Model-Based Embedded Software

Robert Bui
Kevin Albers
Jose Oyola Cabello
Naren Vasanad

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2015-117
http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-117.html

May 15, 2015

Acknowledgement

We'd like to thank our advisors Professor Edward Lee, Professor Sanjit Seshia, and Christopher Brooks for their guidance and encouragement during our project work.

University of California, Berkeley College of Engineering

MASTER OF ENGINEERING - SPRING 2015

Electrical Engineering & Computer Sciences

Robotics & Embedded Software

Model-Based Embedded Software

Robert Truong Bui

This **Masters Project Paper** fulfills the Master of Engineering degree requirement.

Approved by:

1. Capstone Project Advisor:

Signature: ________________________ Date ___________

Print Name/Department: Edward A. Lee / EECS

2. Faculty Committee Member #2:

Signature: ________________________ Date ___________

Print Name/Department: Sanjit Seshia / EECS

Abstract

Model-Based Embedded Software

by

Robert Truong Bui

Master of Engineering in Electrical Engineering and Computer Sciences

Professor Edward A. Lee and Professor Sanjit Seshia

Embedded software is typically developed using traditional programming languages like C and C++. However, these traditional types of programming languages are not well suited for embedded systems development. The model-based embedded software project extends the code-generating capabilities of Ptolemy II to help users develop software using model-based design techniques for ARM mbed devices. In particular, this project primarily focuses on automatically generating C/C++ code in Ptolemy II for Synchronous Data Flow (SDF) and Finite State Machine (FSM) models. This makes it easier to design and debug, leading to faster and more robust software development.

Table of Contents

I. Problem Statement

The Internet of Things (IoT) encompasses all small scale embedded systems which are interconnected wirelessly through the internet and are continuously transmitting data. Currently, programming embedded systems requires knowledge of intricate details of the platform being used and the software is typically written using traditional programming languages such as C and C++. In addition, embedded software for complex systems becomes very long and difficult to understand as it grows. Our project involves the creation of an environment to make designing applications for IoT easier through the use of model-based embedded software techniques. The product abstracts all the finer details of implementation and exposes the features that the designer is concerned with. Today, designers widely use embedded computing devices such as Arduino[1] and mbed™[2], from ARM®, for prototyping embedded applications, because they are open-source and low power. They are also inexpensive and have a large community of developers. The design environment we are developing will specifically target these types of embedded platforms.

Hardware and software of a cyber-physical system can be complex and difficult to implement. "Cyber-physical systems" refers to embedded computer systems that interact and are affected by physical elements (Mueller et al. 2012:219). A technique for designing a cyber-physical system is model-based design, which applies mathematical modeling for designing and verifying systems (Jensen et al. 2011:1666). Our project focuses on the creation of a model-based design environment for programming embedded platforms. In particular, our project targets applications aligned with the Internet of Things.

[1] "Arduino is an open-source electronics platform based on easy-to-use hardware and software. It's intended for anyone making interactive projects." <arduino.cc>
[2] mbed is an ARM based microcontroller that can be used to develop applications for the internet of things. <https://mbed.org/>

Over the course of the project, we created a model-based design environment and demonstrated its use with an embedded platform application. In order to test and determine the effectiveness of the application, the project included designing an example system. The application used to demonstrate the model-based design environment's capabilities was an interactive LED cube that could be controlled with hand gestures. The application was initially developed using regular coding techniques by writing C and C++, and later developed using the model-based design environment for comparison. The models for the components of this application were included in the final application.

Code generation is one of the primary aspects of the model-based design approach. As described by Jensen et al. (2011:1666), the model-based design methodology involves the use of a code synthesizer to produce code that executes the desired models of computation. Typically, designers will write C code that can be programmed on an embedded platform to perform some task. However, model-based design techniques allows a developer to build graphical models that represent their application. This project involves the creation of an environment using Ptolemy II [3] to allow designers to represent their application as graphical models. Based on the model created in the design environment, code can be automatically generated for an embedded platform.

Due to the nature of model-based design and specifically code generation, designers can spend less time writing and debugging code. Rather, designers can focus on the design of their application and verify its expected behavior. The use of a model-based design environment allows designers to represent how they expect their application to perform and allow the software environment to produce reliable code. The modularity of graphical models allows designers to easily reuse models in different applications and change aspects of their design, and

[3] "Ptolemy II is an open-source software framework supporting experimentation with actor-oriented design." <http://ptolemy.eecs.berkeley.edu/ptolemyII/>

the graphical interface allows a user to easily view concurrent processes and how distinct units of a program interact with each other.

II. Industry/Market/Trends

A. Introduction

Open source embedded platforms have become popular for rapid prototyping. The market for embedded platforms has been growing as the number of connected devices continues to increase. Our capstone project aims to contribute to the community of embedded developers by solving the challenges of efficient code generation using the approach of model-based design.

The motivation for this project was twofold. First, a model-based design environment specifically for mbed devices does not currently exist. There are a few competitors, as described further in this section, that provide a graphical interface, but they do not offer a design environment focused on model-based design. Secondly, our project targets an emerging market and offers an opportunity for us to differentiate from our competitors. Embedded platforms have become very popular with hobbyists and the maker community, but there are not many tools such as ours that directly contribute to helping design for applications involved with the IoT. The stakeholders for this project include three segments: end users, sponsors, and customers. End users include hobbyists who work on IoT projects. Since these users will be working on fast prototyping of solutions and also have basic knowledge about building products, this would be the ideal market to target. These users could potentially give feedback of our product to improve and focus it towards being viable to a larger audience. Once the software gains traction amongst hobbyists it will be easier to reach a broader market like students, major companies, and universities. Our sponsors include the EECS Department, Embedded Systems Lab, TerraSwarm Research Center, Professor Edward Lee, Professor Sanjit Seshia, and the project team members

(Kevin Albers, Robert Bui, José Oyola, Naren Vasanad). Our customers will be discussed in detail in the Customers sub-section.

In this section, we use Porter's five forces model to analyze the five major forces in our embedded software market in order to create a go-to-market strategy: competitors, customers, suppliers, new entrants, and substitutes (Porter, 2008). In his article "How Competitive Forces Shape Strategy", Michael Porter (1979) discussed how the "strength of these forces determines the ultimate profit potential of an industry". We describe each of the forces and its effect on our strategy in the sections ahead and provide a strong or weak label. A force that is labeled as strong means that it could have a strong effect on our competitive strategy, whereas a weak force is an area that our strategy could take advantage of. Porter's five forces was important to use because it offers a unique analysis to determine the strength of our product's position, potential to make a profit, and create a strategy to move the balance of power to our favor.

B. Market Trends

Our target industry includes anything which encompasses IoT. Gartner (2014) published a study indicating that the IoT is on the peak of the hype cycle. It is expected that IoT will reach the plateau of productivity, the point where the technology is stabilized, in the next five to ten years. Furthermore, Clarice Technologies (2014) talks about how there will be close to 50 billion devices connected to the internet by 2020. Based on these studies, the IoT industry has the potential to grow immensely in the near future.

Most of these IoT devices will be small scale devices which sense the environment and connect over the internet to communicate with other more complex devices. A Markets and Markets (2014) report expects that by 2019, the IoT market will be close to $500 Billion. IoT has the potential to create waves in many industries worldwide, spanning from medical and wearable devices to transportation and automation, as well as improve social connectivity between people everywhere (Hulkower 2014; Ma et al. 2011).

C. Competitors

There are three main competitors that offer model-based programming with a graphical interface. These include MATLAB's Simulink®[4], National Instrument's LabVIEW[5], and an open source project named PyLab_Works[6].

Mathworks' product, MATLAB, is one of the world's best super calculators that runs on a computer. It uses a scripting language to solve complex computations, often by using calculus. Simulink is an environment within MATLAB that allows programs to be built using graphical blocks. Mathworks has provided an interface, called Simulink Coder, a Simulink extension that allows user to generate and execute code from stateflow models.. This allows people to use Simulink to build model-based programs, then use the interface to and from the Arduino to provide Simulink with the inputs and outputs. However, Simulink must be installed on a computer to run, so the embedded device must be connected to a computer in order to work.

National Instruments improves upon Simulink's flaws with LabVIEW. LabVIEW is similar to Simulink, but it switches the focus from computations with calculus to data analysis and program logic. The best advantage that LabVIEW has over Simulink is the downloadable model. It allows code generated by the model to be downloaded to the embedded platform and run without the help of a computer. While LabVIEW offers substantial advantages for embedded devices compared to Simulink, our solution offers further improvements with the use of model-based approaches.

[4] "Simulink® is a block diagram environment for multidomain simulation and Model-Based Design." <http://www.mathworks.com/products/simulink/>
[5] "LabVIEW is a graphical programming platform that helps engineers scale from design to test and from small to large systems." <http://www.ni.com/labview/>
[6] "PyLab_Works is a free and open source replacement for LabView + MatLab, written in pure Python." <https://code.google.com/p/pylab-works/>

In the open source community, PyLab_Works offers an open source solution that attempts to accomplish model-based embedded programming. It offers a block graphical interface similar to LabVIEW, but it does not have much support. Each block must have written code in Python, meaning it is not completely model-based software.

Our solution differs from our competitors since it’s open source and open platform, whereas MATLAB and LabVIEW require a license to use them. A MATLAB license for personal use costs $149 for non-students, and the basic LabVIEW license costs $999 (MathWorks n.d.; National Instruments n.d.). This license cost is prohibitively expensive to many potential users of these systems. In contrast, our solution is open source and freely available. In addition, our solution is open platform. MATLAB and LabVIEW are closed to specific platforms that the developers have chosen to support. If a user wishes to use one of these software tools with a different platform that is not supported, then there is little he or she can do. By making our solution available to the open source community, it is able to expand and grow the amount of supported platforms. Overall, the threat of rivals is weak, though with a change in strategy, it is possible that these competitors could enter the hobbyist space.

Open source software has been known to disrupt markets dominated by proprietary software in the past. According to IBISWorld, “open-source software (OSS) has been growing as a share of the global software market” (Kahn 2014:31). OSS (such as the Linux operating system) is a threat to some proprietary software, but will also promote interoperability and new software developments (Kahn 2014:31). Since our software is associated with open source software, we anticipate that we can leverage on the OSS structure and increase traction on our product.

The success of our application can be measured with market adoption. A study has shown that the number of updates to open source software created by members of open source communities has increased exponentially in the recent past (Deshpande et al, 2008:205). This further supports our claim that acquiring more users would lead to more development of our project. Handling a community is not a straight-forward task. Øyvind et al. says that it may be beneficial to release the product as executables in the beginning to increase usage and decentralize the control of power with specific tasks having ownerships also that as the product grows (Øyvind et al. 2009:71-72).

Another factor that affects market adoption is the availability of modules. Our application will have a library of modules that are specific to IoT. These modules include sensors, actuator and communication. Making these modules specific to IoT will help differentiate ourselves from competitors who may not have such libraries. These standard libraries will help to create trust in the open source community and hence will help in building traction amongst hobbyists (Øyvind et al. 2010:114).

D. Customers

Our project would make it easier to communicate with development platforms and also to integrate sensors and actuators into a system. Since the technology is still nascent, it gives the project the right opportunity to grow with an emerging market and adapt to changes from customer needs.

Our main target customers are hobbyists and do it yourself (DIY) enthusiasts. These customers have a large variety of products to build their projects with, as well as a competitive market with low prices for embedded platforms. In addition, there are various tools that they can use to develop on their chosen platform as described in the competitors subsection. The most important factor is our reliance on market adoption to promote our product. We need to create a community that develops libraries and examples that are easily accessible to new users. However, open source software adds additional barriers for customer adoption. It can be harder for customers to trust open source software as much as the paid closed source alternatives created by established companies (Bianco et al. 2009). For these reasons, the customer market force is strong.

E. Suppliers

Since our project is built using the Ptolemy II, the affiliated Ptolemy II research group at UC Berkeley is our main supplier. Ptolemy II group relies on donations from research grants and businesses that use the software. Our success will help extend the successful functionality of the Ptolemy II project, making it beneficial for us to succeed. This makes our supplier a collaborator rather than a potential threat to our success.

Furthermore, the fact that this is a research project under one of the most reputed universities in its field helps us differentiate from other competitors. Even if there are competitors in the open source community, the backing of the Ptolemy II project will help gain trust from potential users and hence increase the conversion rate of adoption in our favor.

F. New Entrants

According to Hoover's industry analysis of Computer Aided Design (CAD) software, the DIY movement "has sparked interest in CAD/CAM software among hobbyists and tinkerers" (2015). Our software falls into this category as a form of CAD. This industry opportunity shows that not only will this space be attractive to existing players, who can easily enter the market to compete with their products, but also startups that could use our open source code to build their own similar products to compete with our own. This shows that the threat of new entrants is strong.

G. Substitutes

Hobbyists have the option to continue using tools that they know, which makes programming in languages such as C a substitute to our product. Since it might be too time consuming to learn a new programming method such as using a graphical design environment, many hobbyists might decide it is not worth their time to switch from their current programming methods. We designed our tool to reduce development time when the user has learned how to use it, but over a short period of time this is less obvious to the user and they may become frustrated and return to a familiar tool. In addition, the current communities, such as the Arduino community, have large libraries of tools and project guides, which pose a strong threat to our product adoption. This makes the threat of substitution a strong threat.

H. Critique and conclusion to Five forces

Given the fact that our project is open source and the current trends in the open source community, we are in an interesting position when it comes to our strategy. After evaluating the five forces, it seems that some of these forces may actually end up working in our favor. First, our main supplier, the Ptolemy II project, is actually more of a collaborator. The project participants frequently and on a daily basis increase the capabilities of Ptolemy II and add to the already large codebase. As will be discussed in the section on Intellectual Property, our success is linked with the Ptolemy II project, which was mentioned in the suppliers sub-section. This further incentivizes the Ptolemy II project stakeholders to continue to pursue the project and ensure its success.

In addition, the customers for our project are hobbyists and the open source community. The open source community is known for expanding projects and making the projects suit their needs (Deshpande et al. 2008:198). Therefore, our open source customers can actually become collaborators and help expand the codebase of Ptolemy, adding support for other platforms, and creating sample applications for others to use and learn from.

The open source nature of the project also has the effect that new entrants can end up helping us succeed. Any new open source alternatives to our Ptolemy project will have to compete with Ptolemy's 20-year-long history and codebase, which spans over 3 million lines of code. However, open source projects have another option: to join our community and enhance its reach and capabilities. For instance, a new entrant seeking to create an open source model-based environment for the Raspberry Pi can take advantage of Ptolemy's already existing infrastructure and simply add support for their platform instead of building everything from scratch.

Overall, the five forces in our market are moderate, with the strongest force being the customers. This means that without addressing these forces appropriately, the profit in this industry will not be huge, even if successful. The open source business model adds an additional challenge to profitability. We can mitigate the strong forces with the right positioning.

To bring our product to market, our marketing strategy will be focused on the 4 'P's: product, place, price, and promotion. As mentioned in the subsection on Customers, our target customers and users are hobbyists and makers. By making our product initially open-source, it will be very appealing to this customer segment as they are very willing to try new products especially those that are at no cost to them. We plan to market it differently as well since we are targeting the open source community instead of industry professionals like our competitors. From our marketing study conducted early in the project, we learned that many of these types of users learn about the latest technology through websites and complementary technologies to our product such as embedded platforms like Arduino. Therefore, our strategy will be to ensure our product is easily accessible online by hobbyists.

I. Marketing and Productization

Based on the success of providing our product as an open source solution, there are four ways in which we could begin to monetize our project. The first way would be to to offer technical support for those that are interested in advanced applications. Users could pay to receive help from our technical support staff in using and extending our product for their own needs. This option would be the first one that we would try since it has been successful for other products in the past. In his article, Fitzgerald calls this a value-added service-enabling model which has been very successful for Red Hat, an open source Linux provider (Fitzgerald 2006).

Another alternative would be the use of advertisements. Similar to how desktop and mobile applications are designed, we could incorporate advertisements in our design environment and users would pay a fee in order to use a version without advertisements.

Furthermore, we could offer a professional version of our open source project that would be targeted to advanced users and industry professionals. This version would use a subscription model where customers pay a monthly or annual fee. The professional version would include application specific content and strong technical support and documentation for the most cutting-edge advancements in embedded systems. Fitzgerald also mentions in his article that this would be considered a loss-leader/market-creating model since our first product would be open sourced but a product with more features would be used for monetization (Fitzgerald 2006).

A final option for monetizing our product would be to partner with an embedded platform company and offer our product as part of a bundle. The company would provide the target embedded platform hardware and our software product would complement their device with a custom design environment. An example of this approach would be the mbed collaboration between ARM and several semiconductor companies. In this industry with established competitors, this would be an appealing approach to obtaining market share and brand recognition.

J. Conclusion

Based on our project's unique features and target market, our project has potential to make an impact in the embedded software industry. The IoT era has brought a need for better software design tools and our product helps solves the challenges that designers face. By targeting hobbyists and the maker community, our product enters a space where it can receive market adoption and not directly compete with well-known embedded software competitors. “Open source style software development has the capacity to compete successfully, and perhaps in many cases displace, traditional commercial development methods” (Mockus et al. 2002). Based on our evaluation of Porter’s five forces in this industry, our business strategy should allow our product to make a strong impression in an industry with primarily commercial development methods (Porter 2008).

III. IP Strategy

A. Introduction

Since the Model-Based Embedded Software project is built upon Ptolemy II, it is important to understand the intellectual property surrounding the project before deciding how it should be advanced for commercialization. The concepts and ideas that form the basis of this capstone project are not novel, nor is the particular application that this project seeks to build. In particular, the project is an open source implementation, rather than invention, of the previously existing branch of computer programming known as model-based code generation. Several software solutions already exist that produce code using similar techniques, and they are mentioned later in this section. This makes it highly unlikely that any aspect of the project is patentable. However, this does not mean that the concepts of intellectual property do not apply to this project. This section discusses the intellectual property aspects of the Model-based Embedded Software project and the strategy that can be used to ensure proper use and attribution of our work, as well as the risks associated with infringement of previously existing IP.

B. Open Source Licenses

There are many different open source licenses that are available to protect the work of the open source community. The most widely used open source license, the GNU General Public License (GPL), is an example of what is known as a "copyleft" license, which requires that any work built upon GPL-licensed software must also be distributed under the same license (Lindman et al. 2010:239). This ensures that any GPL-licensed work will forever be freely available for all to use. However, other open source licenses such as the Berkeley Software Distribution (BSD) and MIT open source license are different. These open source licenses, both of which come from academic institutions, allow software covered under the license to be used in any way, including in commercialized software for profit, with no restrictions (Lindman et al. 2010:239). The idea behind this method of licensing is that successful projects coming from these institutions, if available freely for use in successful software, can benefit the institution from where it came by enticing others to provide funding to the institution for further development of the software. An example of successful commercial software built upon BSD-licensed software is Apple's Mac OS X and iOS, both of which are built upon BSD Unix (Engelfriet 2010:49). These open source licenses provide many benefits to those wishing to build upon them, such as software startups, since it does not require the resulting software to have the same license. This means that any other protection can be used for the software, including copyright protection, or even a different open-source license, which would ensure that the software would continue to be available as open source, if that is the goal of the software developer, as is often the case for the open source community (Engelfriet 2010:49).

Since our work is part of a large software collaboration, Ptolemy II, it will be bounded by the same rights of use, the BSD license ("Ptolemy II F.A.Q" 2014). "Ptolemy II is an open-source software framework supporting experimentation with actor-oriented design" and is a part of the

Ptolemy project at UC Berkeley, which is an initiative dedicated to studying models and simulations of embedded systems (“Ptolemy II” n.d.) . The Ptolemy project is well-funded and has many industrial sponsors involved (“Sponsors of the Ptolemy Project” n.d.). The BSD license allows software designed with Ptolemy II to be used for free commercially. Thus, if we decided to extend the software in the future as a separate entity, we would not have any issues commercializing it.

C. Advantages of Open Source Licenses

Furthermore, there are many other advantages for distributing our software through open-source channels. As mentioned in the Industry/Market/Trends section, many large competitors already exist in the embedded software industry. Open-source software offers a way to create market adoption by allowing customers to try a new product for free in order to build a community supporting the software. This is one way that open source software can penetrate a market with large competitors. According to Hoover (2015), “open-source software, which poses a competitive threat to the industry's traditional license-based business model, has grown in popularity in the last decade.” There are many examples of immensely popular open source successes in the past, such as Linux and Apache, and PostgreSQL, which have formulated a threat to proprietary software (Kahn 2014:31; Deshpande et al, 2008:197).

Although open source software can pose a threat to proprietary software, its open nature can also be a disadvantage. Since many of the existing large players have a research and development unit, the entrance of a new player could mean that existing players can simply use the new open source software to improve their solution directly (Engelfriet 2010:49). This is not an issue for copyleft licenses, since they require that any software built on it must also use the same license, but this requirement doesn’t exist for permissive licenses such as BSD (Engelfriet

2010:49). Because permissive open source licenses allow for this to happen, it is very difficult for open source developers to protect themselves.

Currently, two of the largest competitors in the embedded software industry are Mathworks and National Instruments. Their respective products that are similar to our software tool are Simulink and LabVIEW. Each of these products offers a graphical design environment that can generate code for embedded system. Both of these companies have many patents registered involving the design environment, model types, and methods for code generation. In particular, National Instruments has a patent titled “Statechart development environment with embedded graphical data flow code editor”, US patent number 8387002 (Dellas et. al. 2008:1). The patent describes a graphical design environment that uses a model that LabVIEW called statecharts, “a diagram that visually indicates a plurality of states and transitions between the states”, to represent an application (Dellas et. al. 2008:35). Furthermore, in the patent, LabVIEW claims the rights to the invention of code generation for statecharts and specifically the transitions linking the states of a model (Dellas et. al. 2008:35). Although this patent seems similar to our product, it is quite different since it involves statechart models which are not used in Ptolemy II. Rather, our software generated code based on the specific model of computation selected instead of solely transitions and states as done in LabVIEW. Based on the limits of the patent to statechart models, the patent should not overlap with our idea.

D. Concluding Remarks on IP

Ptolemy II has existed for almost 20 years as an open source project and many commercial products have been created from Ptolemy such as Agilent’s Advanced Development Systems (“Links” 2014). Our capstone project extends the functionality of Ptolemy II by offering code generation for models currently supported in Ptolemy II. Since there are currently no novel aspects of our projects that could be patented, open source would be the best alternative approach for the current state of our project.

IV. Technical Contributions

A. Project Overview

Embedded software is typically written using imperative languages such as C. However, these traditional types of programming languages are not best suited for embedded systems development. Lee and Neuendorffer (2004) define embedded systems as the combination of software and hardware in a system where the software reacts to sensor data and issues commands to actuators. Unlike general software, software for embedded systems can have complex timing requirements and are typically held to a higher reliability standard due to their applications directly involving human lives ("Design Challenges" 2008). Therefore, developing embedded software can be time consuming and difficult for developers, especially those creating embedded software for the first time. Our capstone project aims to demonstrate how formal methods, particularly formal modeling, can be used to efficiently design embedded systems applications. In particular, we explored the model-based design of embedded software for open source embedded platforms and created a graphical design environment using Ptolemy II that automatically generates code for an embedded platform.

The scope of the project involved choosing an embedded platform and using it to create an application in order to understand how embedded system applications are traditionally developed. We explored the challenges of developing embedded software for an ARM mbed platform. Furthermore, we assessed the tradeoffs and risks associated with code generation through our graphical design environment using Ptolemy II. Since Ptolemy II has an existing code generator for C code, we became familiar with how the generator works and modified it in order to generate code from models for our target embedded platform based on our example application. We compared the generated code with code written for our embedded platform to determine opportunities for optimization and efficiency.

The timeline of this project was for one academic year. In the first semester, the primary focus was on understanding the model-based design process and creating an application to become familiar with embedded system tools. We chose an application inspired by the Internet of Things which involved the construction of an interactive LED cube that could be controlled by sensors in a data glove over the internet. My work during this semester involved creating models to represent our application and developing the application code to interface the LEDs with the embedded platform based on information received from the sensors. It was important for my work to be carefully developed in order to give our team a foundation for code generation in the second semester of the project which is discussed further in the Methods and Materials section.

In the second semester, the team's main focus was on generating C code based on the code we had written for our application. This involved learning about the code generator architecture in Ptolemy II and extending it to generate code and build the source files for our target mbed platform, the Freescale FRDM-KL25Z. Furthermore, we learned how to model our application in Ptolemy II and solved issues to create a usable design environment. An integral aspect of our project was to document our process in order for future users to extend the code generator for their own applications in the future. My work during this semester involved understanding the existing Ptolemy II code generator and building models to test our application. During this process, I helped find and solve issues related to the code generator and create models that successfully code generate. The outcome of my work helped the team have models for testing their aspects of the project and move further in creating a code generator for more complex models.

B. Relevant Work

Although there are many different formal techniques for designing an embedded system, the focus of this project will be on model-based design. This approach emphasizes the use of mathematical modeling for designing and verifying dynamical systems (Jensen 2010). Model-based design allows a designer to check that their model performs and meets their requirements through a formal process. Models are used to represent the design of a system and help predict the characteristics of a design (Karsai et al. 2003). In his paper, Jeff Jensen (2010) formalizes the model-based design workflow for designing embedded systems in ten distinct steps. The steps involve characterizing the physical process in a model, selecting a model of computation, using a simulation tool that supports the model of computation to simulate and synthesize the software, and testing the application. Since results at different steps can influence the design, the model-based design process is iterative. For our capstone project, we used the model-based design approach to develop a target application. Our capstone project focuses mainly on the software synthesis step of the model-based design process since it involves generating code from a graphical model representing the application using Ptolemy II.

Developed at UC Berkeley, Ptolemy II is a software framework that allows a user to model and simulate embedded systems. Ptolemy II offers a graphical user interface called Vergil that allows a user to create graphical models using various actors and directors, and view the results of simulations. Actors represent components of a system in a model such as mathematical components and are governed by directors, which define the semantics of the model (Ptolemaeus 2014). The semantics of a model, or "collection of rules that define how components in a design should interact", is known as the model of computation (Ptolemaeus 2014).

Ptolemy II is different than many other software environments because it allows a user to choose the model of computation for an application. Based on their application, a user can choose from models of computation such as synchronous data flow, discrete event, continuous, or process networks. Actors in a synchronous data flow model consume and produce a number of tokens each time they are fired. In order to fire, each actor requires a certain a number of tokens to perform their process and produce tokens (B. Lee 1998). The models of computation we are interested in working with are synchronous dataflow and finite state machines. Finite state machines involve a set of inputs, outputs, states and transitions. For each transition in a finite state machine, guards are used as conditions for a transition to occur. Each transition may also have an action that executes when the conditions from the guard are fulfilled ("Finite State Machines" 2009).

In embedded systems modeling, it is often of interest to use hierarchy to allow for the use of multiple models of computation and reduce the number of transitions for a finite state machine (Girault 1999). Hierarchy is a key concept for structuring complex systems as the use of sub-models allow for characterizing the architecture and behavior of a system's design. For our capstone project, we are particular interested in using finite state machines inside a synchronous dataflow model (Alur 2003). In Ptolemy II, finite state machines are described in synchronous data flow models using a modal model actor. The modal model actor is governed by the rules of synchronous data flow, but its refinement is a finite state machine.

Unlike Ptolemy II, which allows for different models of computation, other code generation environments have a fixed model of computation. In these types of environments, the user is constrained to modeling their application based on the model of computation that the software is built upon. For example, MATLAB's Simulink environment uses a continuous time model of computation and National Instrument's LabVIEW uses a model of computation similar to synchronous data flow (Ptolomaeus 2014). The existing C code generation environment in Ptolemy II has the capabilities of generating code for synchronous dataflow and finite state machine models (Tsay 2000). We extended the capabilities of the existing code generator for embedded devices based on the supported models of computation in Ptolemy II.

C. Methods and Materials

In the beginning of our project, the first goal was to narrow the scope. As previously mentioned, the goal of our project was to create an example application using an embedded platform that we could later generate code for using a graphical design environment. Since there are many open source embedded platforms and applications that we could pursue, we needed to choose one platform to design with. There are currently two dominant open source platforms used by the majority of developers: Arduino and Raspberry Pi. Both of these platforms are popular with users because they are easy to use and inexpensive. However, they are very different devices. The Raspberry Pi is essentially a small computer that can run an operating system and the Arduino is a simpler device that is slower than Raspberry Pi but is intended for hardware applications.

Another up-and-coming open source embedded platform are boards with processors using the ARM architecture. These development platforms are called ARM mbed platforms and are created from a collection of industry partners such as Freescale, NXP, and STMicro. There are over 30 mbed devices. The mbed platforms are intended for use in developing devices in the Internet of Things realm. Since mbed devices are more advanced than Arduino devices, our team was interested in using the mbed platform, the Freescale FRDM-KL25Z (referred to as mbed for the remaining paper). As shown in Table 1, the mbed has more speed, memory, and less power than the Arduino Uno. From previous work, code generated files typically have a significant amount of overhead. The mbed offers a platform that would help us ensure we would have enough memory to test our generated code. It also includes an accelerometer, touch sensor, and Arduino-style headers which offer the opportunity to test more supported sensors and Arduino shields. Based on the specifications of the mbed and backing of industry partners, we decided to create our project around the mbed.

Table 1: Arduino Uno vs. Freescale FRDM-KL25Z Specifications

	Arduino Uno	**Freescale FRDM-KL25Z**
Microcontroller	Atmega328	MKL25Z128VLK4
Operating Voltage	5 V	5 V
Flash Memory	32 KB	128 KB
SRAM	2 KB	16 KB
Clock Speed	16 MHz	48 MHz
Accelerometer		X
Capacitive Touch Sensor		X
Integrated Design Environment (IDE)	Arduino IDE	mbed online IDE

After choosing an embedded platform, we wanted to understand the embedded software design process by developing an example application using mbed. Since the end goal was to be able to code generate for the example application, we needed to create an application with a variety of modules and components. Based on the prevalence of the Internet of Things, we decided to create an application involving an interactive light-emitting diode (LED) cube using NeoPixel LEDs that can be controlled by a user using a data glove. The application consists of many areas of embedded systems development that we could code generate for. The data glove, a Virtual Realities DG5, consists of three different sensors: an accelerometer, a gyroscope, and finger bend sensors. A photo from the demo of the application is shown in Figure 1.

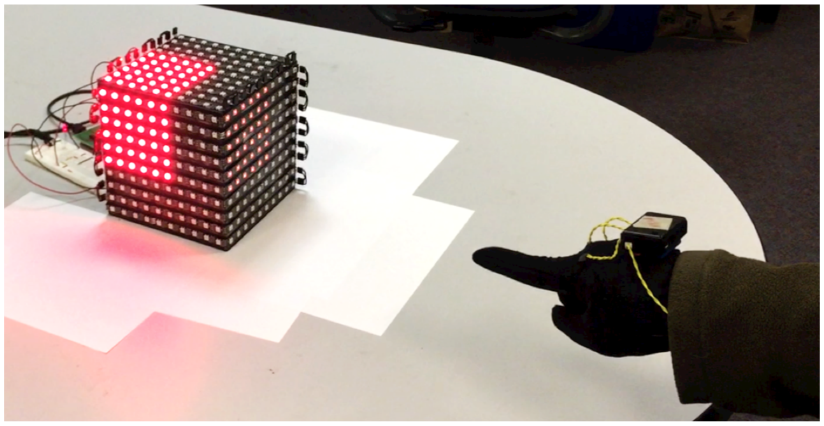


Figure 1: LED Cube Demo with Data Glove

For communication between the major components of the system, our application consisted of a real time network using the TCP/IP socket communication protocol where packets of data were transmitted over Wi-Fi from data glove to a laptop. The laptop routes the data to the mbed device. The mbed receives the packets of data from the laptop through a connected device called the CC3000 using socket communication. The mbed updates the lights on the LED cube based on the information received from the packets. The flow of communication for the application is shown in Figure 2. For our application, only three sides of a cube are used. The LEDs were lighted up to create the image of a three-sided cube, which can be moved and modified based on the gestures recorded from the data glove. José Oyola discusses the creation of the cube further in his paper.

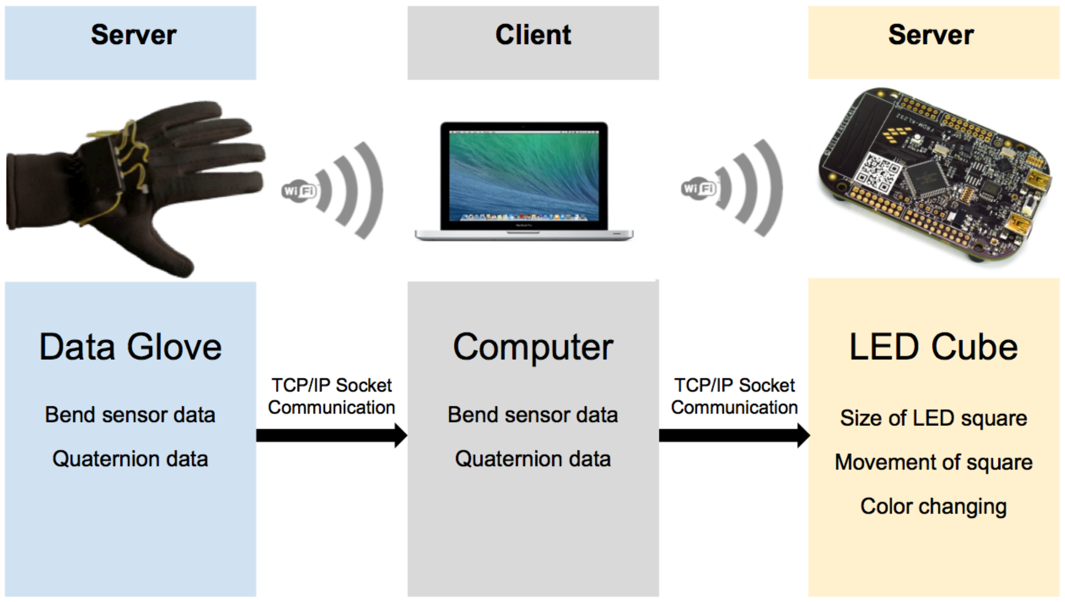


Figure 2 : LED Cube Application Communication Flow Diagram

Overall, the application was modeled as a synchronous dataflow model as shown in Figure 3. This model of computation was chosen because components of our system require data from other components before they can be used and send information to another component. Each packet of data is represented as a token and the numbers in Figure 3 labels the token requirements of each block.

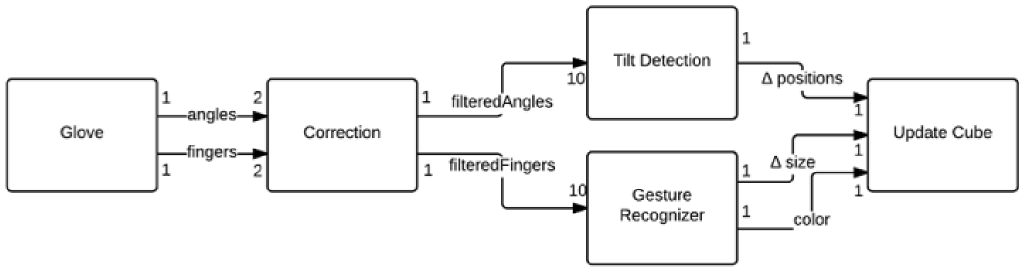
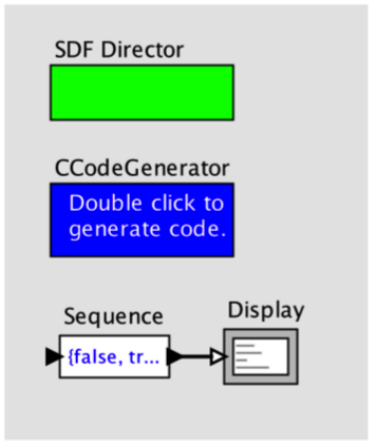


Figure 3: Synchronous Dataflow Model of LED Cube Application[1]

After completing the example application, we began exploring the C code generator in Ptolemy II to begin work on code generation for mbed. First, we looked at the previous work for generating code using Ptolemy II for the Arduino Yun. The goal of the previous work was to be able to blink an LED on an Arduino Yun based on a simple model (shown in Figure 4) created in Ptolemy II. The model consisted of a sequence actor connected to a display actor using the synchronous data flow model of computation. The sequence actor sends true and false alternately to the display actor. Based on the Boolean value received, the display actor sets the LED on or off. Since the display actor is typically used for displaying values to a console, it was modified to execute Arduino code based on the value received at its input. The right side of the Figure 4 shows the code that corresponds to the display actor and used by Ptolemy II in code generation.



```
/***preinitBlock***/
#include "mbed.h"
DigitalOut myled(LED1);
/**/

/*** BooleanPrintBlock($name, $channel) ***/
if ($hasToken(input#$channel)) {
    wait(0.1);
    if ($get(input#$channel)) {
        myled = 1;
    } else {
        myled = 0;
    }
}
/**/
```

Figure 4: Blink LED Model for Arduino Code Generation and Display Actor Code

We found that in Ptolemy II each actor used for code generation must have a corresponding C and Java helper file. These files are looked for when the model is parsed to determine how each actor is related and the code structures it represents. Ptolemy II uses these files to create an abstract syntax tree representation of the code. For synchronous data flow models, during the process of code generation, a schedule of the actors is also created to show the order that each actor should be fired. The schedule is created based on the ordering of the actors in the model. After code generation is complete, many C files are created for the model. A C file is created for each actor in the system based on the code written inside their respective C helper file and the connections to their input and output ports. Another C file contains the firing schedule based on synchronous data flow. A main file is created for running and executing the model based on the parameters set in the director. The code generation process is illustrated in Figure 5.

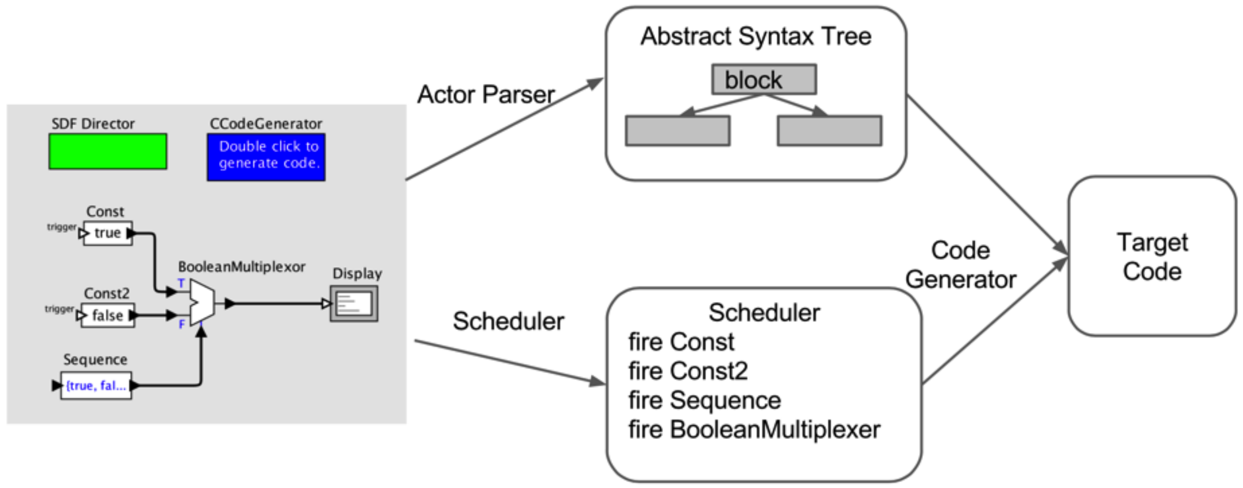


Figure 5: Code Generation Process for a Simple Model

Similar to the Arduino project, our first goal was to get an LED blinking on the mbed. We started by modifying the C helper file for the display actor with the code required to blink an LED on the mbed. After trying to generate code for the mbed for the first time and running it on the online compiler, we learned that there was many changes that needed be made in order to properly compile and run the code. We first made changes to the code in the online compiler to determine what was needed to compile the code. The main issues that we faced were the labelling of files and content of the files. Since the Ptolemy II code generator produces C code, and the mbed online compiler uses a C++ compiler, there are subtle differences that needed to be fixed. We tracked down the changes that needed to be made in Ptolemy II which were in the C Code Generator files. In order to compile the generated code directly from Ptolemy II, we also needed to create a makefile that utilizes a desktop mbed compiler. Naren Vasanad and Kevin Albers talk in more detail about makefiles and the GNU for ARM toolchain.

After successfully getting an LED to blink on the mbed, the next step was to use other actors and a finite state machine. Figure 6 shows an example of a finite state machine model used to change the color of the LED on the mbed. In Ptolemy II, finite state machine models are refined in a modal model actor. In this example, the input to the modal model is a sequence actor, which repeatedly fires true. The finite state machine contains three states, each representing a color of the LED, and transitions each time it receives true and a counter is equivalent to a certain number. When the counter is not equivalent to a certain number, the counter is incremented. The counter is reset each time the finite state machine enters a new state. The modal model fires a token based on the actions of each transition. The actions correspond to the color of the LED that should be set. The embedded code actor uses its input to change the color of the LED based on the code within it. It is treated by the code generator similarly to the display actor as it has a unique C helper file, but it uses the code written by the user within the actor.

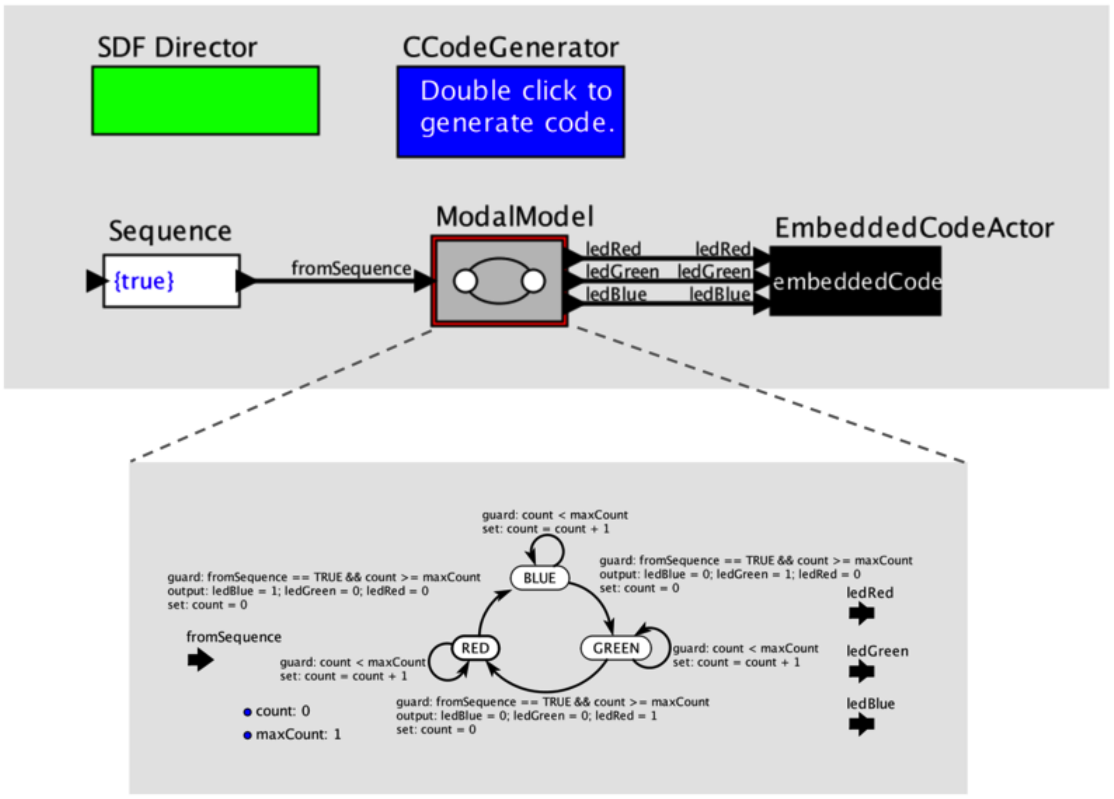


Figure 6: Synchronous Dataflow and Finite State Machine Model of LED

Using the above models, we found that there are flaws in the generated code and memory leaks occur when running the software on our embedded platforms. In particular, the blinking LED model in Figure 4 would only run for about 100 iterations before stalling, and the finite state machine model in Figure 6 would not even run on the mbed. Since our generated code already takes up a large portion of the memory on the mbed, we decided to test our code on a platform with larger memory to verify whether it could be a code size or memory leak issue. Using the Freescale FRDM-K64F, the finite state machine model in Figure 6 was able to run for a few iterations. Thus, we suspected a memory leak was occurring somewhere in the generated code.

In order to help determine the cause of the memory leak issue, we used Valgrind to analyze our generated code. Valgrind is a suite of software tools for analyzing programs in detail (Valgrind). In particular, when given an executable file, Valgrind's memory error detector can be used to determine how memory is being allocated and if data is being lost. Our first goal was to determine how to get an LED on the mbed board to blink infinitely. Using the blink LED model in Figure 4, we generated code based on 100 firings of the sequence actor and created an executable to be analyzed by Valgrind. Based on the output produced by Valgrind as shown in Figure 7, 1,600 bytes of data were definitely lost (16 bytes for each firing). Figure 8 shows the offending files and line numbers in the program where Valgrind found the memory errors.

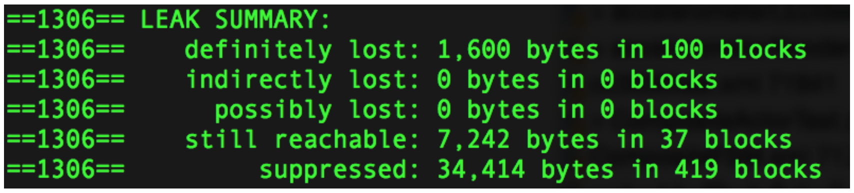
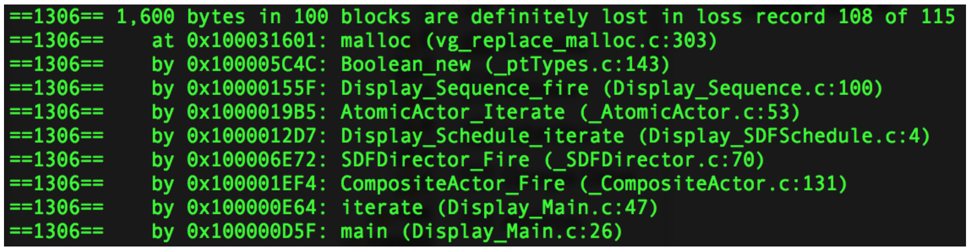

```
==1306== LEAK SUMMARY:
==1306==    definitely lost: 1,600 bytes in 100 blocks
==1306==    indirectly lost: 0 bytes in 0 blocks
==1306==      possibly lost: 0 bytes in 0 blocks
==1306==    still reachable: 7,242 bytes in 37 blocks
==1306==         suppressed: 34,414 bytes in 419 blocks
```

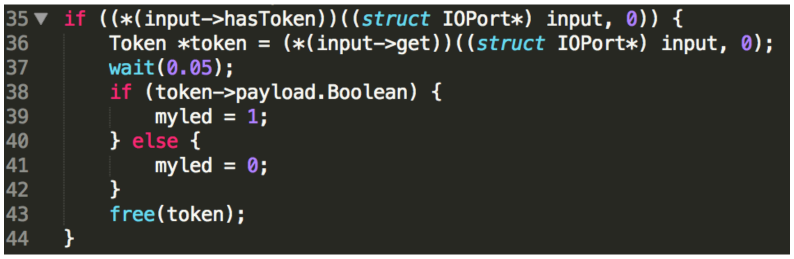
Figure 7: Valgrind Memory Analysis Summary of Blink LED Model



```
==1306== 1,600 bytes in 100 blocks are definitely lost in loss record 108 of 115
==1306==    at 0x100031601: malloc (vg_replace_malloc.c:303)
==1306==    by 0x100005C4C: Boolean_new (_ptTypes.c:143)
==1306==    by 0x10000155F: Display_Sequence_fire (Display_Sequence.c:100)
==1306==    by 0x1000019B5: AtomicActor_Iterate (_AtomicActor.c:53)
==1306==    by 0x1000012D7: Display_Schedule_iterate (Display_SDFSchedule.c:4)
==1306==    by 0x100006E72: SDFDirector_Fire (_SDFDirector.c:70)
==1306==    by 0x100001EF4: CompositeActor_Fire (_CompositeActor.c:131)
==1306==    by 0x100000E64: iterate (Display_Main.c:47)
==1306==    by 0x100000D5F: main (Display_Main.c:26)
```

Figure 8: Files Involved with Memory Leak for Blink LED Code Generation

From analyzing the offending files, we found that tokens received from the Display actor are not being freed. As mentioned during the code generator structure, the $get macro is used to obtain the value of the token at the input of the actor when it is received. However, the macro is only getting the token, but it does not free it after it is consumed. Thus, we have currently fixed the problem by adding code to the actor to create a token and free it after it has been used. The code shown in Figure 9 is used in the Display.c actor to received boolean tokens for the Blink LED model of Figure 4 and results in no memory leaks.



```
if ((*(input->hasToken))((struct IOPort*) input, 0)) {
    Token *token = (*(input->get))((struct IOPort*) input, 0);
    wait(0.05);
    if (token->payload.Boolean) {
        myled = 1;
    } else {
        myled = 0;
    }
    free(token);
}
```

Figure 9: Code for Display.c to Fixed Memory Leaks

Another actor that we were able to temporarily resolve memory leaks for in Ptolemy II is the EmbeddedCodeActor. The EmbeddedCodeActor is a useful actor because it allows us to directly write the C code to be generated. Figure 10 shows a model that uses an EmbeddedCodeActor to light up a NeoPixel LED strip connected to a digital pin on the mbed platform. In this model, the EmbeddedCodeActor (NeoPixel_LED_Strip), receives a sequence of values from IndexSequence and ColorSequence to determine which LED on the strip to turn on and set its color respectively.

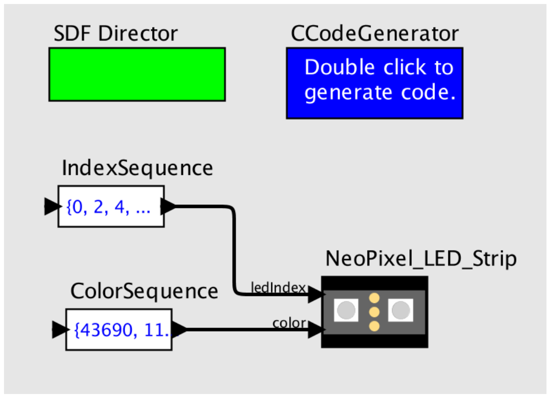
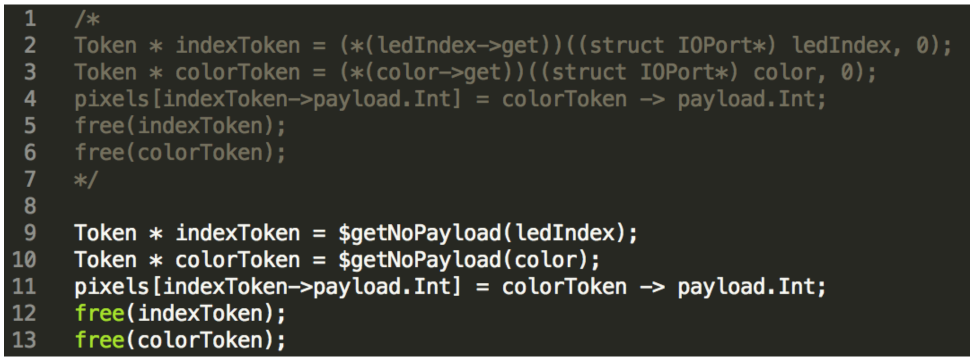


Figure 10: Ptolemy II Model with EmbeddedCodeActor (NeoPixel_LED_Strip)



```
/*
Token * indexToken = (*(ledIndex->get))((struct IOPort*) ledIndex, 0);
Token * colorToken = (*(color->get))((struct IOPort*) color, 0);
pixels[indexToken->payload.Int] = colorToken -> payload.Int;
free(indexToken);
free(colorToken);
*/

Token * indexToken = $getNoPayload(ledIndex);
Token * colorToken = $getNoPayload(color);
pixels[indexToken->payload.Int] = colorToken -> payload.Int;
free(indexToken);
free(colorToken);
```

Figure 11: NeoPixel_LED_Strip EmbeddedCodeActor Code

As shown in Figure 11, similar to the Display Actor in the Blink LED model, we needed to create and free the token received by the EmbeddedCodeActor. To simplify the creation of the code in the actor, we created a macro called $getNoPayload that returns the necessary code as shown in the commented code of Figure 11. Using the code shown above, we were able to light up of the NeoPixel LED without any memory leaks and have been able to use multiple EmbeddedCodeActors for other applications involving our LED cube.

Most of the work for our project was modifying the existing C code generator to program the mbed and ensure that it was producing robust code. EmbeddedCodeActors were beneficial for rapidly testing the code we had written for our LED Cube application in Ptolemy II and ensuring the code generator was producing code that allowed the application to work as expected. Using the EmbeddedCodeActors, we were able to create a model to represent our LED Cube application and successfully generate code for the mbed to replicate the application based on the code we had written on our own. Based on the EmbeddedCodeActors used for the working LED Cube model, we found out how to create and add actors to Vergil in Ptolemy II in order for users to easily recreate our application or extend the actors for their own use. The final model with our created actors is shown in Figure 12, Figure 13, and Figure 14.

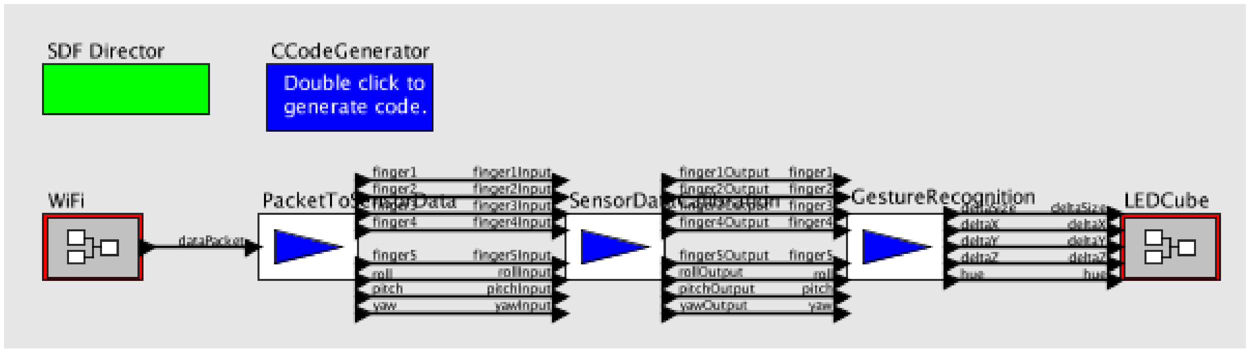


Figure 12: LED Cube Application SDF Model

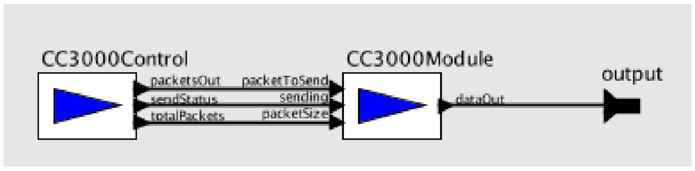


Figure 13: Contents of WiFi Composite Actor in LED Cube Application Model

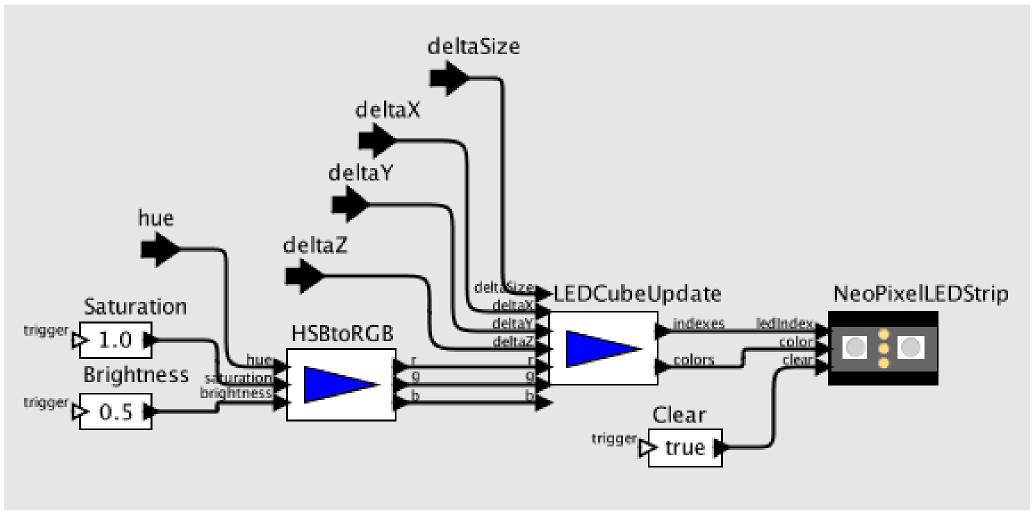


Figure 14: Contents of LED Cube Composite Actor in LED Cube Application Model

D. Results and Discussion

The model-based design process is used to accelerate development time and improve reliability of embedded software. For our capstone project, we began by using traditional methods to write C code for an example application, a lighted LED cube controlled by a data glove. The application was overall successful as a user was able to control the movement, size, and color of a lighted cube based on different gestures using a data glove. Using wireless communication via Wi-Fi, the gestures from the data glove were discerned and the status of individual LEDs were updated. We wrote the embedded software for our application using objected oriented programming concepts to make it as modular as possible for use in code generation. Through the experience of using traditional methods, we learned about the challenges of debugging embedded software and how model-based design could be useful.

By creating a graphical design environment using Ptolemy II for code generation, we were able to successfully generate code for simple models such as blinking an LED and toggling input and output pins. The code was able to compile on the online mbed compiler and then, we were able to create a makefile with an offline compiler in order to directly generate and compile code from Ptolemy II. Using EmbeddedCodeActors, we were able to create a model representing our example LED cube application and successfully generate code to produce a similar functioning application. The use of graphical models has shown to be much easier to use and quicker than writing the code for the application. However, further work needs to be done to optimize the code size and enable the use of more complex models.

V. Concluding Reflections

Although there is still much work that needs to be done in order for code generation in Ptolemy II to be beneficial and a better alternative to traditional programming for embedded devices, our capstone project demonstrates that it is possible to offer an embedded software developer a tool for rapid prototyping using a graphical design environment to represent their application. With graphical models, users can spend less time debugging syntax errors from traditional programming languages and more time developing their applications. By using an mbed device, the Freescale FRDM-K64F, as our target application for code generation, we showed that the use of a graphical design environment can be truly beneficial to developers. Based on the major elements of our LED cube application which used a network of sensors and actuators, we were able to demonstrate with our project the ability to generate reliable code using Ptolemy II to replicate many elements in our cube application such as using the accelerometer and controlling the lighting of the LEDs.

Besides the development of a graphical design environment for code generation, another primary deliverable for our project was the documentation of our work and the creation of a user guide. Since our work focused on one family of embedded devices, the ultimate goal would be the ability for the code generator in Ptolemy II to be expanded to other devices. One of the biggest challenges when our group started working on modifying the code generator in Ptolemy II was the lack of documentation regarding the topics and files surrounding the code generator. The documentation that we have created should enable new developers to extend Ptolemy II for their own platforms using the SDF and FSM models of computation.

Based on the work we have done with our capstone project, there are many areas of code generation in Ptolemy II that could be recommended for future research. From our original schedule created through project management, we had expected to explore code generation for other models of computation such as Discrete Event. However, due to different issues that arose from SDF and FSM models, we did not have the chance to extend our work to other models of computation and aspects of model-based design. Future work could be done to improve the current code generator for other actor types and extend the code generator for other models of computation. Furthermore, there are opportunities to explore model-based testing techniques based on the work we have done to ensure reliability and efficiency of the code generated.

References

Alur, Rajeev, Thao Dang, Joel Esposito, Yerang Hur, Franjo Ivancic, Vijay Kumar, Insup Lee Pradyumna Mishra, George Pappas, and Oleg Sokolsky. "Hierarchical Modeling of Embedded Systems," Proceedings of the IEEE, Vol. 91, No. 1, Jan. 2003.

Audris Mockus, Roy T. Fielding, and James D. Herbsleb. "Two case studies of open source software development: Apache and Mozilla." *ACM Trans. Softw. Eng. Methodol.* 11, 3 (July 2002), 309-346, Web. 16 Feb. 2015. <http://dl.acm.org/citation.cfm?id=567795>

"Buy LabVIEW." - *National Instruments*. National Instruments, n.d. Web. 25 Nov. 2014. <http://www.ni.com/labview/buy/>

Clarice Technologies. "Demystifying the Internet of Things." *Thinking Products: A Weblog by Clarice Technologies*, Clarice Technologies, 6 Mar. 2014. Web. 16 Feb. 2015. <http://blog.claricetechnologies.com/2014/03/demystifying-the-internet-of-things/>

Dellas, Christina M., and Hogan, Kevin M. Statechart Development Environment with Embedded Graphical Data Flow Code Editor. National Instruments Corporation, assignee. Patent US 8,387,002 B2. 26 Feb. 2013. Print.

Deshpande, A. and Riehle, D., *IFIP International Federation for Information Processing*, Volume 275; *Open Source Development, Communities and Quality*; Barbara Russo, Ernesto Damiani, Scott Hissam, Björn Lundell, Giancarlo Succi; Boston: Springer, 2008. pp. 197–209.

Engelfriet, A. "Choosing an Open Source License." IEEE Software 27.1 (2010): 48-49. Print.

"Engineering, Scientific & CAD/CAM Software" Hoover's Online. 2015. Web. 16 Feb. 2015.

Fitzgerald, Brian. "The Transformation of Open Source Software." MIS Quarterly. Vol. 30, No. 3 (Sep., 2006) , pp. 587-598. Web. 16 Feb 2015. <http://www.jstor.org/stable/25148740>

"Gartner's 2014 Hype Cycle for Emerging Technologies Maps the Journey to Digital Business", *Gartner,* 11 Aug. 2014, Web. Nov. 2014. <http://www.gartner.com/newsroom/id/2819918>

Girault, Alain, Bilung Lee, and Edward Lee. "Hierarchical Finite State Machines with Multiple Models of Currency," Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, Vol. 18, No. 6, Aug. 6 2002.

Hulkower, Billy. "Living Online - US - May 2014." *In Mintel*. n.d. Web. 13 Feb. 2015. <http://academic.mintel.com/display/704619/?highlight>

"Internet of Things Market & M2M Communication", *Markets and Markets*, Nov. 2014, Web. Nov. 2014. <http://www.marketsandmarkets.com/Market-Reports/internet-of-things-market-573.html>

Jensen, Jeff. "Elements of Model-Based Design," Technical Report No. UCB/EECS-2010-19, EECS Department, University of California, Berkeley, Feb. 19, 2010.

Jensen, J. C., Chang, D. H. and Lee, E.A., 2011, "A model-based design methodology for cyber-physical systems", *Proceedings of the International Wireless Communications and Mobile Computing Conference* . IWCMC 2011 . pp. 1666-1671. Print.

Justyna Zander, Ina Schieferdecker, and Pieter J. Mosterman, 2011 "Model-Based Testing for Embedded Systems", *CRC Press.* Boca Raton: Taylor and Francis Group, 2013. Web. 16 Feb. 2015. <http://dx.doi.org/10.1201/b11321-1>

Kahn, Sarah, IBISWorld Industry Report 51121: Software Publishing in the US. Dec. 2014. Web. 13 Feb. 2015.

Karsai, Gabor, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. "Model-Integrated Development of Embedded Systems," Proceedings of the IEEE, Vol. 91, No. 1, Jan. 2003.

Lee, Bilung, and Edward Lee. "Interaction of Finite State Machines and Concurrency Models," Proceeding of Thirty Second Annual Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California, Nov. 1998.

Lee, Edward, and Stephen Neuendorffer. "Concurrent models of computation for embedded software," Technical Memorandum No. UCB/ERL M04/26, EECS Department, University of California, Berkeley, July 22, 2004.

Lee, Edward. "Cyber Physical Systems: Design Challenges," Technical Report No. UCB/EECS-2008-8, EECS Department, University of California, Berkeley, Jan. 23 2008.

Lee, Edward. "Finite State Machines and Modal Models in Ptolemy II." Technical Report No. UCB/EECS-2009-151, EECS Department, University of California, Berkeley, Nov. 1 2009.

Lindman, J.; Paajanen, A.; Rossi, M., "Choosing an Open Source Software License in Commercial Context: A Managerial Perspective," *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, 237-44, 1-3 Sept. 2010

"Links." *Ptolemy Project*. UC Berkeley, 26 July. 2014. Web. http://ptolemy.eecs.berkeley.edu/archive/links.htm, accessed February 28, 2015.

Ma, Tao, and Chunhong Zhang. "On the Disruptive Potentials in Internet of Things." *Proceedings 17th IEEE International Conference on Parallel and Distributed Systems: ICPADS 2011:*

7-9 December 2011, Tainan, Taiwan. Los Alamitos, Calif: IEEE Computer Society Conference Publications, 2011. 857-59. Print.

Mueller, W., Becker, M., Elfeky, A., DiPasquale, A., "Virtual prototyping of Cyber-Physical Systems," *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*, 219-26, 30 Jan. 2012-2 Feb. 2012. Print.

Øyvind Hauge, Daniela Soares Cruzes, Reidar Conradi, Ketil Sandanger Velle, and Tron André Skarpenes, “Risks and Risk Mitigation in Open Source Software Adoption: Bridging the Gap between Literature and Practice” *Proceedings of 6th International IFIP WG 2.13 Conference on Open Source Systems, Open Source Software: New Horizons, Notre Dame, IN, USA, May 30 - June 2 2010.* Springer. 2010. Web. 16 Feb. 2015. <http://link.springer.com/book/10.1007%2F978-3-642-13244-5>

Øyvind Hauge and Sven Ziemer, “Providing Commercial Open Source Software: Lessons Learned”, *Proceedings of 5th IFIP WG 2.13 International Conference on Open Source Systems, Open Source Ecosystems: Diverse Communities Interacting, Skövde, Sweden, June 3-6, 2009* Springer. 2009. Web. 16 Feb. 2015. <http://www.springer.com/computer/general+issues/book/978-3-642-02031-5>

Porter, Michael. "How Competitive Forces Shape Strategy." *Harvard Business Review*, vol. 57, no. 2, 137-45. Mar. 1979. Print.

Porter, Michael. "The Five Competitive Forces That Shape Strategy." *Harvard Business Review.* Jan. 2008. Print.

"Pricing and Licensing." *MATLAB and Simulink Overview*. MathWorks, n.d. Web. 25 Nov. 2014. <http://www.mathworks.com/pricing-licensing/index.html?intendeduse=home>

Ptolemaeus, Claudius. Editor. *System Design, Modeling, and Simulation Using Ptolemy II*, Ptolemy.org, 2014.

"Ptolemy II." *Ptolemy Project*. UC Berkeley, n.d. Web. <http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm>

"Ptolemy II Frequently Asked Questions." *Ptolemy Project*. UC Berkeley, 18 Dec. 2014. Web. <http://ptolemy.eecs.berkeley.edu/ptolemyII/ptIIfaq.htm#ptolemy%20II%20copyright>

"Sponsors of the Ptolemy II Project." Ptolemy Project. UC Berkeley. Web. <http://ptolemy.eecs.berkeley.edu/sponsors.htm, accessed 14 Apr. 2015>

Tsay, Jeff. "A Code Generation Framework for Ptolemy II," EECS Department, University of California, Berkeley, ERL Technical Report UCB/ERL No. M00/25, May 19, 2000.

Valgrind. Web. <http://valgrind.org/>.

Vieri del Bianco, Luigi Lavazza, Sandro Morasca, and Davide Taibi, "Quality of Open Source Software: The QualiPSo Trustworthiness Model", *Springer*, 2009, Web. 16 Feb. 2015.