Software Design Document (v.2)
February 25, 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 a 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 the architecture of MAB, and showcasing how each of the features fulfill the needs of our sponsors, as well as how they all connect into the greater whole of MAB.

2. Implementation Overview

As mentioned, the best software solution to the problems described in the previous section was determined to be a GUI. Based on this, we researched methods of implementing this and evaluated what approaches best fit our problem and context. In this section we will outline the technologies our system will use to maximize the desired outcome.

The backbone of our GUI is Electron, a framework developed by GitHub. Electron is used to build GUI applications, and provides a frontend using HTML/CSS and a backend using JavaScript. Electron is incredibly easy to use, and enables our team to create a GUI with minimal developmental overhead. Although Electron is somewhat notorious for being wasteful of memory, this isn't a major constraint for our project. Although we should still be mindful of our GUI's computational performance, most contemporary computers will fare fine using Electron.

Being a GUI framework, Electron naturally divides its work between a frontend and a backend. Electron's frontend is powered by Chromium, enabling our team to design the GUI's look with HTML5 and CSS3. The backend is powered by Node.js, allowing us to provide functionality to our GUI through JavaScript. The combined frontend and backend not only give us the tools to develop a robust GUI, it also allows us to utilize any other code libraries that are compatible with either.

Our solution allows users to build Madeus assemblies in a diagram format; as such, our software must provide users with some kind of "canvas" to work with. Our system implements this using Konva, a JavaScript canvas library for desktop applications (and more). Because Konva is a JavaScript framework, it meshes well with Electron - using Konva via Electron's Node.js backend and its HTML-based frontend we can represent all of the potential shapes and figures a user needs to express their software deployment in terms of the Madeus model.

3. Architectural Overview

MAB's design follows the Model-View-Controller (MVC) design paradigm. In this use case, the model is a simplistic data-structure that dynamically houses the user's created assemblies, the view is an HTML based viewport with Konva objects acting as the building blocks, and the controller is Electron.

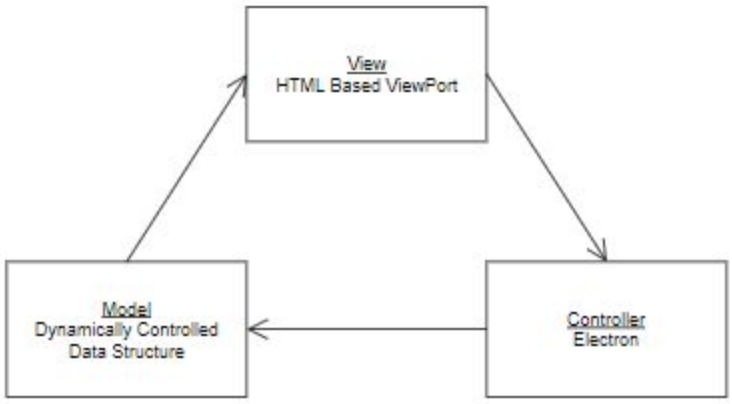

Fig. 1: MVC Integration

The dynamically controlled data structure that is the model in this MVC is a JavaScript array. The array was chosen over many other data structures natively present in JavaScript because of its simplicity. Accessing and appending to arrays are amortized O(1). The constant run time was extremely important for user interaction. The user should never "feel" that anything was happening in the background; the user should only see what is being presented to them in a timely manner.

MAB's view is informed by the model and realized with HTML5, CSS3, and Konva. HTML5 is a standardized system for presenting web pages on the World Wide Web (WWW). Konva is an HTML5 canvas JavaScript framework that extends the 2d context by enabling canvas interactivity for desktop and mobile applications. Konva enables the user to build their assemblies with objects instead of predefined shapes that have no interactivity.

Electron acts as the controller in the MVC. Electron is an open source library developed by GitHub for building cross-platform desktop applications with HTML5, CSS3, and JavaScript. Electron accomplishes this by combining Chromium "an open-source browser project" and Node.js "an asynchronous event driven JavaScript runtime". The use of

Electron as the controller provides seamless integration of needed modules into one package. Electron's native use of JavaScript allows the integration of the chosen dynamic data structure, it's use of Chromium affords the use of HTML5 and CSS3 to easily present information (namely, Konva figures) to the user, and it's overall design allows for the easy packaging of applications for any operating system the user may choose.

4. Module and Interface Descriptions

The architecture of MAB involves six key modules and interfaces. These make up the interface that the user builds assemblies with, the core creation of the assemblies, and the implementation of plugins that perform additional tasks such as generating MAD code, saving and loading assemblies, or simulating assembly deployment. In this section, those six key parts of the architecture will be discussed and linked with each to provide a better understanding of MAB as a whole.

4.1. Graphical User Interface

The Graphical User Interface will be the main tool that the user interacts with to create and manipulate Madeus assemblies. The GUI will provide the actions to directly manipulate the graphical elements of Madeus assemblies such as components, places, transitions, and dependencies. The Graphical User Interface will be minimalist in design. The user will drag and drop components to create them in the workspace. To create a place inside a component the user will double click inside the component where they wish to create the place. Transitions will require the user to left click on the source place and right click on the destination place. Transitions will only be added to valid places. Dependencies will be an attribute of places and transitions, and will be instantiated by the user. The user will left click on the provide portion of the dependency and right click on the use portion of the dependency to connect them. Figure 2 shows a screenshot of the GUI with an assembly being in the process of development.

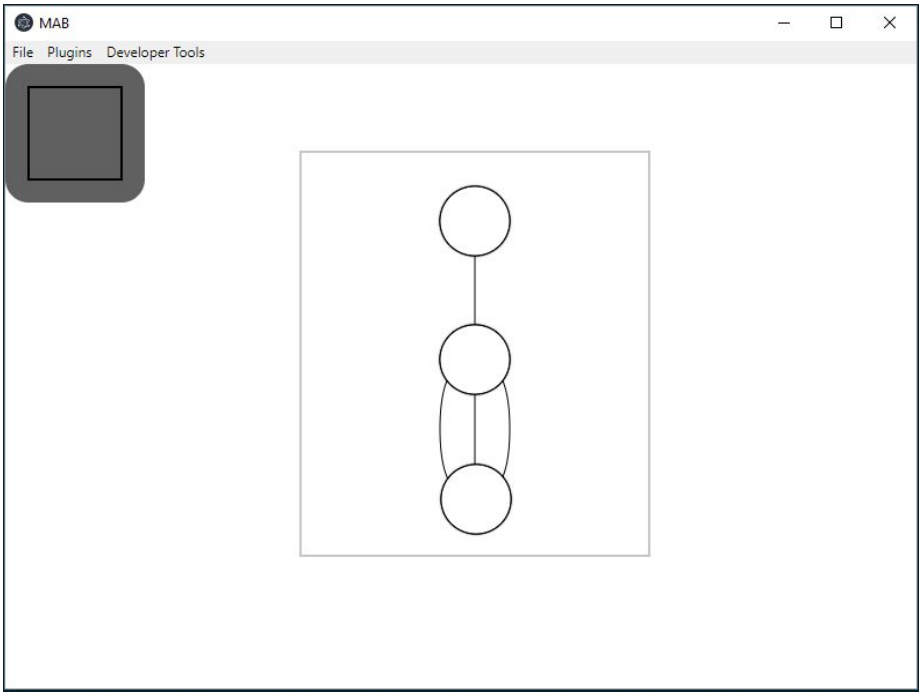

Figure 2: Prototype of GUI

4.2. Assembly Data Structure

For our software project, we need some kind of data structure to represent an assembly. An assembly contains all of the components, places, transitions, and dependencies in a given Madeus software deployment. This assembly is initially empty until the user begins to populate the canvas with Madeus elements, at which point the data structure then encapsulates all of the relevant information needed to represent the user's assembly.

This data structure is integral to the entire functionality of our GUI - without a data structure representing an assembly, users are simply dragging shapes onto a canvas. This data structure is updated dynamically as an assembly is edited in the GUI canvas - it is an in-memory formalization of the diagram in accordance with the Madeus model that is updated as the GUI is used. Our assembly data structure provides information that plugins need, enabling them to do operations such as saving and loading assemblies from and into the GUI, generating MAD code, and simulating the deployment of an assembly.

As the GUI listens for events (i.e. the user creating their diagram) not only are figures being created via Konva, but functions are also being called according to what Madeus pieces are being added/modified, and the appropriate updates are made to the Assembly data structure on the backend, whether it's changing the name of an element or adding a

new place to the “place_list” of a given component. Figure 3 shows the general workflow the assembly data structure concerns itself with.

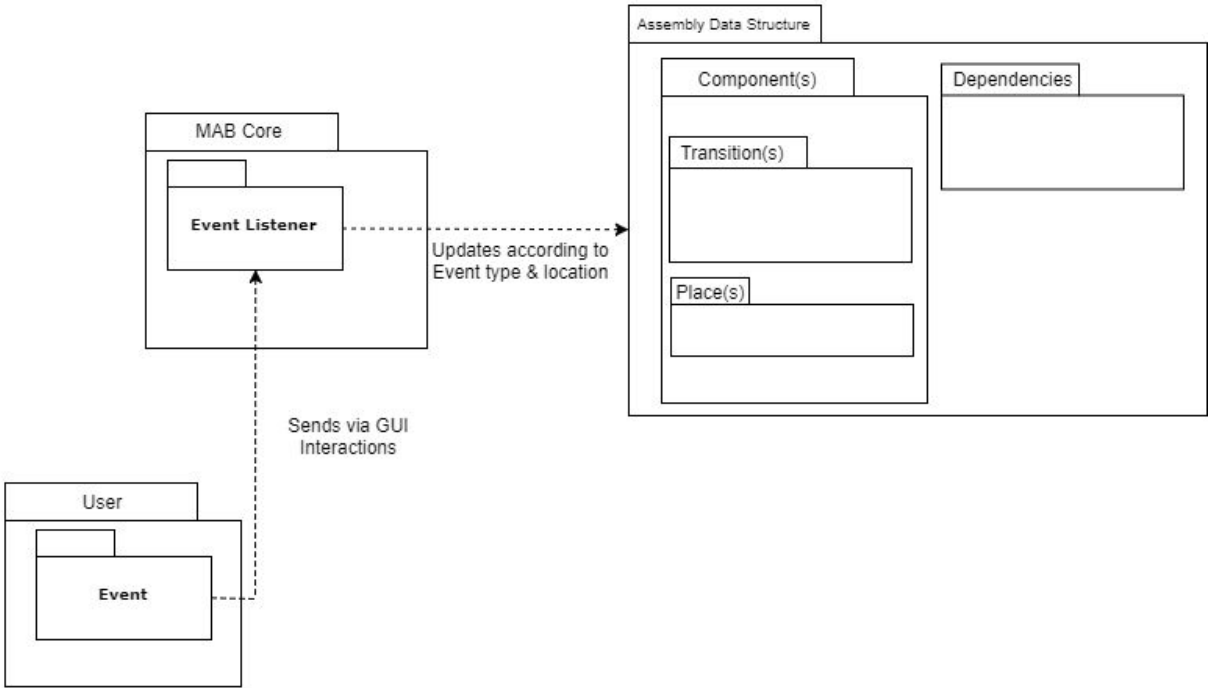

Fig 3: Workflow of Assembly Data Structure

4.3. Plugin Framework and Integration

This component of the architecture is responsible for validating plugins, populating the GUI with buttons related to each validated plugin, and finally executing those plugins on the request of the user. This is vital to MAB’s extensibility, as it will continue to be developed and added to beyond this year, primarily through the use of this plugin framework.

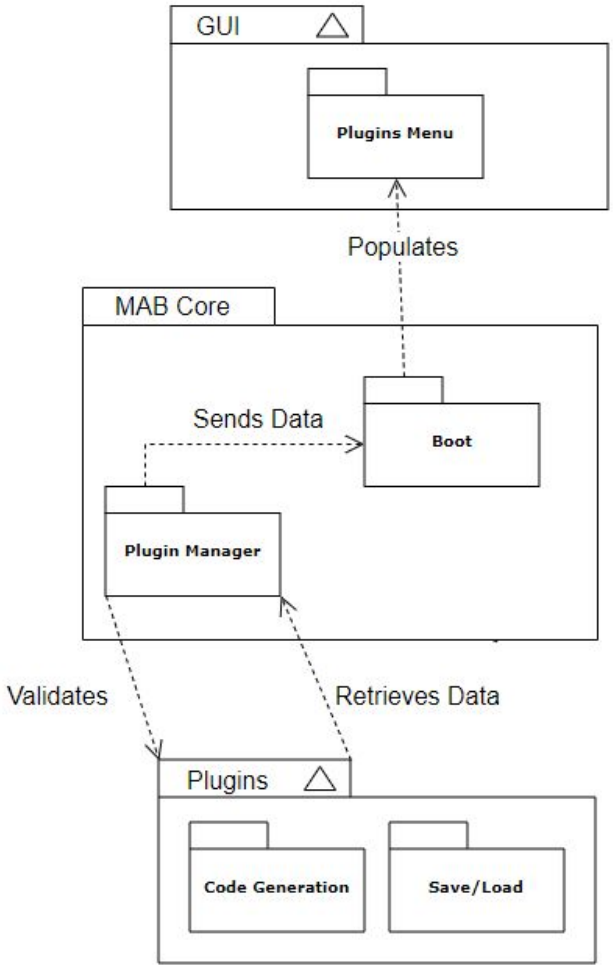

Fig 4: Architecture of the Plugin Framework

The Plugin Manager validates plugins based on the structural requirements detailed in the MAB Plugin Documentation (a separate document, available on our capstone website). In particular, each plugin must have a driver.js file, which uses an IPC Renderer looking for a specific message, dependent on the name of the plugin's unique folder.

When a plugin is validated, its name and file path are stored and passed on to the Boot part of MAB. The Boot is where the application is launched from, and is also the part of the code that generates the GUI. At this point, a button is made that sends the message that its appropriate driver file is waiting for, each time the button is clicked. These buttons populate a particular drop-down menu in the GUI, where they can be quickly accessed by the user either through clicking, or by a quick keyboard shortcut. This system ensures that the plugins can be executed whenever the user wants, and however many times they need it to.

4.4. Code Generation

MAB's code generation is a core functionality provided by the GUI. It converts the user's generated diagram into executable Python code that can be run via MAD (developed by Inria). It has been implemented into the GUI's architecture via an extensible plugin (Fig. 4). This allows the code generation itself to be non-rigid, and act mostly independently of the GUI (aside from the information retrieved from the assembly data structure). The code generation plugin builds as many files as necessary, i.e. one file for each of the created components, and one more overall file for the assembly as a whole.

MAB generates the required Madeus code by heavily relying upon the data structure that was simultaneously being created by the user during assembly creation. The data structure holds objects of types component, place, transition, and dependency. When the plugin detects the type of the object, it generates the necessary code specific to that type. It also detects all associated attributes of each type and continues to generate the necessary/associated code needed. It does this for all the associated types and completes the process by creating one final file, the driver file.

The driver file is responsible for creating any needed MAD objects: assemblies, components, and/or dependencies. The code generation plugin is also responsible for detecting what type of code needs to be created and for determining if the created code is valid.

4.5. Deployment Simulation

The deployment simulation will consist of animations showing the deployment of the Madeus assemblies in action. Each component will have a visual "token" that will transition from place to place through the component's transitions. Multiple transitions coming out of the same place will have child tokens created and animated in parallel, representing MAD's ability to execute transitions concurrently. Each component will be simulated independently of one another unless a dependency exists. Dependencies will display the enabling or disabling in the use-provide or data-use-provide connections to be determined by a token being present in the place and transition of the corresponding components. Figure 5 shows an example of this process, with both of the tokens being present in the initial places of each component.

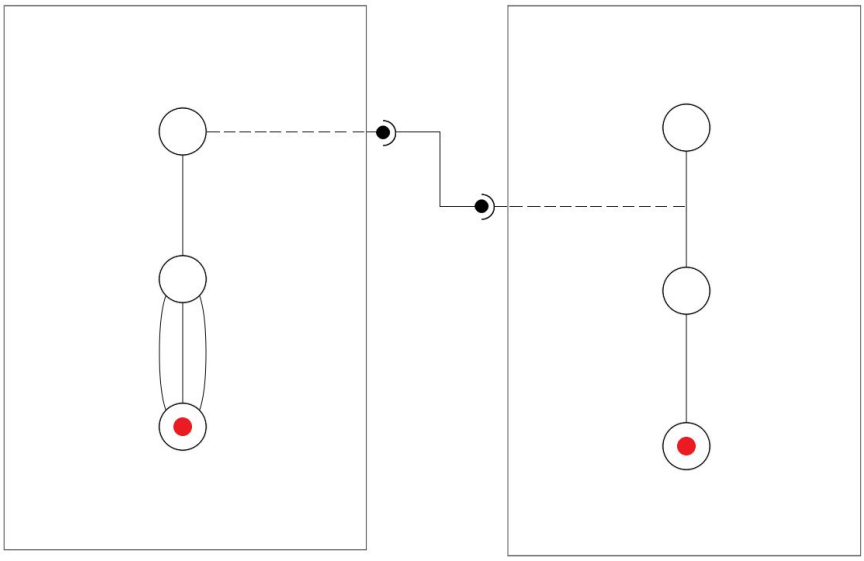

Fig 5: Prototype of Deployment Simulation

4.6. Saving and Loading

This architectural component involves the saving and loading of a MAB user's assembly to and from the YAML file format. This will be implemented into MAB as a plugin, and it must be able to accurately replicate the Assembly's detailed parts, including any connections they may have and their positions in the MAB workspace.

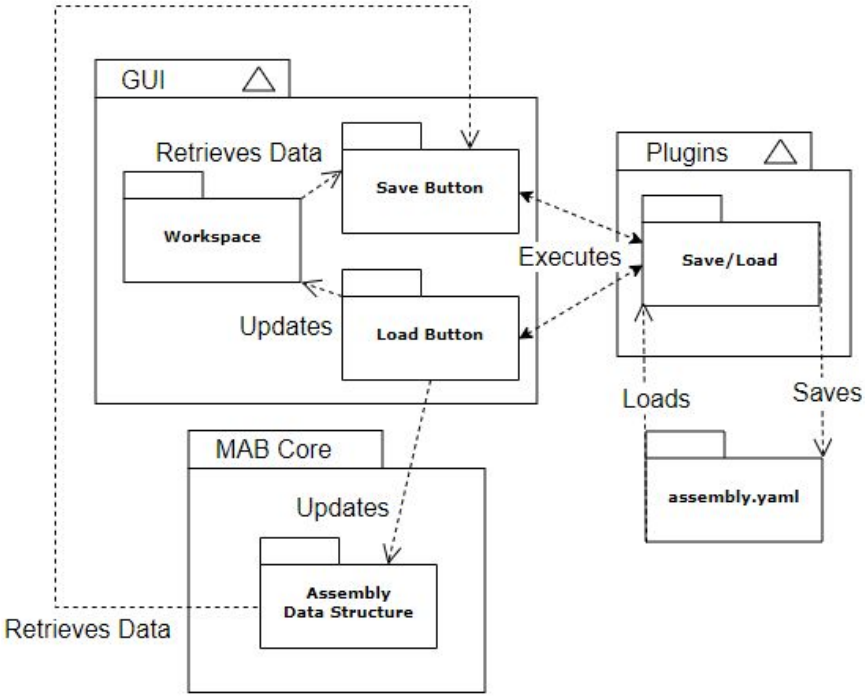

Fig 6: Saving and Loading Process

Since this component will be implemented as a plugin, it will be executed at the user's command through buttons in the GUI. When the user saves their assembly, the data structure will be stored into the YAML file, alongside data from the workspace which will include details such as the coordinates for each part of the assembly and what connections exist. This data will then be stored into the YAML file created or updated at the user's desired location.

When the user loads their assembly, the workspace will be cleared, and the assembly will be visually loaded based on the coordinates and types stored in the YAML file. Then, the data structure that was saved in the YAML file will overwrite the existing data structure, updating it to reflect the loaded assembly.

5. Implementation Plan

Fig. 7: Gantt Chart (February 5, 2019)

Our implementation plan or project's major development process was broken down into three major phases: Graphical User Interface Creation, Data-Structure Creation, and Plugin Framework and Integration. The Graphical User Interface consists of component, place, transition, and dependency creation and manipulation from the user. The next major development process is Data-Structure Creation. The Data-Structure creation consists of saving all assembly objects that the user creates and their attributes. The Data-Structure will play an important role in future plugin implementation. The Plugin Framework and Integration consists of Code Generation, Deployment Simulation, and Saving and Loading of user created assemblies. Each major areas will be developed simultaneously throughout the semester. Certain sub-tasks will depend on other development areas, such as file saving/loading and deployment simulation. These tasks

will be implemented later in the development phase once the tasks they depend on have been completed. Each major development process was divided up for implementation across the team as illustrated in the table below.

Team Member	Tasks
Evan Russell	GUI Creation, Data-Structure Creation, Deployment Simulation
Kyle Krueger	Data-Structure Creation, Code Generation
Wyatt Evans	File Saving and Loading, Documentation of Plugin Support
Melody Pressley	Data-Structure Creation, File Saving and Loading

6. 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 explain the various parts of MAB, which are all working towards alleviating the downsides of Madeus/MAD as well as further enhancing the software deployment field as a whole. We are confident that these six major modules will result in a complete and satisfying software that will survive, and be capable of evolving long after this year, while also fulfilling the present needs of our sponsors.