Model Generation to support Model-based Testing Applied on NASA DAT

--an Experience Report

Christoph Schulze, Mikael Lindvall,
Sigurthor Bjorgvinsson
Fraunhofer Center for Experimental Software Engineering
College Park, Maryland, USA
{cschulze, mlindvall, sbjorgvinsson }@fc-md.umd.edu

Robert Wiegand
NASA Goddard Space Flight Center
Greenbelt, Maryland, USA
robert.e.wiegand@nasa.gov

***Abstract*— Model-based Testing (MBT), where a model of the system under test's (SUT) behavior is used to automatically generate executable test cases, is a promising and versatile testing technology. Nevertheless, adoption of MBT technologies in industry is slow and many testing tasks are performed via manually created executable test cases (i.e. test programs such as JUnit). In order to adopt MBT, testers must learn how to construct models and use these models to generate test cases, which might be a hurdle. An interesting observation in our previous work is that the existing manually created test cases often provided invaluable insights for the manual creation of the testing models of the system. In this paper we present an approach that allows the tester to first create and debug a set of test cases. When the tester is happy with the test cases, the next step is to automatically generate a model from the test cases. The generated model is derived from the test cases, which are actions that the system can perform (e.g. a button clicks) and their expected outputs in form of assert statements (e.g. assert data entered). The model is a Finite State Machine (FSM) model that can be employed with little or no manual changes to generate additional test cases for the SUT. We successfully applied the approach in a feasibility study to the NASA Data Access Toolkit (DAT), which is a web-based GUI. One compelling finding is that the test cases that were generated from the automatically generated models were able to detect issues that were not detected by the original set of manually created test cases. We present the findings from the case study and discuss best practices for incorporating model generation techniques into an existing testing process.**

Keywords— Model-based Testing, Model Generation, State Machines

I. INTRODUCTION

Software testing is an essential task to ensure that the quality of the system under test (SUT) meets the stakeholders' requirements. However, testing is also one of the most time consuming and therefore probably the most expensive aspect of software development [1].In current practice, testers construct test cases manually. Please note that wherever we use the term "test case" in this paper, we refer to executable test cases such as JUnit test programs.

These test cases provide inputs to the SUT while checking that the corresponding outputs provided by the SUT are correct. What exactly constitute input and output varies depending on the nature of the SUT. For GUI testing, the inputs are values entered into input fields, clicks on buttons etc. and the outputs are the displayed values, or the screen that is visible. For non-GUI testing the inputs are typically function calls and the outputs are values returned from these function calls etc. Test cases can either pass or fail. If the test case passes, then the actual output matches the expected. If a test case fails, then the actual output doesn't match the expected. The reason for failing must be investigated to determine if it is due to an issue with the SUT. This testing is functional (a.k.a. behavioral), and thus detect mismatches between actual and expected behavior, but can also detect issues related to performance and capacity.

Model-based Testing (MBT) is a valuable and versatile testing approach for a wide range of software systems. There are many variants of MBT [2]. The variant discussed in this paper uses state machine models to describe the behavior of the SUT. These models are typically created after the system was built and are only used for testing. The main idea is to use these models to automatically generate test cases. Thus instead of manually creating and maintaining a set of test cases, one test case at a time, the tester creates and maintains one or more models and generates test cases from each model. The test cases that are automatically generated can often not be distinguished from the ones that are manually created, since they contain the same actions and asserts. However, the generated test cases are typically longer than the manually created ones.

One advantage this Model Based Test Case Generation (MBTCG) has over manual test case creation is the ability to automatically generate large sets of different test cases. In addition, these models help stakeholders understand the scope of the testing, what is being tested, and what is not being tested. Since these models capture the desired behavior of the SUT they can be used as system documentation. Furthermore the models can be used to generate test cases for other testing tasks, such as stress testing.

In spite of the benefits of MBTCG, the rate of adaption of MBT in industry, especially in non-safety critical applications, is still slow [3]. One obstacle that stands in the way of a more rapid adaption is that testers and software engineers often are not used to creating models. However, they do know how to manually create test cases [4]. In our previous work on MBT

[5][6], we took advantage of these test cases by first manually analyzing them. This analysis provided invaluable insights into how the SUT is supposed to be used, which cannot always be easily determined from documentation. Based on the analysis of the existing test cases we created a model that initially contained exactly the same information as the set of existing test cases. This initial model was then used to automatically generate new test cases. In case the existing test cases did not test all possible combinations of a certain situation, the model was elaborated until it was complete in relation to the particular testing goal.

In this paper we present an industrial case study of a new approach that addresses several problems related to manually creating models. The new approach automatically analyzes existing test cases and automatically transforms the extracted test case information into an initial behavioral model. The model generation algorithm is based on a heuristic state merging approach that is intended to generate an initial model based on a set of test cases. Due to the heuristic nature this model is not always a perfect match of the SUT and requires inspection and sometimes some manual rework. The model is a finite state machine (FSM) where states and transitions represent the assertions and actions in the test cases.

The approach addresses two of the most time consuming tasks of MBT: the model creation and the creation of the mapping between the abstract model actions and the concrete actions that execute against the SUT. In a previous study these tasks consumed roughly 84% of the overall effort [7].

In this case study we show that the new approach is feasible to use on industrial systems because with reasonable effort (48 hours including learning the new approach) a tester was able to generate models from various sets of test cases, which he then used to generate additional test cases from the model. In addition, these new test cases were able to find defects in the system that the original set were not able to.

As part of the case study the tester created several test suites consisting of manually recorded Selenium test cases for three different features of the web interface of NASA's Data Access Toolkit (DAT) system. For each of these test suites the tester automatically generated one or more testing models.

II. BACKGROUND

A. Model Based Testing

MBT employs models to describe the SUT and derives test cases from those models. There are a variety of MBT approaches that usually differ in the modeling notation and the level of tool support. The general structure of the approaches for test case generation from models is typically similar to each other.

In this work we employ finite state machines (FSMs) to describe the expected behavior of the SUT. FSM based testing models describe inputs or stimuli of the system and contains the expected behavior to these stimuli. Figure 1 shows a simplified example model that is used to enter to and remove data from a web system. FSMs consists of transitions that represent stimuli to the SUT (e.g. enter data to the system) and states, which are abstract representations of the state of the SUT (e.g. the data was entered successfully).

An *abstract* test case is a traversal by an *MBT tool* through the FSM from the *start* state until a chosen *stopping criteria* is met. The MBT tool used in this study is based on *Graphwalker* [8]. Graphwalker offers several *stopping criteria*. For example the number of steps to take (e.g. the test case is completely generated when 100 steps – actions and asserts – have been added to the generated test case) or the state/transition coverage to reach (e.g. the test case is completely generated when 80% of the states or transitions have been visited). An abstract test case, is a list of the state and transitions that were encountered during the traversal. A *concrete* (a.k.a. executable) test case is produced by replacing all state and transitions names in the abstract test case for the matching concrete executable code snippets for the action or asserts. Assigning executable code snippets to each action and assert used in the model is called *mapping*.

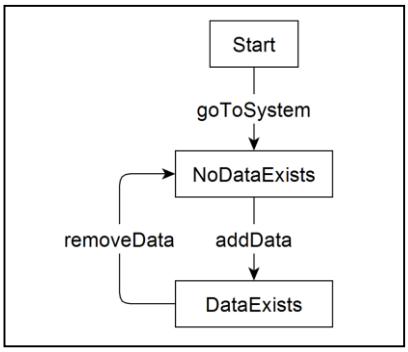

Figure 1: Test Model Example to add data into a web system

B. The System Under Test

The Data Access Toolkit (DAT) is an archive, access and analysis system for NASA mission data. DAT provides an advanced query interface for mission analysts, mission managers, and the flight operation team to search and mine the available data. Users can query the system using a web-based query interface based on representational state transfer (REST) or through a web-based graphical user interface (GUI) to query the underlying PostgreSQL database, which stores metadata.

In our previous work [9] we described how we tested the REST interface of DAT using metamorphic testing [10]. We refer to our previous work for additional details about the SUT. It is important to mention that the version of the DAT system that we tested has relatively high quality both due the fact that it has been around for some time (the first build was produced in 2012) and because of the DAT team's testing efforts. The testing Fraunhofer conducted as a basis for this paper was regression testing and therefore a limited number of detected issues are expected. Specifically we discuss how a Fraunhofer tester tested DAT through the GUI. The features that were tested are: "Request Data", "Manage Repository" and "Manage Templates". The tester used a set of DAT use cases provided by the DAT team to understand the workflow and planned the manual test cases accordingly.

The "Request Data" page is the data access page of DAT. This page allows the user to insert values into input fields that together form a query for data. When this query is submitted, the DAT system responds with the resulting data. The request data page implements a rich query language and offers several options and components for formulating queries. Testing that all

possible request queries return correct data is therefore a daunting task that require a large set of test cases.

The "Manage Repository" page allows the user can navigate a tree-like structure of the DAT repository. The tree-like structure of DAT consists of Repositories, Missions, Namespaces and Archives listed in the highest to lowest in the hierarchy. Thus, an instance of DAT can have many repositories where each repository can have several missions. Each mission can have several namespaces, and each namespace can have several archives.

Using the manage repository feature, repositories, missions, namespaces and archives can be created, edited and deleted. The data itself is stored in archives and can be configured to handle many different data types.

The "Manage Templates" page manages the templates of the system. Templates are used when requesting data and specify how the user wants the output to be reported and formatted. On this page the user can create, edit and delete templates. A template has a name, type, folder and a body encoded as XML. The name is the name of the template, the type declares how the data will be represented, a report or plot. The folder says what folder the templates should be saved in and the body is the declaration of the template in XML format.

It should be mentioned that the DAT team uses an agile software development approach. This development approach requires a testing approach that supports frequent changes and delivers testing results within short time. Therefore it was important to develop a testing approach that supports such short turn arounds.

C. Selenium

Selenium [11] is a browser automation tool that is often used as a testing framework for web applications. Selenium IDE is a Firefox extension that allows recording, execution and editing of selenium scripts. Selenium scripts are stored in the XML-based Selenese language. Selenium scripts contain high level actions such as clicking on an element present on the web page, or asserting the existence of an element on the web page.

Selenium scripts use Java Script to navigate and manipulate the Document Object Model (DOM) of the web application. A Selenium script can be used to simulate how a user uses the system to carry out a certain task. Through the use of various assertions, the script can automatically determine if the expected output is provided by the SUT. Thus such a script is a test case that can be used to automate testing of the GUI of a web-based system.

A Selenese command is a triple: <command, target, arguments>. The first element contains the name of the command (e.g. type), the second element contains the target of the command (e.g. an input field), and the last element contains the arguments for the command (e.g. the text that should be entered into the input field). Java script commands can be added to the test scripts, e.g. to create random names for input fields.

III. Model Generation from Test Cases

We developed two software tools related to MBT: one tool that allows the tester to generate models from Selenese test cases (The Fraunhofer Model Generation Tool), and one tool that generates test cases from such a model (the Fraunhofer MBTCG Tool). The MBTCG tool provides a GUI to the Graphwalker Model Based Testing tool, and adds visual debugging capabilities that help the user to identify issues in the models.

The models can be visualized and edited using the Yed graph editor [12]. In this section we will describe the algorithm that transforms test cases to FSM models. We will also describe the workflow of the overall approach: 1) creating the manual test cases, 2) automatically creating models from those test cases, 3) automatically generate test cases from the models, and 4) automatically executing large sets of generated test cases.

A. Model Generation Algorithm

The algorithm transforms a set of Selenese test cases into a FSM testing model by employing a set of heuristic transformation rules, based on observations of our manually created testing models and test cases. Our FSM testing model is a quintuple $(\sum, S, s_0, \delta, F)$, where:

- $\sum$ is the input alphabet. The input alphabet is determined by all non-assert actions in a test case. Two Selenese actions are considered equal if their command, target and arguments are equal.
- S is a finite, non-empty set of states. The states are determined by the assert statements. Two states are considered equal if their corresponding assert statements are equal. The algorithm adds helper states in case of two or more consecutive non-assert actions.
- s_0 is the initial state of the FSM and an element of S. The label of the initial state is *Start*.
- δ is a state-transition function: $\delta: S \times \sum \rightarrow S$
- F is a (possibly empty) set of final states. In the case of our testing models final states are states that do not have any outgoing transitions.

We created a set of rules that transform selenium actions into transitions and states. The rules determine when to merge states and when to add helper states in case of two or more consecutive non-assert statements. The transformation rules are encoded as follows (see Figure 2).

The merging approach is implemented as a two-pass algorithm. In the first pass (lines 1 to 6) it traverses each test case and merges consecutive assert statements together (line 4). In case of consecutive non-asserts helper states are interleaved into the test case (line 6).

In the second pass the state machine is constructed from the modified test cases. The algorithm starts by creating the state machine and the start state s0 (lines 7-8) and then iterates through each test case in the test suite. It then resets the *lastVisited* helper variable to the start state (line 10) and then iterates through the actions of the modified test case. In lines 12-13, the algorithm checks if the action is an assert-action (future

state) and if that state already exists in the state machine. If it does not exists it will be created in the next line.

```
//First Pass
1  for each TestCase in TestSuite
2    for each action in TestCase
3      if consecutive-assert-actions
4        merge assert-actions
5      if consecutive-non-assert-actions
6        insert helper state between non-assert-actions

//Second Pass, on transformed test cases
7  create state-machine
8  create state s0
9  for each TestCase in TestSuite
10   lastVisited = s0 //always start in start state
11   for each action in TestCase
12     if action is assert-action and not in state machine
13       create state (action)
14     if action is non-assert-action
15       if action already exists in lastVisited state
16         if existing.target equals action.target
17           lastVisited = existingAction.target
18           continue
19         else
20           Error: Indication of possible non-determinism
21           continue
22       else
23         create transition(action)
24         transition.source = lastVisited
25         transition.target = State(nextAssertAction)
26         lastVisited = transition.target
```

Figure 2: Pseudo Code of the two pass model generation approach

Lines 14-26 handle the non-assert actions (future transitions). It first checks if the action already exists as a transition in the current state (line 15). If it does it checks if the target of the transition is the same as the next state in the test case (line 16). If it is then the *lastVisited* state will be set to this next state and the algorithm continues with the next action in the test case (lines 17-18).

If the target state of the existing transaction and the current test case action are not the same the algorithm will throw an error message to the user (line 20). In this case the approach would produce a non-deterministic state machine. This is an indication of an error in the test cases that can appear through manual modification of the recorded selenium test case. This case has been introduced into the algorithm as a sanity check for the test cases.

In case that no correspondent transition exist in the current state, a new transition will be created from the current action and connected to the *lastVisited* state and the next state in the test case. The *lastVisited* state will be modified to reflect the new state.

B. Workflow Overview

In this section we will describe the workflow of our approach, from the creation of manual test cases to model generation to test case generation. Instead of manually creating a model, which is the common manner to conduct MBT, the new approach starts with constructions of example system usages. That is, the tester creates a set of Selenese test cases by using the record feature in the Selenium IDE. Each test case can be replayed in the Selenium IDE and potential defects in the test case can be addressed. Then the tester automatically generates a model from these test cases, which is used to generate more test cases. The workflow is supported by the two tools described above. The tools manages the model generation, model creation and test generation. The workflow of the approach is as follows:

1. The tester analyzes the features of the website and decides which test cases to create. The division is either done by web-page or by use-case or a mix of both.
2. The tester records a set of test cases for each feature using the Selenium IDE.
3. Once the tester debugged the test cases, the tester uses the tool to generate a model for each feature.
4. The tester visually inspects the model and fixes issues, or adds missing behaviors.
5. The tester generates test cases from each model.
6. The tester executes the test suite and analyzes the results of the execution. Failed test cases are especially analyzed.

After finishing the separate features of the system, the tester now combines all test cases into a larger model of the system by repeating steps 3-5 for all test cases to generate system level test cases.

The tester can use different test generation strategies and block out parts of the model in order to guide the test generation into specific parts of the model. This can be useful for stress testing the system by creating certain objects over and over. E.g. in the example the tester could block off the *set template type report* transition in order to make sure that only templates of the *type plot* are created.

IV. Case Study

The tester applied the approach described in the previous section to test the features of the DAT web-interface which consist of the web pages *Request Data*, *Manage Templates* and *Manage Repository*. This section will describe the features that we tested, the manually created test cases and the models generated from these test cases. Furthermore we will provide effort data for applying the approach

A. The Tester

A student intern at Fraunhofer created the test cases manually, generated the model, generated test cases, executed the test cases and analyzed the test results. Before starting his internship, he had no background in Selenium IDE or Selenese, or MBT. Thanks to his internship at Fraunhofer, he had about 2 months of experience with modeling and model based testing.

B. An example of applying the approach

In this example we explain steps 1-5 from the overview in the previous section with a simplified example. This will help understanding the algorithm, the workflow and the benefits of the approach. As an example we chose the *Manage Template* feature from the DAT system. The full description and size of the test cases the corresponding models and generated tests for the feature can be found in the case study section. We use a simplified example due to size constraints. In the case of the DAT system dividing up the features was straight forward since

each feature had its own web page and the features are relatively independent of each other.

For this example, the tester recorded three test cases (see Figure 3) that test different variations of the possible inputs and options for creating a template. The tester replayed them using the Selenium IDE to detect and remove any potential issues. In some cases the tester also manually modified the test cases by adding commands that were not automatically inserted during the Selenium recording. For example, storing a value from the website in a variable that will be used in a later assertion on other pages. Adding such a command is supported by Selenium IDE and requires a right-click on the value to store in addition to the variable name.

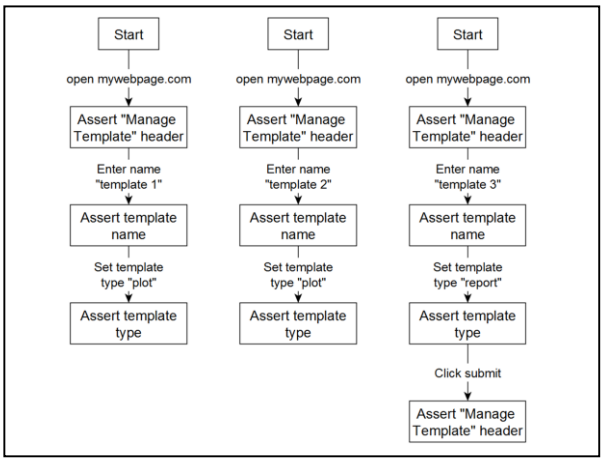

Figure 3: Three simplified test cases for the manage template feature

When the manual test cases have been debugged, they are loaded into the model generation tool. In this example, the model in Figure 4 was created. The three test cases are merged in three merging points namely: the *Assert manage template header*, *Assert template name* and *Assert template type* action. Thanks to the merging, the behaviors can now be interleaved to create 6 instead of three different inputs and can be repeated over and over again, thereby adding many more potential behaviors.

Before generating test cases from the generated model the tester first inspects it. Since the recording of test cases is a manual task there is the potential for errors. An example of an issue caused by such an error is a wrongly recorded assert statement, which can manifest in unwanted paths or states in the model, or states that are not fully connected with the rest of the model. The model can be modified directly by adding, removing or modifying states and transitions. However, we have found that the best way is to instead edit the test case accordingly and regenerate the model. In this way, the model is never manually edited. The tester can also add, modify or remove test cases and quickly regenerate the model.

The tester who was studied in the case study explained that after some practice he developed a habit to plan the test cases based on the model he wanted to generate. Thus, he from the beginning had an image in his head about what the model should look like, which paths would be logical and what states should be created. He further explained that when he sees a path that he does not expect or a state that leads nowhere the tester knows that there is an issue with the model.

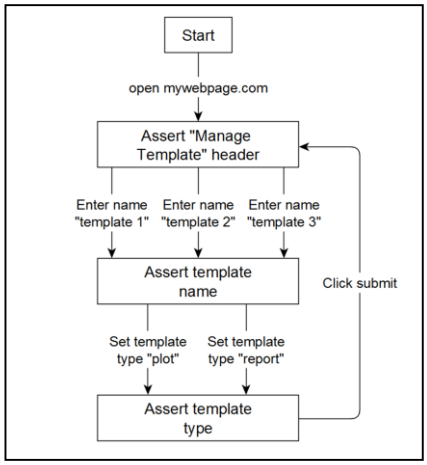

Figure 4: The Model of the combined simplified test cases

After the tester has verified the model he generates additional test cases from it and executes them against the system. He then inspects failing test cases. A test case can fail because of two reasons. Either the error is a true positive, which means that the test case failed because of a bug in the SUT, or the error is a false positive which means that the test case failed because of an issue in the model or in the manual test cases. The tester must inspect the generated test cases that failed and their execution in order to determine the nature of the issue. If the model or test case is incorrect, then the tester corrects them. If the test case failed because of an issue in the system, then the tester documents the detected issue including how to reproduce it, and reports it to the DAT development team. We will now describe how we tested the features of DAT using this approach.

C. The Request Data Feature

On the Request Data page the user inputs the parameters for a request to the DAT system. The page offers many different options and input fields. The user can enter a start date and time and then has the option to choose either an end date and time or to enter a duration (in days, hours, minutes, or seconds). Additionally the user can choose the different datasets to be searched. Testing this feature manually is difficult because of the many ways a query can be formulated and thus a large number of manual test cases would be needed to cover all possible options.

In the next step the user can add mnemonics. The mnemonics are the identifiers for the different sensory data that are stored in the system. In order to pick a mnemonic the user has to choose the mission to select mnemonics from. In the next step the user has to decide which properties (there are 14 different properties to select from) the user wants the system to return for each mnemonic. There is no known limit to the number of mnemonics that can be added.

To test this feature, the tester created 19 manual Selenese test cases with an average length of 15 instructions and generated a model from them. The generated model has 55 states and 66 transitions. 32 of the states were based on assertions while the other 23 states were helper states introduced by the model

generation algorithm. From this model, the tester created 100 test cases using a random traversal strategy. The generated tests had an average of 39 selenium commands. As indicated above, these test cases are executable and require no editing. The tester therefore immediately loaded the entire test suite into Selenium IDE, which executed the test cases automatically.

D. The Manage Templates Feature

The Manage Templates page allows the user to manage reporting and plotting templates. On this page the user can create, edit and delete templates. When creating new templates there are 2 input boxes and 2 dropdown menus (with two and 5 options to choose from in the dropdown menus respectively).0

The tester created 5 manual test cases for the manage templates feature with an average length of 38 instructions. The generated model has 22 states and 27 transitions. 12 of the states are based on assertions and 10 are helper states. From the generated model the tester created 100 test cases using a random traversal strategy. The generated tests had an average of 35 selenium commands.

E. The Manage Repository Feature

On the Manage Repositories page the user can manage repositories, missions, namespaces and archives. Each of these items has a name and a description. The user can create, edit and delete repositories, missions, namespaces and archives.

The tester created 5 manual test cases for the Manage Repository feature that had an average of 27 instructions. The generated model had 28 states and 32 transitions. 6 of the states were based on asserts the other 22 states were helper states. For the test generation of this feature the tester used the blocking property of the MBT approach. This allowed the tester to focus the test generation on certain parts of the model without directly editing the model (i.e. deletion and rerouting of transitions was not necessary thanks to blocking). He did this by blocking certain transitions, which means that during test case generation, these transitions were not available. By blocking certain transitions the tester made sure that for example only archives could be created to study how the system handles the creation of a large number of archives.

The tester generated the following test cases from this model:

- A set of 10 test cases that cover the whole model and create repositories, missions, namespaces and archives. The test cases had an average lengths of 377 commands.
- Two test cases that only create one repository, but several missions, namespaces and archives. The test cases had an average of 1867 commands.
- One test case that creates one repository and one mission but several namespaces archives. The test case had a length of 1397 commands.
- One test case that creates one repository, one mission and one namespace but several archives. The test case had a length of 1864 commands.
- One test case that only create repositories. The test case had 1399 commands and created a large number of repositories.
- One test case that creates one repository and several missions. The test case had 1608 commands.
- One test case that creates one repository, one mission and several namespaces. The test case contained 1556 commands.

When the tester was creating the repository model he wanted to be able to create many entries in the same test case but was running into issues with naming because all names were static. The tester was able to find a way around that by using JavaScript code in the manual test case where he used the StoreEval to store a function in a variable. The tester stored a pseudo random JavaScript function that the test case always calls when creating a new template. The function creates a unique string throughout the run of the test case thus avoiding potential name conflicts. The function needed to be a pseudo random function because if a generated test case would fail, the tester needed to be able to run the exact same test case to be able to properly debug the issue.

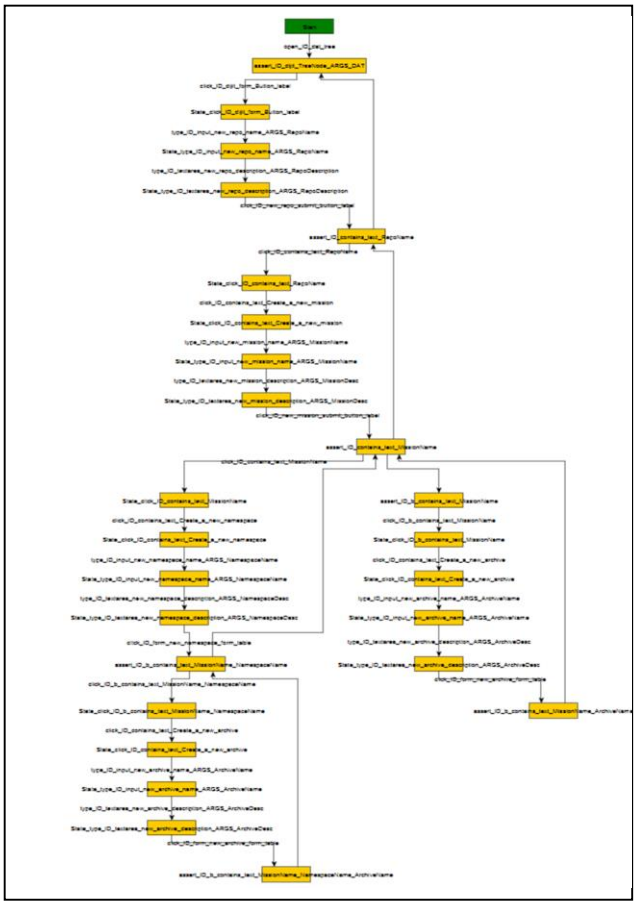

Figure 5: Generated Model for the Manage Repository Feature. The model has 28 states and 32 transitions.

F. Effort

The effort for the tester to learn how to create the test cases and transform them into models was about 8 hours. The effort for the tester to create the three suites of test cases that were input to model generation was 4 hours (manage repository), 6 hours (manage templates) and 12 hours (request data) depending on the model. The effort seems to be proportional to the complexity of the feature.

Executing 100 test cases with an average of 39 command each takes about 50 minutes. Each command takes about 0.77 seconds but this can be controlled via the speed setting in the Selenium IDE. The maximum speed is however often not feasible to use since the highest speeds often cause test cases to fail due to page loading issues.

It took about 12 hours to debug and run samples of the test cases. It takes about 16 hours to run all test cases, however, it should be noted that this is computer time so the effort for the tester is negligible.

The analysis of the failed test cases took the tester about 6 hours. This includes inspection of the failed tests and documenting them for the DAT team. In total, creating the test cases, generating models and test cases, running the test cases, and analyzing the failed test cases took about a week plus one day for learning (i.e. 48 hours).

G. Issues applying the approach

The tester observed two false positives in the generated test cases and investigated their causes. The first issue appeared in test cases that were generated from the Manage Repository model. The reason was that the manual test cases included a wait action after the creation of each repository, mission, namespace, or archive. This was done since the website reloads after the submit button is pressed and the tester has to wait since otherwise the next assert statement would try to assert for elements that have not been loaded yet. The wait time in the manual test cases was set to 1500ms, but after creation of several entries in the system the reloading took more and more time and the test cases started failing. The tester increased the wait time in the manual test cases and recreated the model, which fixed this issue.

The second issue the tester encountered was caused by a dynamic identifier of an element that would change due to reloading the page. As mentioned in the previous section, Selenium tries to choose the best locator for an object but sometimes the best locator turns out to be a dynamic identifier, which changes between runs or by refresh of the web page. We also encountered such behavior in a previous case study [7] and therefore the test cases fail at that point. To address this issue the tester had to change the type of locator of the recorded manual test. The tester had to choose a higher element in the DOM that was static and then used the xpath (i.e. a Selenium feature used to locate elements in an HTML document) to find the correct element in relation to the static element. In addition the tester also used the "contains" function and the "last" function in the xpath to navigate to the correct element.

H. Detected Issues in the System Under Test

All three models generated test cases that detected issues that were previously unknown by the DAT team. These issues were not detected when the manually created test cases were executed. These issues are:

1) Performance issue with representing large amount of data in the browser when requesting a table of data.

This issue was detected on the "Request Data" page, which occurred when the tester was running test cases from the Request Data Model. The reason was that a generated test case had a large time range. The test cases included a large number of data points because of the large time range, which led to degraded performance of the GUI. Eventually, this made the test case fail. DAT (the browser actually) then returned a non-user friendly message that the script was "no longer responsive".

2) Data is retained in fields when creating multiple templates but not in other cases.

This issue was detected when a test case attempted to assert a dropdown menu on the "Manage Template" page with test cases generated by Manage Template Model. The reason was that when a template is created and the type is set, the type was always identical to the previously created template. Upon further investigation the tester noticed that values were retained in an inconsistent way. The issue was that when the site was reloaded, the information of the last created template was not retained, but when two consecutive templates are created the information was retained. Thus, this behavior is inconsistent as to whether the data is supposed to be retained or not.

3) Templates are overwritten without any warning.

This issue was detected on the "Manage Templates" page with tests generated from the Manage Template Model. When the test case asserts that the correct type of the template has been created, the type was sometimes incorrect. This was because only the XML of the template was being overwritten but not the type of the template. Let us assume there is an existing template with the name "tempTemplate", with the type "report" and the xml "first xml". If the test case creates another template with the same name "tempTemplate", but with the type "plot" and the xml "second xml", an inconsistency will occur. The reason is that this will lead to the template having the same name "tempTemplate", the type "report" and the xml "second xml". Using the updated "tempTemplate" will return an error due to this issue. This issue was noticed before the tester created a pseudo random function to create different named templates.

4) Hidden view limit of archives.

This issue was detected on the "Manage Repository" page with test cases generated from the Manage Repository model where test case created a large number of archives. The issue is that there is an undisclosed limit on how many archives the system can display in the hierarchy. The limit turned out to be 100 visible archives. If there are more than 100 archives in the system, they are not displayed and the hidden archives are impossible to reach. However, the user can still continue creating more archives although this limit has been reached. This issue was detected because the generated tests failed at exactly 100 archives.

V. Discussion

We have presented an approach where models are generated from test cases instead of being manually created. We will now discuss strengths and limitations of the approach as well as some other topics related to using it.

A. Strenghts of the Approach

Automated model generation – Since the new approach can generate a model from existing test cases, it overcomes the hurdles related to creating models for MBT.

Automated mapping - Since the model contains *all* information provided in the Selenese test case, the mapping problem is also addressed. The mapping problem means that for each transition and state in the model, the tester must assign executable statements in the form of selenium commands and assertions, which can be both tedious and error prone. The new approach automatically copies all executable statements into the model and thus minimizes the need for manual mapping.

Reduces the necessary skill level and facilitates learning MBT – Since the new approach reduces the need for mapping, which typically is somewhat difficult as well as time consuming, the necessary skill level is reduced. Since the new approach is based on test cases, which testers know how to create, and these test cases represent examples of how the SUT it used (test cases), and since the tool shows what the corresponding model looks like, it is easier to understand and learn MBT.

Well prepared for regression testing of the next version of DAT – Since the DAT team uses an agile development approach it is important that the testing approach can support quick turnarounds. With this approach we believe that this is the case and that a new DAT version of the GUI can be tested within one day since it takes 16 hours of unsupervised computer time to run all test cases. In case the GUI has changed significantly, we expect that creating new test cases and corresponding models will add between 8 and 16 hours to the effort.

B. Limitations of the Approach

Some limitations are due to the fact that we use MBT for test case generation, which has some well-known limitations. E.g. sometimes it was difficult to identify the root cause of an issue due to the fact that the generated test cases were long and are therefore difficult to comprehend. Also since there can be similarities in the generated test cases the same issue can often manifest itself in several test cases. E.g. 10 failing test cases can have the same root cause but this is usually only clear after they have all been inspected. Just because MBT can generate a large number of test cases does not mean that it is always helpful to do so. It is often better to guide the test generation using blocked commands and test different scenarios, which the tester did in a few instances.

Another limitation with the approach is that we have experienced that more advanced testing requires extended FSMs (EFSM), which allows guards and state variables, but the tool currently generates FSMs. The Fraunhofer Test Case Generation tool supports EFSM, thus a manual step is required to turn the initial models into EFSMs as necessary.

C. Does this testing approach replace regular MBT?

We think this approach complements regular MBT because it provides a quick way to create initial models. In addition, it provides an excellent way of teaching MBT to testers who are unfamiliar with MBT. For example, the authors have already successfully used this approach in a graduate testing class to teach MBT to software engineering students.

D. Does practice matter?

Practice has a large influence on the approach, as expected, since the quality of the manually recorded test cases directly impacts the generated models. Furthermore, the tester mentioned that after some practice he already "knew" what the resulting model would look like when he recorded the test cases for it. This probably improved his ability to create test cases that would result in correct model and to inspect and debug the generated models. Thus practice made him more efficient.

E. Does this approach apply to non-web-based system?

In its current state the approach is tied to Selenium and therefore to testing of web-based systems. However, web-based systems are very similar to other GUI based systems and we see no reason why the approach could not be extended to other GUI based systems. The one restriction we place here is that for the approach to work well for testing GUIs of non-web-based system, a tool similar to Selenium IDE should be available for that GUI.

F. Comparing the models to manual MBT

We compared the generated models in this cases study to the manually created models in our previous case study [7]. One observation is that the generated models are on a lower level of abstraction. The manually created model in previous studies have no direct reference to the SUT and instead used abstract actions. The manually created model used for example an abstract command called *submitdata* to enter data into the system, whereas the automatically generated model is more concrete and for example has a reference to the id of the button directly in its label. The effect is that the generated models are slightly more difficult to understand due to the fact that they contain more details.

Another observation is that the labels in generated models are much longer than the labels in the manually created models. The reason is that the generated labels contain nearly all the information from the Selenese commands. The effect is that the readability of the generated labels is less than the manually created labels.

Another observed difference is the lack of sub models in the generated models. In the manually created models, the tester typically groups similar actions into sub models, which introduces a hierarchy to the model that is helpful for model comprehension. Currently the generated models are flat. The effect is that larger models are more difficult to navigate and to understand.

G. Proposed Improvements

It is important that the generated models are easy to understand and maintain, which is something that can be improved. We were able to group certain commands together. E.g. selecting the frame of a web page was always grouped

together with the next action in the test case, which is usually a click action. This made the models smaller and easier to manage and understand, but in order to be more similar to the manually created models we want to evaluate ways to automatically introduce abstractions.

We are especially researching ways to make labels of states and transitions more concise. Furthermore, we want to find ways to automatically create sub models during the model generation process. We believe that one way to achieve this is by rerunning the test cases and automatically collecting additional execution data (e.g. DOM models like in [13], page names). This data could then be used to automatically refine and validate the model. One strategy is to automatically group all states and transitions that reside at a specific URL or all states and transactions that belong to a certain a page with a specific title.

Analyzing the test failures is a time consuming task since at the moment it involves rerunning the failed test to observe the test failure. In order to reduce the debugging and test case analysis time we will automatically add Selenese code to each generated state that will take a screenshot of the webpage and log the state name and time.

VI. Threats to Validity

The threats to validity discussion is based on the model by Wohlin et al [14]. They define four classes of threats to validity, namely threats to internal, external, construct, and conclusion validity.

A. Threats to internal validity

Threats to internal validity are caused by factors that were not considered but might have influenced the results of the case study.

The results of this study show that the presented model generation approach was able to detect defects in the SUT. However, we have to consider the possibility that the tester who conducted the case study knew the issues before conducting the study and he could have therefore tailored the test cases and resulting models in a way that would make sure that the defects get detected. If that is the case the results could not be attributed to the approach but to the knowledge of the test engineer.

We believe this risk to be absent from our case study for the following reasons. Three of the identified defects were not known to the tester when this study started. The tester had encountered an issue similar to the fourth detected issue (retained template values). However, this issue (or none of the other issues) was not detected by the manually created test cases.

B. Threats to external validity

Threats to external validity are concerned with whether we can generalize the results outside the scope of our study.

We have to address different threats here. The first one is that the study was only performed by one tester and secondly the tester only tested one system. We are aware of this limitation. This is an initial feasibility study for our new approach and we plan to compare the results from this study with follow up studies that will have more testers, different systems and also different versions of systems.

Another threat is concerned with the knowledge of the tester. It is possible that another tester might not have been able to use the approach as effectively and therefore would have taken more time and/or might not have detected the same issues. And although it is easy to generate a large number of test cases with MBTCG in some cases the tester has to block certain parts of the model in order to generate test cases for a specific scenario. This requires intuition and training. We addressed this threat by compiling a comprehensive tutorial for our test engineers that covers regular manual MBT (e.g. how to construct models for a system and how to generate effective test cases for different testing goals). We believe that this tutorial will lessen the variance between different testers. However, a natural variance between individuals is always given in these types of studies, no matter what kind of training is given to them.

The approach was designed specifically for Selenese and therefore only for web-based system. However, if there are similar record playback tools with a similar feature set for other types of GUI systems, we believe this approach would be applicable to them as well.

C. Threats to construct validity

Threats to construct validity assess if the correct measurements were used in the case study.

For this study we used direct measures such as the effort in terms of the number of hours spent on a certain task, the number of issues detected, and the size of the test cases and the generated models. We did not use derived or subjective measurements. The measures are therefore good indicators for comparison with other approaches and future case studies. The tester was given instructions to log the time that he worked on the different tasks so that the effort would be reasonably accurate.

D. Threats to conclusion validity

Threats to conclusion validity cover issues that affect the ability to draw the right conclusions from a case study.

Our conclusions are that the approach is feasible to use on industrial systems similar to NASA DAT in the described context. We currently do not see threats to that conclusion, mainly due to the nature of a case study like this one where no comparison to a control group is done. For such comparisons, controlled experiments are required.

VII. Related Work

Torens et al [15] present an approach that generates models from existing test cases of a train control system. The approach employs pre/post condition in the test cases that are written in the Object Constraint Language (OCL) to generate models. The usage of pre/post conditions is similar to our usage of assert statements to create the models. The paper was a feasibility study of the model generation approach and describes the steps to obtain a model but they offer no effort data nor do they discuss its application to test a real system.

There are three other techniques that are frequently used for model generation. One type of techniques are based on Angluin's L*algorithm [16]. These approaches use a *learner* that knows all possible input and output actions of the system and a *teacher* that can for a given sequence of input actions

provide the correct output action. In our context the teacher would be the SUT and the learner would create model hypothesis and compare them to the SUT until it has found a model that corresponds to the behaviors of the SUT. Tretmans [1] discusses how learning algorithms can be applied in the context of MBT. In our approach we do not learn a model of the system but construct one from a set of test cases with heuristic rules.

Another frequently used approach for model inference is based on the k-tail algorithm [17]. The k-tail algorithm creates candidate models by observing execution traces and employing heuristics to merge different executions together. Although k-tail based algorithms can be applied on test case sequences often there are not enough manual test cases to create an accurate model and often system logs are analyzed since they offer more example behaviors.

Lastly another common approach to generate models are based on observing the state of the system, or an abstract representation thereof. Marchetto et al [13] present an approach that abstracts the DOM model of a web application into a state model and then generates additional test cases. They applied it on an open source to-do list manager application with seeded defects. We applied our approach on an industrial web based system. The techniques in these three categories are very powerful and can test many behaviors of the system. However, oracles often have to be added manually after the fact, which is not always trivial due to the size of the models. Our approach leverages the oracles from the test cases and encodes them into the generated model automatically.

VIII. Conclusion and Future Work

We presented a new approach where some of the hurdles related to adopting MBT were addressed. In earlier studies, we have observed that testers are not used to creating models, but they are used to creating executable test cases such as JUnit test programs and Selenium test scripts. The new approach avoids some of the hurdles related to manually creating models for testing by instead analyzing existing test cases and automatically generating models from those test cases.

There are many advantages with such an approach. For example, the tester can focus on creating test cases using tools like Selenium IDE where each test case can be automatically recorded by providing inputs to input fields, clicking on buttons, and adding assertions in proper places. Once the test case is created, the tester can ensure that it works properly by playing it again.

By generating a model from a set of such test cases, we showed how new test cases can be automatically generated. This also addresses the problem that testers typically do not have the time to create enough test cases. Since the model contains all information provided in the Selenese test case, the mapping problem is also addressed. The new approach automatically copies all executable statements into the model and thus minimizes the need for manual mapping.

Maybe the most important points are that the case study shows that this approach is feasible since the case study was conducted in very reasonable effort, and that new defects were detected by the generated test cases – defects that were not detected by the manually created test cases.

The approach was applied using Selenium on a web based system, namely NASA's DAT system. In the future we will evaluate if the approach can be extended to non-GUI systems and JUnit test cases to determine the wider applicability of the approach. We will also improve the model generation tool and plan to conduct a controlled class room experiment in order to further evaluate the costs and benefits of the new approach.

Acknowledgment

This work was conducted as part of a NASA supported SARP project. The authors wish to acknowledge support from SARP as well as from the DAT team.

References

[1] J. Tretmans, "Model-Based Testing and Some Steps towards Test-Based Modelling," in *Formal Methods for Eternal Networked Software Systems*, Springer Berlin Heidelberg, 2011, p. pp 297–326.

[2] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2010.

[3] A. Hartman, M. Katara, and S. Olvovsky, "Choosing a test modeling language: a survey," pp. 204–218, Oct. 2006.

[4] D. Ganesan, M. Lindvall, D. McComas, M. Bartholomew, S. Slegel, B. Medina, R. Krikhaar, C. Verhoef, and L. P. Montgomery, "An analysis of unit tests of a flight software product line," *Sci. Comput. Program.*, vol. March 2012, Mar. 2012.

[5] C. Schulze, D. Ganesan, M. Lindvall, D. Mcf Omas, and A. Cudmore, "Model-based testing of NASA's OSAL API — An experience report," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013, pp. 300–309.

[6] V. Gudmundsson, C. Schulze, D. Ganesan, M. Lindvall, and R. Wiegand, "An Initial Evaluation of Model-Based Testing," in *IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[7] C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman, "Assessing model-based testing: an empirical study conducted in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, 2014, pp. 135–144.

[8] "GraphWalker: The Open Source Model-Based Testing Tool." [Online]. Available: http://graphwalker.org/. [Accessed: 06-Jun-2015].

[9] and R. E. W. Mikael Lindvall, Dharmalingam Ganesan, Ragnar Ardal, "Metamorphic Model-based Testing Applied on NASA DAT," in *ICSE SEIP*, 2015.

[10] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen, "How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?," *IEEE Trans. Softw. Eng.*, vol. 40, no. 1, pp. 4–22, Jan. 2014.

[11] "Selenium - Web Browser Automation." [Online]. Available: http://www.seleniumhq.org/. [Accessed: 06-Jun-2015].

[12] "yEd - Graph Editor." [Online]. Available: http://www.yworks.com/en/products/yfiles/yed/. [Accessed: 06-Jun-2015].

[13] A. Marchetto, P. Tonella, and F. Ricca, "State-Based Testing of Ajax Web Applications," in *2008 International Conference on Software Testing, Verification, and Validation*, 2008, pp. 121–130.

[14] C. Wohlin, P. Runeson, M. Höst, M. C Ohlsson, B. Regnell, and A. Wesslén, "Experimentation in software engineering: an introduction," *Springer Netherlands*, vol. 15, no. 1, 2000.

[15] C. Torens, L. Ebrecht, and K. Lemmer, "Starting Model-Based Testing Based on Existing Test Cases Used for Model Creation," in *2011 IEEE 11th International Conference on Computer and Information Technology*, 2011, pp. 320–327.

[16] D. Angluin, "Learning regular sets from queries and counterexamples," *Inf. Comput.*, vol. 75, no. 2, pp. 87–106, Nov. 1987.

[17] A. W. Biermann and J. A. Feldman, "On the Synthesis of Finite-State Machines from Samples of Their Behavior," *IEEE Trans. Comput.*, vol. C–21, no. 6, pp. 592–597, Jun. 1972.