Requirements Specification (v.3)
December 10, 2018

Team Amadeus

Mentor: Austin Sanders
Sponsors: Dr. Hélène Coullon & Frédéric Loulergue
Members: Wyatt Evans, Kyle Krueger, Melody Pressley, Evan Russell

Accepted as the Baseline Requirements for the Project:

Sponsor Signature: x____

Team Signature: x____

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 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.

2. Problem Statement

Although the Madeus model itself is highly efficient at concurrently deploying software, the act of defining the deployment process in terms of MAD is difficult. The workflow of our client's software is prone to various problems, primarily because users have to deal with the shortcomings of expressing their design in code. These problems are exacerbated even further when a user is creating complex, deeply interconnected assemblies with many different parts. The two biggest issues with the workflow of our sponsor's software are as follows:

- Tedious Design Implementation:
  The Madeus model at its core is powerful, without being too complex. The steps required to define a deployment in MAD, however, are tedious and time-consuming. After a user knows what their deployment should look like using Madeus elements (which doesn't involve any programming), the user then is required to manually specify every aspect of their deployment again in code using MAD classes. Not only is this tedious because a user has to express their deployment twice, but also because every detail of the deployment needs to be defined in code, regardless of how small. Madeus and MAD are exceptional because of their power, but a large portion of the workflow could use automation.

- Difficult to Edit:
  Another problematic part of our client's software is that it's difficult to refactor or modify a deployment, regardless of the size of the change. Deployments in MAD are usually concurrent, and there are many interconnected parts used in many different places. As such, the process of modifying a MAD deployment is prone to numerous errors - whether it's through the deletion of an element in some places but not others, changing a name in some places but not others, or failing to fully connect a new state to all of its required transitions.

The current process requires developers to conceptualize an assembly first, and then write code that matches their envisioned assembly. It is much easier to express an assembly as a diagram than it is to express one as code. Therefore, a useful change to this process would be to put more emphasis on the visualization of what the assembly should look like, instead of focusing on the tedious structuring and maintenance of the code as it goes through the development process.

3. Solution Vision

The team at Inria wants the Madeus model to be more accessible to anyone interested in it, so that the efficiency of deployment 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, currently 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.

Developers will also be able to simulate the deployment process in our GUI, so that they can make sure the assembly they have created will function in the ways they are intending it to.

Additionally, our GUI will allow developers to edit their assemblies easier. Instead of manually changing the code in all of the necessary ways, the developers will only need to move objects in the GUI and all changes to the code’s structure will be handled for them.

This GUI will help increase usability by allowing what would usually be done manually in Python code to be done graphically instead. By providing clear visualizations, allowing drag-and-drop assembly building, and MAD code generation, we hope to give users easier access to the Madeus model, and make MAD a more ideal tool for use in software deployment.

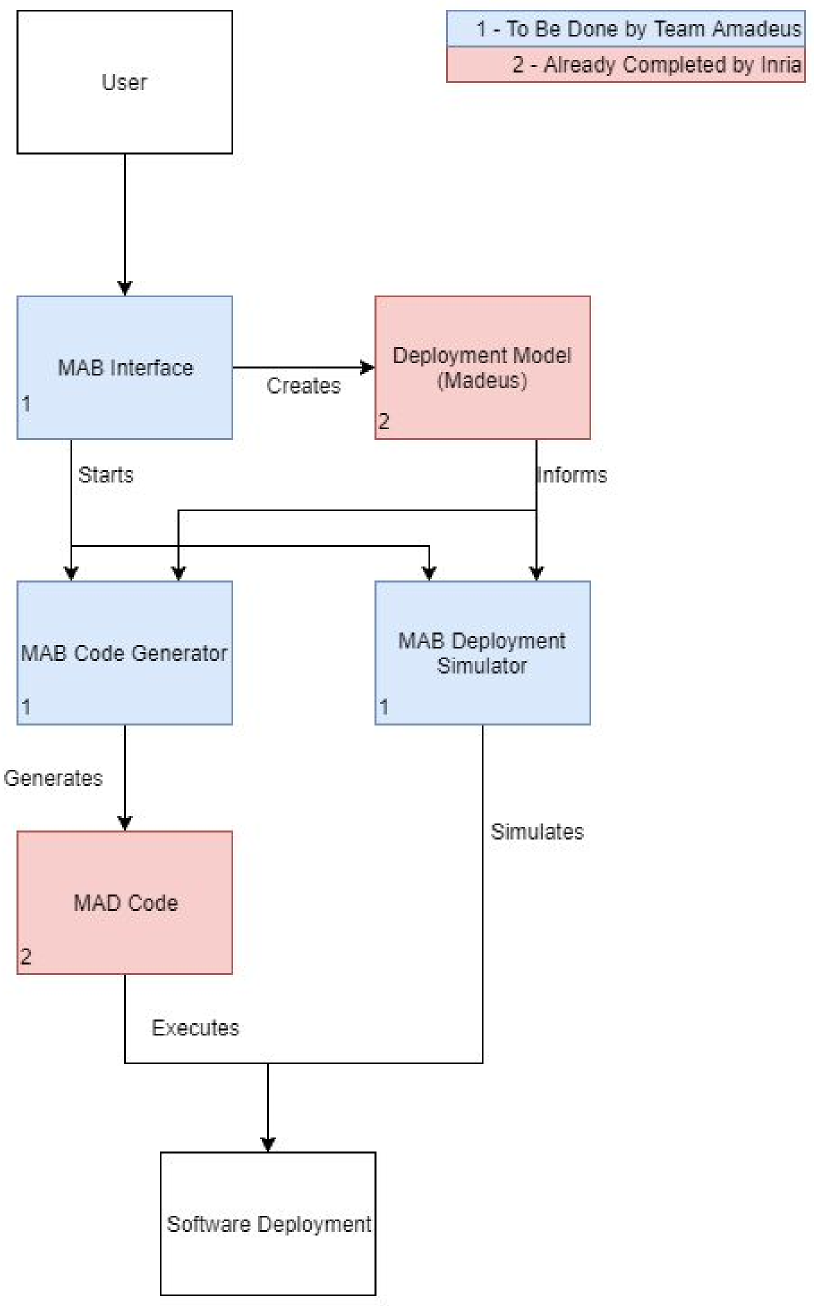


Figure 1: High Level Overview of System

An overview of our solution is shown in Figure 1. This overview illustrates elements of our system ("MAB Interface", "MAB Code Generator", and "MAB Deployment Simulator"), their relation to our client's existing projects ("Deployment Model", "MAD Code"), and also how *all* of these elements relate to the end-goal of software deployment.

4. Project Requirements

The primary focus of this document is to detail the requirements which we have collected over the last few months. These requirements will form the foundation from which we will build our project, in order to ensure that we will meet our sponsors' needs with as few complications as possible. In this section, we will detail our 4 overarching domain-level requirements, and then break down our 11 top-level requirements across three categories: Functional, Performance, and Environmental.

4.1. Domain-Level Requirements

These are the core, broadly described features which our software must have in order to meet the requirements of our sponsors, and successfully solve their problem.

D1.Visualization
Our users must be able to prioritize the visualization of the assembly they are trying to create. Our GUI must allow the users to build a diagram out of the parts of a Madeus Assembly as those parts are implemented in MAD. To this end, we must provide the user with the necessary functionality and creative options to create any assembly that can be written using MAD.

D2.Extensibility
It is impossible for us to predict every feature that a user may need or want for our software. While we cannot implement everything, we can ensure that our software will be capable of handling a wide variety of additional features or updates in the future. Therefore, our software must be highly extensible to allow end-users to add or adjust features and functionality.

D3.Generation
Since the user will be focusing on the visualization of their assembly, our software will need to handle the bulk of the code. Our software must be able to convert the assembly visualization created by the user in the GUI, into accurate MAD python code. This code will require the user to manage certain features that our GUI is not capable of managing, such as the details of transitions and dependencies, as well as any other backend features that are necessary for the

user’s unique assembly. However, the code that we create should be an accurate representation of the user’s assembly otherwise.

D4. Simulation

We must provide a way for our users to simulate the deployment of a Madeus assembly, as implemented in MAD, without needing to actually go through the deployment process. We need to show visually how the deployment will progress throughout all components of the assembly. We must also allow the users to give estimated runtimes for each part of the deployment, so that the user will be able to observe the deployment simulation that will most accurately meet their needs.

4.2. Functional Requirements

1. *Top Level Requirement*: “Drag and Drop” method for creating assemblies (**D1**)
Users of MAB should primarily interface with their work through a Drag and Drop interface. This method is the most convenient way of building assemblies.

1.1. Madeus elements should be capable of being dragged into a project workspace
- 1.1.1. A component should be able to be dragged into the workspace
- 1.1.2. A place should be able to be dragged into a component
- 1.1.3. A transition should be able to dragged to a place
  - 1.1.3.1. After being dragged to a place, the transition should also be connected to a second place.
  - 1.1.3.2. The two places a transition is connected to must exist within the same component
- 1.1.4. Provide ports (data and service) should be able to be dragged into a component, with one or more associated places.
- 1.1.5. Use ports (data and service) should be able to be dragged into a component, with one or more associated transitions.
- 1.1.6. Data-provide ports should be able to be linked to a data-use port in a separate component.
- 1.1.7. Service-provide ports should be able to be linked to a service-use port in a separate component.

1.2. From within a workspace, existing elements should be capable of being dragged around and rearranged
- 1.2.1. Dragging a pre-existing component should also drag all elements within the component

1.2.1.1. Dragging a component should maintain any existing dependencies with other components

1.2.2. Dragging a pre-existing place should maintain all currently-existing transitions

1.2.2.1. A place should not be capable of being dragged outside of its parent component

1.2.3. Dragging one end of a pre-existing transition must result in that end being assigned to a place.

1.2.3.1. After reassignment, both places to which a transition is connected must exist within the same component.

2. *Top Level Requirement*: Code Generation (**D3**)
MAB must have the capability of generating executable MAD code based on user generated diagrams. This allows a user to execute their deployment on any system they desire.

2.1. Component Creation

2.1.1. All places will be inserted into a single list.

2.1.2. All transitions will be inserted into a single dictionary.

2.1.2.1. The transition name is the key, and the values are tuples of size 3 which contain a start place, an end place, and associated function (In that order).

2.1.2.2. A place-holder function with sleeps will be created per transition with user inputted names.

2.1.3. User inputted dependencies will be inserted into a dictionary.

2.1.3.1. Dependency names will be the key, the value will be a tuple of size 2 with dependency type, and a list of potential transitions (In that order).

2.2. Assembly Driver Creation

2.2.1. Component object instantiation based on previous component file(s) creation.

2.2.2. Assembly object instantiation.

2.2.3. Add component object(s) to the Assembly object.

2.2.4. Add dependency connection between previously added components. Dependency Variables:

2.2.4.1. First component name

2.2.4.2. Dependency name

2.2.4.3. Second component name

2.2.4.4. Dependency name.

2.2.5. MAD object instantiation from the assembly object.

2.2.6. MAD deployment i.e. mad.run()

3. *Top Level Requirement:* File Saving (**D2**)
Our software must be capable of saving and loading MAD assemblies generated in the MAD Assembly Builder to and from the .yaml file format.

3.1. Save Assemblies to .yaml format

3.1.1. Save feature will be implemented through an object in the GUI (such as a button)

3.1.2. Develop a file structure for MAD assemblies in .yaml

3.1.2.1. Determine key-value pairs that associate with parts of the Assemblies

3.1.2.2. Key-Value tiers in order of Assembly->Components->Internal Structure->Details

3.1.3. Saved .yaml file must have the correct structure and indentation (representative of the saved Assembly)

3.2. Loading Assemblies from .yaml format

3.2.1. Load feature will be implemented through an element in the GUI (such as a button)

3.2.2. Read in the requested file and display the associated assembly

3.2.2.1. The resulting dictionary will be read and converted into GUI objects as described in the *Code Generation* requirement (4.2.3)

4. *Top Level Requirement:* Unobtrusive Alert System (**D2**)
Our system should feature an unobtrusive alert system ("UAS") that alerts users of any potential errors that could be present in their assembly, such as a cycle in the transitions.

4.1. The UAS should be implemented via the software's plugin system.

4.2. The UAS plugin implemented by Team Amadeus will be manifested as a console-style text box visible in the main GUI.

4.2.1. Checking assembly will be done either via MAD's own "check_warnings" function, or a MAB-specific implementation of the "check_warnings" function.

4.2.1.1. Checker runs when a new Madeus part is added to the assembly.

4.2.1.2. Checker runs when a project is loaded into MAB.

4.2.1.3. If the checker returns a warning, display the warning in the UAS.

4.2.1.4. If the checker does not return a warning, display a confirmation message.

5. *Top Level Requirement:* Plugin Support (**D2**)

The software must be extensible in order to allow future changes or additions to the GUI and the GUI's functionality. All possible future changes or additions cannot be predicted, so this section will detail the specific attributes that must be extensible, as well as the ways we will work to make the rest of the software extensible.

5.1. Security

5.1.1. Controlled Access

5.1.1.1. Software must be written such that plugin developers will have access to certain features of the code for the purposes of changing/adding to that code

5.1.1.2. Base functionality of the code will not be explicitly written to allow change via Plugins

5.2. MAB must be extensible by plugins

5.2.1. MAB must provide a shared library system to allow for plugin storage and access

5.2.1.1. Designated Plugin Folder

5.2.1.1.1. Designated Plugin Folder must be accessible to the user

5.2.1.2. Plugin Checker

5.2.1.2.1. Program must detect files in Designated Plugin Folder at launch or shortly after launch

5.2.1.2.2. Program must check that a Plugin has the correct filetype (.py) before utilizing the Plugin

5.2.2. MAB must be capable of integrating multiple plugins

5.2.2.1. Software must be able to integrate multiple Plugins

5.2.2.1.1. Plugins must be of the proper file type to function (.py)

5.2.2.2. User will be allowed to attempt implementation of any kind of plugin

5.2.2.2.1. Plugins may not work if they interfere with core functionality

5.2.2.2.2. Plugins may not work if they are of an incorrect file type

5.2.2.2.3. Plugins may not work if they cause an error

5.2.3. Specific features of MAB must be extensible through this plugin framework

5.2.3.1. Specific graphical components of MAB must be extensible through this plugin framework

5.2.3.1.1. Colors may be changed

5.2.3.1.2. Text may be changed

5.2.3.1.3. Text may be added

5.2.3.1.4. GUI object locations may be changed

5.2.3.1.5. GUI object models may be changed

5.2.3.1.6. GUI objects may be added

5.2.3.2. Specific functional features of MAB must be extensible through this plugin framework

5.2.3.2.1. Additional file types (other than .yaml) may be added for the purposes of saving to

5.2.3.2.1.1. The developer of such a Plugin is required to provide the code for file conversion and saving

5.2.3.2.2. Error Handling

5.2.3.2.2.1. Language Support

6. *Top Level Requirement*: Animation for Simulating MAD Deployments (**D4**)
In order for the user to be assured that their assembly will behave as intended, they must be able to simulate the deployment process. To this end, the system must include a simulated visualization of the deployment process in-GUI.

6.1. The Simulation should be animated to illustrate the deployment

6.1.1. MAB will have a "Simulate" button that initiates the simulation of the deployment

6.1.2. Components will be animated to show a process running through their deployment

6.1.3. The simulation will portray deployment speeds according to a time range estimate inputted by the user.

6.1.4. The user's estimated range will allow the system to simulate a best and worst-case scenario via the upper-bound and lower-bound of the range.

6.1.5. Transitions that are concurrent will be animated concurrently.

6.1.6. Dependencies will be animated and distinguishable between the user and provider components of that dependency.

6.1.7. An animation will display when a component has reached its final place.

6.1.8. If the user does not input an estimate for a transition it will default to a range of 1 to 2 seconds.

4.3. Performance Requirements

1. *Top Level Requirement*: Assembly Creation
   Significant thought should be put towards the assembly creation process. Our system should enable users to create a deployment in the form of an assembly in a timely manner; these metrics will help us determine how well users can interface with our system.

   1.1. A simple 2-component assembly with 3 places per component and 1 dependency (of any type) between the 2 components should be able to be built within 10 minutes by someone that has read the original paper on the Madeus model ("Madeus: A formal deployment model").

      1.1.1. Each primary element of the Madeus model (i.e. "component", "place", "transition", and "dependency") should be immediately accessible from the main GUI screen for use in a user's assembly.

   1.2. An assembly representing the concurrent deployment of an Apache server with a MariaDB database server should be able to be built within 15 minutes, given an example of what the assembly should look like.

2. *Top Level Requirement*: Assembly Simulation
   For the simulation to be useful to the users, the simulation times must be accurate, depending on the user's estimated ranges.

   2.1. MAB will simulate the deployment of user created Madeus assemblies.

      2.1.1. The simulated deployment will be within 5 seconds of the total expected assembly deployment time ranges inputted by the user.

      2.1.2. The simulation will display an accurate elapsed timer for each running component.

3. *Top Level Requirement*: Saving and Loading Assemblies
   Our software must be capable of saving and loading Assemblies that have been saved to .yaml files in a timely manner.

   3.1. Saving a user's Assembly to a .yaml file should take no longer than 10 seconds for any assembly composed of fewer than 250 graphical parts as defined in the MAB GUI

3.2. The GUI object that the user must interact with to save their assembly must be accessible within 3 mouse clicks from the Assembly Development screen

4.4. Environmental Requirements

1. *Top Level Requirement*: Generated Code
Since MAD is based in Python, the code our GUI generates must be in Python with proper MAD syntax.

    1.1. The MAD Assembly Builder will be able to generate code on any hardware it is installed on.
        1.1.1. Generate executable Python code from user created diagrams.

2. *Top Level Requirement*: Cross-Platform
MAB must be able to be used on all major operating systems in order to reach as many potential users as possible.

    2.1. The MAD Assembly Builder will be able to execute on multiple software platforms. These software platforms include:
        2.1.1. Linux (Ubuntu / Kubuntu / Xubuntu / Lubuntu (Saucy and above))
        2.1.2. Windows (Windows 10)
        2.1.3. OS X 10.7 and above (64-bit)

    2.2. MAB will not require any external dependencies or installation requirements.

5. Potential Risks

Although the Madeus model is strictly defined, its use can potentially involve the distribution of complex software onto complex systems. Improper distribution of software onto these systems could be disastrous; because of this, it's important to explicitly analyze any potential risks that could come as a result of our work. More specifically, we need to make sure our GUI provides an accurate description of the Madeus model. We will define three potential risks we anticipate for our project and discuss the repercussions associated with these risks.

1. **Generated Code may not accurately represent the GUI diagram (*Impact Rating 4/5*)**

    Likelihood:

This risk likeliness would be increased with the complexity of the MAD model.

Severity:
If the generated MAD code has an extra transition or place that was not in the user created diagram, the user could not rely on the software to accurately create their deployments. It would result in incorrectly deployed software or users not using our GUI.

Mitigation:
We will mitigate this risk by testing all edge cases that may break our code generation.

2. **Maintaining Software integrity with the addition of Plugins (*Impact Rating 3/5*)**

   MAB needs to be extensible with plugins without compromising its basic functionality. A plugin needs to be modular and not require another part of the software to be modified or removed to function.

   Likelihood:
   This risk is associated with developers that wish to expand on our software.

   Severity:
   This risk could cause our software to become corrupted and not function as intended.

   Mitigation:
   We will mitigate this risk by packaging our software together with its core fundamentals that aren't mutable by the user, however, additional plugins may rely on fundamentals of the GUI that we create.

3. **Simulation time may be inaccurate (*Impact Rating 2/5*)**

   Likelihood:
   This risk is associated with the users hardware. The performance of the simulation needs to take into account a hardware requirements, and its ability to fluently simulate software deployments in the correct time.

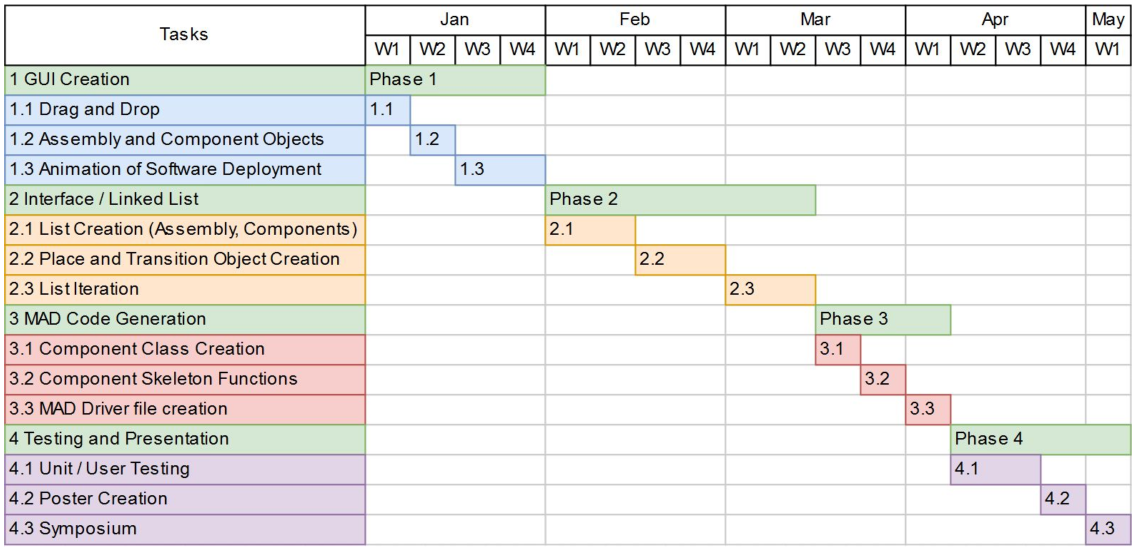
   Severity:

The overall simulation time will have an affect on the users ability to estimate deployment times for their deployment models.

Mitigation:
This risk will be mitigated by maximizing the performance and minimizing the overhead of the animation in order to accurately portray the deployment time of the accumulated sleeps at each transition.

6. Project Plan

Our project execution plan was developed with milestones in mind. We created the Gantt chart below (Figure 2) to help illustrate our development schedule. We separated our development schedule into phases, each phase represents a milestone in our development. These milestones include GUI creation, Interface / Linked List Creation, Generation of MAD code, and Testing / Evaluation.



Tasks	Jan				Feb				Mar				Apr				May
	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4	W1	W2	W3	W4	W1
1 GUI Creation	Phase 1																
1.1 Drag and Drop	1.1																
1.2 Assembly and Component Objects		1.2															
1.3 Animation of Software Deployment			1.3														
2 Interface / Linked List					Phase 2												
2.1 List Creation (Assembly, Components)					2.1												
2.2 Place and Transition Object Creation							2.2										
2.3 List Iteration									2.3								
3 MAD Code Generation											Phase 3						
3.1 Component Class Creation											3.1						
3.2 Component Skeleton Functions												3.2					
3.3 MAD Driver file creation													3.3				
4 Testing and Presentation														Phase 4			
4.1 Unit / User Testing														4.1			
4.2 Poster Creation																4.2	
4.3 Symposium																	4.3

Figure 2: High Level Gantt Chart

The Gantt chart above serves as a high level overview illustrating our development schedule going into the spring semester. Our Prework phase consists of tasks we were carrying out this semester and will conclude with our technology demo showcasing each top level requirement.

Beginning spring semester we separated the development schedule into 4 phases each representing a milestone in the development. Phase One will be our GUI creation. GUI creation will include Drag and Drop functionality, Assembly saving and loading, and assembly deployment simulation. Phase Two is our interface / linked list creation that is created dynamically in regards to the GUI. Our interface will include a data structure that acts as the "software glue" between our GUI and the generated MAD code. Phase Three will be creating the functionality of the interface that generates the MAD code. This includes a set of places, a set of transitions, and a set of dependencies. As well as functions for each transition that will hold the sleeps inputted by the user. The time intervals for each phase were assigned by our estimate of the required time it would take to accomplish each milestone. This Gantt chart is exceedingly high level and is tentative. Certain phases may be reduced or extended or even overlap with one another.

7. 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.

This Requirements Specification document aims to clearly define our sponsor’s problem statement, our proposed vision for the solution, our Functional, Performance, and Environmental requirements, our project plans, and any potential risks associated with our project. We are confident with our planning and development schedule that we will build a Graphical User Interface that satisfies our sponsor’s needs.

8. Glossaries and Appendices

1. *Glossary*

Madeus: A theoretical model for software deployment developed by researchers at Inria
MAD: Madeus Application Deployer. The formal implementation of Madeus, also developed by researchers at Inria. Written using the Python programming language
MAB: MAD Assembly Builder. The current title for the software being developed by Team Amadeus at NAU. MAB is a graphical user interface for developers to create MAD assemblies using a drag-and-drop system, as well as grant developers the ability to simulate the software deployment process.
Assembly: A collection of Components, which represents how the Components are connected and work together through Dependencies.
Component: A piece of the Madeus model, located inside of Assemblies. These are the largest pieces of the assemblies, and vary in complexity. They are representative of the software parts that must be deployed. A Component's deployment is represented by Places, Transitions, and Dependencies.
Place: A piece of the Madeus model, located inside of Components. These are representative of the state the software deployment is at in a component.
Transition: A piece of the Madeus model, located inside of Components. These are how the deployment process goes from step to step. They contain the functionality of the deployment, but in MAB they are given placeholder sleep functions for the purposes of simulation.
Dock: A piece of the Madeus model, located inside of Components. These are the implied links between Places and Transitions, which are represented in MAB as the pointers in our linked list data structure used for representing the MAD assembly.
Dependencies: The requirements for a Component to be deployed. In Madeus all Dependencies are broken down into two types: Data, and Service. Each Dependency instance must have a Provider and a User.

2. *Appendices*

Official MAD Documentation- https://mad.readthedocs.io/en/latest/
MAD Repository on GitLab- https://gitlab.inria.fr/Madeus/mad
Team Amadeus Website-
https://www.cefns.nau.edu/capstone/projects/CS/2019/Amadeus-S19/