Software Testing Plan

Version 1.0

April 5, 2019

Team Amadeus

Mentor: Austin Sanders

Sponsors: Dr. Hélène Coullon & Frédéric Loulergue

Members: Wyatt Evans, Kyle Krueger, Melody Pressley, Evan Russell

Table of Contents

1. Introduction

Software deployment is an integral part of modern software development. Whether installing new security software on all the computers in an office network, or updating an app on thousands of devices across the cloud, software needs to be deployed often and efficiently. However, deploying large, complex pieces of software can be a difficult matter, and since all software is unique, all software deployment processes must also be unique.

There have been numerous solutions developed to make this process easier, such as Ansible or Kubernetes. However, most are inefficient and take much longer to deploy than they should, or are made for simpler micro-deployments. So far there have been no significant solutions that take advantage of concurrency and parallelism to the extent that they could.

Our project sponsor, Dr. Hélène Coullon, is a researcher with the STACK team at Inria - the French national research institute on computer science. Their work has produced Madeus: a theoretical model for software deployment. Madeus defines the deployment process in parts via a well-defined mathematical syntax and a corresponding Petri net-inspired diagram. The model also expresses every dependency between different software components. This enables software deployment to be performed concurrently, with different components executing deployment independently until a dependency is required. (See Section 8, pg. 16 for more details).

MAD (the Madeus Application Deployer), also an Inria project, is a Python implementation of the Madeus model; its goal is to allow users to deploy software according to the model. MAD provides an explicit syntax to Madeus by defining all aspects of it within Python modules. Together, MAD and Madeus have been found to deploy software up to twice as fast as some of the competing software. However, the efficiency of MAD's execution and deployment has come at the cost of simplicity and ease of use.

The team at Inria wants the Madeus model to be easier and more accessible for developers, so that the efficiency of this model can be fully realized. Thus, our solution is a GUI that enables users to utilize the Madeus model via the "Petri net-inspired diagram[s]" described above, rather than the specifics of a Python class. Our GUI, named the MAD Assembly Builder (MAB), will serve as a visualization tool for developers so that they can focus on the creation of a diagram, and generate functional, deployable,

MAD code representative of their assembly without having to go through the tedious process of coding it themselves.

In this document, we will be detailing our software testing plan for the development of MAB. Software testing is an important part of the software development process; it is how developers are able to determine the quality and usability of a piece of software to ensure that the product they are developing is working as it is meant to. For MAB, this testing phase is important in order to guarantee that the software correctly generates Madeus assemblies. If this integral task, and the tasks that facilitate it, are significantly flawed, then the resulting assemblies will be flawed as well. At best, this would make MAB useless - at worst, it would lead to the deployment of incorrect assemblies, potentially causing widespread problems for the software being deployed (as well as the systems they are deployed on) which could prove costly for clients utilizing our software.

Our software testing plan involves three kinds of testing: unit testing, integration testing, and usability testing. For unit testing, we will be ensuring that the important functions used for the creation, management, saving, and loading of Madeus assemblies all lead to the correct results. Due to the nature of MAD as a graphical user interface and the language used being Javascript, unit testing will not be used as much as in other projects, since there are not many areas where it is applicable. For integration testing, we will be going through a series of tests to ensure that the front-end user interface interacts correctly with the back-end data structure. Additionally, we will be testing that the UI and data structure are properly integrated with the four plugins we have developed, and finally checking that the plugins do not cause problems for each other. For usability testing, we will outline our plan for conducting tests with people who have similar backgrounds as our envisioned end-users with the purpose of ensuring that our software can be used as intended.

2. Unit Testing

Unit testing is a level of software testing where individual units/components of software are tested. The purpose of unit testing is to validate that each unit of software performs as it was designed. A unit is the smallest testable part of any software - for example, individual functions may be a unit. They usually have one or several inputs and usually have a single output. When procedural programming is used, a unit may be an individual program, function, procedure, etc. When object-oriented programming is used, the smallest unit is a method, which may or may not belong to a base/super class, abstract class, or derived/child class.

JavaScript being the main programming language in use inherently introduces challenges for unit testing. Some of those challenges are that JavaScript does not natively support unit testing in browsers, JavaScript cannot understand some system actions with the use of other languages, some JavaScript is written for a web application that may have multiple dependencies, JavaScript introduces difficulties with page rendering and DOM manipulation, and etc.

To combat the inherently introduced JavaScript challenges, we are going to perform individual tests of MAB's most important functions. The most important functions in the MAB codebase (including pre-packaged plugins) are:

1. addNewComponent()
2. addNewPlace()
3. addNewTransition()
4. addNewServiceDependency()
5. addNewDataDependency()
6. addNewConnection()
7. saveAssembly()
8. loadAssembly()
9. createComponentString()
10. createAssemblyString()
11. bootstrap()

Many of MAB's most important functions require the preexistence of certain elements. For example, a place can only be created inside a pre existing component, a transition can only be created between two pre existing places, and so on. For testing documentation purposes all valid arguments imply these pre existing dependencies have been established.

Function Being Tested	Description	Expected Result	Equivalence Partitions
1	Add a new component to the workspace by calling the addNewComponent() function with valid arguments.	The data structure will accurately represent the newly created component. The user interface will show the newly created component.	N/A
2	Add a new place to a component by calling the addNewPlace() function with valid arguments.	The component's place data structure will accurately represent the newly created place. The user interface will show the newly created place.	N/A
3	Add a new transition by calling the addNewTransition() function with valid arguments.	The component's transition data structure will accurately represent the newly created transition. The user interface will show the newly created transition.	N/A
4	Add a new service dependency by calling the addNewServiceDependency() function with valid arguments.	The component's dependency data structure will accurately represent the newly created dependency. The user interface will show the newly created service dependency.	N/A
5	Add a new data dependency by calling the addNewDataDependency() function with valid arguments.	The component's dependency data structure will accurately represent the newly created dependency. The user interface will show the newly created data dependency.	N/A

6	Add a new connection by calling the addNewConnection() function with valid arguments.	The connection data structure will accurately represent the newly created connection. The user interface will show the newly created connection.	N/A
7	Create a new save yaml file by calling saveAssembly() function with valid arguments.	A corresponding yaml file will be created and will accurately represent the created assembly.	N/A
8	Call the loadAssembly() function with valid arguments.	An assembly will be dynamically built in the user interface's workspace based off the textual representation inside of the chosen yaml file.	Correct file type, Correct serialized structure of each element type, Correctly ordered serialization
9/10	Call the createComponentString() and createAssemblyString() function with valid arguments.	Corresponding Python files will be created and will accurately represent the currently created assembly being built in the user interface.	N/A
11	Call the bootstrap() function in the Simulate Deployment plugin with valid arguments.	A read-only layer will be added to the workspace which will spawn corresponding self-moving objects that will simulate the deployment of the currently created assembly being built in the user interface.	N/A

The controls of MAB are implemented through event listeners of specific inputs. This ensures MAB's codebase has been written with restrictions in place that will prohibit the user from performing an illegal action or to call a constructor function with erroneous inputs. The selected inputs therefore automatically create the equivalence partitions. If the user is attempting to create a new place outside of a currently existing component, MAB will ignore these actions and wait for a legal action to occur. However, there is a risk of attempting to load a file that is not a correctly serialized assembly. For this reason we need to implement a series of equivalence partitions to test against to ensure a file the user is attempting to load into MAB is a valid assembly.

3. Integration Testing

Integration testing is used to ensure that different modules of a software interact with each other properly. For MAB we will be using integration testing to test that our front-end user interface and our back-end data structure are linked correctly, and then test that the back-end data structure is accessed and updated properly by our four plugins: Generate Code, Simulate Assembly, Save Assembly, and Load Assembly. To this end, we will be using a top-down integration approach, beginning with the user interface and then branching into the plugins.

Test Case ID	Objective	Description	Expected Result
1	Test the link between the user interface and the data structure.	Build an assembly with all pieces, and observe the updated data structure during each step. Compare this with a premade test data structure.	The data structure will dynamically and accurately represent the assembly being built in the user interface at every point in the creation process. The resulting data structure should match the premade data structure.
2	Test the link between the data structure and the "Generate Code" plugin.	Build an assembly with all pieces, and execute the "Generate Code" plugin. Compare this to premade python files containing a Madeus assembly	The generated python code will accurately match the assembly that was built. The resulting files should be the same as the premade files.
3	Test the link between the data structure, user interface, and the "Simulate Assembly" plugin.	Build an assembly with all pieces, and execute the "Simulate Assembly" plugin.	The simulation that runs will accurately match the assembly that was built, in regards to visual accuracy and runtime.
4	Test the link between the data structure and the "Save Assembly" plugin.	Build an assembly with all pieces, and execute the "Save Assembly" plugin. Compare the result with a premade YAML file representing the same assembly.	The generated YAML file will accurately match the assembly that was built, translated to a human-readable format. The contents of the resulting YAML file should match the contents of the premade YAML file.
5	Test the link between the "Load Assembly" plugin, the data structure, and the user interface.	Execute the "Load Assembly" plugin, selecting a YAML file created by the "Save Assembly" plugin. Compare the result to a hand-made assembly made from the same YAML file.	The user interface and the data structure will both be updated to accurately show the assembly as it is described in both of the YAML files. Both of the YAML files should show identical assemblies in the MAB user interface.

These test cases have been selected because they highlight all of the direct interactions between the modules of MAB. Since MAB is built as an extensible software, and we have implemented many of its features as plugins, there are no long chains of module interactions. All of the pre-made MAB plugins only rely on the data structure, the user interface, or both, and therefore do not need to be tested for integration with any other plugins.

4. Usability Testing

The purpose of usability testing is to give our system the opportunity to be used by real users. Close observation of the ways that users use our software (and in some cases, the ways they don't) during this stage of testing can highlight potential mistakes our team has made throughout the course of development. This phase is particularly useful because it can test the integrity of certain components in our software that can't be tested programmatically - for example, the ability of our UI to visually communicate its functionality to an end-user.

MAB will primarily be used as a way to interface with MAD; as such, our user base will be comprised mostly of other software engineers, or at the very least people familiar with (or interested in) software deployment. All of this implies some level of technological familiarity of our general targeted audience - our prospective individuals for usability testing should reflect this. Therefore our candidates will consist of students majoring in Computer Science at NAU who are at least in their Junior year. This guarantees our testing group will have a similar technological background as our expected end-users.

We plan to briefly introduce users to our system and then put them through a series of initial tests that will help our team identify any potential flaws in our system, especially in terms of the UI. Afterwards we will provide a short survey so that users may verbalize any difficulties they had that we didn't pick up on through observation.

Test Case ID	Objective	Description	Expected Result
1	Build a simple assembly	Given a written description, users should build a 2- component assembly with 2 places each, 1 transition each, and 1 dependency each, with 1 connection total.	Given a brief introduction to our system, users should be able to build the target assembly within 60 seconds (1min).
2	Build an intermediate assembly	Given a written description, users should build a 3- component assembly with 3 places each, 4 transitions each, and 1-2 dependencies each, with 2 connections total	Given a brief introduction to our system, users should be able to build the target assembly within 150 seconds (2.5min).
3	Build an advanced assembly	Given a written description, users should build a 4-component assembly with 4 places each, 5-8 transitions each, and 1-2 dependencies each, with 4 connections total	Given a brief introduction to our system, users should be able to build the target assembly within 300 seconds (5min).
4	Save/Load an assembly	After making any assembly, users should be able to save an assembly, restart the program, and load an assembly without being instructed on where saving/loading functionality lies.	After making an assembly, a user should be able to save/load within 60 seconds (1min).
5	Survey	Users will complete a 3-question survey: “Are there any specific things you found yourself struggling with?”, “What would you like to see changed about the system?”, and “On a scale of 1-10 how would you rate your experience today?”	On completion of the survey, our team should have a better understanding of the user's thoughts on our system and any shortcomings we may have not gathered through observation.

5. Conclusion

Software deployment can be a complex process; many solutions have been developed, such as Ansible or Kubernetes, but these often lack performance, resulting in slow deployment times. Madeus is a highly efficient software deployment model that leverages any opportunities for parallelism, which significantly improves deployment times.

MAD is a Python implementation of the Madeus model, allowing users to program a Madeus assembly and execute the representative deployment process. However, MAD can be complicated and tedious to implement and its parallelism creates complexity when it comes to understanding the dependencies between the different tasks in its deployment process. It is also difficult to edit, because changing one element of an assembly could require numerous other parts of the code to be changed.

Our Graphical User Interface will help visualize, create, and maintain the complex parallelized deployment schemes that drive MAD. As a result, it will reduce the complexity for the end-user wanting to use Madeus/MAD to deploy a distributed software system.

Additionally, our plugin framework and the features that are implemented through this framework will ensure the longevity of the software. The documentation we are developing alongside MAB, as well as the open source nature of the software, will help to facilitate the development of future plugins, so that as the needs of the developer inevitably expand and change, MAB can change as well in order to better meet those needs.

This document aims to outline our software testing plan is a way that explains not only our plan, but our reasoning for having this particular plan. This document outlines details of this plan such as unit testing, integration testing, and usability testing, which are the three testing styles that we are using for this project. With this testing plan in place we hope to ensure that our product will work as intended, and satisfy our clients.