University of Arkansas – CSCE Department
Capstone II – Preliminary Report – Spring 2024

Software Platform for Modular, Multiscale Simulations of Large, Complex Biological Systems: BYOSim

Benton Anderson, Bishop Butler, Zachary Harris,

Rithyka Heng, Isaac Withrow

Abstract

Biological systems can be highly complex and highly multi-scale, so building computational models of biological systems can be a challenge. These models are the result of a multitude of years of work done by a research team, and they are often made in such a highly specific manner that they become incompatible with models from other research teams. Differences in programming languages are another large issue that contribute to the inability to simulate multiple different models in conjunction.

However, if a platform could be developed to streamline the process, then researchers would not need to spend nearly as much time developing these models. Instead, they would be able to utilize pre-existing models in their simulations, speeding up research in their field as they wouldn't have to develop these models all on their own.

To accomplish this, we will devise a framework and software that allows for the variables, or outputs, of one model to be passed in as parameters, or inputs, of another model, regardless of the programming language or format of the biological model. Though other solutions have been proposed, ours is unique in its use of backtracking and equilibrium detection. It will simulate all models as they affect each other. The solution should be user-friendly, performant, and general, accepting all types of models, regardless of their implementation.

1.0 Problem

Running multiscale biological simulations with multiple models is resource intensive and incredibly costly. This is because there is no efficient and effective way to run the sorts of simulations that combine many different individual biological models. Different models may use different programming languages, or some may not be written in a programming language at all, and with this language barrier, it can be difficult to integrate various models. Even if the language obstacle is overcome, the problem of how to run these models together arises.

The creation of computational biological models is a time-consuming process that can take several years to decades to implement. These models are often complex, spanning a wide range of both temporal and spatial scales from intermolecular interactions to full-tissue simulations. Researchers are often focused on different highly specific phenomena, so they create models, possibly in many different modeling engines. They are unable to use the implementations of others because there is no efficient nor effective way to get their models to communicate with the pre-existing implementations and simulate them together. Thus, researchers must implement the other models themselves to be able to run a simulation with their own models. Because of this, model implementation continues to remain a time-consuming and resource-intensive process, leading to slower progress in the field of biology. If there is no solution to this problem, research that could be done to save lives may not be possible because large and complex systems are difficult to build models for on a piece-by-piece level. Having a tool that allows for something like the human body to be modeled in this way could lead to breakthroughs in medicinal fields and, in the long run, provide the ability for researchers to advance their research at a faster pace.

The long-term goal of computational biology is to be able to treat diseases and improve the health of patients with targeted prescriptions. If we are able to simulate an entire organism, then we could run simulations using different treatments and determine the most effective treatment for each patient. These simulations would be able to use models that represent a medical digital twin of the patient and determine the effects of each treatment on the patient based on the results of the medical digital twin's simulation results. Because of this long-term goal, one of the short-term goals of computational biology is to be able to simulate multiple models on a smaller scale. But, without a tool that allows researchers to quickly test different models, this is very difficult.

2.0 Objective

This project's objective is to reduce the cost of constructing complex multi-scale biological models. Our solution focuses on allowing users to utilize pre-existing models when constructing their own biological model. The framework that handles this should not introduce too much overhead, and the pre-existing models should require minimal modifications to be compatible with other models. The goal is to have the solution be user-friendly and performant: users should be able to get results quickly. There should be minimal friction in setting up their program, and the simulation should remain computationally efficient regardless of the model's complexity and range of scales.

Any researcher should be able to plug-and-play in our interface; users will be able to input models of any number of programming or modeling languages (along with the corresponding simulator), specify the simulation details in a configuration file, and run the simulation. The simulation details will indicate which parameters and variables match up across the different models as well as the timescales for the simulation. It is also important that all data involved in the simulation be formatted in a flexible yet standardized manner: this includes not only the configuration format but also the format of the data produced by simulations and data being passed between different modules. It should be straightforward to interpret and modify, if necessary.

Through the use of equilibrium detection and backtracking, our interface should be more efficient than other attempts at a biological model integration platform. When a model's variables are stable, then it is more time efficient to stop simulating that model and start simulating another instead of just letting the model run. And, because the model's variables are stable, the resulting inaccuracy will be negligible. Our application will utilize equilibrium detection to determine when a model's variables are stable and switch to another model. Equilibrium detection will lead to an efficient solution because of the time saved. Backtracking allows our application to maintain accuracy and performance. When a significant change is detected in a model, the simulation will pause that model and send the relevant information to the models with parameters that depend on the variable that changed. The other models, which have completed their simulation time step, will backtrack to the time where the significant change was detected. With backtracking and equilibrium detection, our application will be performant and maintain its accuracy.

3.0 Background

3.1 Key Concepts

To understand this project's conceptual solution, its implementation, and its design choices, one must understand some key concepts of computational biological systems modeling and software technology. The relevant key concepts of computational biological systems modeling are the interpretation of models as functions, the idea of interdependence, and the organization of different systems into a hierarchy dependent on time scales. With these concepts, the idea of backtracking, a major component of this project's conceptual solution, can be explained. The relevant key concepts of software technology are application programming interfaces, the client-server model, multithreading, inter-process communication, and open-source code distribution. These concepts help explain the organization and implementation of the project solution.

Computational models of biological systems can be viewed as functions—that is, their outputs depend solely on their inputs. They are deterministic and have no internal state that could affect their outputs. This idea of model outputs being determined only by their inputs will be integral for the backtracking solution described later.

Interdependence is the idea that the entities in a system are dependent upon one another in a cyclic manner. For this project, we consider one entity dependent upon another if the output of the independent entity in some way determines the output of the dependent entity. The output of the independent entity does not necessarily need to be passed directly into the input of the dependent entity—the data may undergo some transformation while maintaining the dependency. A simple example of interdependence follows: entity A, whose input is X_A and output is Y_A, and entity B, whose input is X_B and output is Y_B, are interdependent when $X_A = f(Y_B)$ and $X_B = g(Y_A)$, where f and g are non-constant functions.

Models can be organized into a hierarchy depending on their time scale. More specifically, interactions that occur over longer periods of time can be set on top, with smaller time scale interactions beneath, assuming that the various models are dependent on one another. It is an atomistic view of the world. As a generic example, take a model simulating a brain and its related systems. There are interactions that occur over milliseconds or less, like the release and reception of neurotransmitters, while there are also interactions that occur over seconds or minutes, like the flow of blood into the brain. They are related and interdependent but occur on different time scales, so it is useful (for optimization reasons described later) to separate them on that basis while still representing and maintaining their connections and dependencies.

Computational models will, for the foreseeable future, always have a trade-off between precision and performance. Backtracking, a core component of this project’s solution, attempts to improve performance by reducing the quantity of computations performed during any given time period. It has one key assumption on top of the concepts previously discussed: “insignificant” changes in any model’s outputs can be ignored. This enables two optimizations: the ability to stop simulating models that have reached “equilibrium” and the ability to run dependent models as though they were independent, albeit for small periods of time. The definitions of “significant change” and “equilibrium” are determined by the user. Thus, backtracking can be summarized as follows: 1) interdependent models can be run simultaneously with fixed inputs until those inputs experience some user-defined significant change, at which point the models will be reverted to the time of the significant change and rerun with the updated inputs, and 2) when models reach a user-defined equilibrium, they will no longer be simulated and their outputs will be held constant until at least one of their inputs experience a user-defined significant change.

These key ideas of computational biological systems modeling are vital to our project and the completion of our objective. There are also important concepts that are key to many software applications that we also utilize. The software engineering concepts mentioned at the beginning of this section are commonly used in all kinds of software applications, and our project is no exception.

An application programming interface, or API, is an interface for programs to utilize one another’s functionality. Its purpose is to decouple programs so that they can be reused elsewhere as easily as possible. More generally, however, it is a description of the functionality of some program—how to use it, what data it needs, what it returns, etc. This allows for a set of standardized functions to be available in any environment and in any language, given that the API is defined precisely enough. Because of this, it was decided that API design was just as, if not more, crucial than the implementation.

The client-server model is a common software structure which consists of server processes and client processes. Typically, the client requests some resource or service from a server across a network. However, our project utilizes the server as a sort of “conductor” or “orchestrator” that commands the clients and relays information between them, and the server and clients will exist as distinct threads on the same machine. Essentially, our solution, the server, will coordinate the user-defined models, the clients. This centralizes control and simplifies the inter-model interactions.

Multithreading is the division of a process into multiple concurrent “threads” of execution. A single processor can emulate the ability to perform the work of multiple processors in this way by taking advantage of the relatively long idle times when accessing memory. When stalling or idling, the processor can switch to another thread and perform its instructions until the scheduler decides to switch threads once again. This technique can greatly enhance performance, and it is a necessity when simulating models of the scale this project targets.

Inter-process communication is the sharing of data between separate processes or threads. There are various methods of inter-process communication, such as shared memory or pipes, but our solution makes use of network sockets for its flexibility and simple usage. Sockets are typically used to allow different machines to communicate over a network, but they are also capable of same-machine communication. Of course, there are different types of sockets with different properties. Our solution utilizes stream sockets which exist on the TCP layer of the network stack. Although a connection must be established between the two parties, it ensures in-order and reliable, or error-correcting, data transmission. Although it may be slower than other solutions like shared memory, it is much simpler and safer to manage and is likely to work on any system and in any language.

Our solution is planned to be released as an open-source project on GitHub. This means that our API design documentation, implementation, and other related deliverables will be available to view, download, and extend online for free. We decided to use GitHub, an online repository that stores countless open-source projects, among other things, for its popularity and so that users would be familiar with its layout.

These software technology concepts help maintain a robust and efficient design and implementation focused on this project's objective. They form the foundation and guiding principles of our solution.

3.2 Related Work

We are not the first to tackle the problem of computational biological model integration. Vivarium [1] offers a platform for integrating multiple mechanistic biological models and running them together. They use modular design methodologies and utilize a plugin system to allow users to easily integrate existing systems with newly developed models. The platform includes a simulation engine that takes in models as input and integrates these models to run them as coupled systems over various timescales. Vivarium, however, runs the integrated system step-by-step. Our proposal is to implement equilibrium detection and backtracking instead of a similar step-by-step system. This reduces the computations made and increases the simulation efficiency compared to Vivarium’s step-by-step strategy.

SBML, also known as the Systems Biology Modeling Language [2], is a file format for depicting models in a form that can communicate with other systems. It was designed to create a standard language for computational biological models. This represents one approach to the problems discussed above. To ensure that different models can work together as an integrated whole, researchers could agree to use a common modeling language. That is the goal of SBML: to provide a common modeling language. Our approach, on the other hand, involves a different strategy; our objective, as described above, is to develop an interface in which users can input models with different languages and simulate these models together, regardless of the language.

Tellurium [3] provides an interface to a simulation engine. It allows users to build models in Antimony, a human-readable version of SBML. The simulation engine can then simulate these SBML models. There are a few differences between this interface and our proposed interface. First, we want users to be able to plug-and-play without the necessity of developing a model inside of the interface. Second, we want the user to be allowed to use models in multiple programming and modeling languages. Third, we want to be able to integrate different pre-existent models and have them communicate with each other in one simulation.

4.0 Approach/Design

4.1 Use Cases & Requirements

Use Cases:

- Researchers will be able to plug-and-play multiple biological models instead of developing their own platform to simulate the models.
- Accessible for government researchers, university researchers, and any other interested parties via GitHub.
- Running the model simulations will allow researchers to test hypotheses by saving the results of the model variables, which the researchers can use to compare with expected results.
- Platform will run simulations in a time and resource efficient manner and make predictions that researchers can use to diagnose diseases and develop treatments.
- Researchers can simulate multiple biological levels, such as cells and organelles, by setting appropriate time steps for each model and connecting the correct models.
- Multiple computational biological models can be run such that outputs, or variables, of some models will be inputs, or parameters, of other models.

Requirements:

- User-friendly
  - Minimal user modification to existing models
  - Minimal user definition of code (some will be necessary)
- Minimize resource consumption
- Limit wasted simulation time
- Implementation-agnostic model support

The target audience for our solution is biological researchers that study various unique biological phenomena or the interactions between them. The main use case for this will be the simulation of multiple different models, regardless of their implementation, in a way that the variables from one module become the parameters of another module, and the simulation for the model that just had its parameters set is run.

4.2 Detailed Architecture

Our solution consists of four main components: a configuration file, user-defined modules, our server back-end, and a web interface. The modules, server, and web interface primarily interact via socket messages, with the configuration file being read and modified by the server.

Solution Infrastructure

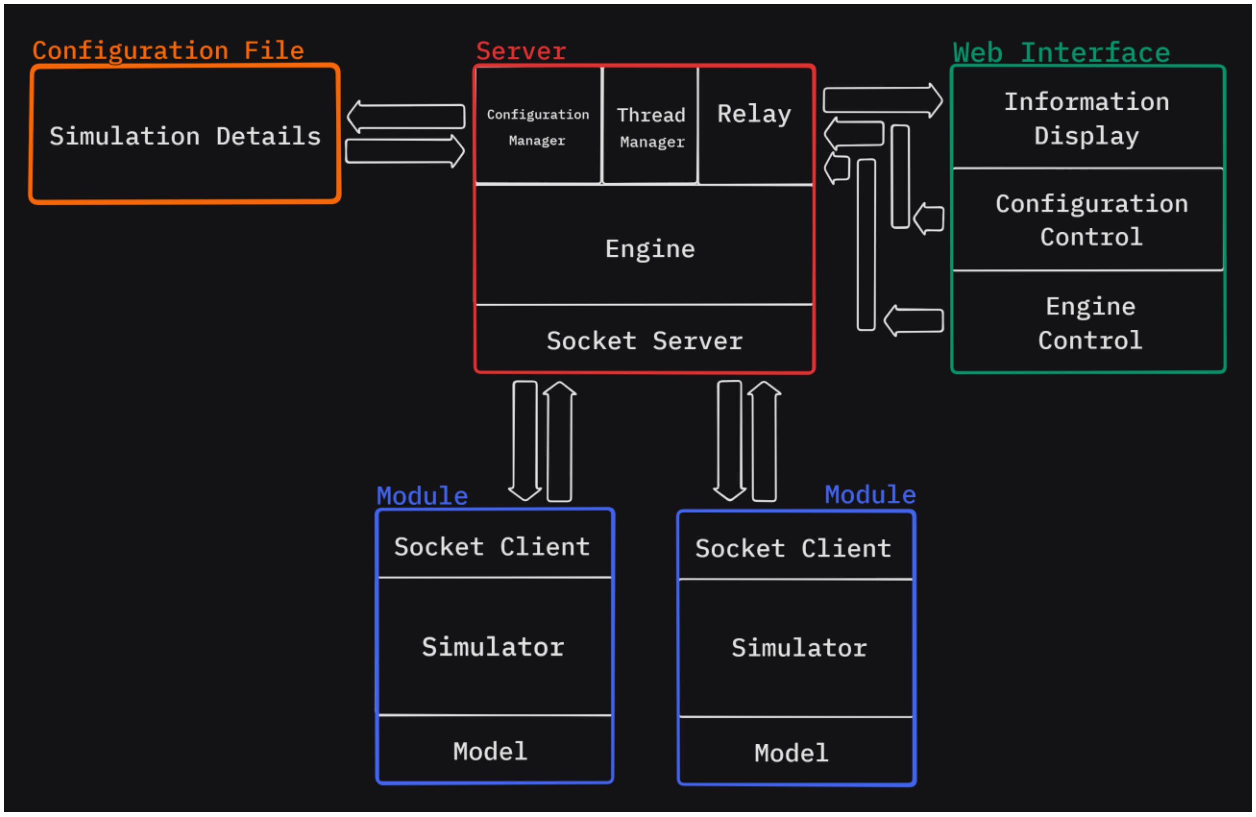


Fig 1. A diagram of solution components and their connections

There are three types of external entities which the server interacts with: configuration files, users (through the web interface), and modules, which wrap the various models and simulators. Each interaction is dealt with by separate components within the server.

The configuration component of the server manages the simulation's configuration file. It imports settings, such as the models to simulate, their variable-parameter connections, and initial parameter values, into the engine component. It also sends these settings to the relay to be displayed on the web interface. It may also export settings set on the web interface into the configuration file. The configuration manager handles these import and export operations, providing a seamless interaction between the configuration file and the server components. This ensures that the simulation remains configurable and adaptable, allowing users to adjust parameters as needed to explore different scenarios and configurations.

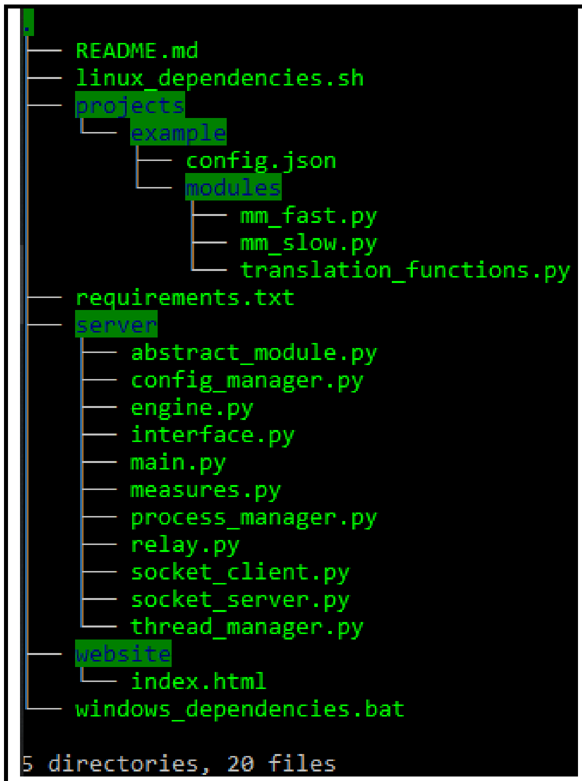
The relay of the server manages the web interface, which is where the user will be able to easily view information about the simulation, configure the simulation settings, and control the simulation. User input received on the web interface is relayed to the engine or configuration components of the server via the relay. Similarly, information to be displayed is relayed from the engine and configuration components to the web interface via the relay.

The socket server component communicates with the user-defined modules via sockets, sending instructions to the simulators and relaying variable information between the models. The specific messages are generated by the engine component based on the simulation configuration and the outputs of the model simulations. To make socket communication easier, socket server and socket client utility classes were made. These have a handful of methods that abstract away the low-level details of creating socket connections, accepting connections, receiving variable-length messages, and sending messages. Instances of the socket server have one primary thread which accepts new socket connections, calls a connection handler function, and starts new threads for each connection. These secondary threads then continuously receive any incoming data from the socket connections and call a message handler function. The connection handler function is a property of the socket server, defined by the creator of the socket server instance, to allow the creator of the socket server instance to change state depending on the details of new connections and to define different message handlers for different socket connections. For example, the engine, with one socket server, defines one message handler for the relay and one message handler for modules. Similarly, the message handler allows the creator of the socket server instance to change state and respond to messages from socket connections. The socket client class is identical to the socket server class except for the fact that instead of having multiple threads, there is only one message receiving / handling thread, since socket clients do not accept connections. The relay also has its own variation of the socket server to support the WebSocket protocol used by the web interface.

Directory Structure

To download and use our solution, users must first clone our repository and add new projects as folders within the `projects` folder. Each project folder must have a `modules` folder and a `config.json` file as its top-level children. The modules folder will hold their module implementations (`.py` files) alongside a copy of the `abstract_module.py` file for their module classes to extend from. Each project folder will also generate an `outputs` folder, within which a new folder will be generated for each simulation run. These simulation result folders will hold a mass of `.json` files storing lists of variable values. The exact format of this is still under development.

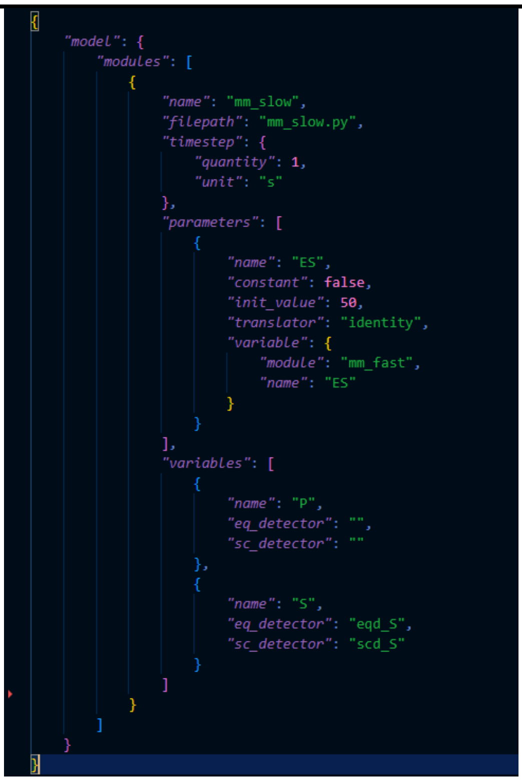
To run a simulation, the user must first run a project. This involves starting the engine and other necessary components required to simulate an integrated model. To run a project, the user must navigate to the root directory of the repository folder and run `main.py`, passing in the name of the project (a directory name) they would like to run. From here, they can run the simulation and make modifications to the configuration via the web interface.



```
.
├── README.md
├── linux_dependencies.sh
├── projects
│   └── example
│       ├── config.json
│       └── modules
│           ├── mm_fast.py
│           ├── mm_slow.py
│           └── translation_functions.py
├── requirements.txt
├── server
│   ├── abstract_module.py
│   ├── config_manager.py
│   ├── engine.py
│   ├── interface.py
│   ├── main.py
│   ├── measures.py
│   ├── process_manager.py
│   ├── relay.py
│   ├── socket_client.py
│   ├── socket_server.py
│   └── thread_manager.py
├── website
│   └── index.html
└── windows_dependencies.bat

5 directories, 20 files
```

Fig 2. The current directory structure



```
{
    "model": {
        "modules": [
            {
                "name": "mm_slow",
                "filepath": "mm_slow.py",
                "timestep": {
                    "quantity": 1,
                    "unit": "s"
                },
                "parameters": [
                    {
                        "name": "ES",
                        "constant": false,
                        "init_value": 50,
                        "translator": "identity",
                        "variable": {
                            "module": "mm_fast",
                            "name": "ES"
                        }
                    }
                ],
                "variables": [
                    {
                        "name": "P",
                        "eq_detector": "",
                        "sc_detector": ""
                    },
                    {
                        "name": "S",
                        "eq_detector": "eqd_S",
                        "sc_detector": "scd_S"
                    }
                ]
            }
        ]
    }
}
```

Fig 3. A minimal template configuration file

Configuration Files

The configuration files are JSON files which define the integrated model by storing details about the inter-model connections and the user-defined module methods. The configuration files are written to by the relay and read by the relay and the engine. It is not recommended to manually modify these files: instead, the configurations should be modified via the web interface.

There are two configuration files: the server file and the configuration file proper. The former stores a server object, and the latter stores a list of modules. The server object includes an engine object and a webserver object. The engine object includes the socket object with the address and port number for the engine. The webserver object includes an http object and a socket object. Both objects contain an address and port number for the servers.

The modules list stores module objects: these consist of the name of the module, the module file path, the timescale that will be used to run the module simulation for that amount of time, and two lists. The first list is a list of parameter objects that will serve as the model's inputs. The parameter objects include the parameter name, the initial value, the name of the translator function, and the variable it will be passed into. The second list is a list of variable objects. The variable objects include the name of the variable as well as the names of the equilibrium and significant change detection functions.

Module

A module is a user-defined class implementing our abstract module class definition. It packages a model, a simulator, and other functions necessary for integrated simulation into one object with a standard interface for our engine to use. These modules will be instantiated and run by the engine in their own threads.

The abstract module class is a tool that helps users define the modules they want to add to our program. It has basic abstract methods that they implement to simulate their models. It also has methods that our engine directly accesses that help simulate the model, save variables, and assess outputs, significant change and equilibrium detection. We have already implemented these functions along with a function that saves the module variables into a file using Json.

The module class contains seven components: the model, the simulator, the parameters and its values, the variables and its values, the translation functions, the detector functions, and a socket client. The model refers to the representation or analogy of the phenomenon being observed, or the computations which transform inputs to outputs. The simulator refers to the entity which performs the model computations. The parameters are the model's inputs, usually a constant or the translated output of another model, and the variables are the model's outputs. The translation functions are what translate one model’s outputs into a useful input value for another model; they each have only one input and one output, meaning that they cannot aggregate the values of multiple variables nor produce multiple parameter values on its own. The detector functions detect if some variable has reached equilibrium or if some parameter (pre-translation) has experienced a significant change. The socket client connects to the engine to allow for multi-threaded communication.

Modules only understand one command: “simulate”. It comes with the start and end times between which the module should simulate and a dictionary storing the untranslated input parameter values that the module should use after translation. The module will automatically translate these raw parameter values before storing them in the parameters buffer and passing them into the user-defined simulate method. The simulate method can either return or set the resulting variable values into the variables buffer. These values are then sent to the web interface via the engine and relay for the user to see. The modules also automatically run their significant change and equilibrium detection functions upon completion of their simulation task, sending a message to the engine containing a list of variables that have experienced a significant change or equilibrium, the time of the detection, and the value of the variable at that detection time. Significant changes overpower equilibrium detections if they are not disjoint. If there are no detections, no message will be sent.

In extending the abstract module class, the user must implement at least the simulate method, one translator, and one detector alongside the minimal constructor with parameter-translator mappings. They must also call the abstract module class’s constructor as well as a “ready” function at the end of their initialization procedure to let the engine know that it is ready to simulate. They are otherwise free to modify the module initialization function and add any additional methods, translators, and detectors as necessary.

Server

The server is the “back-end” of our solution, the “main” program that starts and controls everything else. It consists of the thread and process managers, the model and server configuration managers, the engine, and the relay. The thread manager and process manager are static utility classes that make it easy to spawn, keep track of, and kill the various threads and processes required for the engine, web server, and modules. Modules run in processes, while everything else runs in threads. The configuration managers are other static utility classes that provide one channel of access to the project's configuration. The model configuration accesses the project-specific configuration specifying details about the modules and their connections, and the server configuration accesses a configuration specifying details about socket and webserver ports and addresses. These utility classes are initialized once the program starts, and because they are static classes, their functionalities are globally accessible. The engine orchestrates the simulation of the integrated model by controlling the modules, relaying information between them, and consolidating the results of the simulation for the user to view. The relay is the intermediary between the web interface and the integrated model and its simulation. The server will typically run on the user's machine as one multi-threaded process.

The server begins by reading the first command-line argument, specifying the name of the project the user wants to work on. It checks whether the project directory and configuration file exist before validating the configuration file, generating one if it doesn't already exist. Then, it initializes the thread manager and configuration manager before starting the engine and relay in their own threads. Lastly, it opens the web interface in the user's default browser.

The engine, running in its own thread, starts a socket server and listens for messages from modules or the relay. When the simulation is inactive, the engine only receives messages from the relay. The relay relays the user's commands to the engine, commands like "start simulation". When the simulation is active, the engine receives messages from both the relay and the modules. Before it does anything, however, the engine must ensure any old processes from previous simulation runs are killed, reset any integrated model information, read the configuration file, and create a new output folder immediately after the simulation is ordered to begin. This allows the engine to start the modules' threads and build a dependency graph for all the model's variables and parameters. Once this succeeds, then the engine can begin its primary duty of controlling the simulation of the integrated model. It will instruct modules to simulate for specific periods of time and to detect if the module's model has reached equilibrium or experienced a significant change. In return, the modules will give the engine data about itself, information like the values of its variables, the results of equilibrium detection, and the results of significant change detection. The engine then relays those variable values to the modules that require it, disables the simulation of modules that have reached equilibrium, and backtracks the modules dependent on those which have experienced a significant change.

In a little bit more detail, simulation initialization involves killing all module processes related to the last simulation (if it exists), validating and extracting data from the project's configuration file, creating a new folder for simulation outputs, and initializing model and module information. Model information includes building a dependency graph where nodes are modules and directed edges are labeled with variable names. If module A points to module B via an edge named "S", that means that module B has at least one parameter which depends on the value of S from module A. Module information includes a Boolean signifying whether the module is awake or asleep (for the sleeping optimization), a time-sorted queue of target times that the module should simulate to, and a store of variable output values, labeled by time. Starting the modules involves creating a new process to first import (using the filepath listed in the configuration file) then instantiate the module, kicking off the module's socket connection, listening loop, and any other model initialization procedures. After all of this, the engine sets its `waiting` flag to false and finally enters the simulation loop.

The simulation loop only runs when the engine's active flag is true, and it is more a sort of multi-threaded back-and-forth "conversation" between the engine and the various modules than a traditional loop. It begins with the engine commanding every module to simulate for one timestep (in whatever unit that module's timestep is). The modules will return variable values, which the engine will store. As the engine receives variable data, it will build up parameter data for modules that are `waiting`. If a significant change is detected, the engine checks the graph generated before and tells all modules that are children of the module (in which the significant change) along an edge whose name matches a variable that experienced a significant change occurred to backtrack. Backtracking involves adding the time of the significant change to the list of target times that the dependent module should simulate to. Eventually, the module will reach the closest timestep-aligned time before the significant change, simulating until the significant change with the old parameters, then simulate from the time of the significant change to the next aligned timestep with the new parameters.

The relay is a python program designed to act as an interface between the website code and the rest of the underlying python code. The relay will receive simulation data from the engine and send them to the web interface to visualize and will receive configuration data and simulation controls from the interface and send them to the engine so that the simulations can be controlled, and the configuration file can be updated and used by the engine. We have completed the relay; the relay can send messages from the interface to the engine and messages from the engine to the interface. It uses sockets to send to and receive from the engine, and it uses WebSockets [4] and asyncio [5] to send to and receive from the web interface. When the user uploads a module file via the interface, the relay gets the module information and sends it to the interface. When the user starts the simulation or wants to generate a configuration file, the relevant information is passed to the relay, and the relay puts the configuration settings in the configuration file and sends the simulation control to the engine.

Web Interface

The web interface is the "front-end" of our solution and primary location for user input and interaction; a webpage that allows the user to expose the module information and then modify the parts integrated model, by specifying that models parameter connections to the other modules, control its simulation, and view the results of the simulation and other information regarding it. It is accessed via the web browser and the index.html file will be opened upon running main.py. The web interface is programmed using a mix of HTML for the webpage, and JavaScript for the functions and underlying dynamic creation for the HTML elements, as well as the WebSocket communications to the python relay.

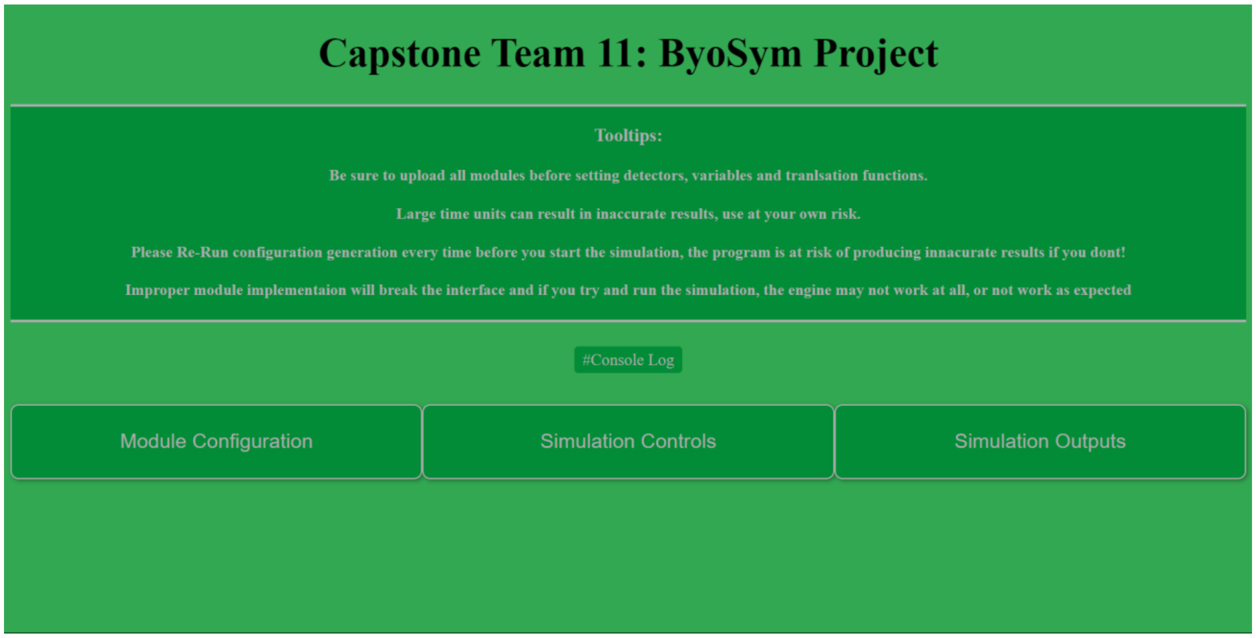
The web interface features three main tabs. The first is the module view (Fig 5.), which contains a place for users to upload modules, and modify simulation configurations. The second is the simulation control (Fig 6.), which houses the buttons that can control the simulations execution, giving the user more direct control over the engine. The last view is the simulation output (Fig 7.), which contains output views per variable. There are three buttons to switch between each view, which will hide all views except for the one that it requested. The main view also contains the project title “BYO Sim” which stands for “Bring Your Own Simulation(s)”, and it also contains a set of tooltips for the user for when they use the interface. There is also a little HTML text element that acts as the console, showing the user that their requested operation was completed when they press a button.

The module view is the main section of the web interface, where the user sets up the configuration of modules for the simulation (Fig 5). The user starts with the upload module button, which prompts them to select a file. Once the user selects this file, they click the upload module button, which sends the name of the file over to the python relay via a WebSocket, which gathers the data needs from them module. The data it gathers is the dictionaries containing: the parameters and translation functions, and then the detectors and variables. It also returns the name of the module, which is used as the button name, and in the config generation function. It then sends this data back to the web interface and creates a hidden new module div and a button to show it. It does this by dynamically creating all required elements HTML elements with JavaScript, and it does the same for each of the parameters. The JavaScript creates a new div for the module, along with a new button for it, containing the name of the module. Within the div also resides hidden HTML elements that denote the file name and the module name. While the user cannot see that, what they will be able to see is what is created next. The JavaScript will take the list of parameters that it obtained from the relay and loop through them, creating another div containing the text box, and 3 dropdowns. Each parameter div has a unique ID to identify it, for example “Module_x_paramx”, and this is used when the getData function is called. When the user clicks on that module button, they can see all parameters that are contained inside of that module, along with options for that parameter. There is a text box for setting the parameters initial value, and 3 drop down lists. These lists contain all the variables from all modules, all translation functions, and all available modules. The variables list is used to set the connection between a variable in one module, to the parameter that is in the current parameter div, and the module dropdown is used to select the module that variable is from. The translation module contains the list of translation functions that can be used to translate parameters between modules. These lists are kept the same across all modules by constantly updating a master list in the JavaScript code, that then allows the lists to get re-updated every time a new module is added. There is also a time unit selector, allowing the user to select the time unit for their simulation (Days, Hours, Seconds, Minutes, etc.) and set a unit of that time the simulation will run for. The user can upload any number of modules and set their options for each inside of their respective tabs. Next, the user can generate the simulation config by clicking the “Generate Config” button. This calls a function called getData which loops through all module divs’ and each parameter div inside of them and generates a JSON string containing the information needed from each module, and then sends that string to the python relay. The python relay then takes this information and passes it along to the configuration manager, which writes the simulation config. The user can then move onto the simulation control view, where they can send commands to the engine.

The simulation view contains four buttons. These buttons are for controlling the simulation, and each one sends a WebSocket message to our python relay. Those 4 buttons are: “Start Simulation”, “Stop Simulation”, “Pause Simulation”, and “Resume Simulation”. These controls offer the user more control over the simulation. When a button is pressed, a message with the control is sent to the relay, which then calls the engine to do the requested function. Then the simulation runs and will return the output to the web interface via the relay. A view of the simulation control screen can be seen in Fig 6. After that, the user can move onto the output view.

The output view contains a drop-down list containing all variables from the simulation. The user can then select a dropdown list option and view a value-time graph on the web interface for that variable, assuming that the data type of that variable is a number. We also offer the user a place to view a dump of all the data generated in the simulation within their project directory. A view of this page can be found in Fig 7.

At the very end we were able to get out styling finished, to add a cleaner look to the web interface that is slightly easier on our user's eyes, that can be viewed by either opening index.html or in the Figures below. As for unimplemented features and plans, we were not able to finish the backtracking and significant change and equilibrium detection. Future work would involve completing these algorithms and enhancing the user experience of our project.


Capstone Team 11: ByoSym Project

Tooltips:

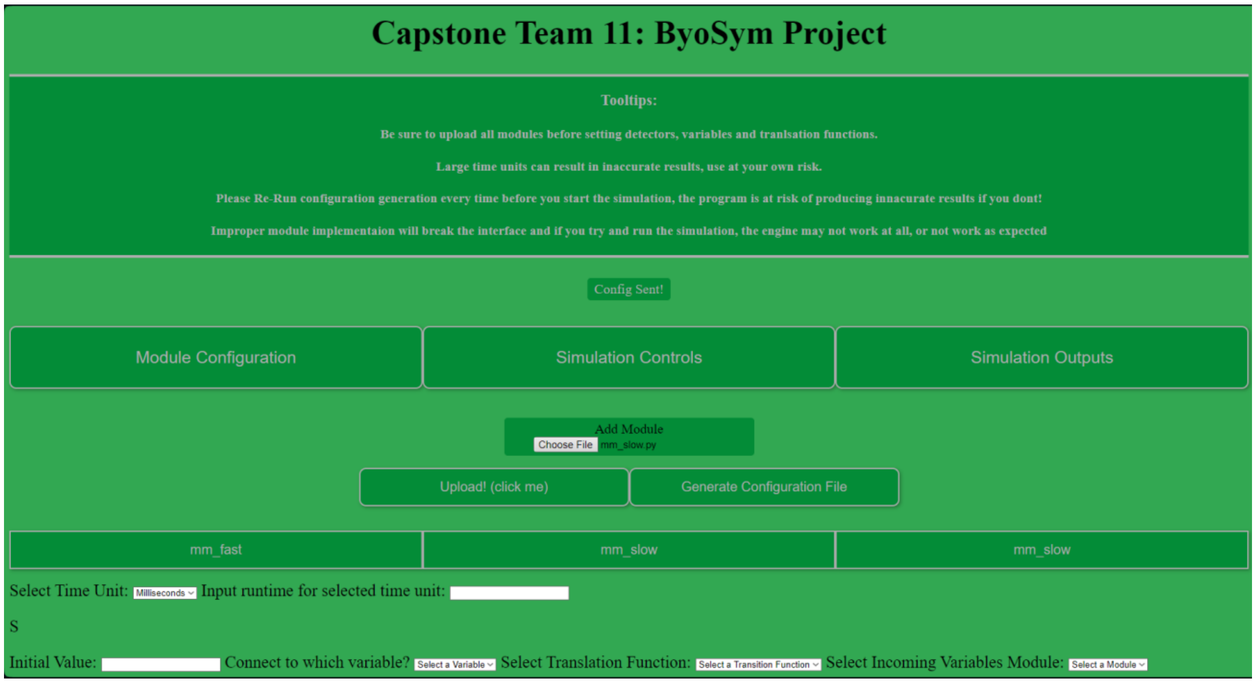
Be sure to upload all modules before setting detectors, variables and tranlsation functions.

Large time units can result in inaccurate results, use at your own risk.

Please Re-Run configuration generation every time before you start the simulation, the program is at risk of producing innacurate results if you dont!

Improper module implementaion will break the interface and if you try and run the simulation, the engine may not work at all, or not work as expected

Fig 4. Initial web interface screen



Capstone Team 11: ByoSym Project

Tooltips:

Be sure to upload all modules before setting detectors, variables and tranlsation functions.

Large time units can result in inaccurate results, use at your own risk.

Please Re-Run configuration generation every time before you start the simulation, the program is at risk of producing innacurate results if you dont!

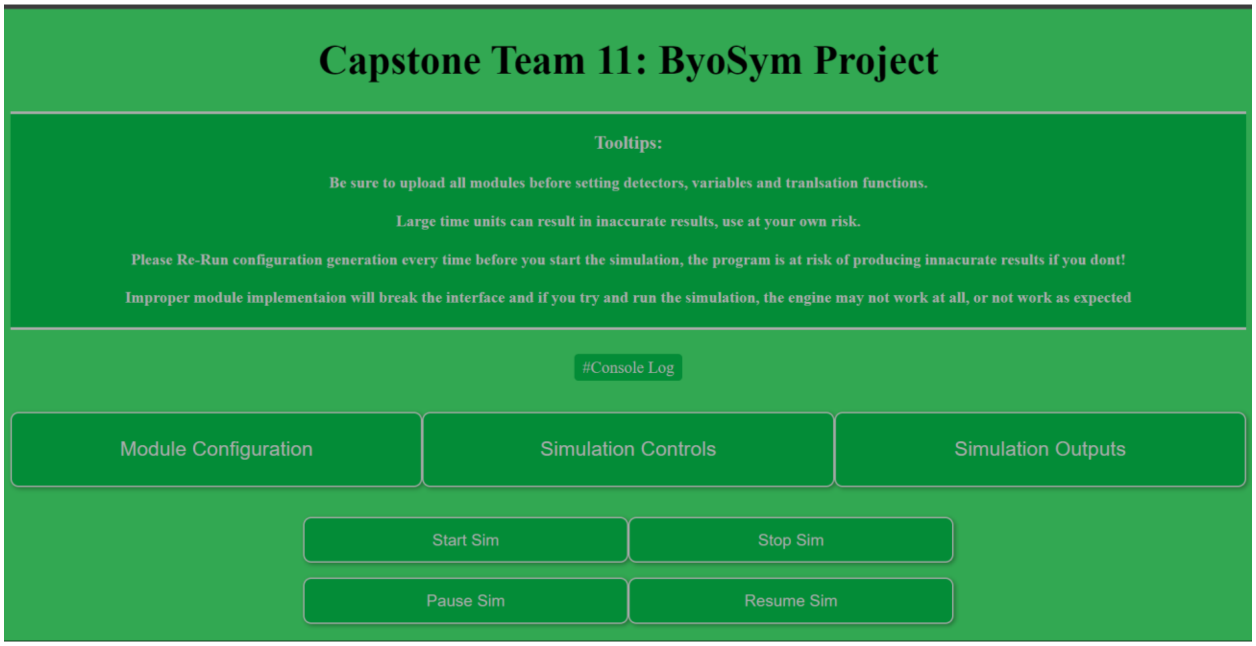
Improper module implementaion will break the interface and if you try and run the simulation, the engine may not work at all, or not work as expected

Select Time Unit: Milliseconds Input runtime for selected time unit:

S

Initial Value: Connect to which variable? Select a Variable Select Translation Function: Select a Transition Function Select Incoming Variables Module: Select a Module

Fig 5. Module screen with sample modules/parameters created



Capstone Team 11: ByoSym Project

Tooltips:

Be sure to upload all modules before setting detectors, variables and tranlsation functions.

Large time units can result in inaccurate results, use at your own risk.

Please Re-Run configuration generation every time before you start the simulation, the program is at risk of producing innacurate results if you dont!

Improper module implementaion will break the interface and if you try and run the simulation, the engine may not work at all, or not work as expected

Fig 6. Simulation Control Screen

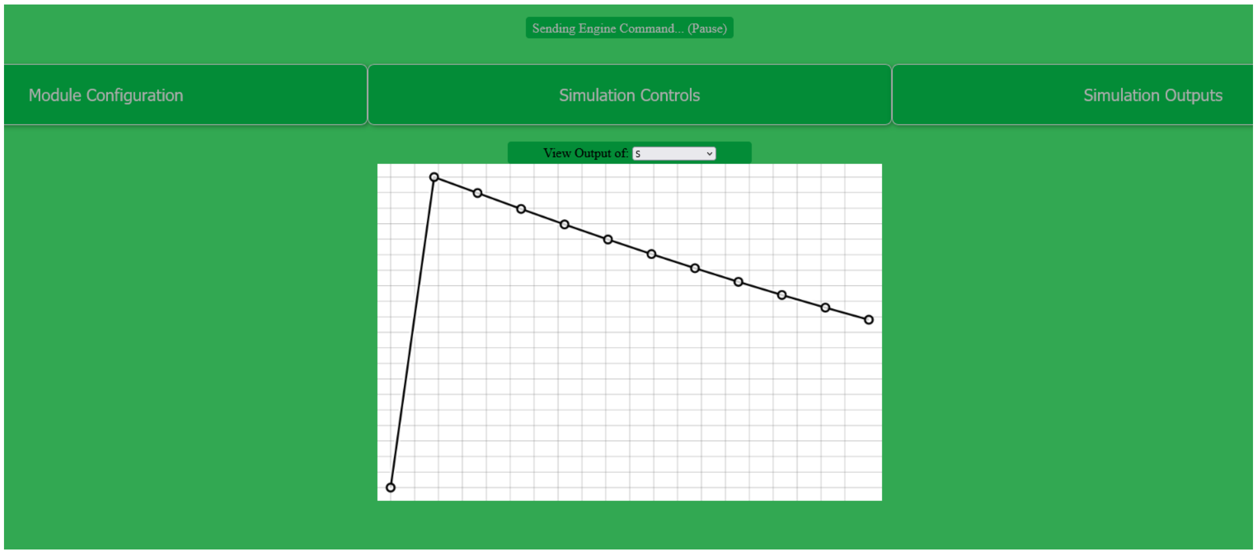


Fig 7. Simulation Output Screen

4.3 Testing and Results

Our project sponsor, Dr. Harris, provided us with two modules to test with: mm_fast.py and mm_slow.py. We tested our solution by comparing the simulation results produced by our engine with the results produced by manually recording and passing variables between modules. The figures in this section show the quantities of ES and S complexes in a two-species Michaelis-Menten enzyme-substrate binding example. Figures 8 and 9 show the original results, and figures 10 and 11 show the results using our engine.

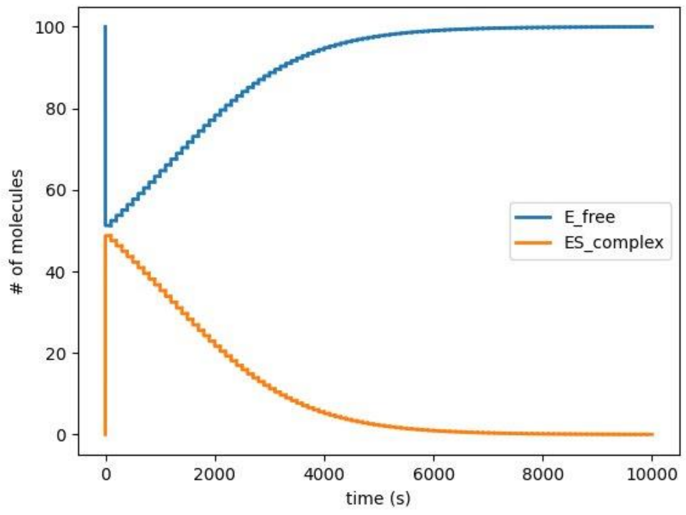


Fig 8. Original simulation results for fast module

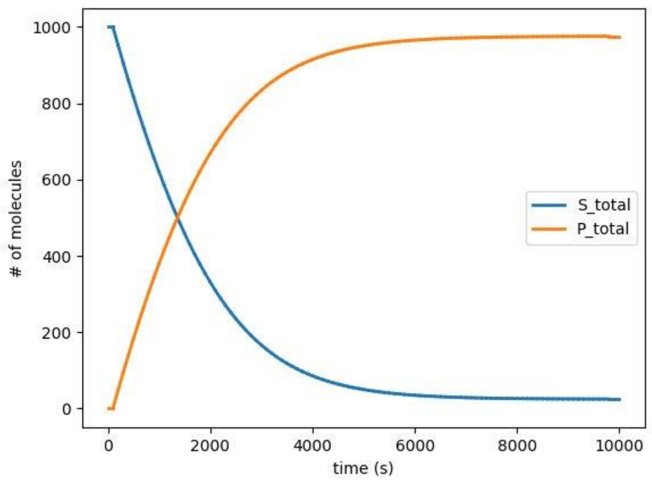


Fig 9. Original simulation results for slow module

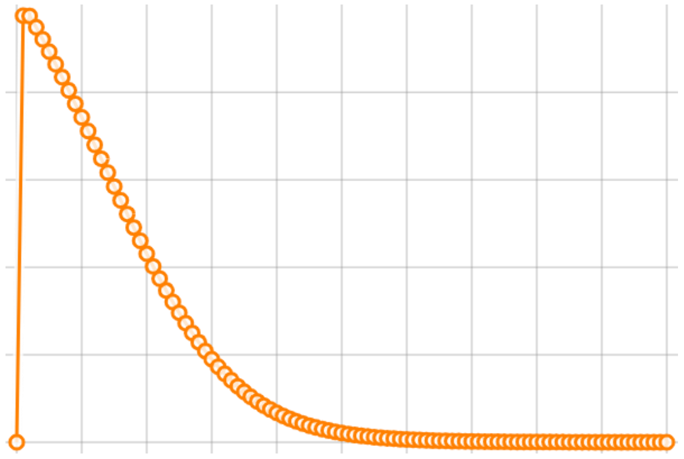


Fig 10. Concentration of ES complex over time, from fast module

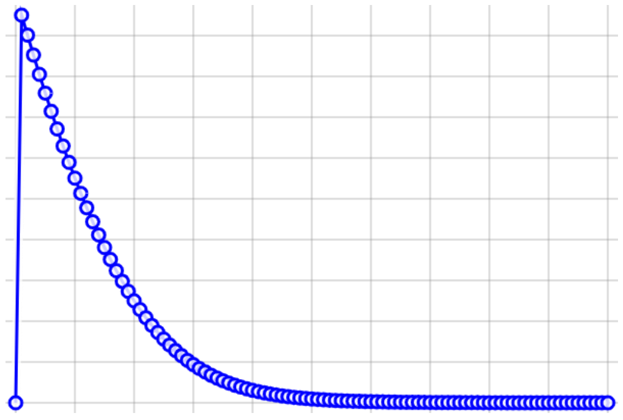


Fig 11. Concentration of S complex over time, from slow module

Our curve from figure 10 approximates the orange curve from figure 8, and our curve from figure 11 approximates the blue curve from figure 9. Based on these results, our engine can successfully simulate these biological models.

4.4 Risks

Risk	Risk Reduction
Use of Large Timesteps can result in inaccurate results	Tooltip warnings to users when they enter time steps telling them of this possibility.
Program is not User Friendly	Making our program do most of the work, minimizing the modifications the user must make to work with our solution. We plan to design things in a way that the UI and other aspects will be intuitive to work with for researchers
Invalid Configuration File	Tooltip warning the user to re-run the configuration file generation every time before they start the simulation again.
Invalid Module Implementation	The user is giving a tooltip warning them that improper module implementation will break the interface, and if they try and run the simulation then things may either not work at all, or not work as expected.

4.5 Tasks

The tasks are broken up into 4 different categories: Initial Research, Implementation, Testing, and Documentation. Initial Research will take up most of the Capstone 1 semester, while Implementation, Testing, and Documentation will take place during the Capstone 2 semester.

Initial Research:

- Research into Vivarium and other past proposed/attempted solutions
- Research into SBML, an encoding for biological models
- Research into other types of models, and the simulators required for those models
- Research into relevant framework creation, socket programming/inter process communication, simulation engines, and multithreading
- Research into the simulators we find to determine how they input and output variables, and how we can manipulate that for making example modules for testing

Implementation:

- Define the abstract module class, including (Done)
    - Relevant functions that must be implemented by the user
        - Module constructor to initialize model and simulator
        - Importing Parameters / Extracting Variables
        - Running Simulation
        - Significant Change and Equilibrium Detection Functions
        - Translation Functions
    - Functions that we need to implement
        - Translator for messages from engine

- Design Configuration file format (Done)
- Define Socket message formats for the various communications (Engine-Module, Relay-Engine, Relay-Web interface) (Done)
- Create Web Interface (Done)
    - Locally host a webpage on the user's machine (no outside internet access) (Done)
    - Build user interface with buttons, text boxes, and other interactive elements (Done)
    - Importing user modules into the program (Done)
    - Populating variable/parameter fields for modules (Done)
    - Create user functionality to define connections between modules (Done)
    - Generation of the config file (Done)
    - Build total simulation output view (In Progress)
        - Variable / time graphs (for every individual variable)
    - Build output dump functionality for simulation output (In progress)
        - Individual Modules
        - Total Simulation
- Create Thread Manager (Server Program) (Done)
    - Functions for thread starting and stopping simulations and other components
- Create Engine (Done)
    - Start/Stop functionality for full simulation (control given to user though WI)
    - Functionality to set the initial values using the module class passed in
    - Functionality to set timesteps, which are given by user
    - Functionality for reading config file and defining socket connections between modules
    - Functionality to call module functions for backtracking, equilibrium detection, and significant change

Testing:

- Work with sponsor to create a sample set of modules for testing of the program (Done)
    - Get a set of models that simulate things that could work together
    - Choose a simulator for the models
    - Create modules for each model
- Run through the users process of importing the modules and using the web interface (Done)
- Test the simulation and passing of variables from one module to another (In Progress)
- Refine design/fix bugs as needed (In Progress)

Documentation:

- Document the module class and its definitions, and what the user needs to implement into this module (In Progress)
    - Example implementations of module class for simple model and different simulators
- Document usage of the program (in the form of a Readme) (In progress)
    - Installation instructions
    - How the server works
    - Using the program with modules

4.6 Schedule

Here is our original schedule:

Tasks	Expected Dates
1. Background investigation into Vivarium, other proposed solutions, SBML, and example models and simulators	11/14 - 11/28
2. Define Abstract Module Class, Config File, Basic Socket Message Format	11/27 - 12/03
3. Make Thread Manager Server Program	01/15 - 01/21
4. Implement socket communications	01/22 - 01/28
5. Build Basic Outline Web Interface (with functionality)	01/22 - 02/11
6. Make Engine program	01/29 - 03/31
7. Make sample module classes	02/05 - 02/18
8. Refine Website to final design	02/19 - 03/03
9. Test systems for functionality and Fix any Issues	03/04 - 03/31
10. Document	04/01 - 04/28

With any group project, the schedule is never followed perfectly. Our project is no exception. The abstract module class, the thread manager, and the socket communications all took longer than expected. As a result, we have modified our schedule to the following:

Tasks	Expected Dates
1. Background investigation, Abstract Module Class, Config Manager, Socket Message Format, Thread Manager, socket communications	Completed
2. Finish Relay and Config Manager communications	2/27 - 3/15
3. Finish Interface (with functionality)	2/27 - 3/25
4. Implement Engine	1/29 - 3/31
5. Test systems for functionality and fix any issues	3/24 - 3/31
6. Document	4/1 - 4/28

As we moved into the final stages of the project, the amount of time needed to implement certain features took longer than expected, just like it did the time before. To reflect the more accurate representation of how long it took for us to complete our tasks, we have updated the table below to show the final actual completion dates of the tasks above.

Tasks	Actual Start/ Completion
1. Background investigation, Abstract Module Class, Config Manager, Socket Message Format, Thread Manager, socket communications	10/11/2023 - 2/19/2024
2. Finish Relay and Config Manager	2/22 - 4/16
3. Finish Interface (with functionality)	01/30 - 4/22
4. Implement Engine	2/20 - 4/21
5. Test systems for functionality and fix any issues	2/22 - 4/22
6. Document	4/18 - 4/22

4.7 Deliverables

- **Server Program –** The main server program will be released on GitHub for the user to download, including the web interface.
- **Module Abstract Class** – The code for the module abstract class will be provided so that users can implement the modules and use our server program with them.
- **Sample Empty Config file** – an empty configuration file with the initial fields for a model filled out (minus information for those fields) will be provided so that users can see the format of the config file.
- **Project Report** – A final report containing a brief overview of the problem, and proposed solution, how the solution solves the problem, and what work was completed/not completed.
- **Documentation** – A readme on the use and design of the server program will be released on GitHub, as well as documentation on the abstract module class and how it should be implemented. A template for how the sockets will communicate will also be provided with the documentation of the module class.
- **Website** – A project website detailing the members of the project, the problem, objective and solution of the problem and any related documents needed for the project should be stored here.

5.0 Key Personnel

Benton Anderson – Benton Anderson is a senior Computer Science major in the EECS Department at the University of Arkansas. Relevant coursework completed includes Software Engineering, Elementary Differential Equations, Ethics and the Professions, Public Speaking, Computer Networks. His relevant experience includes a summer internship at ArcBest Technologies. For this project, he will work on the solution infrastructure, the relay, and the web interface.

Bishop Butler – Bishop Butler is a senior Computer Science major in the EECS Department at the University of Arkansas. Relevant courses taken include Software Engineering, Programming Paradigms, Operating Systems, and Computer Networks. No internship experiences relevant to the project. He will work on the back-end engine along with the other project members.

Zachary Harris – Zachary Harris is a senior Computer Science major in the EECS Department at the University of Arkansas whose relevant coursework includes Software Engineering, Programming Paradigms, Operating Systems, Public Speaking, Ethics and Professions, and Computer Networks. No internship experiences relevant to the project. He will be responsible for the solution infrastructure, the web interface, and other related things within the server, along with assisting other project members.

Rithyka Heng – Rithyka Heng is a senior computer science major in the Electrical Engineering and Computer Science Department at the University of Arkansas. He has completed the following relevant courses: Software Engineering, Computer Networks, Linear Algebra, Elementary Differential Equations, Public Speaking, and Ethics and the Professions. He has the relevant experience: a summer internship at J.B. Hunt, during which he learned about and practiced developer operations and software engineering skills. This student will be primarily responsible for the solution infrastructure, module design and code, and the configuration design.

Isaac Withrow – Isaac Withrow is a senior in computer engineering in the Computer Science and Computer Engineering Department at the University of Arkansas. He has completed relevant courses such as Software Engineering, Operating Systems, Computer Organization, programming foundations I and II. He will program back-end engine stuff.

Matthew Patitz, Professor – Matthew Patitz is an Associate Professor in the Electrical Engineering and Computer Science Department at the University of Arkansas. He is the primary professor overseeing all EECS Capstone 1 projects for this semester. He provides relevant help and guidance to us for our papers, presentations, and within the project that we are working on.

Leonard Harris, Professor – Leonard Harris has been an assistant professor in the Biomedical Engineering Department at the University of Arkansas since August 2020. He has a PHD in chemical engineering from Cornell University and a BS in chemical engineering from the University of Colorado. His expertise lies in modeling and simulating complex biological systems. He has worked as a post-doctorate researcher in the Department of Computational and Systems Biology at the University of Pittsburgh School of Medicine and in the Vanderbilt University School of Medicine.

Justin Dykstra – Justin Dykstra is a PhD student in the Biomedical Engineering Department at the University of Arkansas with a BS in mechanical engineering from John Brown University. His research involves combining machine learning with mechanistic modeling to predict effective anti-cancer drug combinations.

6.0 Facilities and Equipment

- Facilities:
  - JB Hunt Building for a place for us to code on campus
  - Mullins Library for meeting as team to practice and discuss
- Equipment:
  - Personal Laptops for developing and running code

7.0 References

[1] Agmon, Eran *et al.* "Vivarium: An Interface and Engine for Integrative Multiscale Modeling in Computational Biology," Bioinformatics, Oxford University Press, 2022

[2] Hucka, Michael *et al.* "The Systems Biology Markup Language (SBML): Language Specification for Level 3 Version 1 Core," Journal of Integrative Bioinformatics, De Gruyter, 2015

[3] The Tellurium Simulator, https://tellurium.analogmachine.org/

[4] WebSockets, https://websockets.readthedocs.io/en/stable/

[5] asyncio, https://docs.python.org/3/library/asyncio.html