues	4	4	4	4
	@veryfyMonthsCount	6	3	3	3
Filter	@goToObjects	2	0	3	2
	@selectRoom	3	5	5	5
	@verifyObjectsFiltered	25	5	5	5
	@selectRoomUncorrect	3	3	3	3
Compare	@goToCompare	2	3	3	2
	@selectRoom	3	3	3	3
	@verifyWidget	10	10	10	10
	@verifyChart	0	0	0	0
	Total	63	39	45	42

Table 2 : Number of lines for each rule for each version of each feature

Translation rules are quite simple. They consist in associating executable code to abstract instruction. For our example, the executable code corresponds to 0 up to 25 lines of code, for a total of 189 lines of code (see Table 2). Variation implementation details are thus expressed in a very concise way and localized. It becomes easy to make them evolve.

4.3 Discussion and Analysis

Our focus is to show the feasibility to express test scenarios for multiple UI versions of the same application, and to measure the effect of the factorization. The factorization effect can be evaluated through the difference of size between abstract and executable tests (Table 1). The factorization contribution is clearly visible. With Tobias, it is easy to increase artificially this difference, by playing on the combinatorial feature of the tool. But we deliberately limit the combinatorial exploration (e.g. we check only three months, instead of the twelve).

TSLT language does not allow expressing directly loops, return statements, exception handling nor proper functions in the translation rules. This constraint has for origin to guaranty that the combinatorial engine of Tobias will always succeed in the process of translating abstract scenarios into executable tests. Here, those constructions are necessary to express oracle condition and for scrolling handling navigation on the mobile version. The limitation has been bypassed during the experiment by separating code of the loops in a different file. To be able to express all the translation rules in TSLT, the language has to be extended. This does not affect the relevance of the approach.

As it can be seen on Table 2, some translation rules correspond to zero line of code. The reason is that there is no corresponding instruction within Selenium/Selendroid (e.g. it is not possible to check that an image is the one which is expected). This is directly linked to the testing framework expression power. It is independent of the approach.

5. Conclusions and Perspectives

We are concerned by the validation of several UIs provided for the same application for different contexts. Our motivation is to prepare the validation of adaptive applications, where tests have to be chosen with respect to the context. To do that, we would like to be able to automate test in a cost effective way.

The work described here is a first step toward this goal and should be considered as a feasibility study. Abstract scenarios and translation rules were both expressed in an existing language called TSLT, associated to a testing tool called Tobias. It is a combinatorial tool which aims at unfolding scenarios to explore all combinations that are defined by the scenario. It was not originally designed for UI testing but for JUnit test script generation. That was the main reason why it was chosen. The fact that the translation of abstract scenario into executable scripts can be done in a combinatorial way is an advantage since it helps in the process of factorizing code (and thus being cost effective).

Even if our illustrative example is simple, the different versions were built to explore a variety of widgets. Different navigation paths were also considered. This helps us to be confident in the fact that the factorization can be generalized. Moreover, translating abstract scenarios into executable ones can also be carried out for other testing framework than JUnit, since Tobias is designed to fit other testing frameworks. However, the example shows that the TSLT language is not fully appropriate as it is designed now (see. Sect. 4)

The example also shows that the approach can decrease the work of creating and maintaining testing suites. The size of the abstract scenarios with the translating rules is much smaller than the size of the final test scripts, which share a lot of identical code. Writing them is not simpler, but definitively shorter.

Our perspectives are to consolidate the work by exploring larger examples, other versions of interfaces and testing frameworks. This will help us to evaluate more precisely the amount of manual effort required with respect to the automated one and the approach genericity. Once this step is achieved, we will explore the possibility to associate translation rules to a context definition, and then to provide a framework that is able to generate tests during the execution, to fit the current execution context. The final step will be to generate automatically the abstract tests from a model, like those proposed in the Cameleon framework [19] and/or to produce automatically the translation rules such as in TESTAR [10].

Bibliography

[1] Coutaz, J., and Calvary, G. HCI and software engineering: Designing for user interface plasticity. Human-Computer Interaction: Development Process (2009).
[2] Muccini, H., Francesco, A. D., and Esposito, P. Software testing of mobile applications: Challenges and future research directions. In 7th Int. Workshop on Automation of Soft. Test (AST), IEEE (2012), 29-35.
[3] Beizer, B. Software testing techniques. Dreamtech Press, (2003)
[4] Grechanik, M., Xie, Q., and Fu, C. Maintaining and evolving GUI-directed test scripts. In IEEE 31st Int. Conf. on Software Engineering (ICSE) (2009), 408-418.
[5] Gao, J., Bai X., Tsai W. T., and Uehara T. Mobile application testing: a tutorial. IEEE Computer, 47:2 (2014), 26-35.
[6] Jung, H., Lee, S., and Baik, D.-K. An Image Comparing-based GUI Software Testing Automation System. In SERP (2012), 318-322.
[7] Xie, Q., and Memon, A. M. Designing and comparing automated test oracles for GUI-based software applications. ACM Transactions on Software Engineering and Methodology (TOSEM) 16, 1 (2007).
[8] Aho, P., Suarez, M., Kanstrén, T., and Memon, A. M. Industrial adoption of automatically extracted GUI models for testing. In EESSMOD@ MoDELS (2013), 49–54.
[9] Aho, P., Suarez, M., Kanstren, T., and Memon, A. M. Murphy tools: Utilizing extracted GUI models for industrial software testing. In IEEE 7th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW), (2014), 343–348.
[10] Vos, T. E., Kruse, P. M., Condori-Fernández, N., Bauersfeld, S., and Wegener, J. TESTAR: Tool support for test automation at the user interface level. Int. Journal of Information System Modeling and Design (IJISMD) 6, 3 (2015), 46–83.

[11] Nieminen, A, Jääskeläinen, A., Virtanen, H., Katara, M. A Comparison of Test Generation Algorithms for Testing Application Interactions, 11th Int. Conf. on Quality Software (QSIC), (2011), 131-140.
[12] Amalfitano, D., Fasolino, A.R., Tramontana, P. A GUI Crawling-Based Technique for Android Mobile Application Testing. IEEE 4th Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW), (2011), 252-261.
[13] Holzmann, G. J., and Smith, M. H. An automated verification method for distributed systems software based on model extraction. IEEE Trans. on Software Engineering (TSE), 28, 4 (2002), 364-377.
[14] Grilo, A., Paiva, A., and Faria, J. Reverse engineering of GUI models for testing. In 5th Iberian Conf. on Information Systems and Technologies (CISTI), (2010), 1-6.
[15] Nguyen, B., Robbins, B., Banerjee, I., and Memon, A. Guitar: an innovative tool for automated testing of GUI-driven software. Automated Software Engineering (ASE) 21, 1 (2014), 65-105.
[16] Yuan, X., Cohen, M. B., and Memon, A. M. Towards dynamic adaptive automated test generation for graphical user interfaces. In IEEE Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW) (2009), 263–266.
[17] Oliveira, R., Dupuy-Chessa, S., and Calvary, G. Equivalence checking for comparing user interfaces. The 7th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, ACM, (2015).
[18] Triki, T., Ledru, Y., du Bousquet, L., Dadeau, F., and Botella, J. Model-based filtering of combinatorial test suites. In Fundamental Approaches to Software Engineering (FASE). Springer, (2012), 439-454.
[19] Calvary, G., Coutaz J., Thevenin, D., Limbourg, Q., Bouillon, L., Vanderdonckt, J. A Unifying Reference Framework for Multi-Target User Interfaces, Interacting With Computers, Vol. 15/3, (2003), 289-308

GUI-based Testing in the Brazilian Software Industry: A Survey

Rafael A. P. Oliveira[1], Jorge Francisco Cutigi[2]
[1]University of Sao Paulo, USP, São Carlos
rpaes@icmc.usp.br
[2]Federal Institute of Sao Paulo, IFSP, São Carlos
cutigi@ifsp.edu.br

Abstract: GUI (*Graphical User Interface)*-based testing consists of exploring front-end resources to exercise a Software Under Test. Despite the fact that GUI-based testing strategies are in constant evolution, the distance between practice and theory remains a problem in this field. Due to this, the practice of automated and systematic GUI-based testing is severely limited by costs and the need for specific tools. Aiming to produce insights on the current practice and the needs of GUI testing in the Brazilian software industry, in this paper we conducted a survey that contributes to deep analyses and discussions on the practice of GUI-based testing in the scenario of the Brazilian Software industry. Based on the answers from 49 practitioners from leading companies, our survey reveals that problems such as the lack of knowledge, lack of supporting tools, and tight deadline so large projects, lead most of the Brazilian companies to perform manual and ad-hoc GUI testing instead of exploring automated and systematic approaches. Then, the main contributions of this paper, in addition to quantitative and visual information, is a discussion associated with a proposal of some future directions towards the incorporation of GUI testing strategies by Brazilian companies and companies from other emergent countries.

Keywords: GUI-based testing; Test automation; Graphical User Interface

1 – Introduction

Currently, the global economy has been guided by technological advances in *Information Technology* (IT). Having an active software industry can increase the representativeness of a whole country in a global economic context. Software testing activities are intimately associated with a powerful and influential software industry, and are seen as being one of the most important practices to increase the software quality and productiveness [14]. Software testing collaborates to ensure that the software product is in accordance with its specifications, reducing costs of maintenance [14]. However, some emergent countries do not have good practices on software testing, limiting their ability to participate more effectively in global economic activities. For instance, Brazil has a representative local software industry that needs direction so that it can to act better in international scenarios. To do so, initiatives from academia need to be implanted in practice by industry.

A contemporary practice in software testing is GUI testing (*Graphical User Interface)* (also known GUI-based testing) that consists of exploring front-end resources to exercise a *Software Under Test* (SUT) [1]. Technically, a GUI consists of an array of graphical components, widgets, and properties that allows final users to interact with the SUT's underlying code (back-end code). In a GUI, each graphical component dispatches an event to respond to user's interactions from keyboards, mouse, *touchscreens*, etc [2]. Then, test cases can be designed through sequences of input events that exercise the GUI's widgets. Theoretically, GUI testing represents an essential step towards the proper verification of the consistence among a system and its specification [3]. Nowadays, even if all of the underlying code has gone through systematic VVT activities (*Validation, Verification, & Testing* activities), it is not recommended to deliver a SUT without the application of a GUI testing strategy. Given these concepts, it is possible to affirm that, in conjunction with other testing techniques and criteria, GUI testing is absolutely necessary to build successful software [4,5]. However, in practice there are huge gaps between theory and practice that causes automated and systematized GUI testing to be avoided by most of the practitioners in software industry [7]. These gaps are mainly due to different reasons: lack of proper tools, maintenance of testing scripts, manual efforts, generation of useless test cases, and difficult to design effective test oracles.

Aiming to produce insights on the current practice and needs of GUI testing in the Brazilian software industry, in this paper we conducted an online survey. Through this survey, we measured common practices and testing strategies adopted by a representative portion of leading companies of software development in Brazil. The main goal of this survey is to create quantitative data to support ideas and initiatives towards the productive use of GUI testing in practice. Using professional social media and personal contacts, we invited more than 70 practitioners of top IT companies in Brazil to participate of our research. Based on a set of data collect from more than 50 practitioners, we designed the results of this study. Our survey is prepared to answer the following five *Research Questions* (RQs):

RQ1: Do the practitioners from the Brazilian software industry know all of the GUI testing concepts?

RQ2: What are the GUI testing strategies being applied in practice (in accordance to a chronological taxonomy defined by Alegroth E. [6])?

RQ3: What are the reasons to not apply GUI testing in practice?

RQ4: Does the Brazilian software industry implement supporting tools for GUI testing?

RQ5: Do the practitioners know the importance of the GUI testing for the quality of the final product?

Besides the quantitative analysis and visual information on the practice of GUI testing by leading Brazilian companies, the contributions of this paper are threefold:*(1)* it produces insights for future directions on the practice of GUI testing strategies; *(2)* it presents an effective analysis questionnaire that can be adapted and applied in other countries under different contexts; and *(3)* it provides a massive discussion with critical points of view and comments on what is expected from the GUI testing in the software industry of an emergent country.

2 – GUI-based Testing

Regarding the code level, a GUI is a **3-tuple: *{W, P, V}*:W** is a set of Widgets (Labels, Forms, Buttons, Lists, Sliders, Spinners, Menus, etc);**P** represents a set of properties to each Widget (Size, Color, Background-color, Translucence, Shape, etc.); and **V** addresses to a set of valid Values associated to each Property **P**. Technically, this 3-tuple representats a concept known as *"GUI state"* that is a set of values to *W*, *P*, and *V*, representing an instance of a GUI. Regarding the software testing point of view, a consequence of the 3-tuple representation is that the GUI testing is different from regular testing techniques.

Historically, 1991 was the year in which the first research efforts on exploring GUI resources to test the SUT's functionalities were published [1]. In a recent analysis on the GUI testing history, Alegroth E. [6] states that during the last decade (from 2005 to 2015) this area evolved the following three chronological approaches that can be seen as generations: (1) Record/Playback-based; (2) GUI model-based approaches; and (3) Visual-based approaches. This taxonomy is based on the source of information and on the level of automation associated with the GUI testing techniques. Below we present details of each testing generation:

Generation 1 (G1): this first generation is totally scripted and it uses record/playback tools that allow the testers to record testing sequences. Then, after new implementations and code updates, these testing sequences can be playedback automatically. Due to the usage of screen coordinates or specific code targets to perform the test record, scripts maintenance in this generation is a painful and costly activity;

Generation 2 (G2): this generation, which can be called the component-based generation, explores a model to represent all of the possible GUI events and interactions. From such a model, which can be derived from rippers implemented through reverse engineering techniques or even from manual scripts, it is possible to generate test cases and design several test oracles. This generation has a high level of automation, however the high number of test cases and ripper/script-dependency are some of the drawbacks of its approaches;

Generation 3 (G3): also known as *Visual GUI Testing* (VGT), this generation uses processing image techniques and resources from image recognition algorithms to exercise the GUI and assert the SUT's correctness through the GUI representation displayed on the screen. Then, approaches in this generation are script-based and code-independent. Once this generation represents a novel trend on GUI testing, there is an open field of research efforts to be done regarding test data generation and test productivity. Despite the fact that only initial research have been conducted [6], this generation seems to suffer from several robustness problems, mainly due to costly image processing algorithms [8].

The GUI testing generations are complementary and good testing strategies are given through the combination of approaches from two or more generations. Regarding GUI testing, for instance, there are valuable research efforts and tools that could be adopted in practice, for instance, PBGT-approach [13] for GUI testing of web applications, GUITAR [12] and Murphy tools [11] for desktop applications, Sikuli [10] for VGT testing, and so forth.

3 – Survey Design and Execution

In order to make survey access and distribution easy, we developed the questionnaire[1] using a free online service. We planned a questionnaire with objective questions (multiple choice), open questions (text), and scale questions, totalling 21 questions.

We divided the questionnaire into 2 sections: at the first section, we presented the participants with a small introduction regarding GUI Testing. In this introduction, we presented the three generations of GUI testing (Section 2) and we named as ***Generation 0* (G0)** all of the GUI Testing executed manually with no tool support at all. In the second section, we added the survey questions. The first five questions were about the professionals and the company that they worked for. The next five questions were about a testing activities (testing levels, automation, etc). The next eight questions were related to GUI testing, which asked

[1]access online the complete questionnaire: http://goo.gl/forms/WFk4vHM6R6

about the professional’s knowledge and how this kind of testing was performed at the Brazilian companies. Finally, the three last questions got information about the contact of the responder and an open space to comments about the questionnaire or GUI testing.

We developed specific questions to answer our research questions. The questionnaire was e-mailed to practitioners that are involved with software development, focusing on testing practitioners of leading Brazilian IT organizations. We started to distribute the questionnaire on June 8, 2015, in which only testing practitioners were the target. After June 13, 2015, we prepared a new distribution batch for broader group of software practitioners, not only testers. The next batch was sent after June 17, 2015, in which we collected another good amount of responses. We received 56 responses. Among the 56 responses seven were discarded because some of their answers were not complete enough, missing some necessary points to our analysis. After that selection, our survey comprised 49 responses.

4 – Survey Results

Analysing the first part of our survey, one can notice that we had responses from different major business, such as information technology, consulting, finance, education, agribusiness, healthy, etc. We also got information of some government industries, representing around 16% of the respondents. 31% of all respondents work for large companies (more than 1000 employees) and 27% work for small companies (less than 50 employees). About the respondent's job positions, we have almost 27% responses from practitioners related directly with software testing/quality activities, while others are directors, project managers, analysts or developers.

Regarding general software testing activities in Brazilian companies, we asked some important basic questions, aiming to know about testing levels and their automations. Figure 1 presents the results obtained in a bar graph, in which each section is how automated the level testing is, or if this level is not used in the company. We can notice that most of the companies perform the testing activities, although a good amount of companies make it manually or ad-hoc. We also asked how satisfied the respondents are about the software testing activities performed in their company: only 10.2% of the respondents are very satisfied with the testing in their company, while 22.4% are very unsatisfied. Around 63.7%, declared to be partially in accordance with the testing activities performed by their companies.

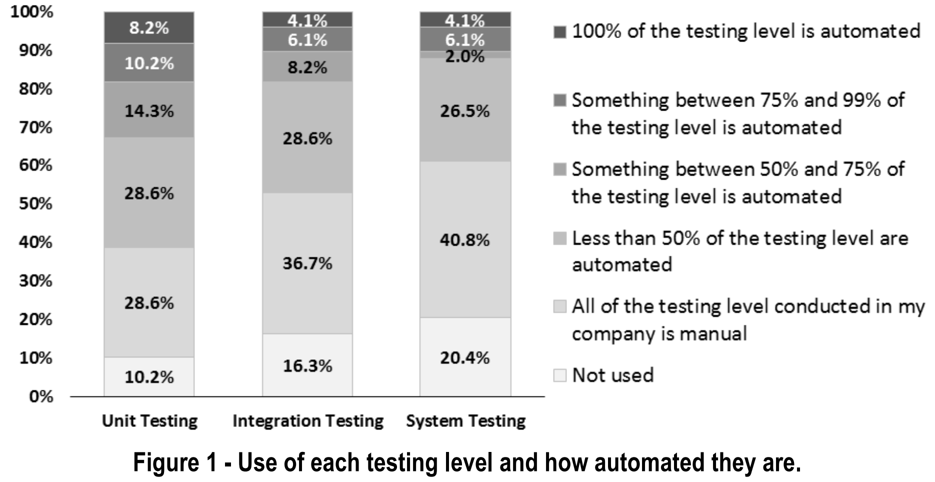


Figure 1 - Use of each testing level and how automated they are.

The section below presents potential answers to our RQs based on the collected through our survey.

4.1 – Answers to RQs and Discussion

In the paragraphs below we present potential answers for our RQs (presented in Section 1) and some comments on the implications and evidences collected from the survey.

Regarding the *RQ1*, our survey has revealed that few practitioners (only 69% of the respondents) had previous knowledge of concepts and techniques of GUI testing. Then, a considerable amount of practitioners (31% of the respondents) affirmed they did not have any previous knowledge of GUI testing. We expected more practitioners with previous knowledge on GUI testing. In addition to that, we believe that these critical numbers are due to two main reasons: (1) defective undergraduate IT courses with few or no software testing lessons; and (2) GUI testing is a relatively new concept and the Brazilian software industry did not incorporate testing strategies and artefacts to address this issue.

With regard to *RQ2*, that is addressed to identify which are the GUI testing generations being used in practice by Brazilian software industries, our survey reported four main findings: (1) an extensive amount of the companies (87.8%) use manual and ad-hoc GUI testing approaches from G0; (2) few practitioners explore approaches from G1 (38.8%) and G2 (26.5%); (3) few companies (8.2%) use modern approaches from G3; and (4) a considerable portion of the Brazilian software industry is careless about GUI testing. Figure 2 presents all of the data collected from our survey to identify the usage of the testing generations in practice. The left side of Figure 2 presents a bar plot with the percentage of usage of each generation. The right side of the same figure illustrates the numbers of each survey respondent using each GUI testing generation.

The answers from our survey to RQ2 show us that most of the practitioners use manual approaches (G0) or record/playback strategies (G1) to test their GUI applications. This evidence was already expected by us for two main reasons: (1) GUI testing is a new practice, then it is natural that most of the companies are still incorporating its practice in their daily activities; and (2) the lack of mature GUI testing tools cause some companies explore approaches that are more human-dependant (G0 and G1).

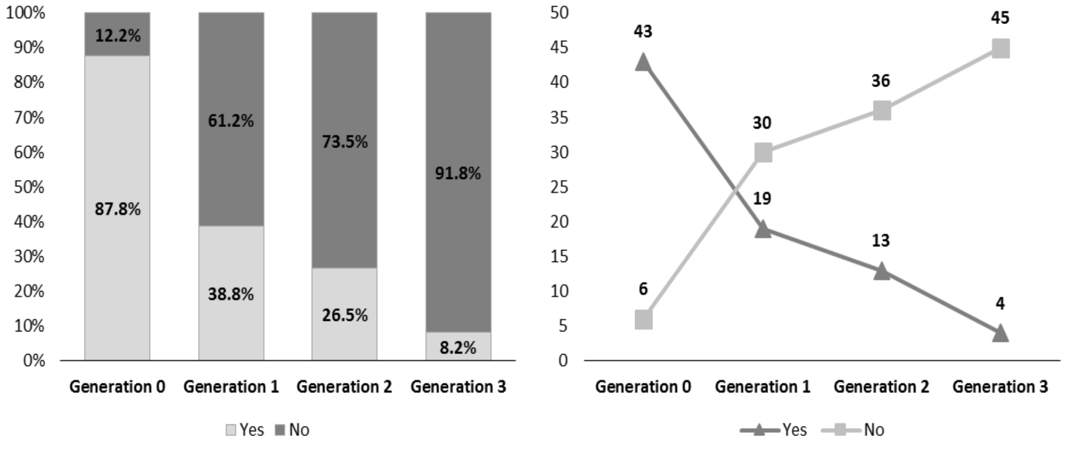


Figure 2 -Rates (left) and numbers (right) for the practical use of each GUI testing generation.

RQ3 addressed raised evidence about reasons and barriers to not employ GUI testing in practice. To obtain data towards answering this question, we adopted open questions: GUI testing practitioners and even for practitioners that are negligents about GUI testing. After analysing several responses, we classified the answers into four categories: (1) *negligence or careless* about applying GUI testing; (2) *time issues*; (3) *lack of knowledge*; and (4) *lack of supporting tools*. Figure 3 presents the final results of our analysis. For 36% of the practitioners, the "lack of supporting" tools represents the main limitation to apply GUI testing in practice; 29% of the practitioners complained about "time issues"; 18% declared that sometimes the "GUI testing is neglected" in their projects; and 17% declared believe the "lack of knowledge" and capacity is the main drawback on using GUI testing in their companies. Still about RQ3, we believe the results were near to what we expected. This is due to two main reasons: (1) the lack of proper supporting GUI testing tools for different programming languages, platforms, and paradigms is already known among researchers and practitioners in this field. In addition to that, the lack of automated resources leads to time issues associated to the limitation of its usage; (2) although, we did not expect that a large amount of practitioner would declare that sometimes the GUI testing is neglected.

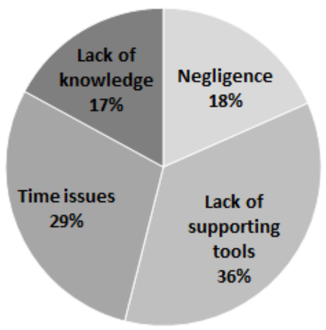


Figure 3 - Main reasons to not apply GUI testing.

With regard to RQ4 that investigates whether GUI testing tools are implemented by companies to be used in their own projects, 8% of the respondents affirmed that it is a common practice to implement their own testing tools. From the data collected, one can infer that the particularity of some projects makes 8% of the companies implement GUI testing artefacts. Consequently, 92% do not implement their own automated or semi-automated testing tools. However, this huge number does not mean that Brazilian industry have mature tools supporting, but it show us that software testing is sometimes neglected or performed manually.

Still regarding RQ4, analysing details of cases in which the professional affirmed their company implements their own tools for GUI testing, we noticed two main evidence: (1) there are cases in which the company does not implement a complete tool, however they establish a particular software architecture and a testing pattern to keep their test sets easy to change; (2) the supporting tools implemented can be classified as G1-tools because most of them consists of scripts that reproduces a real-user interacting with the SUT's GUI through its GUI using coordinates and components' IDs.

Finally, to answer RQ5 that consists of checking the level of priority given by Brazilian practitioners for GUI testing, we prepared a scaled question that starts from one (1) ending in five (5). Our data shown that 42.9% of the respondents considered the GUI testing very important for the SUT’s quality, and 36.7% considered the GUI testing important. Figure 4 presents a bar plot in which the answers of the participants are synthesized through visual information. Regarding the answer of this question, one can considered that most of the professionals recognize the importance of the GUI testing for the SUT’s quality, however number of them refuse or neglect to adopt automated and systematic GUI-based testing strategies in their daily activities.

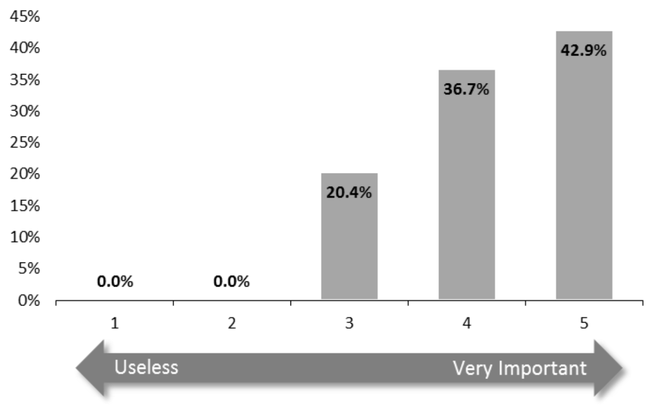


Figure 4 - Level of importance of the GUI testing for Brazilian IT practitioners.

In an additional analysis, we systematically evaluated some of the data of our survey to represent a draft of the state-of-the-practice of the GUI testing in Brazil. Considering that the most prolific GUI testing scenarios must be composed by the combinations of tools from two or more generations (preferentially, combinations from G2 and G3), we analysed the answer from each practitioner individually to determine the combinations of GUI testing generation approaches. Then, for the survey response, we analysed the combination of tools from the four different generations presented. Figure 5 shows the final numbers for this analysis arranged through a Venn diagram.

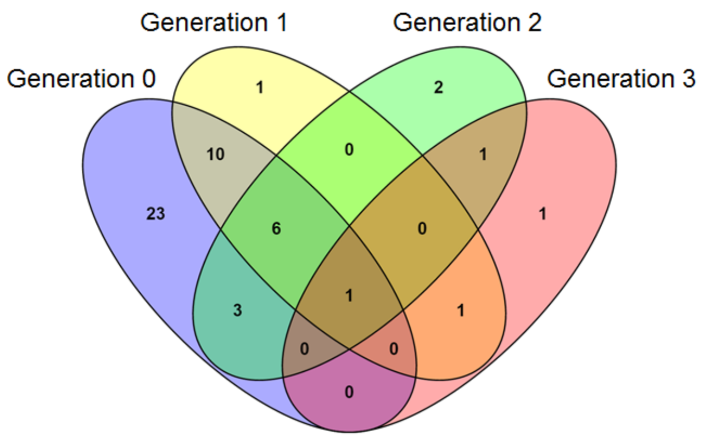


Figure 5- Venn diagram for the combination and usageof each GUI testing generation. (created using [9])

Analysing the visual information provided by the diagram, one can notice several important points: (1st) 23 practitioners (in blue bubble) declared their company uses only approaches from G0; (2nd) 10 respondents said that in their companies are used approaches from G0 and G1 (intersection between blue and yellow bubbles); (3rd) six respondents declared their companies combine approaches from G0, G1, and G (intersection between blue, yellow, and green bubbles); and (4th) only one respondent said his company works along all of the testing generations (central area and intersection of all of the bubbles).

Regarding this combination of generations, it is possible to mention that the Venn diagram and the data from our survey reveal an immaturity of the GUI testing in the Brazilian industrial scenario. The immaturity of the area is evident mainly for three reasons: (1) most of the combinations involves the G0 and G1 that are considered outdated facing the current scenarios; (2) in general, few respondents declared their company combined tools from different generations, demonstrating most of the companies have to deal with drawbacks and limitations of particular approaches; and (3) few companies are using the two newer generations and their combination. Regarding the combinations, we highlight that the ideal scenario expected consists of more companies using approaches from G2, G3, and their combinations in different contexts (in the intersection between green and pink bubbles).

After analysing the data collected from our survey, we believe that some directions regarding GUI testing are fundamental to increase the Brazilian competitiveness in the software industry. We see some initiatives that are necessary to improve the scenario of GUI testing in Brazil. Firstly, it is necessary for *"more collaboration between academia and industry"* towards the development of open source testing tools and testing resources applicable to GUI testing in different contexts, mitigating testing costs and human efforts. Secondly, customers, universities, and government have to be engaged *"towards the exigency of more reliable"* and faithful software products. In this context, customers should be more energetic in demanding well tested software. Universities should incorporate more quality issues in their IT courses offering specific lessons of topics on contemporary testing practices (such as, GUI testing). In addition to that, the Brazilian government and Universities should be associated to coordinate funding programs to allow researchers to realize local workshops and conferences about contemporary IT topics. Finally, the Brazilian software industry by itself should *"look for capacity programs for their employees"*, preparing their professionals for contemporary practices.

4.2 – Threats to Validity

The threats of this study are presented on three perspectives:

Credibility: the main threat to the credit of the findings collected from this study is associated with the survey population. As mentioned in Section 3, we opened the survey, not only for testers, but also for practitioners from different job positions (developers, analysts, managers, etc). Thus, we take the risk that some of our survey respondents are not able to provide precise information on GUI testing practices.

Dependability: we see a threat to the consistency of our findings associated with the fact that some of our survey questions (mainly the open questions) could make the results inconsistent. Then, to mitigate this threat, we discarded some questions and defined some assumptions and performed a few to make our analysis possible.

Confirmability: regarding the real association of the answers to our RQs and the data collected, we believe there is threat associated to our fourfold categorization about the drawbacks on using GUI testing in the RQ3. This threat is due to our previous knowledge on GUI testing. To mitigate that threat, the categorization was suggested by two different authors and validated by a specialist.

5 – Conclusions

GUI testing is a contemporary VVT activity that explores UI aspects to exercise the SUTs aiming to check inconsistencies. The GUI testing field is advancing and it will be soon still more practical and useful. Based on five previously defined RQs, this survey study reveals several critical points about the practice of GUI testing by Brazilian practitioners from leading software companies: (1) few practitioners (69%) know GUI testing concepts, supporting tools, and resources; (2) most of the GUI testing strategies are still manual and ad-hoc; (3) lack of proper automated GUI testing tools and time to market are the main excuse and barriers to the effective use of contemporary GUI testing strategies in industry; and (4) despite the fact that most of the professionals know the importance of GUI testing, they continue refusing to use it in their daily activities. In this regard, as a contribution, this paper provides some initiatives towards increasing the practical usage of GUI testing and other contemporary software testing practices in the Brazilian software industry. Specific open source tools, academia engagement, and efforts to train professionals are some of the directions to be followed. As a future work, in addition to collecting answers from more practitioners, we intend to run this survey in countries whose software industry is at similar levels as Brazil (India, Israel and Ireland) and in reference countries (USA, China, and Germany), establishing then some basis for comparisons between Brazilian industry and other countries, besides opening opportunities to write new papers to be discussed in key conferences.

References

[1] *“Graphical User Interface (GUI) Testing: Systematic Mapping and Repository”* by Ishan Banerjee, Bao Nguyen, Vahid Garousi. and Atif Memon, Information and Software Technology, Vol.55, I. 10, October 2013, pp 1679-1694.

[2] *“GUI Testing: Pitfalls and Process”* by Atif M. Memon. Computer, IEEE Computer Society Press. vol. 35, no. 8, 2002, pp. 87-88.

[3] *“What Test Oracle Should I Use for Effective GUI Testing?”* by Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. In Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003), Montreal, Canada, pp. 164-173. 2003.

[4] *"A systematic capture and replay strategy for testing complex GUI based java applications"*, by O.E. Ariss, D. Xu, S. Dandey, B. Vender, P. McClean, B. Slator. in: Conference on Information Technology, 2010, pp. 1038–1043.

[5]*"Modelling and testing hierarchical GUIs",* by A.C.R. Paiva, N. Tillmann, J.C.P. Faria, R.F.A.M. Vidal, In: Proceedings of the 2005 Workshop on Abstract State Machines (ASM 2005), Pairs, France, pp. 8-11, 2005.

[6] *"On the industrial applicability of visual GUI testing"* by E. Alegroth, ´ Department of Computer Science and Engineering, Software Engineering (Chalmers), Chalmers University of Technology, Goteborg, Tech. Rep., pp. 1-187, 2013.

[7] *"Making GUI Testing Practical: Bridging the Gaps"* by Pekka Aho, Matiaz Suarez. Atif Memon, and Teemu Kanstrén, in The Proceedings of The 12th International Conference on Information Technology - New Generations (ITNG 2015), Las Vegas, NV, USA, pp. 439-444, 2015.

[8] *"Conceptualization and Evaluation of Component-based Testing Unified with Visual GUI Testing: an Empirical Study"* by Emil Alégroth, Zebao Gao, Rafael A. P. Oliveira. and Atif Memon, in The Proceedings of 8th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2015), Graz, Austria, pp. 1-10, 2015.

[9]*"An interactive tool for comparing lists with Venn's diagrams"* by Oliveros, J.C. (2007-2015) Venny.http://bioinfogp.cnb.csic.es/tools/venny/index.html

[10] *"Sikuli: using GUI screenshots for search and automation"* by Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. In Proceedings of the 22nd annual ACM symposium on User interface software and technology (UIST 2009). ACM, New York, NY, USA, pp. 183-192. 2009

[11] *"Murphy Tools: Utilizing Extracted GUI Models for Industrial Software Testing"* by Pekka Aho, Matias Suarez, Teemu Kanstren. and Atif Memon, In Proceedings of the Testing: Academic & Industrial Conference (TAIC-PART), Cleveland, OH, USA, 343-348 2014

[12] *"GUITAR: an innovative tool for automated testing of GUI-driven software"* by Bao N. Nguyen, Bryan Robbins, Ishan Banerjee. and Atif Memon, Automated Software Engineering, 2013, pp. 1-41, Springer US.

[13] *"A Pattern-Based Approach for GUI Modelling and Testing"* by Rodrigo Moreira, Ana Paiva and Atif Memon, In Proceedings of the 24th IEEE International Symposium on Software Reliability Engineering (ISSRE 2013), Pasadena, CA, pp. 288-297, 2013

[14] *"Software Testing Research: Achievements, Challenges, Dreams"* by Antonia Bertolino. In Proceedings of the Workshop on the Future of Software Engineering (FOSE 2007). IEEE Computer Society, Washington, DC, USA, pp. 85-103, 2007

Closing Gaps between Capture and Replay: Model-based GUI Testing

Oliver Stadie and Peter M. Kruse

Berner & Mattner Systemtechnik GmbH, Berlin, Germany,
{oliver.stadie|peter.kruse}@berner-mattner.com

Abstract. Testing software as a black box can be time consuming and error-prone. Operating and monitoring the graphical user interface is a generic method to test such systems. This work deals with convenient and systematic testing of GUI software systems. It presents a new approach to model-based GUI testing by combining the strengths of four well-researched areas combined: (1) the intuitive capture&replay method, (2) widget trees for modeling the GUI, (3) state charts and (4) the classification tree method. The approach is implemented as a prototype and is currently under validation on a real GUI. The presented approach includes the whole test cycle, from scanning the GUI and model-based test specification to the automatic execution of tests.

Keywords: Automated GUI Testing, Systematic GUI Testing, Model-based Testing, Classification Tree Method, State Chart, Capture&Replay

1 Introduction

There are many approaches, how to test software. Today, many software systems provide a graphical user interface (GUI) to ease the users to access these systems. When testing these software systems, the GUI can be used to test the software from the user-perspective (black box testing) or to test the GUI itself. In current systems, the GUI takes up to 60 percent of the total source code [1]. Testing generally causes 50 percent of the total costs of software development [2]. By automating GUI tests, up to 80 percent of the costs could be saved compared to manual GUI testing [3].

Especially for regression testing, a major problem is that changes to the GUI should not require manual steps in order to adapt the test [4].

Semi and fully automatic methods have been published, in order to simplify the GUI testing process [3, 5]. Nevertheless, the approach most widely used is still capture&replay [6]. This gap between explored and used methods could be due to a lack of intuitiveness, learning and lack of tool support.

In this work we develop a method to support the GUI testing process by combining existing methods. The developed method is to be implemented in a tool that can be used for any type of GUI, regardless of the underlying technology. Specifically, the following methods and models are used: the *capture&replay method* [7], *widget trees* [8], *state charts* [9] (esp. UML state diagrams), and the *classification tree method* [10].

Table 1. Combined Methods and Models in This Work

Method	Advantages	Disadvantages
Capture&Replay [7]	- Intuitive Usability - Widespread - Quick and Easy Specification of Individual Sequences - Simple Means to Scan the GUI	- High Maintenance Costs - Low Stability against Changes
Widget Trees [8]	- Detailed Modeling of GUI States - Convenient Management and Overview by Hierarchies	- Stability to Changes Uncertain
State Charts [9]	- Modeling and Selection of System Behavior (Sequences) - Relatively Stable to GUI Changes - Easy to Learn	- Difficult Automated Construction
Classification Tree Method [10]	- Classification of the Input Data Space, Reduction of the Necessary Test Cases - Systematic Derivation of Test Cases - Established in Practice - Suitable for Functional Black Box Testing	- Can be Too Large for Complex Systems *(Splitting into Several Trees Circumvents the Problem)*

These methods are to be combined in the following sections, with the aim to obtain as many of the benefits as possible and to eliminate as many of the disadvantages as possible (Table 1).

2 Background

The widespread capture&replay tools work as follows: The tester records a manually executed sequence of actions on the GUI. This is the capture phase. Then the recorded sequence can automatically be executed on the GUI repeatedly. That's the replay phase. One advantage of capture&replay tools is that they can be easily learned and used. A draw-back is that they are not inherently systematic, so the quality of the recorded test depends of the skills of the tester [7].

Memon pays attention to both the importance and the lack of GUI test methods [7]. Methods used are often unsystematic, ad hoc or too expensive.

Surveying recent works related to model based GUI testing [11–13], reveals that there are two dominant methods for modeling GUIs. One common approach in model-based testing are state charts [5]. A second approach is the model of Memon et al. consisting of *GUI forest*, *event-flow graphs* and *integration tree* [14].

Widget Trees are another approach to modeling of GUI-states [8]. Widget trees focus on the modeling of all elements in the widget hierarchy, while state charts model the behavior and possible navigation paths through the system.

A systematic method for test specification is the classification tree method [10]. TESTONA is a test tool that implements the classification tree method [15, 16].

3 Approach

The developed method supports the tester in testing a model-based GUI. The method forms a cycle, which (1) starts and analyzes the GUI, it then (2) creates GUI models, from these (3) derives test sequences and ultimately (4) executes these sequences again on the GUI (Figure 1).

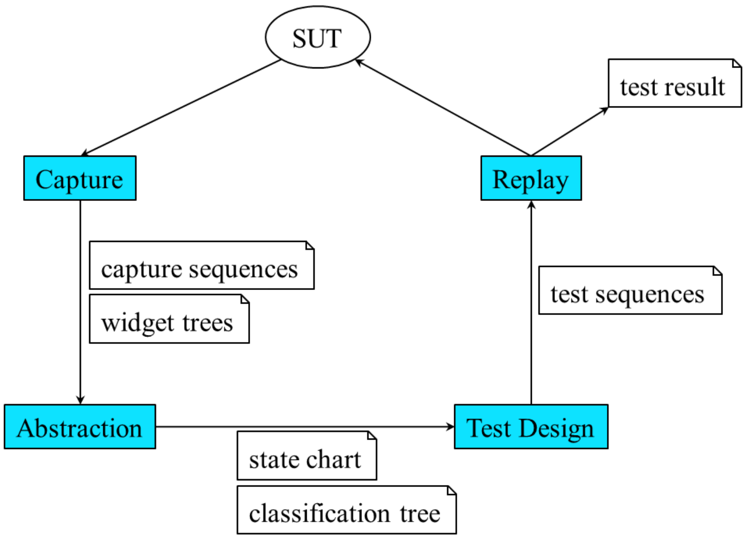


Fig. 1. Workflow

1. The tester initiates the capture process part. The tester performs to be recorded sequences, by using peripherals[1] (mouse and keyboard), on the system under test (SUT).
2. The inputs in the periphery and the output of the SUT can be observed and models of GUI are created and updated from it. After the tester completes a sequence, they can restart the capture process any number of times to record additional sequences.
3. The created GUI model is presented to the tester for test sequence generation.
4. Finally, the tester triggers the test sequence execution. The SUT is started automatically for each test sequence and manipulated automatically via generated, simulated peripheral inputs.

3.1 Capture

The first phase of the process ensures the creation of capture sequences. After starting the SUT, its GUI is scanned to determine its initial state. The tester makes any number

[1] Device used to put information into or get information out of the computer. Also called input/output device. [17]

of inputs on the keyboard and mouse during each recording. Each of these inputs affects the SUT. Thereafter, the method links the input to the last recorded GUI state and reads the new GUI. Each GUI scan is stored as a widget tree.

Stabilization: For all widgets in the widget tree, the details are reduced and only names and types are kept, resulting in a discarding of e.g. widget dimensions, pixel positions, colors, IDs (similar to [18]). This is done for increased robustness, esp. in regression testing. The interrelation of widgets is only maintained using their location in the widget tree.

Merging: Multiple similar user actions without consequences to the widget tree can be merged, e.g. typing several single letters into a textbox or moving around the mouse without actually clicking (assuming there are no interactive reactions cause by the typing and no hover-reactions of traversed GUI elements). This is done do limit state explosion.

Each capture sequence contains a (merged) chain of traversed states and transitions of the SUT. Each transition has an atomic action as its trigger. Each state consists of a stabilized widget tree. The model of capture sequences is a non-empty set of capture sequences, which in turn are modeled as traversed states and actions performed.

3.2 Abstraction of Models

After capturing has been completed, abstraction of GUI Models is performed. The capture sequences lack a relationship between the sequences and their branches. Therefore, we use the following heuristics to merge the capture sequences:

Treat equal what looks equal: Since operation is performed on stabilized widget trees, the algorithm merges sequences so that equal looking steps (containing widgets with same name and type) are merged into single states of the resulting state chart.

Hierarchy: To create state chart hierarchies, a state on the first level of the state chart is created for each modal window. Each window state has a set of sub states for different application modes. All widget trees from all capture sequences with the same type structure (discarding all other properties, such as name) are merged into single application modes (all on the second level of the state chart). Each application mode state contains a set of sub states, derived from the text of the widgets in the widget tree. So we use the similarity in widget tree structures for creation of states on second level and the differences in widget tree properties (especially widget text) for creation of states on third level of the state chart.

Concurrency: Orthogonality is created using a set of predefined widgets, such as the main menu and pop-up menus. Once a menu is used in any sequence recorded, it is considers always accessible, independent of actual access in recorded sequences. The behavior of each such menu is modeled in its own orthogonal region in the state chart.

The first and the third heuristic here increase the possible number of variations and permutations in later test design. In contrast to the plain playback of linear sequences in conventional capture&replay. This generalization might however lead to non-executable sequences.

Each GUI-model consists of a set of widget trees, a state chart and a classification tree created from the state chart (as described in [19]). Each widget tree is assigned to exactly one state in the state machine.

3.3 Specification of Test Sequences

In the third phase, test sequences are specified. First, the classification tree part of the GUI model is used to identify test sequences and—as in the ordinary classification tree method [19]—described in a test matrix. Each sequence also represents a path through the state chart. The state machine is used to constrain possible test sequences. The sequences determined meet coverage criteria such as state coverage or path coverage.

Each test sequence defines a sequence of states to reach. Since the state chart has a higher abstraction level than the captured sequences, non-caputured sequences may occur here. The handling of infeasible paths in sequences is not yet automated and therefore left to the tester. The required actions to traverse the states are defined in the state chart. As such, events carry those input values (e.g. for text fields) that were recorded in the initial capture phase. The tester can however adopt these as part of test specification.

3.4 Test Execution

In the fourth phase, the previously specified test sequences are executed automatically. At this stage, all given test sequences are treated sequentially. At the beginning of each test sequence, the SUT will be started automatically. Similar to [18], it is required that the SUT always starts into the same initial state. The tester needs to take care of this (e.g. by resetting the SUT preferences prior test execution).

Each test step is then processed from each test sequence. First, the GUI is scanned to identify its widget tree. The to-use widget is searched in the widget tree. The action to be carried out by operation of the peripherals is then simulated. The tester can define a delay time between execution steps. Otherwise all events are fired as fast as possible, potentially leading to the SUT not receiving all events.

After all steps of a test sequence are performed, the SUT is terminated. After completing all the test sequences, this phase ends.

Simple test results can be produced here by comparing the actual with the expected widget trees after each step. This also includes reporting whether each action could be performed.

4 Evaluation

The developed solution has been implemented as a plug-in for TESTONA[2] and currently works for all GUIs in Windows operating systems (Figure 2). To this end, several existing works—as frameworks and libraries—have been reused.

The approach used here conforms to a general structure for GUI tests [20]: A testing framework consisting of a technology-independent GUI model and the general test sequence specification were already implemented as XML specifications.

The implementation of the developed method is currently under evaluation quantitatively and qualitatively for several GUI systems. The results of evaluation are intended

[2] `http://www.testona.net`

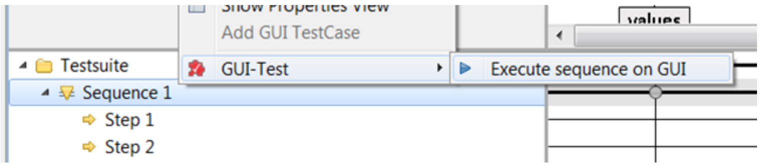


Fig. 2. Execution of Tests

to provide indications of the practicality of the developed test design process and about the quality of the prototype.

We have obtained some first results. In our approach widgets are described in terms of their name and type. This choice is sufficient for inferring a good abstraction of the model in the Windows applications tested, as for example most of the time button names were similar to their caption. This might be due to the Windows API or simply be good design of tested applications (Windows Calculator, TESTONA tool itself). With a growing body of applications under test, we might need to consider different stabilization rules.

Currently, keyboard and mouse inputs by the tester are the only

Table 1 lists the four methods used in our approach with their individual strengths and weaknesses. We will now evaluate, whether the combination of methods helps to overcome weaknesses without sacrificing the strengths.

4.1 Capture&Replay

A general problem with capture&replay are the high maintenance costs due to low stability of captured sequences against changes. By using widget trees and by scanning the GUI during test execution, our approach reduces the maintenance costs with increased stability against changes of the GUI. Details on cost reduction and on how stable the approach actually is, have not yet been provided due missing evaluation in detail.

The advantages of capture&replay are all preserved. Our combined approach also relies on the intuitive usability. The recorded individual sequences are, however, used to abstract a general GUI model, which allows more variation in later test specification.

4.2 Widget Trees

The stability of widget trees against changes in the application has not yet been assessed. Without a large-scale evaluation, we cannot yet overcome this problem.

The advantage of widget trees, both detailed modeling of GUIs and the introduction of hierarchies for better overview, are both kept.

4.3 State Charts

The construction of state charts is a challenging task. By introduction of heuristics provided, this weakness is completely resolved.

The strengths of state charts are all maintained. The influence of GUI changes to the stability of state charts have not yet been verified.

4.4 Classification Tree Method

By only including relevant parts of the state chart to the classification tree, the size of trees is kept small.

The mentioned advantages are all preserved. The automated generation of test sequences is considered helpful.

5 Related Work

Bauersfeld and Vos also implement a tool for testing GUI system: GUITest [18]. Their tool provides the following features, also present in our implementation: *a)* Works on all native GUIs, which are recognized by the Windows API. *b)* SUT must not be instrumented. *c)* Allows the user to define their own actions. *d)* Generated Test sequences can be stored and played back.

In contrast to GUITest our tool offers the following features: *a)* GUITest specializes in robustness tests. That is, it searches automatically for random test sequences through the GUI, without necessarily representing realistic or target-oriented user behavior. The point is to find errors. Our implementation is especially useful for functional testing. The aim here is to test if specific requirements are met and the SUT fulfills its intended purpose. *b)* Compared to GUITest our implementation displays the model of GUI and test sequences.

Memon et al. also offer an implementation—similar to this work—for model-based GUI testing, with prototypical capture, semi-automatic modeling and automated execution [14]. While Memon et al. model the SUT with GUI forests, event-flow graphs and integration trees, this work uses state chart, widget trees and classification trees.

We assume, that state charts are more suitable, because they are more compact due to hierarchies and orthogonality. Test models intended for end-user (Tester) presentation should be understandable. In this case, state charts might be better, esp. when dealing with self-transitions, which can occur in GUIs. However, state charts are not considered better *per se*.

Nguyen et al. combine state charts with the classification tree method [21]. They choose paths on the state chart, representing abstract test sequences. For each of these paths, they construct a classification tree. With these trees several specific test sequences are specified for each path. Nguyen et al. see the strength of the state charts in the specification and selection of sequences (consecutive events) and the strength of the classification tree method in the selection of specific input parameters and a meaningful reduction of the input parameters.

In the context of web applications, there are similar approaches [22, 20]. While this also is a challenging field, our work focuses on native applications outside the browser.

6 Conclusion

The developed tool enables the user to comfortably create GUI models by capturing. GUI models are then used for systematical test design in terms of the classification tree method. Resulting test scenarios can be automatically executed on the SUT. Such scenarios allow to test the GUI itself or to use the GUI for black-box testing the underlying system.

Despite the problems worked out the combination of the four methods *capture&replay*, *widget trees*, *state charts*, and *classification trees* seem to be much-promising and suitable for the testing of GUI systems. Weaknesses of single methods were overcome without accepting many sacrifice of their strengths. Many of the problems identified will be addressed in future work.

Future work will also concentrate on a large scale evaluation and comparison with capture&replay tools in terms of efficiency and effectiveness considering both, initial creation and maintenance efforts.

References

1. Brad A Myers. User interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 2(1):64–103, 1995.
2. Frederick P Brooks. *The mythical man-month*, volume 1995. Addison-Wesley Reading, MA, 1975.
3. Michael Turpin. Survey of gui testing processes. 2008.
4. Atif M Memon and Mary Lou Soffa. Regression testing of GUIs. *ACM SIGSOFT Software Engineering Notes*, 28(5):118–127, 2003.
5. Imran Ali Qureshi and Aamer Nadeem. Gui testing techniques: A survey. *International Journal of Future Computer and Communication*, 2(2), 2013.
6. Stephan Arlt, Cristiano Bertolini, Simon Pahl, and Martin Schäf. Trends in model-based gui testing. *Advances in Computers*, 86:183–222, 2012.
7. Atif M Memon. Gui testing: Pitfalls and process. *Computer*, 35(8):87–88, 2002.
8. Sebastian Bauersfeld, Stefan Wappler, and Joachim Wegener. A metaheuristic approach to test sequence generation for applications with a gui. In *Search Based Software Engineering*, pages 173–187. Springer, 2011.
9. David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
10. Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Softw. Test., Verif. Reliab.*, 3(2):63–82, 1993.
11. Valéria Lelli, Arnaud Blouin, Benoit Baudry, and Fabien Coulon. On model-based testing advanced GUIs. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–10. IEEE, 2015.
12. Pekka Aho, Matias Suarez, Atif Memon, and Teemu Kanstrén. Making GUI testing practical: Bridging the gaps. In *Information Technology-New Generations (ITNG), 2015 12th International Conference on*, pages 439–444. IEEE, 2015.
13. Tanja EJ Vos, Peter M Kruse, Nelly Condori-Fernández, Sebastian Bauersfeld, and Joachim Wegener. TESTAR: Tool support for test automation at the user interface level. *International Journal of Information System Modeling and Design (IJISMD)*, 6(3):46–83, 2015.
14. Atif M Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 260–260. IEEE Computer Society, 2003.
15. Peter M Kruse and Magdalena Luniak. Automated test case generation using classification trees. *Software Quality Professional*, 13(1):4–12, 2010.

16. Eckard Lehmann and Joachim Wegener. Test case design by means of the CTE XL. *Proceedings of the 8th European International Conference on Software Testing, Analysis and Review (EuroSTAR 2000), Kopenhagen, Denmark, December*, 2000.
17. Philip A Laplante. *Dictionary of Computer Science, Engineering and Technology*. CRC Press, 2000.
18. Sebastian Bauersfeld and Tanja EJ Vos. Guitest: a java library for fully automated gui robustness testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, pages 330–333. ACM, 2012.
19. Peter M Kruse and Joachim Wegener. Test sequence generation from classification trees. In *Proceedings of ICST 2012 Workshops (ICSTW 2012)*, Montreal, Canada, 2012.
20. Peter M Kruse, Jirka Nasarek, and Nelly Condori Fernandez. Systematic testing of web applications with the classification tree method. In *Proceedings of the XVII Iberoamerican Conference on Software Engineering (CIbSE 2014)*, 2014.
21. Cu D Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining model-based and combinatorial testing for effective test case generation. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 100–110. ACM, 2012.
22. Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. Rich internet application testing using execution trace data. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 274–283. IEEE, 2010.

Using property-based testing to automate test case generation and diagnosis in web-based graphical user interfaces

Laura M. Castro[1], Clara Benac-Earle[2], Henrique Ferreiro[1], Macías López[1], and Miguel Ángel Francisco[3]
lcastro@udc.es, cbenac@fi.upm.es, hferreiro@udc.es, mlopez@udc.es, miguel.francisco@interoud.com

The nature of user interactions, together with the inherent state explosion to be faced when formalising the behaviour of a GUI, makes efficient and effective testing of graphical user interfaces (GUIs) still very challenging nowadays. Many methods and tools commonly used for either white-box or black-box system testing are unsuitable in practise for different reasons, yet the final quality of the system needs to be assessed more than ever.

In this paper, we show how we can use property-based testing (PBT) to automate test case generation, execution and diagnosis, when working at GUI-level, for systems in which such GUI is a web-based GUI. By opening this possibility, we believe that significant improvement can be brought to the quality of software products, at the same time that the effort needed to achieve that quality level is substantially decreased.

Introduction

A key area of software development is designing, development and testing of human-computer interfaces. Human-software communication takes place via such ‘user interfaces’, among which the most common are ‘graphical user interfaces’ (GUIs). GUIs are often the only part or aspect of a software system the users have access to, thus the way in which they exercise the software functionalities.

With the proliferation of web applications in the last decade, web-based interfaces have become one of the most popular types of GUIs. They represent a lightweight, flexible, and maintainable way of building the client side of a system, up to the challenge of continual availability of web applications [4]. In this domain, technologies and standards such as JavaScript and HTML5 are present as the user-interaction component in the vast majority of web services and web applications.

Unfortunately, GUIs (of any kind) are the software components which usually undergo the least complete and systematic testing process [10] in a software development project. Not only is testing, especially when manual, and thus labour-intensive and slow, the software development phase that suffers the most squeezing under pressure [4], the intrinsic characteristics of GUIs add significant difficulties when trying to apply techniques and tools which do a good job in other aspects of software testing, such as unit testing or integration testing. The specific challenges of GUIs testing, in close relationship with its event-driven behaviour and its visual essence, hinder the development and popularisation of methods and tools to assist developers in this task. This is in contrast to the spreading of testing techniques applied in testing other kinds of components that we are witnessing recently, such as property-based testing (PBT) [13]. Also, it can pay a bigger price when high availability is expected, as is the case of web applications [5, 11, 19].

PBT has been applied to a number of open testing challenges over the last few years [7,8,12,16,18], and has been quite successful in increasing testing coverage, efficiency, and overall effectiveness. Here, we make the following contributions:

– We introduce the first PBT-based testing approach for web-based GUIs, enabling automatic test case generation, execution, and diagnosis.

– We implement a testing library that implements this approach, which features two complementary aspects:

• a set of general properties, applicable to common interactions present in many web-based GUIs;

• a test model for general browsing, adaptable to be used with different frameworks and technologies commonly used for building web-based GUIs.

– We validate our approach and library in an industrial system, to analyse its generalisability and possible threats to validity.

Related work

The automation of GUI testing is an open research problem. As we have already mentioned, the nature of GUIs presents serious challenges, such as their visual nature, or the simultaneity of the bidirectional conversation between user and software.

There are two general approaches to finding faults in web applications: static analysis and dynamic analysis (i.e. testing). The first has limited potential, due to the dynamism intrinsic to web-based GUIs (in which more frequently than not parts of the GUI are generated on-the-fly using JavaScript and other client-side scripting languages), while the latter face a large input space and the requirement of simulating user interactions. In the state of practice manual generation of inputs that leads to displaying different pages is required [9].

The difficulties related to exercising the different parts of a GUI during testing are augmented by the great number of technologies used to develop them, and particularly their bond to particular environments, operating systems, and software or hardware platforms. Consequently, most successful attempts usually focus on a particular choice of technologies, as is the case of the FitTest project (which provides a solution for Java-based desktop GUIs [1]), or HP QuickTest Professional (which provides a VisualBasic solution for Windows systems).

In the web domain, practitioners have chosen systematisation of GUI testing by means of testing tools such as Selenium [3], or Sahi [2], that use direct interaction with the browser to provide partial testing automation. Specifically, they automate test case execution once the test case has been defined (either by manually writing script-based scenarios, or using a record-replay facility). However, since these automations are constrained to test case execution, they leave test design still mostly as an intuition-driven manual testing activity [22]. This makes it very difficult to provide any indication about testing advance or coverage, and when bugs are found they are rather hard to debug and fix.

Slightly more advanced tools generate tests by executing an application on concrete input values, and generate additional values by solving symbolic constraints derived from exercised control-flow paths, but these have not been practical, especially in the area of GUI-testing [9].

Last but not least, the high availability of web applications often leads to the deployment of new GUIs for existing services in one of these two variants: (1). in co-existence with an older version; (2). as a full replacement. In any case, the new web-based GUI can be developed regardless of new functionalities being deployed together with the GUI or the new GUI version being rolled out only for aesthetic purposes (which may involve using a different UI framework). And, in any case, all the testing effort invested in testing a previous version of a same-functionality web-based GUI is not reusable for the new GUI.

The aim of this work is to develop and validate a testing methodology (and supporting tool), that allows the automatisation of web-based GUI testing, including test case generation, execution, and diagnosis, that is agnostic to the UI technology used to implement the GUI and thus allows for a maximum degree of reuse of test efforts.

PBT and web-based GUI testing

Property-based testing (PBT) is an automatic testing strategy that has lead to a family of powerful automatic testing tools. These are generally tools for test-case generation and execution based on specifications. Usually working as a library, they allow the developer/tester to write down program specifications, using a regular programming language, in the form of properties which should not be violated. From those specifications, the tools automatically generate, run, and check the results of large numbers of random test-cases to see whether the properties fail or not. The use of random inputs in doing so does well in comparison with more systematic black-box testing techniques [14], with a great gain on cost efficiency. All in all, the more advanced PBT tools provide mechanisms to control data distribution as a means to improve effectiveness [12].

The applicability of PBT to GUI testing has already been explored [15]. However, this previous approach required some initial manual exploratory exercising of the GUI by the developer or tester, in order to generate an initial set of interaction traces. Then, from these traces, a model was inferred and used to subsequently produce test cases and execute them (cf. Fig. 1).

In our work, we make a different use of PBT for GUI testing, adapted to the particularities of web-based GUIs. Namely, we focus on two common characteristics to many web-based GUIs:
– *Common patterns of interaction*, such as log-in pages, or add-element to a table-like structure (shopping cart, client list, etc.).
– *Continuous browsing*, in which essentially a user would expect to continuously and seamlessly interact with the web application without encountering any unexpected errors.

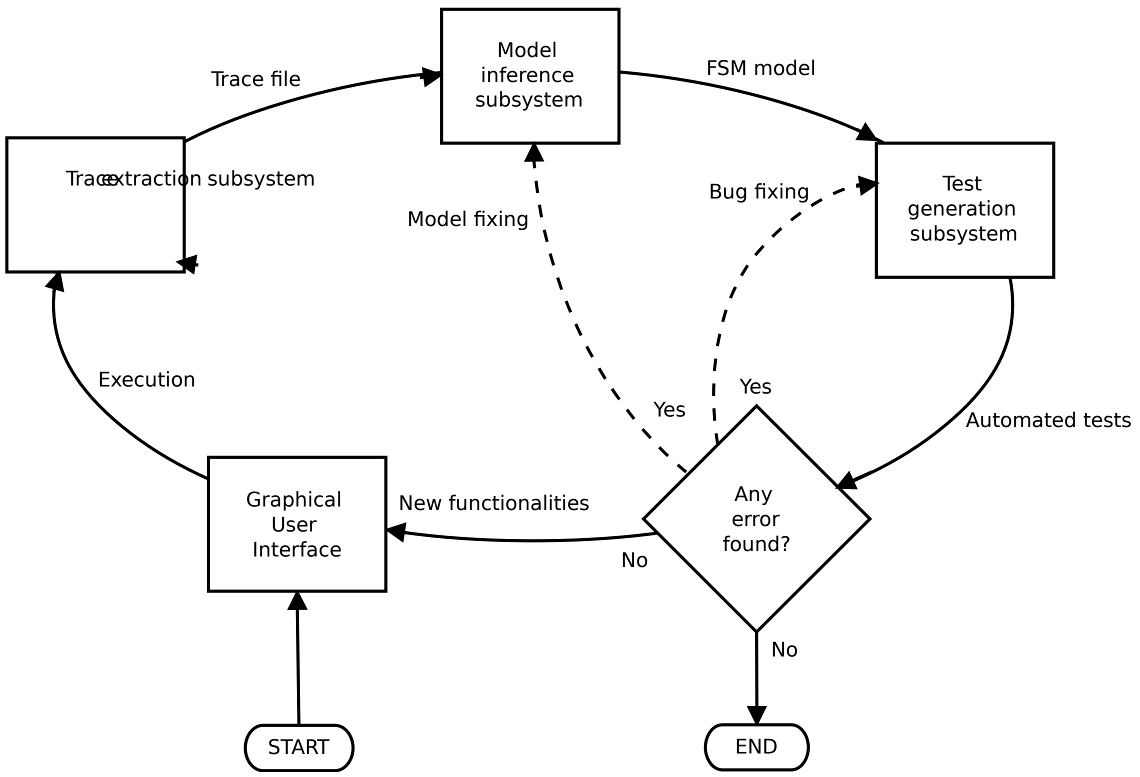


Fig. 1: PBT-based interaction trace & model inference GUI test cycle.

Consequently, our contributions here are twofold. To assist web application developers and testers in designing their test scenarios, and allow them to adopt PBT as a testing strategy, we build a library of reusable general properties, which define common patterns of interaction. An example of such general properties would be the one shown in Source 1.1.

Here, *element_data()* represents a generic data generator to fill in the required fields depending on the element to be added to the current page. From such a property, a PBT tool can instantiate the data generator many times and execute the property body many times, automatically. Should the property body evaluate to false any time, the PBT tool will not only inform of the discrepancy (i.e. possible bug found that is triggered by a certain generated input), but also shrink such input [8] in order to find a smaller, easier to debug example that triggers the error.

The key actions used in the property body, INTERACTION_ON_PAGE... and PAGE_CONTAINS_... , together with

```
prop_add_element_to_table() ->
   ?FORALL(ElementData, element_data(),
      begin
         ResultPage = ?INTERACTION_ON_PAGE_TO_ADD_ELEMENT(ElementData),
         ?PAGE_CONTAINS_ADDED_ELEMENT(ResultPage) == true
      end).
```

Source 1.1: Generic property for adding list elements in a web-based GUI

Algorithm 1 The failure detection algorithm (general model behaviour)

```
Precondition: valid initial $URL$ (web-GUI start page)
Precondition: browsing session $b$ initialized to $URL$
Precondition: maximum test sequence length $T$ randomly selected

function RunTestSequence($b, URL, T$)
    $actions \leftarrow get\_actions(b, URL)$
    $testLength \leftarrow 0$
    while $(testLength \leq T) \wedge (actions \neq \emptyset)$ do
        $action \leftarrow oneof(actions)$                    ▷ random choice
        $URL \leftarrow run\_action(action)$                  ▷ may lead to a different page
        $actions \leftarrow get\_actions(b, URL)$             ▷ may change even if page does not
        $testLength \leftarrow testLength + 1$
    end while
end function

Postcondition: maximum test sequence length $T$ reached
Postcondition: no browsing errors in $b$
Postcondition: valid $URL$
```

many others relevant to web-based GUI testing, are heavily dependent on the underlying technology and frameworks used to build the GUI. Most of them are based on HTML5 and/or JavaScript, but define their own building blocks, use their own naming conventions for the elements to be found on a web page, etc. Consequently, to enable a technology-agnostic definition of PBT properties, we build a general browsing test model, in the form of a general skeleton that defines the key actions, and it is complemented with particular instantiations of specific interactions depending on the specific web technology.

This model defines the general, technology-agnostic property that a PBT tool will attempt to falsify during testing, which is presented in Algorithm 1.

Implementation: the webUI-test library

Once the abstraction of the methodology is in place, the key is to define the actions, which are also present in the patterns of interaction used in specific properties.

To that end, we have developed the webUI-test library, which implements both the general continuous browsing model and the more specific interaction patterns. The general architecture of the tool is presented in Fig. 2, where gray-shadowed elements are the reusable components, while their specialisations are GUI-specific.

To test a given GUI using webUI-test, a WebUIModel specialisation must be implemented, including specific *setup()* and *teardown()* methods, together with a WebUIactions

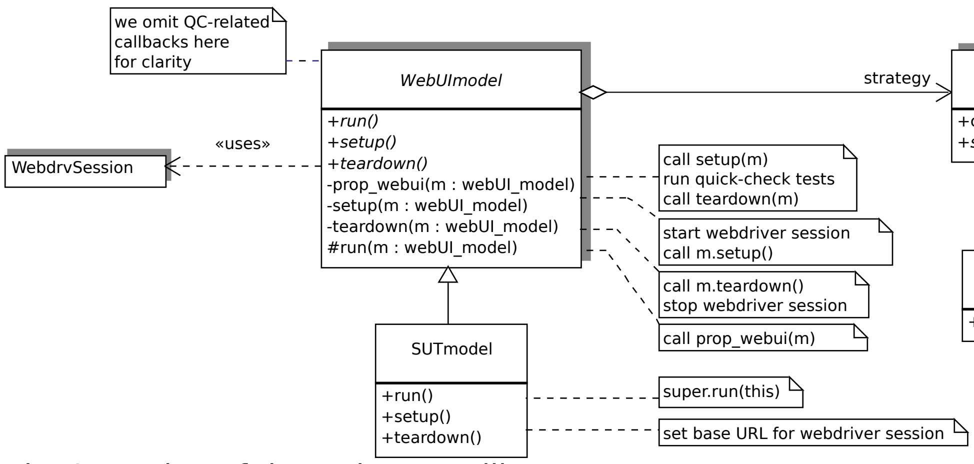


Fig. 2: Design of the webUI-test library.

```
% general browsing
get_actions() ->
  {ok, L} = get_links(),
  {ok, B} = get_buttons(),
  {ok, I} = get_inputs(),
  {ok, [{link, Link}     || Link <- L]   ++
       [{button, Button} || Button <- B] ++
       [{input, Input}   || Input <- I]}.

% framework-specific implementation of actions
get_links() ->
  webui_actions:find_elements_by_type(?XPATH_LINK,   [visible]).

get_buttons() ->
  webui_actions:find_elements_by_type(?XPATH_BUTTON, [visible]).

get_inputs() ->
  webui_actions:find_elements_by_type(?XPATH_INPUT,  [visible]).
```

Source 1.2: Implementation snippet for action detection.

specialisation, including specific ways of retrieving action elements for the GUI under test. Source 1.2 shows an example of interaction elements implementation specific to a XPath-compatible UI framework.

Pilot evaluation: the VoDKATV pilot study

In order to assess the validity of our approach, we apply webUI-test to one of the GUIs in the VoDKATV system. VoDKATV is an IPTV/OTT middleware that provides end-users access to different services on a TV screen, tablet, smartphone, PC, etc., allowing an advanced multi-screen media experience. Architecturally, it is a distributed system composed by several components, which are integrated through web services, for which different web-based GUIs are designed, implemented and deployed.

As many other services, many VoDKATV GUIs present the user with a log-in page in which a valid combination of username

```
setup() ->
  login().

login() ->
  {ok, LoginField}    = webui_actions:find_element_by_id("loginName"),
  {ok, PasswordField} = webui_actions:find_element_by_id("password"),
  {ok, [Submit]}      = webui_actions:find_elements_by_type(?XPATH_INPUT,
                                                      [visible,submit]),
  ok = webui_actions:set_element_value(LoginField,    "vodkatv"),
  ok = webui_actions:set_element_value(PasswordField, "vodkatv"),
  ok = webui_actions:activate_element(Submit).
```

Source 1.3: Implementation of VoDKATV custom setup/teardown.

and password must be provided before proceeding. Once logged in the system, there is virtually no restrictions to the interaction sequences that a user can perform on the web-based GUI. Thus, Source 1.3 shows how a VoDKATV-specific WebUImodel specialisation would include this initialisation step, where the use of the generic WebUIactions encapsulates the use of the corresponding VoDKATV-specific actions definition.

With this pilot, we have been able to use the webUI-test generic model and actions, defining one specific test model and two framework definitions to test the old version and the new version of two functionality-equivalent GUIs that coexist nowadays in production. This has effectively reduced not only the amount of test source code to be written, but also has augmented the extent and systematisation to which both GUIs were being tested.

Discussion

One of the main strengths of our work is that, unlike many others, we follow a purely black-box dynamic approach that does not require any static analysis [4], nor does it require access to the server-side in any way. As a consequence, we do not explicitly offer coverage measures within the framework itself [4], but coverage of the server side is still achievable using whichever coverage tools are suitable. All in all, since our interest is GUI testing, not system testing, server-side coverage is not the most relevant metric, although it is the only way of detecting server-side dead code (i.e. unreachability of server-side functionalities from the GUI). Instead, we evaluate the impact of our approach by analysing test effort reduction, that is to say, reduction in test code LOC, together with percentage of reuse of test code. These metrics are better suited than GUI-coverage metrics precisely because of on-the-fly generation of web pages on web-based GUIs. In fact, coverage of GUI, when it can be partially dynamically generated is an open problem.

Even without accessing the server side, the use of PBT instead of traditional unit testing tools, allows us to exploit, during test generation process, the output of previous interactions [4]. These values are collected dynamically from the web pages that are transversed, which are accessed directly via the browser (using WebDriver [3]), thus maintaining our approach free of any dependencies with any web-based GUI implementation framework [4, 19]. As for test input minimisation upon bug detection, most research focus on value minimisation [9], and in doing so combine symbolic and concrete execution. Stateful PBT is capable of minimising the test sequence itself, eliminating interactions superfluous to reaching the uncovered failure. Also, the symbolic execution is usually another source of binding of research tools to specific languages, frameworks and/or systems, which we avoid. In contrast, the possibility of writing test data generators in a programming language is much more powerful and expressive than inference, and compensates for the obliviousness of black-box input minimisation to the properties of such inputs [9].

Previous work has already pointed out the interest of non-determinism in testing web applications [4], since it manifests important aspects of them. Alternatively, one could profile users to narrow test scope, something that has already been done, in combination with PBT, for non-functional testing [6].

Absence of bias and subjectivity in determining a bug is also a strength of PBT, because of the as is of any work that uses an automated oracle [4]. However, GUI errors such as displaying of the wrong page, or part of the GUI being blocked or overlaid and thus not visible or interactuable because it is blocked or overlaid cannot be detected by oracles that only look for execution or script errors, in contrast with those that directly inspect web-page elements.

Ricca et al. [19] use a static UML model of a generic web application to drive their testing; we instead use a stateful model of web-based GUI interaction for the same purpose. We argue that the static structure of the application itself is not that important for GUI testing. While the first represents a white-box approach, our black-box does not need to differentiate between dynamic or static HTML content, since at the point we interact with it (the browser), it makes no difference. Ricca et al. [19] use this for optimisation purposes, eliminating from the web app static pages not containing forms. We cannot benefit from this, but again we do not need access to the server side internals either, which is definitely an advantage in the area of web systems, which tend to evolve rapidly [19] (and more so their web-based GUIs). More so since few works consider problems related to web site evolution and maintenance [21].

Other tools for automatic testing of dynamic webpages systematically explore all paths to a certain bound [20]. However, they need a human tester to prepopulate user-interaction profiles. Other approaches pre-record traces of user interactions [17]. While eliminating the need for human intervention, and maintaining the relevant random component [4], the use of specific interaction profiles could be added to our approach, as mentioned before [6].

Last but not least, considerating interactions such as reload, going back, etc. seems very meaningful in web-based GUI testing, and has not been done before to the best of our knowledge. In our approach, these interactions can be treated in the same way as any other.

Conclusions

Exploring systematisation and automation possibilities for GUI testing is a very active and promising research area. In this work, we show how we enable the use of PBT in the domain of web-based GUIs in a way that enables test-effort reuse by abstracting out the GUI-framework specific interactions and liberates from test case design using a generic continuous browsing model.

References

1. FITTEST. http://crest.cs.ucl.ac.uk/fittest/ (2010-2013)
2. Sahi: Automation testing tool for web applications. http://sahipro.com/ (2015)
3. Selenium: Web application testing system. http://seleniumhq.org/ (2015)
4. Alshahwan, N., Harman, M.: Automated web application testing using search based software engineering. IEEE/ACM International Conference on Automated Software Engineering. pp. 3–12 (2011)
5. Anderson, R., Srinivasan, S.: E-satisfaction and e-loyalty: A contingency framework. Psychology and Marketing 20(2), 123–138 (2003)
6. Arts, T.: On shrinking randomly generated load tests. ACM SIGPLAN Workshop on Erlang. pp. 25–31. ACM (2014)

7. Arts, T., Castro, L., Hughes, J.: Testing Erlang data types with Quviq QuickCheck. ACM SIGPLAN Workshop on Erlang. ACM Press (2008)
8. Arts, T., Hughes, J., Johansson, J.: Testing telecoms software with Quviq QuickCheck. ACM SIGPLAN Workshop on Erlang (2006)
9. Artzi, S., Kiezun, A., Dolby, J., Tip, F., Dig, D., Paradkar, A., Ernst, M.: Finding bugs in web applications using dynamic test generation and explicit-state model checking. IEEE Transactions on Software Engineering 36(4), 474–494 (2010)
10. Belli, F.: Finite state testing and analysis of graphical user interfaces. International Symposium on Software Reliability Engineering (2001)
11. Brashear, T., Kashyapb, V., Musantec, M., Donthud, N.: A profile of the internet shopper: Evidence from six countries. Journal of Marketing Theory and Practice 17(3), 267–282 (2009)
12. Castro, L.: Advanced management of data integrity: property-based testing for business rules. Journal of Intelligent Information Systems 44(3), 355–380 (2015)
13. Derrick, J., Walkinshaw, N., Arts, T., Benac, C., Cesarini, F., Fredlund, L., Gulias, V., Hughes, J., Thompson, S.: Property-based testing - the ProTest project. Lecture Notes in Computer Science 6286, 250–271 (2010)
14. Hamlet, D.: When only random testing will do. International Workshop on Random Testing. pp. 1–9 (2006)
15. Iglesias, D., Castro, L.: Property-based testing for graphical user interfaces. Journal of Computer and Information Technology 1(3), 60–71 (2011)
16. López, M., Castro, L., Cabrero, D.: Feasibility of property-based testing for time-dependent systems. Lecture Notes in Computer Science, vol. 8112, pp. 527–535 (2013)
17. McAllister, S., Kirda, E., Kruegel, C.: Leveraging user interactions for in-depth testing of web applications. Recent Advances in Intrusion Detection, Lecture Notes in Computer Science, vol. 5230, pp. 191–210 (2008)
18. Nilsson, A., Castro, L., Rivas, S., Arts, T.: Assessing the effects of introducing a new software development process: a methodological description. International Journal on Software Tools for Technology Transfer 17(1), 1–16 (2015)
19. Ricca, F., Tonella, P.: Analysis and testing of web applications. International Conference on Software Engineering. pp. 25–34 (2001)
20. Tanida, H., Prasad, M., Rajan, S., Fujita, M.: Automated system testing of dynamic web applications. Software and Data Technologies, Communications in Computer and Information Science, vol. 303, pp. 181–196 (2013)
21. Warren, P., Boldyreff, C., Munro, M.: The evolution of websites. International Workshop on Program Comprehension. pp. 178–185 (1999)
22. Zhu, Z.: Study on beta testing of web application. International Conference on Computer and Automation Engineering 1, 423–426 (2010)