Computational Heart Physiology Simulator

Team ION

Richard Stack  Brian Call  Brad Pease
Devin Lane  Peter Lavallee

Project Sponsor
GCCIS Ph. D. Program

Faculty Mentor
Dr. Naveda

Project Overview

The GCCIS PhD department at RIT and its researchers are in the process of creating predictive models of the heart in order to better understand the long-term effects of abnormalities. They have currently implemented computer representations of known mathematical models of heart functions in order to simulate and visualize them. By creating a computer simulation of the heart, researchers will have the ability to run experiments and conduct research that would otherwise be impossible. It will also allow doctors to form better predictions when diagnosing patients. Overall the platform aims to achieve a new way to represent the inner workings of the heart.

Currently, these researchers have each constructed their own implementations of the heart model. The algorithms used to perform the calculation and the supporting functionality that go along with it are different for each implementation. As a result, the programs are mostly incompatible and cannot be integrated together. Even similar entities such as a matrix class differ between implementations. It is very inefficient for a researcher to create new models because of the lack of code reuse. Extending functionality also becomes a difficult task because each implementation is unique and consists of thousands of lines of complex math code. The PhD program has identified the lack of a common core as a major roadblock in the development of this platform and is in need of a more viable solution.

The overall goal of this project is to create a base framework to achieve a standard representation of the heart model. Researchers will then be able to individually add their own extensions to the system without conflicting with functionality that has been added by others. The framework will give the researcher the ability to add new data and functions to the underlying heart representation and will allow them to access and extend functionality that already exists. The project will not focus on translating the current mathematical models into computational code. That task is up to the expertise of the PhiD program and its researchers. Instead, this project is concerned with creating a foundation that the GCCIS researchers can easily learn how to expand upon in order to implement their specific contributions. The system will focus primarily on the standardization of computational models and reuse of components.
In addition to a common heart model representation, another main goal of the system is to include the ability to run a simulation using functionality that developers have added. A researcher will be able to specify which computational extensions he or she wants to run on the heart model and then perform that simulation. So not only does the project offer the user the ability to add functionality to the system but it also offers an engine that will process the additional functionality and then run specified computations.

From a business perspective, this system will eventually seek government funding and thus will be open source and not a commercial venture. However, the GCCIS PhD program wishes to use this platform as a chance to boost its recognition and attract other researchers into heart modeling. It also aims to make a contribution to the scientific community by providing a platform that better enables collaboration between researchers, some of whom are located in different parts of the world. Success of this project from the GCCIS PhD program’s perspective is measured by the ability of the project to enable enhanced collaboration between its researchers and those around the world, and the ability of the project to attract additional researchers to RIT. Additionally, the program would wish to receive recognition for its efforts that might earn it additional funding or sponsorship.

Upon completion of this product, researchers working to develop modeling systems for the human heart will have a platform on which to base their work, allowing them to benefit from increased collaboration, standardization, and reuse of their contributions. This enhanced reuse and standardization will dramatically reduce the time required for implementation and allow them to easily contribute their work back to the community.

**Basic Requirements**

The most fundamental component of the system is the underlying framework that makes up the computational heart representation. This framework is at the heart of the project and all of the other functionality stems from this common core.

The framework also stresses extensibility as one of its prime characteristics. It needs to be modular in order to accommodate for the incorporation of researcher contributions. The platform must allow for extension points when new representations are required. The system must provide a method by which extension points are added and a formula that can be followed to ensure that additions to the platform happen cleanly. It is also necessary for the framework to provide a means to make contributions reusable to allow for faster and easier extension development. Code contributions need to be easily accessible to anyone so they can either extend it or use it in their own code.

Another major component of the system is the simulation engine. Once the system has been populated with computation extensions, the user needs to be able to actually run them. The user will be able to select which computations should be performed, select which order they will run in, and then specify how many iterations of time the simulation will run for. The simulation engine will then check the list of specified computations and
will handle any dependencies between extensions. The engine will then compile all of the needed extensions and instantiate the model according to the node data supplied by the researcher. Once the engine has finished the necessary preprocessing it will begin to actually run through each computation extension that has been selected. After running through each computation in order, the engine will then restart the process over again for as many time iterations as specified.

The system will also contain developer tools to ensure that everything is consistent and make the framework more accessible to researchers. They will be able to specify what their extension needs as input from the system and what it will offer as output to the underlying heart model. There will also be a code generation tool which will generate skeleton code of each extension so it will fit the structure of the system and ease researchers into making their contributions. The project will include a way to facilitate deployment of extensions once they have been created so they will be sure to conform to the standards of the framework.

The entire project needs to be highly portable in order to accommodate for every type of user. Researchers operate on different platforms so the framework does not rely on specific features of a particular operating system. The project therefore needs to be able to operate efficiently on Windows, Mac OS X, and Linux.

The performance of the system is also crucial. Computation extensions will consist of many lines of code and the simulation engine will be processing lengthy numerical calculations. This processing can easily get out of hand depending on the number of nodes being calculated and the complexity of the computation itself. There are no strict runtime requirements at this time but the system should at the very least be comparable to the existing C++ implementations that the researchers have now.

Constraints

There were surprisingly few constraints on this system. The GCCIS PhD department defined a set of requirements with us and then left the design choices up to us so we were in control of the choice of language, technologies, and structure of the architecture. The system itself is not reliant on a particular operating system and through the use of Mono it will be executable on most major operating systems. We were slightly constrained on time as the length of senior project is only six months long. The entire project, however, will be a multiple year endeavor and may possibly include the work of another senior project team. To combat the short development time we determined the scope of what we were capable of completing and then ensured that the proper documentation was present so future teams could easily learn how what we were aiming to achieve and how the framework is currently structured.
Development Process

A key factor for the success of this project is the placement of ground work for future project teams to be able to continue and maintain the work that already exists. Since the requirements and resulting design is so crucial to the success of this project, a more structured process has its advantages over a less structured agile processes. Furthermore, once gathered, requirements are not likely to change enough that it would render the design invalid. With these factors in mind, a process in which the requirements and design are focused on in the beginning stages of development, before focusing on implementation would be appropriate. Our first thought was to use the Waterfall methodology which is a classic process that is well defined and well understood within the software community. As we tried to develop requirements to the point where we could move into the design phase we realized that to fully define our requirements we would need to know some information about our design. This need for feedback from the next stage of development caused us to select a variation on the Waterfall method. The Sashimi model has similar characteristics in that it has the same stages of development but where the processes differ is in the definition of when the stages of development occur.

Within the Sashimi model, the current stage of development and the next stage of development intersect so that both stages can be worked on concurrently. One risk that arises with the Sashimi method is that we will start to move into a third stage of development while still working with the previous two. We could start work on a stage pre-maturely, while the previous stages may still need work. We must ensure this does not happen by verifying that we have thoroughly exited a development stage before proceeding to the next two stages. It should be noted that our Sashimi model, as shown below, singles out the Maintenance stage. The reasoning for this is that the Maintenance stage is out of scope for our project.

Project Schedule: Planned and Actual

The project schedule was developed by first determining what we actually needed to deliver and what was in scope. We determined that since the requirements of this
project were not simple or strictly defined by the sponsor from the beginning that the early stages of the first quarter would consist of creating a clear set of requirements. Then from there we also decided that the design of the system was the most important aspect of creating a successful framework so we allotted the rest of the first quarter to clearly defining and documenting the framework design. Then by working backward from the final deliverable we set milestones that we needed to meet in order to achieve success. We wanted to have the final release of our implementation done in week eight in order to leave time for the presentation and to finish up whatever process work and documentation we had left. We then set two more iteration releases in week six and week two of the second quarter of the project. So in summary we wanted to have the core requirements completed by the middle of the first quarter and the design finished by the end. Then we had to meet implementation releases by the end of weeks two, six, and eight of quarter two with everything compiled and submitted by the end of the quarter.

During the first quarter we did experienced a schedule slip. We originally wanted to have the design finished during week eight of the first quarter but when we reached that time we realized that we were nowhere close to having the design finished. We decided to move the due date to the end of the quarter and then we looked at where we were spending our time to determine where the schedule slip actually happened. This led us to start tracking the time spent in each phase metric which will be discussed later. By examining our progress throughout the quarter we determined that we were having problems agreeing on a concise set of requirements. By constantly updating and editing our requirements document we were not able to get the design work off to a productive start. After realizing this, we all agreed on what the functional requirements of the system should be and then locked the document so we could focus our attention to developing a framework to meet all of the requirements.

Other than the schedule slip we experienced in the first quarter, the rest of our schedule went smoothly. We successfully met each release of the implementation on time and ensured that we had enough time to complete all of the necessary documentation and to provide our sponsors with a successful project at the end of the quarter.

**System Design**

Before designing the conceptual structure of the system, we needed to decide what language we were going to use and how we were going to address our high priority non-functional requirements. We were originally going to implement the system in C++ in order to maintain consistency with the GCCIS PhD department as this language was what they were currently using to create their systems. After further review however, we came to the conclusion that creating the project in C# would be more beneficial as the language had features that would directly address our major issues. By compiling the C# code using the tool Mono, it could be compiled to run on each major operating system easily which answered our need for portability. C# also offers a feature called extension methods which allow functions to be imported into a particular namespace and then accessed as if they were implemented within that class. By taking advantage of this feature the system is able to provide users with an easy way to create reusable functions
that can be accessed by any other extension. There was a bit of a tradeoff when it came to performance but the difference had minimal impact on the system and the useful features of the C# language were more beneficial than a small decrease in speed.

At the most abstract view of our design, the architecture of our system is split into four distinct subsystems. These consist of the main framework library, the simulator component, supporting tools, and an eclipse plug-in. As shown in figure 2, The library is the central component of this framework and all of the other components depend on it.

![Figure 2](image)

The library component consists of a standard internal data representation of heart data. It provides the functionality for creating extensions to the system and contains the means to understand and compile these extensions. The framework represents the heart model data using the concept of an attribute store. This attribute store will contain arrays of data for each attribute that the system computes. The way heart model data is represented conceptually is by using a collection of nodes at various physical locations around the heart, basically creating a grid mesh of the physical structure. Then each node contains a value for an attribute such as electric potential. The attribute store contains an array which consists of the value of that attribute at each node. Instead of maintaining a list of nodes and the values of their individual attributes, the attributes are stored instead and nodes are just a conceptual representation which is present as a specific index of each attribute array. The attribute store itself consists of any number of attribute spaces. The attribute spaces represent all of the attributes that belong to a certain type of node as the physical mesh grid nodes are only one form of data point in the heart representation. Attribute spaces can be further broken down into attribute sets which represents all of the attributes offered by a single extension. This way each attribute can be accessed using the unique name of the extension.
The extension mechanism is also implemented in the framework library. An entire contribution by a developer which will contain his or her extensions has been named a module. There are three different kinds of extensions that a module can offer to the system thus there is an extension base class which is sub-classed by each type of extension. A data type extension allows the user to add custom types to the framework so they can be used by anyone. A function extension is a reusable function that can be added to the framework and be accessed by other modules. A computation extension is the heart of the module and this contains the math code written by the researcher which will perform computations on the data within the heart model. A module can consist of any number of data and function extensions but can only contain at most one computation extension.

When a researcher wants to create a computation extension, they will need to specify what attributes are needed as inputs and what outputs the extension will change in the system. Computation extensions do not actually have access to the attribute store however. Instead, the data is copied into a node class structure and then passed to the extension. This allows developers the ability to utilize the capabilities of a class object instead of having to manipulate bunches of values in an array. Plus, it enforces persistence within the system because the extensions are not directly changing values located within the heart model. When the extension is done performing its computations, the values within each node are then copied into their corresponding positions within the attribute store.

The simulator component is designed to execute the code in each computation extension according to a configuration that the user specifies in order to actually perform a simulation. The simulator will initiate the attribute store and will then run computation extensions that will update the values within the attribute store. When each extension is run, it will copy the attribute data into nodes, manipulate the node data by running the computation extension, and then copy the resulting modified data back into the attribute store. Figure 3 describes this process.
individually for the total number of time steps and then proceed to do the same for the
next computation extension and so on. The step configuration can utilize any
combination of these two techniques in order to perform a simulation.

The low-level support tools of the system attempts to offer useful line tools to aid
in the development of modules. There are three command line tools that will be
developed for the researcher. The creation tool will generate skeleton code based upon
the XML manifest that is defined by the user. It will create skeleton code for computation
extensions, function extensions, and data type extensions. A build tool is also included
and this will utilize core library functions to determine dependencies, generate the
necessary compilation command for Mono, and then run that command. The deployment
tool will simply copy the files into the right places in the directory structure of the system
in order to ensure that everything is where it needs to be.

In order to provide researchers with an easy way to define an XML manifest file
that states what attributes their module needs, our system offers an eclipse plug-in that
gives users a simple graphical way to create their manifest. This eclipse plug-in abstracts
the framework away from the researcher so they can specify the parameters of their
modules without needing to know how the framework itself will use it. The plug-in will
also ensure that their manifest automatically fits the schema required by the framework.

The design of the heart model within the framework turned out to be the toughest
part of the lower level design to create. The attribute store needed to be extensible by
unique modules yet organized in a way that other extensions would be able to easily
access the data stored within it. We determined that storing the attribute data within a
bunch of node classes was inefficient because of the required data operations and the fact
that there could be multiple kinds of nodes. We did not want to lose the structure of a
node class object however, so we chose a solution that integrated both. The framework
sees the data as an array while a developer see the data as a node object containing only
the attributes that he/she needs.

We also came to the conclusion that the framework needed to be easy to learn to
accommodate the needs of the researchers who have little coding experience. The
framework is implemented in C# which not a standard language used in the scientific
community. Since developing modules for this framework would be a new endeavor in
itself we wanted to offer the users of the system the most help we could offer. That is
why we made the decision to include as much tool support as possible.

Process and Product Metrics

The process metrics we tracked were project slippage and stage distribution per
week. For project slip, we maintained a comparison between the issues that were late and
those that were on time. If the ratio was too high then a problem was present and needed to be addressed. We used the stage distribution metric to track how we were spending our time each week. During each meeting, we recorded how much time we were spending on each individual stage of our process. Then we combined that with the total amount of hours spent on tasks related to each stage of development. The result would show us exactly where we were spending our time so we could see if there was a slip according our schedule. This metric was also very important because of our development process. The sashimi model allowed us to be in more than one stage at any given time so we used this metric to manage the current state of the process. It let us know if we were jumping into the next stage prematurely or if we were back tracking too much and not making any forward progress.

For product metrics, we calculated function points per class in order to gauge the complexity of each subsystem. This would reveal to us which classes carry the most weight and where the majority of functionality lies. Along with the functions per class, we also had a desired max cyclomatic complexity (the number of nested conditions within our code) of four. This ensured that the code was easier to understand as too much nesting would provide unreadable code. As a result of this, we were able to break some of these conditions into separate functions when it seemed fit.

As far as results are concerned, the process metrics turned out to be very valuable to us. In order to visualize the results, we took the data that we have recorded in the Redmine content management system and created graphs. Here are the results of our stage distribution metric:

Winter Quarter:

Spring Quarter:
As you can see in the first graph, our schedule slip happened in week seven of the Spring quarter as the amount of work completed was half that of the week before and this led us to have to produce a high amount of work in the following weeks. Also notice the jump in administrative tasks towards the end of each quarter. This is attributed to the preparation of our presentation.

We also measured our estimated time vs. actual time which the results for winter and spring quarter are depicted in the following graphs:

Winter Quarter:

Spring Quarter:
As shown above, the team did a decent job estimating tasks, but tended to be on the safe side with overestimates. Also note that the actual of Week 10 for both the winter and spring appear with less than actual simply because the graphs were made before time was logged by all team members.

The following graphs depict the percentage of On-Time Issues for both the winter and spring quarter. This metric takes the total late issues over the total number of issues for the iteration (Iterations were end Week 2, Week 6, and Week 8).

Winter Quarter:

Spring Quarer:
The biggest dip over the two quarters was Week 3 of spring quarter. This is simply because week 2 ended an iteration (so the total issues reset to 0) and as a team we created issues to cover the bulk of what was needed for our second iteration.

The following charts show the Max Cyclomatic Cycles for the three main parts of our system (libheart, libsimulator and tools):
Out of all the classes (46 total classes), only one has a cyclomatic complexity (number of nested conditions) above our desired max of 4 (AttributeDependencyCalculator has a cyclomatic complexity of 6). All other classes have a cyclomatic complexity of less than 3, with the average being 1.2. There was a total of 16 (34.8%) with a complexity of 0.

The following charts show the average number of functions per class in each of our three main systems:
Because of this metric we were able to ensure that one class wasn’t doing more than it should. The average functions (including properties) per class is currently 6.8 and a standard deviation of 4.6. There are a total of 12 classes one standard deviation away from average, and of those, only three of them were two standard deviations away from average. Both the Compiler and the SimulationDriver shared the max with 19 functions in the class and several Exception classes shared the min with 1 function in each.

Product State at Time of Delivery

The state of the final product met all deliverables that were in scope. Each component has been successfully implemented and tested. The library accommodates for extensions to the model and the simulator can run through these extensions. There were no unplanned features added as we spent a great deal of time designing and implementing what we already had in scope. There were no discrepancies between the product and what we aimed to deliver.

Outside of the core extensions (Point3 and Matrix), there has been no extensions made for the framework. As stated in our scope, this is the responsibility of the research community to create extensions as they see fit. The project contains documentation on architecture and low-level design to make it easier to maintain for future teams, since we knew from the beginning it will be a long-term project. Overall, the final release was stable and includes the groundwork for this multi-year project.

Project Reflection

This senior project experience has been very positive for the entire team but it also came with some problems that we successfully struggled through. The lack of clear requirements during the beginning phases made it hard for everyone to understand exactly what we needed to create. The domain was new to all of us so it was tough to bridge the gap between our knowledge and the needs of the sponsor. We spent weeks just doing requirements elicitation to try and understand what we needed to do. The lack of a concise description of the project left each team member with their own individual perspective on how the project actually functioned. This variation in understanding forced us to spend time ensuring that everyone was on the same page. This lack of a common mindset made the requirements phase longer than we had planned which was one of the key factors driving the schedule slip we experienced in the first quarter.

At the start of the second quarter we decided to change the team leader. This was not because of poor performance; in fact it is quite the opposite. In the first quarter we had a tough time dividing up work between each member. Especially toward the end of the first quarter, more and more of the work was concerned with designing the framework. Our team leader at the time was the one who had most of the ideas for design and had a better understanding of how all of the pieces fit together. This required him to lead both design work and project management which turned out to be an unfair amount of work. At the beginning of spring quarter, we decided to change team leaders in order
to better accommodate the skills of each team member and to lighten the workload on the previous leader. This turned out to be a very beneficial decision as the second quarter went very smoothly with few problems

Overall, this project went very well. We each worked together well as a team and our process and communication led us to utilize the unique aspects of each individual’s personality. We learned that organization is the key to success as things were a little messy when we first began but as we fell into a routine that worked for everyone everything went very smoothly. By managing our time efficiently and using our skills as software engineers we have delivered a complete and successful project.

References

Glossary of Terms

CHPS - Computational Heart Physiology Simulator. Pronounced "CHiPS".
Heart Model - A numerical representation of the heart.
Physical Heart Model - A numerical representation of the physical (e.g. position) data of the heart.
Numerical Model - An equation or set of equations that computes the behavior of the heart from a specific perspective (e.g. Electrical Propagation).
Module - A collection of type, function, and computation extensions that implement numerical models or supporting functionality.
Extension - An addition of functionality to the system. This can be a type extension, function extension, or computation extension.
Type Extension - Provides additional data types to the system.
Function Extension - Provides reusable supporting functions to the system (e.g. sum(x, y)). These functions are not tied to any Computation Extension to encourage re-use.
Computation Extension - Provides the ability to add data to and manipulate data in the Heart Model.
Attribute - A numerical datum of the heart used by a Numerical Model (e.g. Conductivity or Elasticity).
Attribute Set - A set of attributes contributed by a single module.
Attribute Space - A collection of related attribute sets.
Attribute Store - The internal representation of the Heart Model as a set of Attributes.
Node - A structure used by Computation Extensions to represent Heart Model data.
Glue Code - Defines a translation to and from the Attribute Store to Nodes for a given Computation Extension.
Manifest - A document that describes the extensions provided by a module along with its meta-data.
Recipe - An ordered list of modules that the simulation engine will run.
Schema - Formal definition of the Manifest file format.