

**The Edward S. Rogers Sr. Department of  
Electrical and Computer Engineering**

University of Toronto

**ECE496Y Design Project Course**

**Group Final Report**

Title: Move Smart Contract Synthesis and Verification

Team #:	2020270	
Team members:	Name:	Email:
	Pranavbhai Patel	pranavbhai.patel@mail.utoronto.ca
	Kevin Raihanizadeh	kevin.raihanizadeh@mail.utoronto.ca
	Arjun Mittal	arjun.mittal@mail.utoronto.ca
	Rohit Arora	roh.arora@mail.utoronto.ca
Supervisor:	Veneris, Andreas	
Section #:	2	
Administrator:	Phang, Khoman	
Submission Date:	March 30th, 2021	

## Group Final Report Attribution Table

This table should be filled out to accurately reflect who contributed to each section of the report and what they contributed. Provide a **column** for each student, a **row** for each major section of the report, and the appropriate codes (e.g. ‘RD, MR’) in each of the necessary **cells** in the table. You may expand the table, inserting rows as needed, but you should not require more than two pages. The original completed and signed form must be included in the hardcopies of the final report. Please make a copy of it for your own reference.

Section	Student Names			
	Pranav	Kevin	Arjun	Rohit
Group Highlights	MR, ET	RD	MR, ET	MR, ET
Individual Contributions	RD - Pranav	RD - Kevin	RD - Arjun	RD - Rohit
Executive Summary	MR	RD	MR	MR
Introduction	MR	RD	MR	MR
Final Design	ET	ET	RD	RD for 2.3.5
Testing and Verification	RD	ET	ET	RD for 3.7, MR
Summary and Conclusions	ET	RD	ET	MR
Appendix A				RD, MR
Appendix B	MR, ET			RD
All	FP, CM	FP, CM	FP, CM	FP, CM, OR1

### Abbreviation Codes:

Fill in abbreviations for roles for each of the required content elements. You do not have to fill in every cell. The “**All**” row refers to the complete report and should indicate who was responsible for the final compilation and final read through of the completed document.

RS – responsible for research of information

RD – wrote the first draft

MR – responsible for major revision

ET – edited for grammar, spelling, and expression

OR – other

“All” row abbreviations:

FP – final read through of complete document for flow and consistency

CM – responsible for compiling the elements into the complete document

OR - other

If you put OR (other) in a cell please put it in as OR1, OR2, etc. Explain briefly below the role referred to:

OR1: References

OR2: enter brief description here

### Signatures

By signing below, you verify that you have read the attribution table and agree that it accurately reflects your contribution to this document.

<b>Name</b>	Pranavbhai Patel	<b>Signature</b>	Pranavbhai Patel	<b>Date:</b>	2021/03/30
<b>Name</b>	Kevin Raihanizadeh	<b>Signature</b>	Kevin Raihanizadeh	<b>Date:</b>	2021/03/30
<b>Name</b>	Arjun Mittal	<b>Signature</b>	Arjun Mittal	<b>Date:</b>	2021/03/30
<b>Name</b>	Rohit Arora	<b>Signature</b>	Rohit Arora	<b>Date:</b>	2021/03/30

## Voluntary Document Release Consent Form

To all ECE496 students:

To better help future students, we would like to provide examples that are drawn from excerpts of past student reports. The examples will be used to illustrate general communication principles as well as how the document guidelines can be applied to a variety of categories of design projects (e.g. electronics, computer, software, networking, research).

Any material chosen for the examples will be altered so that all names are removed. In addition, where possible, much of the technical details will also be removed so that the structure or presentation style are highlighted rather than the original technical content. These examples will be made available to students on the course website, and in general may be accessible by the public. The original reports will not be released but will be accessible only to the course instructors and administrative staff.

Participation is completely voluntary and students may refuse to participate or may withdraw their permission at any time. Reports will only be used with the signed consent of all team members. Participating will have no influence on the grading of your work and there is no penalty for not taking part.

If your group agrees to take part, please have all members sign the bottom of this form. The original completed and signed form should be included in the hardcopies of the final report.

Sincerely,  
Khoman Phang  
Hamid Timorabadi  
ECE496Y Course Coordinators

### Consent Statement

We verify that we have read the above letter and are giving permission for the ECE496 course coordinator to use our reports as outlined above.

Team #: 2020270

Project Title: Move Smart Contract Synthesis and Verification

Supervisor: Andreas, Veneris

Administrator: Khoman Phang

<b>Name</b>	<u>Pranavbhai Patel</u>	<b>Signature</b>	<u>Pranavbhai Patel</u>	<b>Date:</b>	<u>2021/03/30</u>
<b>Name</b>	<u>Kevin Raihanizadeh</u>	<b>Signature</b>	<u>Kevin Raihanizadeh</u>	<b>Date:</b>	<u>2021/03/30</u>
<b>Name</b>	<u>Arjun Mittal</u>	<b>Signature</b>	<u>Arjun Mittal</u>	<b>Date:</b>	<u>2021/03/30</u>
<b>Name</b>	<u>Rohit Arora</u>	<b>Signature</b>	<u>Rohit Arora</u>	<b>Date:</b>	<u>2021/03/30</u>

## **Group Highlights** (author: K. Raihanizadeh)

The team has completed all planned milestones in their respective deadlines following the Gantt chart in Appendix A. The latest being the reverse verification flow defined as Milestone 5: Model Generation from Code from the Gantt chart. This is done by inserting a Move smart contract into the code editor of the application shown in Figure 27. The user can then generate a transition system which is necessary before the verification flow can execute. The steps involved in this are: (1) running the contract through the Move language parser ensuring correct syntax. It is then (2) transformed into a transition system shown in the model editor in Figure 9, which can then be (3) verified for safety and liveness properties using the verification plugin.

The team has also completed the forward verification flow stated in Milestone 3, generating a Move contract from a user-defined transition system. The user can define their model through the model editor panel, defining states and transitions associated with the system. Safety and liveness properties can also be defined which are tested when verifying the correctness of the model through the verification plugin. Once the model is verified, the associated smart contract (code) is generated which can both be viewed in the code editor panel or downloaded.

In addition to code generation, UI refinement was completed as stated in Milestone 6. The goal of refinement involved making improvements to the verification workflow. This primarily consisted of developing a new screen for specifying the safety and liveness properties used when verifying a model. Previously, the user specified properties through an unintuitive interface, and the properties had to be re-written on each verification run. The new screen developed by the team persists all properties to the model and presents a better user experience. Persisting properties to individual models is especially important in facilitating collaboration and developing multiple contracts at once.

Lastly, the team originally planned on hosting the application on the web, however, this was deemed unnecessary and dropped. Despite no longer hosting the application, a docker image for the application was still created and it allows users to develop in a consistent environment. Furthermore, it allows any stakeholders interested in the project to easily set up a local copy of the application for exploration purposes, simplifying the setup. Refer to the Individual Contributions sections for details on individual responsibilities mentioned above.

## **Individual Contributions** (author: R. Arora)

Throughout the project, the team practiced Agile when developing. As a result of agile development, the team often shipped smaller components instead of large tasks or milestones at once. Due to the nature of software development itself, the team also participated in pair programming. This section outlines an overview of all tasks individual team members participated in. All members participated in Move Language Preliminary Research and VeriSolid Exploration before branching out to more specific tasks.

## Individual Contributions - Arjun Mittal (author: A. Mittal)

Arjun did a lot of work centered around discovering the core components of the Move Language, and an associated parser capable of crawling the Move language syntax. With the key components mapped out, this assisted greatly in modifying the core verification algorithms and code templating from VeriSolid. This also assisted in developing a mock auction contract for testing the verification and code generation workflow. Additionally, Arjun also wrote docker files to unify testing/development environments.

Table 1: Summary of Arjun's Technical Work

Task	Description	Status
Parser Implementation + Augment Statement	The core parser functionality was leveraged from a generic library known as tree-sitter and integrated with the move-tree-sitter package defining the syntax rules. Arjun also wrote an algorithm to parse the resulting tree structure of the syntax tree.	Completed
BIP Templating + Verification	Within the verification flow, Arjun worked on specifically integrating the BIP templating, formatting checks, and NuSMV usage. The BIP templating formats were leveraged from VeriSolid. The formatting checks and NuSMV usage were also adapted from the VeriSolid algorithm.	Completed
Mock Auction Contract	Arjun composed a mock auction contract that can be represented as a Finite State Machine (FSM) within the VeriMove GUI. The main components of the auction contract include: create, bidding, and withdrawal stage and the associated guards, and actions defined in Move.	Completed
Docker Orchestration	Arjun wrote a docker-compose YAML file to orchestrate the correct start-up and shutdown of VeriMove services. Additionally, the docker-compose file ensured a uniform environment.	Completed
Documentation	Arjun worked with Rohit to create a GitHub wiki documenting all necessary development processes of the system.	Completed

## Individual Contributions - Pranav Patel (author: P. Patel)

Early in the project, Pranav assisted in the verification flow and later transitioned to primarily UI changes. Through pair programming or discussion sessions, Pranav also provided input in other areas such as a second set of eyes for the sample Auction contract. The Verification Properties visualizer was developed in entirety by Pranav using webgme-react-viz support. In addition, Pranav was responsible for integrating the code editor with the code generation process. See Table 2 for tasks specifically owned by Pranav.

Table 2: Summary of Pranav's Technical Work

Task	Description	Status
Define Model Editor Panel	Pranav investigated VeriSolid's custom Model Editor Visualizer. Pranav concluded that a custom visualizer was not needed, and the base WebGME Model Editor can be leveraged.	Completed
FSM to Augmented Transition System	Pranav finished writing the augment model functionality as a separate module which leverages the augment statement work completed by Arjun. The implementation is taken from VeriSolid with minor changes where Solidity-specific logic was included.	Completed
Verification Properties Visualizer UI	The initial verification flow had users enter CTL properties through WebGME's plugin GUI. This GUI was not user-friendly and did not persist properties to the model itself. As an alternative, Pranav wrote a visualizer to handle CTL property specifications.	Completed
Integrate Visualizer into Verification Flow	The verification flow was updated to leverage the newly defined Verification Properties Visualizer.	Completed
Configure CI Process	To facilitate ongoing development with minimal regressions, Pranav set up GitHub Actions which enforced all automated tests must pass for any new changes to be accepted.	Completed
Display generated code in Code Editor	The code generation flow outputs a code file. However, this code was not visible in the application itself. Pranav updated the Code Editor Visualizer to allow users to view generated code directly in the editor instead of having to view it separately.	Completed

## Individual Contributions - Kevin Raihanizadeh (author: K. Raihanizadeh)

After the initial investigation, Kevin's work primarily revolved around code generation from a transition system. Alongside Arjun, Kevin helped with researching the syntax and semantics of the Move language. Through pair programming, Kevin worked on creating Move templates to format the code when creating a smart contract from a transition system. Kevin then aided Rohit in creating the FSM generation functionality.

Table 3: Summary of Kevin's Technical Work

Task	Description	Status
NuSMV Verification	Kevin helped add the existing NuSMV tool to model verification.	Completed
Friendly UI for verification	Kevin aided Pranav in creating the verification properties screen to add and remove properties from being checked in the model.	Completed
Generate code	Kevin and Arjun worked on creating templates to map FSM characteristics into a compilable Move contract.	Completed
Investigate reverse flow	Kevin helped with the initial investigation on the implementation of the reverse flow, determining whether it should apply to a general contract or a contract that meets the team's generation specifications.	Completed
Syntax Highlighting for Move	Kevin and Rohit used Rust Syntax highlighting for the code editor as it is in line with Diem's standards.	Completed
FSM Generation	Aiding Rohit, Kevin developed supplementary functions (parsing model attributes) for the creation of the FSM generation functionality.	Completed



## Individual Contributions - Rohit Arora (author: R. Arora)

After the investigation stage, Rohit created a Figma schematic of how the interface of the application should look along with Pranav, and Arjun. This UI was later scrapped due to the difficulties with working on the front-end components of WebGME. This is further described in Appendix E. Rohit's focus then shifted to building and creating core features of the application, including the CTL verification properties generator functions. In the later stages of the development, Rohit completed the FSM generation feature, created a full licensing file, and wrote the documentation surrounding the usage around the application. Table 4 displays the most important tasks for Rohit.

Table 4: Summary of Rohit's Technical Work

Task	Description	Status
Define Move Code Editor	Using the open-source package, Ace Editor, Rohit added a functional code editor to the application.	Completed
CTL Properties Verification	Rohit wrote the CTL properties generator functionality as separate modules. Implementation is similar to VeriSolid but minor changes where Solidity-specific logic and unnecessary inputs were removed.	Completed
Syntax Highlighting for Move	Rohit and Kevin added Rust Syntax highlighting for the Code editor as it is in line with Diem's standards. There is no complete syntax highlighting specified for Move as of yet, so the Rust approach was taken.	Completed
FSM Generation	To have the application create FSMs from Move code, Rohit developed multiple components including, creating nodes for states & transitions and having the FSM appear on the Model Editor panel.	Completed
Licensing	Rohit created a licensing document that contained all the copyright information for all the packages, as well as transitive packages to enable public deployment.	Completed
Documentation	Rohit worked with Arjun to create a GitHub wiki documenting all necessary development processes of the system, as well as how to use the application.	Completed

## **Acknowledgements** (author: All)

We gratefully acknowledge Andreas Veneris for allowing us to work on this project and giving us the resources to actually go about development.

We are grateful to Keerthi Nelaturu for her continued assistance throughout the project. She played a critical role by being an amazing mentor for our team. She spent her time improving our understanding, giving us more resources to look through, and guided our development process.

This project was greatly supported by the VeriSolid, and WebGME teams. The projects developed by these teams acted as wonderful resources that highly influenced the design choices of our project.

We have to express our appreciation to Khoman Phang for his comments on earlier versions of the documents.

## **Executive Summary** (author: K. Raihanizadeh)

### **Background, Motivation & Goal** (author: K. Raihanizadeh)

Smart contracts are agreements specified as code that are stored and executed on blockchain networks. Once established on the blockchain, smart contracts become immutable and cannot be edited. A new blockchain has recently been introduced by the name of Diem, and its associated programming language, Move, for defining smart contracts. Like many programming languages, Move lacks formal semantic definition and developer familiarity. This can lead to unexpected bugs and contracts that are much more vulnerable to malicious attacks. This has happened previously with the Ethereum Network, where attacks on the DAO and the Multisignature Parity Wallet Library led to more than \$330 million worth of cryptocurrency being stolen. To verify the correctness and avoid potential security breaches, the method of formal verification is necessary. It verifies correct smart contract design through mathematical models for safety and liveness properties. Currently, no tool that does the above exists for Move. With this in mind, the goal of this project is to create an application that helps developers write functionally correct Move smart contracts by applying the method of formal verification.

### **Requirements** (author: K. Raihanizadeh)

Basic requirements we defined for the following application include accessibility through a web browser, the ability to define an FSM known as a transition system diagram, be able to perform model verification, explain erroneous design states, and generate compilable Move code.

### **Final Design** (author: K. Raihanizadeh)

To accomplish this goal the team has created an application called VeriMove, consisting of a graphical interface, a Move language parser, a verification system, and an FSM generator. The graphical interface consists of two primary components, a code editor and a model editor. A user can create a smart contract defined as a finite state machine through the model editor. Defining the contract as a finite state machine abstracts semantics of the Move language simplifying the creation process. The user can then specify security properties the model must adhere to, ensuring its correctness using the verification system. If successful, the application will output a fully functional Move smart contract. If the user instead already has a Move smart contract, they can input it into the code editor portion of the application. Leveraging the FSM generator, the system is transformed into a transition system so it can be processed by the verification system.

**Testing** (author: K. Raihanizadeh)

The primary method of tests was to write automated test cases that verify core functionality of the application which all changes must adhere to. We also used manual testing for simulating end to end tests. Lastly, testing by similarity was used on modules and algorithms already proven to work such as algorithms in the verification system.

**Applications and future work** (author: K. Raihanizadeh)

This project is applicable to any developer wanting to ensure functional correctness of a Move smart contract before being deployed on the Diem network. Future work for the project includes supporting other languages used on the blockchain as well as incorporating the official Move compiler when it is released.

## **Table of Contents**

1.0 – Introduction	1
1.1 – Background Research	1
1.2 – Motivation	1
1.3 – Project Goal	2
1.4 – Requirements	2
2.0 – Final Design	4
2.1 – System-level Overview	4
2.2 – System Block Diagram	5
2.3 – Module-level Description and Designs	6
2.4 – Assessment of final design	12
3.0 – Testing and Verification	14
3.1 – Web Application	15
3.2 – Design Verification	17
3.3 – Move Code (Input and Output)	20
3.4 – Model Definition Capabilities	22
3.5 – User Interface Accessibility	22
3.6 – Code Editor Syntax Highlighting	23
3.7 – Reverse Code Flow - FSM Generation	23
4.0 – Summary & Conclusions	25
5.0 – References	27
6.0 – Appendices	28
Appendix A: Gantt Chart	28
Appendix B: Financial Plan	33
Appendix C: Glossary	34
Appendix D: Generated Auction Move Code	36
Appendix E: Figma Schematic	37
Appendix F: Verifying Move syntax	38

## **1.0 - Introduction** (author: K. Raihanizadeh)

This report documents the motivation, design, implementation, and testing done for the synthesis and verification of Move Smart Contracts. The report concludes with suggestions of possible improvements and future work.

## **1.1 - Background Research** (author: A. Mittal)

The development of blockchain technology has led to the rise of users developing their own decentralized applications. The most common occurrences of this can be found on applications launched on the Ethereum Blockchain Network. A new blockchain has recently been introduced by the name of Diem, and its associated programming language, Move, for defining smart contracts [1]. However, with the advent of new technology, comes new security concerns as well. Like many programming languages, Move lacks formal semantic definition and developer familiarity [2]. This can lead to unexpected bugs and contracts that are much more vulnerable to malicious attacks [2]. The open-source community attempted to remedy these problems in the Ethereum network, by defining a tool, known as VeriSolid, for formal verification and synthesis of smart contracts written in a language known as Solidity. Formal verification is verification through mathematical models that guarantees from all possible inputs any output is not vulnerable [3]. Abstracting away details of the Solidity language for the user, VeriSolid uses a model-based verification approach to produce correct-by-design Ethereum Contracts [3]. In contrast to other verification techniques which exhaustively aim to search for errors found in other smart contracts, VeriSolid looks to satisfy developer specified properties of a design [3].

## **1.2 - Motivation** (author: K. Raihanizadeh)

Move attempts to address some of the security concerns associated with Solidity smart contracts by introducing four key features: statically typed resource definitions, improved flexibility in code composition, limiting external dependencies with implied trust obligations, and finally on-chain verification of common safety properties [2]. While these features add to the security of Move and the Diem blockchain, they do not ensure the correctness of the code itself. As seen on the Ethereum Network, attacks on the DAO and the Multisignature Parity Wallet Library led to more than \$330 million worth of cryptocurrency being stolen due to their functionally incorrect implementations [3]. Moreover, due to the nature of the platform these decentralized applications are deployed on, vulnerabilities in contracts cannot be patched easily nor can malicious

transactions be reverted [3]. This brings up the concept of deploying and shipping functionally correct contracts once. There have been attempts to solve this problem through the use of buckets of either automated vulnerability discovery tools or formal verification of correctness [3].

The Move language still in its early stages does not have an abundance of developers versed in it. The unfamiliarity with the Move language can further increase the chances of security or functional concerns. Developing in a manner that abstracts specific details of Move, such as VeriSolid's model-based approach, would help in simplifying the creation of a contract to bridge the learning curve for new developers and further limit potential security threats.

The problem with current automated vulnerability discovery tools is they rely on commonly found errors in previous smart contracts to find bugs in new smart contracts [3]. However, they are unable to discover vulnerabilities specific to the new smart contract requirements. Formal verification on the other hand provides guarantees for security against both common and uncommon vulnerabilities. Similar to methodologies found within the robotics industry, VeriSolid aims to create correct-by-design smart contracts through a rigorous formal verification process [3]. Such a tool does not exist within the Diem community and warrants a great deal of attention.

### **1.3 - Project Goal** (author: K. Raihanizadeh)

The goal of the project was to create an application that aids in the synthesis and verification of Move smart contracts on the Diem blockchain, using the correct-by-design methodology.

### **1.4 - Requirements** (author: R. Arora)

Table 5 below outlines the team's project requirements. A hard constraint on the application is that it must be accessible via a browser. Initially, a requirement was for the application to be hosted and accessible through a web URL, after contacting the client this was deemed unnecessary as it was sufficient for potential users to download the application on their own system similar to VeriSolid.

Table 5: Outline of the system requirements, both top-level and sub-requirements.

ID	Project Requirement	Description
1	Web Application	<b>Constraint:</b> Must be accessible by a user in this form.
2	Design Verification	<b>Functional Requirement:</b> Design must validate contracts using formal verification.
2.1	Liveness Properties	<b>Functional Requirement:</b> Design must verify liveness properties.
2.2	Safety Properties	<b>Functional Requirement:</b> Design must verify safety properties.
2.3	Deadlock Freedom	<b>Functional Requirement:</b> Design must verify deadlock freedom.
2.4	Erroneous Behaviour Explained	<b>Functional Requirement:</b> Given an erroneous contract, verification results must indicate incorrect state(s).
3	Move Code (Input and Output)	<b>Functional Requirement:</b> The user must be able to specify Move code to the design and the design is able to generate Move code.
3.1	Move Statement Support	<b>Functional Requirement:</b> The user must be able to use core language constructs including expressions, loops, and control flow statements.
3.2	Generated Code	<b>Functional Requirement:</b> Generated code for a given model must compile.
4	Model Definition Capabilities	<b>Functional Requirement:</b> Design must support the creation of a formal transition system $S = (C, T)$ where $C$ is a set of configurations and $T \subseteq C \times C$ is a relation [4]
5	Browser Compatibility	<b>Objective:</b> Application can function on at least two of the following browsers: Google Chrome, Firefox, Microsoft Edge, Safari
6	User Interface Accessibility	<b>Objective:</b> The application complies with as many WCAG (Web Content Accessibility Guidelines) 2.0 and 2.1 rules as possible. The more rules satisfied, the higher the accessibility of the application.



7	Code Editor Syntax Highlighting	<b>Objective:</b> The code editor has syntax highlighting that covers expressions, loops, and control flow statements.
---	---------------------------------	--

## 2.0 - Final Design (author: A. Mittal)

### 2.1 - System-level Overview (author: A. Mittal)

The five main components of this project are a graphical interface, verification system, language parser, code generation, and a finite state machine (FSM) generator. These components can be seen working together in the System block diagram in Figure 2. On a high level, the GUI interface supports the definition of both a transition system and a Move smart-contract. These serve as inputs into the verification workflow, which validates the design and provides the necessary counter-examples if the model contains any flaws. If the input is a Move smart-contract, the application will first transform the design to an equivalent transition system before continuing with verification. This transformation requires the need for a language parser to understand the syntax structure of Move. Given a valid design, the final step is code generation which only executes if the input is a transition system. Figure 1 outlines the flow of this process.

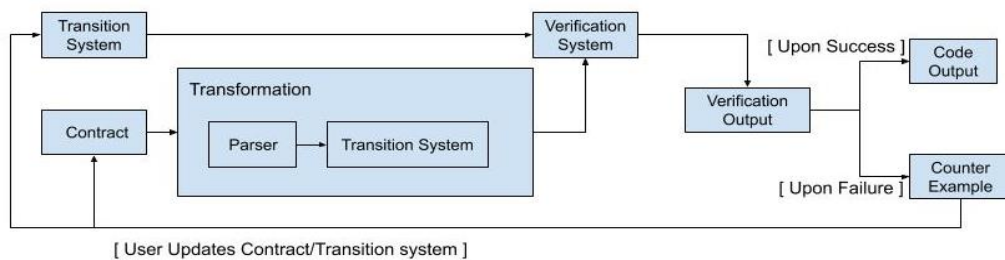


Figure 1: Verification and synthesis workflow

## 2.2 - System block diagram (author: P. Patel)

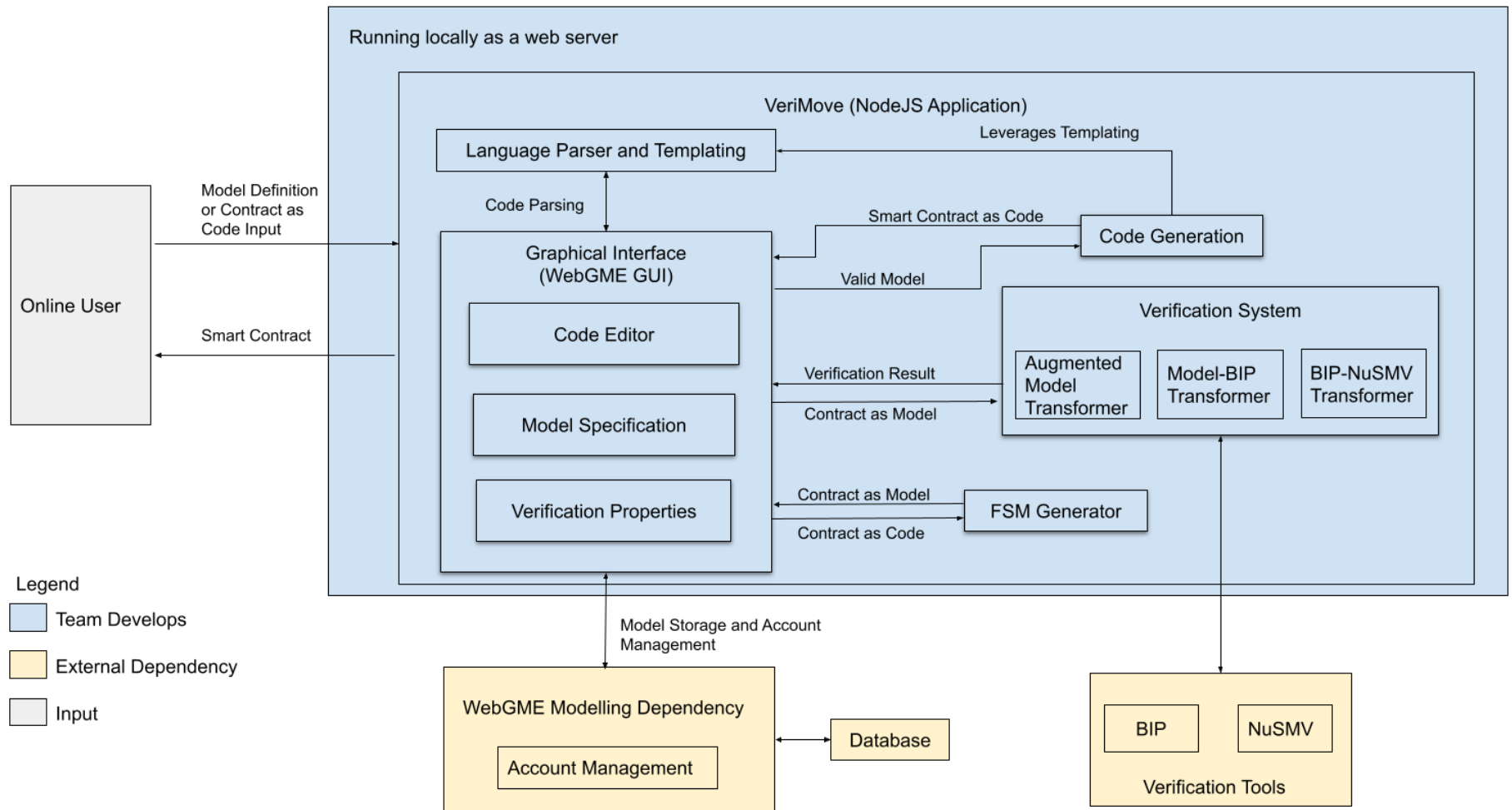


Figure 2: System block diagram

## 2.3 - Module-level Description and Designs (author: A. Mittal)

### 2.3.1 - Graphical Interface (author: A. Mittal)

The graphical interface is composed of two main components, a model creator for defining the smart contract design in a language-agnostic way, known as a transition system, and a code editor for directly specifying Move smart contracts. The team leveraged WebGME in the development of the interface. WebGME is a generic web/cloud-based modeling tool that supports version control and collaborative editing [5]. This tool forms the basis for the model creation and storage aspects in the graphical interface. The team's development work on this component involved extending the functionality provided by WebGME to support custom domain-specific needs.

The components involved in defining a complete transition system include: an initial state, a final state, a create transition, a regular transition, and a regular state. To illustrate a Move smart contract design using a transition system, the team will be using an Auction example seen in Figure 3.

1. CreateAuction (CA) - Initial State: Initiating a new auction contract.
2. AcceptingBids (AB) - Regular State: Bidders submit their bids and deposit Diem.
3. FinishAuction (FA) - Final State: Auction owners release winning bid.
4. Create - Create Transition: Default transition associated with the initial state.
5. Start, Bid, Withdraw, Finish - Regular transition: Corresponding actions that can be taken within their respective state.

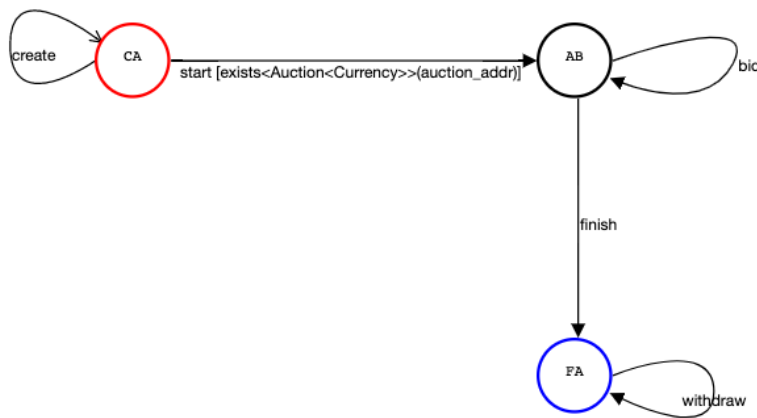


Figure 3: Move Auction Smart Contract Transition System

The initial state defines the entry point into the Move smart contract. Consequently, the final state defines the exit point of a Move smart contract. These two states are explicitly differentiated from a regular state due to the special properties associated with them. Specifically, when a new Move smart contract is instantiated, the default state is set to the initial state. Moreover, there can be numerous final states but only one initial state.

Attributes	
guards	<a href="#">⊗ Edit content ...</a>
input	<a href="#">⊗ Edit content ...</a>
name	<a href="#">⊗ start</a>
output	<a href="#">Edit content ...</a>
statements	<a href="#">Edit content ...</a>
tags	<a href="#">Edit content ...</a>

Figure 4: Transition Components

#### Edit Attribute "statements"

```
1 let auction = borrow_global_mut<Auction<Currency>>(auction_owner_addr);
2 assert(auction.auction_start + 432000 == DiemTimestamp::now_seconds());
```

Figure 5: Transition Statement for Start Transition  
(Auction Smart Contract)

Each transition defines an action a user can take within that state. Upon the successful execution of that action, the contract transitions to the successor state which may be itself. Each transition is composed of 4 parts including guards, inputs, outputs, and statements shown in Figure 4. Each of these components is defined using the Move language and verified with the language parser covered in section 2.3.3. The action to be executed within the transition is defined through the statements component shown in Figure 5. The execution of that action is conditional that the guard statements evaluate to true, also defined by the user. These guard statements can be seen on the transition system directly next to the transition name as shown in Figure 1 in square brackets [guard statement], i.e.: start transition. The input component defines the expected values needed for the successful execution of the transition. Additionally, the transition also has the option to return information about the action taken via the variables defined in the outputs component.

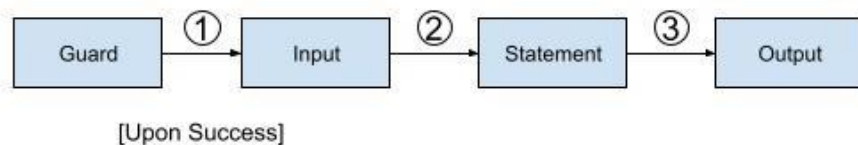


Figure 6 - Transition Components execution

A special transition, known as the create transition is included by default in every Move smart contract as it is responsible for the instantiation of all objects/variables associated with the life cycle of that contract. As the smart contract itself is a stateless object, the create transition stores all necessary information in a special Move construct known as a resource. This resource is responsible for tracking the Diem currency, highest bid, and current state in the case of the Auction contract.

Furthermore, the team implemented a custom code editor within WebGME for displaying generated code after the verification process and for writing custom Move code. A lot of the functionality found within the code editor was built using the Ace library.

### 2.3.2 - Verification System (author: A. Mittal)

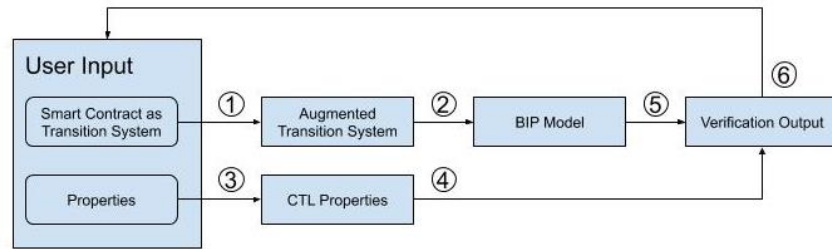


Figure 7: Verification flow from a transition system

Upon having successfully created/generated a transition system, there are three main steps involved in the verification of the transition system. This includes transforming it into an equivalent augmented system, BIP model transformation, and NuSMV verification. As described in section 2.3.1, users have the ability to define the action being taken in a transition through Move statements. An example of these statements can be found in Figure 5. There is no restriction on the number of Move statements that can be placed within the action component. For this reason, the first step is to simplify the transition system by replacing transitions that have complex statements with a series of transitions that have simple statements defined as variable decelerations and single Move expressions. Breaking up these complex statements into simple components is handled by the language parser and described in detail in section 2.3.3 and is

known as the augment system transformation. The algorithm and associated proof of the augmented system transformation can be found in the VeriSolid white paper [3].

The next step is to run BIP model transformation. This tool stands for Behaviour, Interaction, Priority and represents a standardized correct-by-design modeling tool [3]. The following is a one-to-one mapping from the augmented transition system. This indicates all transitions, states, guards, and actions of the augmented transition system can be mapped directly to the transitions, states, guards, and actions of BIP. The format of the BIP template can be found within the VeriSolid white paper [3].

The system finally proceeds with the NuSMV verification. The following tool verifies safety and liveness properties defined by the user. Safety properties check for the non-reachability of a set of erroneous states. Liveness properties on the other hand check that a set of states are reachable ensuring states are not deadlocked. Users define these properties through logic formulas known as CTL properties. The team's system provides natural language templates that are mapped to logic formulas during the verification process. These templates include:

1. [Action 1] cannot happen after [Action 2]
2. [Action 1] can only happen after [Action 2]
3. If [Action 1] happens, [Action 2] can only happen after [Action 3] happens
4. [Action 1] will eventually happen after [Action 2] happens

Actions in the aforementioned templates can be replaced by transition names from the associated transition system. Templates 1, 2, and 3 define safety properties, and template 4 defines a liveness property. One additional step is transforming the BIP system into an equivalent NuSMV system via the BIP-to-NuSMV tool. Finally, feeding the NuSMV transition system with the associated CTL properties into the NuSMV tool completes the verification process. The NuSMV tool provides the verification results or a counterexample if the system design violates one of the CTL properties.

### 2.3.3 - Language Parser (author: A. Mittal)

The Move language parser plays a critical role in a number of processes in both the code generation and verification aspects. The language parser is needed to break up complex Move statements into variable declarations and single Move expressions for the augmented transition system transformation. Furthermore, before the completion of the code generation, the result is fed through the code parser to ensure the syntactical correctness of the smart contract.

The team's system leveraged two libraries for the creation of the language parser which includes tree-sitter as well as move-tree-sitter. Tree-sitter is a parsing library that builds a syntax tree describing the various components of a Move statement. Tree-sitter takes a grammar file that defines the grammar of the language to be parsed. The team needed to define a Move grammar file describing the syntax of the language which was found through the Move-tree-sitter package.

After the successful creation of the syntax tree from a Move statement, the system would traverse the leaves of the tree structure breaking up the complex components into simple Move variable declarations and Move expressions. An example of the parsing structure generated from the sample struct using the Move language is shown below in Figure 8.

```
(source_file
  (struct_definition
    name: (struct_identifier)
    type_parameters: (type_parameters
      (type_parameter
        (type_parameter_identifier)))
    fields: (struct_def_fields
      (field_annotation
        field: (field_identifier)
        type: (apply_type
          (module_access
```

Figure 8: Structure of parsing a struct construct in the Move language

### 2.3.4 - Code Generation (author: A. Mittal)

There are two main steps involved in the generation of this smart contract. The first is mapping all transitions to an equivalent Move function. The body of the function is made up of the guards and actions specified by the user. The input and output of this function are also specified within the transition as shown in Figure 4. The second step is adding the generic Move resource to keep track of the state and any other necessary variable/object within the smart contract. All

references to global variables and state are replaced with the appropriate calls to the Move resource. Before returning the generated code, the output is parsed by the Move language parser to ensure syntactical correctness.

### **2.3.5 - Finite State Machine Generator** (author: R. Arora)

The FSM generator takes Move code within the code editor and defines a simple transition model with two states. The first state is an initial state where the contract is initialized. The second state is the core state where all functions within the smart contract are defined. Doing this enables the contract to be verified for safety and liveness properties [6] and allows for identification of incorrect behaviour within the contract.

To do this, the generator parses the code within the Code Editor and identifies several attributes from each function including the name, the inputs, the outputs, and the statements within the function (the body). The generator then creates 2 state nodes where the initial state handles the contract resource creation and the core state. The generator creates multiple transitions that loop into the core state, where each function is found in the original Move smart contract.

Figure 9 below shows how the generated model appears in the GUI using the code generated from the team's predefined Auction model (seen in Appendix D). Two state nodes are shown, one initial (called c) and one state node (called core). The initial node handles the contract object definition and creation, which is then sent to the core node. The core state node has multiple transitions looping back to itself which are created from the functions defined in the Move code.



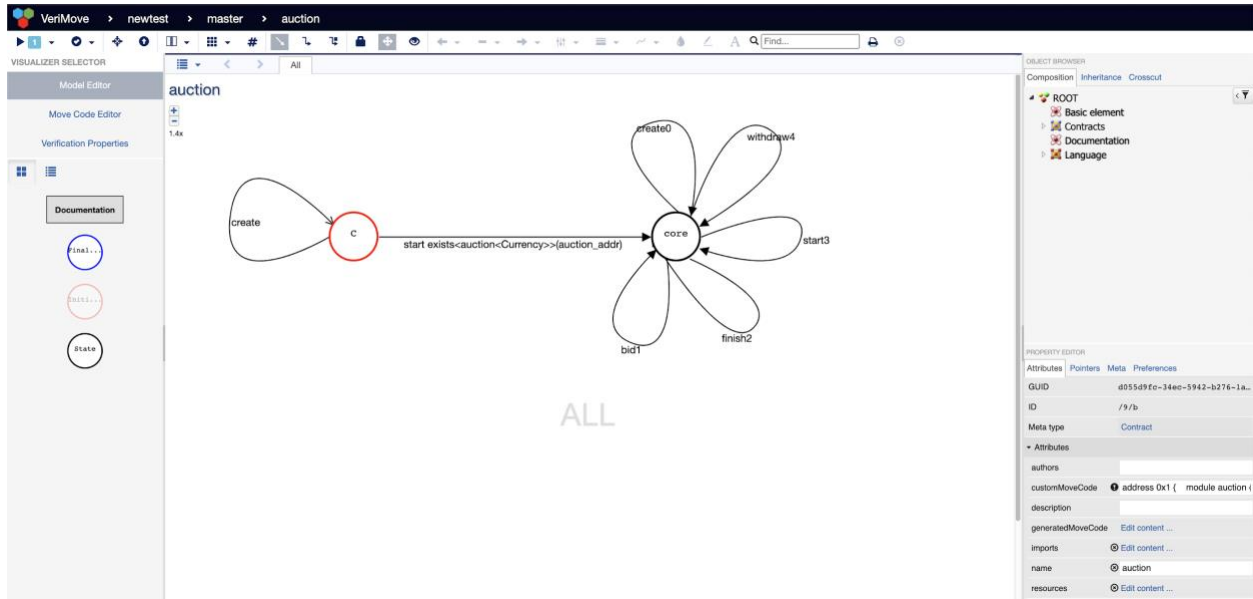


Figure 9: Using the FSM generator to create an FSM from the generated Auction code from Appendix D

## 2.4 - Assessment of final design (author: A. Mittal)

The completed final project accomplished all aspects of the original specification. Given either a transition system or a Move smart contract, the system is able to verify it for functional correctness covering both safety and liveness properties specified by the user. The following project was an extension from VeriSolid where the team aimed to develop a similar tool for Move. As the current field of cryptocurrencies and blockchains is rapidly growing there is no de facto language in the current ecosystem used by the masses. For this reason, in the initial discussions on the design of the project, it was decided we keep track of the similarities between the Solidity and Move verification systems in hopes to devise a generic framework that can be easily extended for other languages. While this was not a primary task, it would guide the design to be more modular and hence afford greater maintainability. Currently, the program only supports the Move language, but this was developed in a manner that makes it easily extensible both in terms of more features and even language extensions. This was accomplished through three primary design choices.

The first design choice is the use of WebGME. WebGME has the ability to easily integrate more features through the use of plugins. Plugins are generic JavaScript interfaces that allow the

extension of WebGME functionality by writing custom classes. This indicates that more features can be developed and integrated seamlessly without having to re-write any portion of the existing application.

The second design choice is the use of a generic language parser. Previously a custom Solidity language parser had been written for the VeriSolid application. However, for the current application, tree-sitter was leveraged as it defines a standardized format for defining syntax rules enabling the project to be expanded for use with other languages. Furthermore, tree-sitter is built around a rich open-source community that often has pre-developed syntax rules for popular languages.

The final design choice is separating core verification logic from language-specific syntax. The core verification logic is developed around transition systems that do not depend on any specific Move features. Rather language translations are handled by the parser and templating tools. These design decisions allowed the team to create a clean and modular framework that supports easy future development activity.

### 3.0 - Testing and Verification (author: P. Patel)

This section details the verification of the final design and whether it meets the defined project requirements. The verification plan largely consists of three verification methods: (1) automated testing, (2) manual testing, and (3) verification by similarity. Automated testing involves writing unit tests for various code modules. New changes to the project will not be accepted without existing tests passing. In cases where automated tests cannot easily be written, manual testing will be done. Manual quality assurance testing will also be done for testing the finished product and any user interface functionality. Lastly, verification by similarity is applied for modules or algorithms that already exist and are proven to work such as when testing the augment system transformation from VeriSolid. Table 6 is a summarized verification of the final design. For proofs and detailed results, see the follow-up sections which go in-depth into each requirement.

Table 6: Verification table for the project requirements outlined in Table 5.

ID	Project Requirement	Verification Method	Result
1	Web Application	<b>TEST:</b> Try to open the application in a browser.	PASS Section 3.1
2	Design Verification	<b>SIMILARITY:</b> The same algorithms used in VeriSolid are used in the application, guaranteeing the correct use of formal verification.  <b>TEST:</b> Define a set of models that should fail verification and a set of models that should pass verification. Run tests.	PASS Section 3.2.3
2.1	Liveness Properties	<b>TEST:</b> Define a model that violates a specified liveness property and define a model that passes a specified liveness property. Manually test the models as inputs and confirm the expected outcome matches the actual outcome.	PASS Section 3.2.3
2.2	Safety Properties	<b>TEST:</b> Define a model that violates a specified safety property and define a model that passes a specified safety property. Manually test the models as inputs and confirm the expected outcome matches the actual outcome.	PASS Section 3.2.3

2.3	Deadlock Freedom	<b>TEST:</b> Define a model that fails deadlock freedom and define a model that passes deadlock freedom. Manually test the models as inputs and confirm the expected outcome matches the actual outcome.	PASS Section 3.2.3
2.4	Erroneous Behaviour Explained	<b>TEST:</b> Manually QA that any erroneous behaviour is displayed to the user.	PASS Section 3.2.3
3	Move Code (Input and Output)	<b>TEST:</b> Confirm users can write Move code in the code editor and in the attributes of a model. Confirm generated code is visible in the code editor after the verification and synthesis flow.	PASS Section 3.3
3.1	Move Statement Support	<b>TEST:</b> Define a set of code segments consisting of supported statement types and unsupported statement types. Write automated tests that confirm the application runs as expected when the statement types are supported and returns an error when they are not.	PASS Section 3.3.1
3.2	Generated Code	<b>TEST:</b> Confirm generated code passes compilation using the official Move language compiler.	UNTESTED Section 3.3.2
4	Model Definition Capabilities	<b>SIMILARITY:</b> The application leverages the same implementation for the creation of a transition system as VeriSolid.	PASS Section 3.4
5	Browser Compatibility	<b>TEST:</b> Manually QA that the application opens in each browser and confirms core functionalities work. If a core functionality fails, that browser is declared unsupported.	PASS Section 3.1
6	User Interface Accessibility	<b>TEST:</b> Use an external testing library for WAG accessibility rules and see how many tests pass, the more the better.	UNTESTED Section 3.5
7	Code Editor Syntax Highlighting	<b>TEST:</b> Manually QA the coverage of the syntax highlighting. The more language constructs covered (expressions, loops, and control flow statements) the better.	PASS Section 3.6

### 3.1 Web Application (author: P. Patel)

A constraint of the design is that the system must be a web application. This means that it should be accessible through a browser. An objective is to support at least two known browsers. The application was tested on the three most common browsers: (1) Google Chrome, (2) Mozilla Firefox and (3) Safari. No issues were noticed across any of the browsers. Other browsers should

also work assuming they support modern Javascript. Figure 10 shows the application being used in Mozilla Firefox and Figure 11 in Google Chrome, thus meeting the requirements.

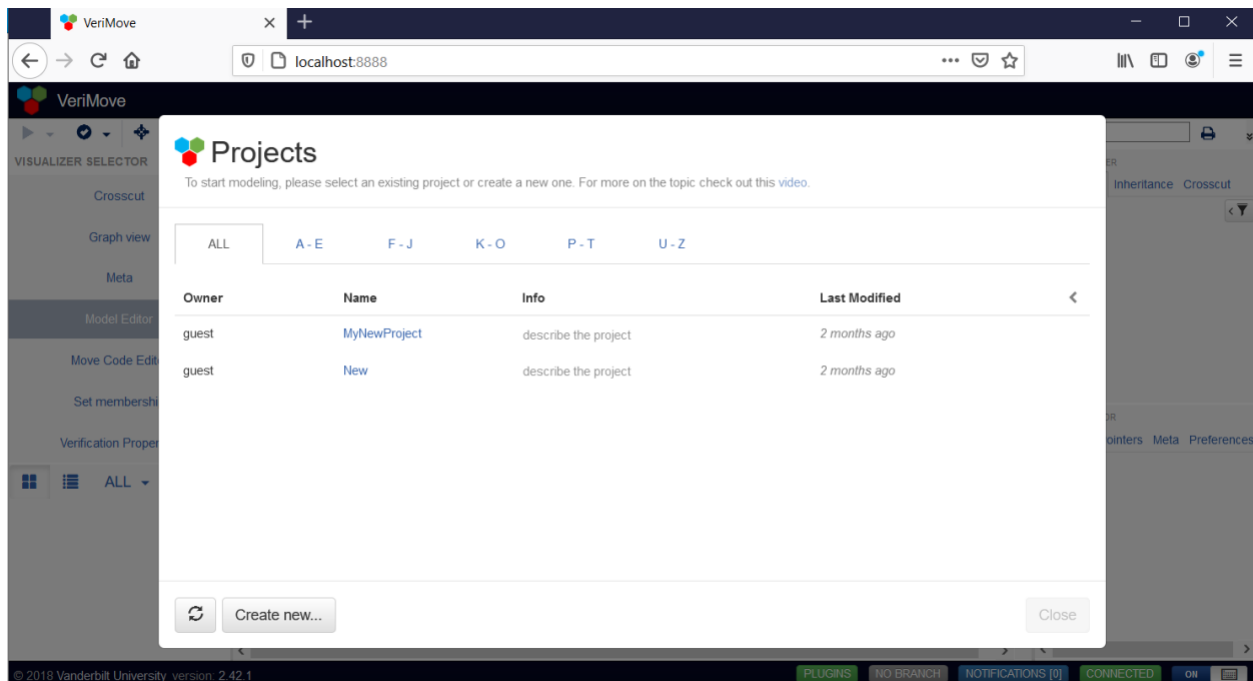


Figure 10: The application being accessed at the URL localhost:8888 on Mozilla Firefox.

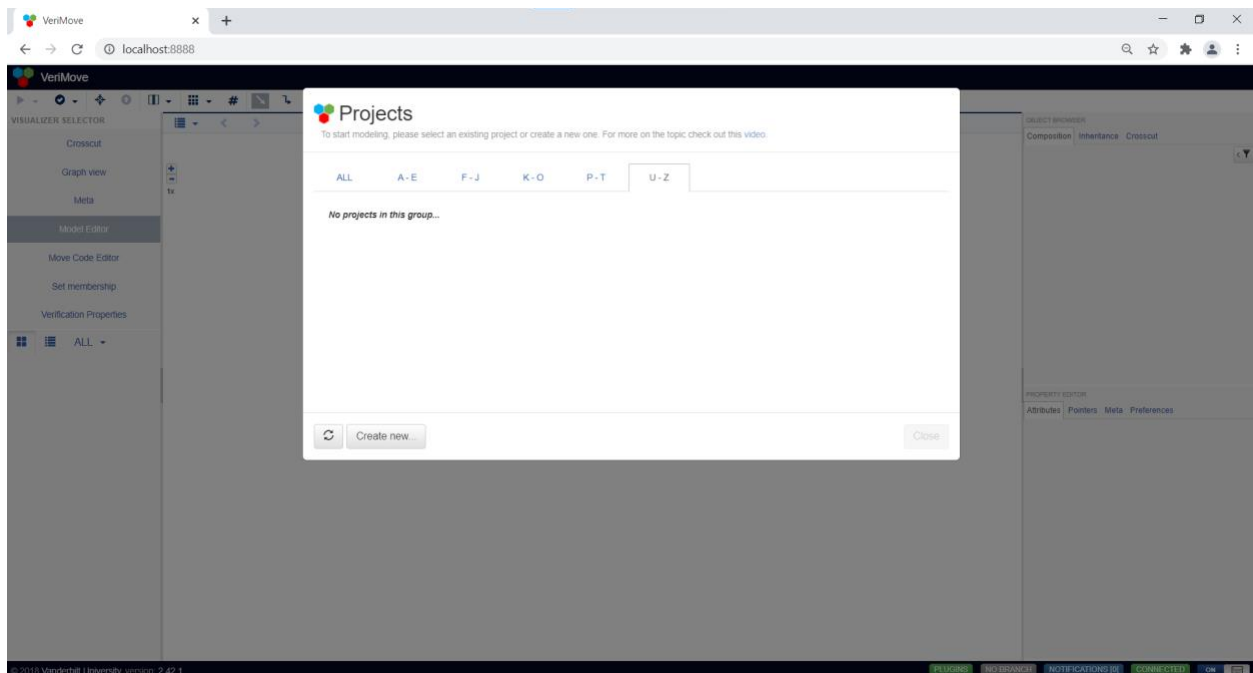


Figure 11: The application being accessed at the URL localhost:8888 on Google Chrome.

### 3.2 Design Verification (author: P. Patel)

A core requirement of the application is that it must be able to validate contracts using formal verification. The verification methodology is based on VeriSolid and is correct by the method of similarity. Apart from testing the end result, the module itself is composed of subcomponents that are also independently tested: (1) Model Augmentation and (2) CTL Property Parsing.

#### 3.2.1 Model Augmentation (author: P. Patel)

Model augmentation is tested and verified through automated tests written using Mocha, a Javascript testing framework. The augmentation process is composed of two algorithms: (1) augmentStatement and (2) augmentModel. These algorithms are defined and proven in the VeriSolid whitepaper. To ensure correct implementation, ten tests are written and if all ten tests pass, it is clear that the augmentation module is working as expected. Figure 12 shows a sample test run in which all tests are successful.

```
augmentTransitionSystem
  #augmentStatement
    general expressions case
      ✓ should properly augment states and transitions
  #augmentModel
    ✓ should properly augment a model
  compound expression case
    ✓ should handle the case where there is zero statements
    ✓ should handle the case where there is one statements
    ✓ should handle the case where there is more than one statements
  if expression case
    ✓ should handle the case where there is no alternate path
    ✓ should handle the case where there is an alternate path
  while expression case
    ✓ should properly augment states and transitions
  statement type unsupported case
    ✓ should throw an error
  #augmentModel
    ✓ should properly augment a model
```

Figure 12: Test output for the augment module.

#### 3.2.2 CTL Property Parsing (author: P. Patel)

This module is used for parsing CTL properties inputted by the user and translating them to the format accepted by the NuSMV toolkit. Automated tests are written to test the proper handling of each type of CTL property. As seen in Figure 13, tests for each case are written and they all pass.

```

CTLPropertiesForAuction
#generatingCTLProperties
First Template case
✓ bid cannot happen after finish
✓ start or bid cannot happen after finish
Second Template case
✓ withdraw can happen only after finish
Third Template case
✓ If start happens, withdraw can only happen after finish happens
Fourth Template case
✓ Finish will eventually happen after start happens
#generatingCTLPropertiesTxt
First Template case
✓ bid cannot happen after finish
✓ start or bid cannot happen after finish
Second Template case
✓ withdraw can happen only after finish
Third Template case
✓ If start happens, withdraw can happen only after finish happens
Fourth Template case
✓ finish will eventually happen after finish
#generatingCTLFairnessProperties
Second Template case
✓ withdraw can happen only after finish
Third Template case
✓ If start happens, withdraw can happen only after finish happens

```

Figure 13: Test output for the CTL Properties module.

### 3.2.3 Testing Model Verification (author: P. Patel)

Model verification is considered to be working if, for a given model, the application can properly verify liveness properties, safety properties, and deadlock freedom. The verification is handled by the NuSMV toolkit and can be assumed to be correct as it is used in many commercial systems. The scope of testing model verification is seeing if the overall process of specifying a model, CTL properties, and running verification are successful. Based on the model shown in Figure 14 and the properties specified in Figure 15, the verification output shown in Figure 16 was generated. The output shows that both the liveness property and the safety property fail and the reasoning is provided. The requirement of “Erroneous Behaviour Explained” is met through this output, however, the design has room for a better user experience. Overall, additional models can be tested but this one example is enough to show that the verification requirement and its sub-requirements are met as expected.

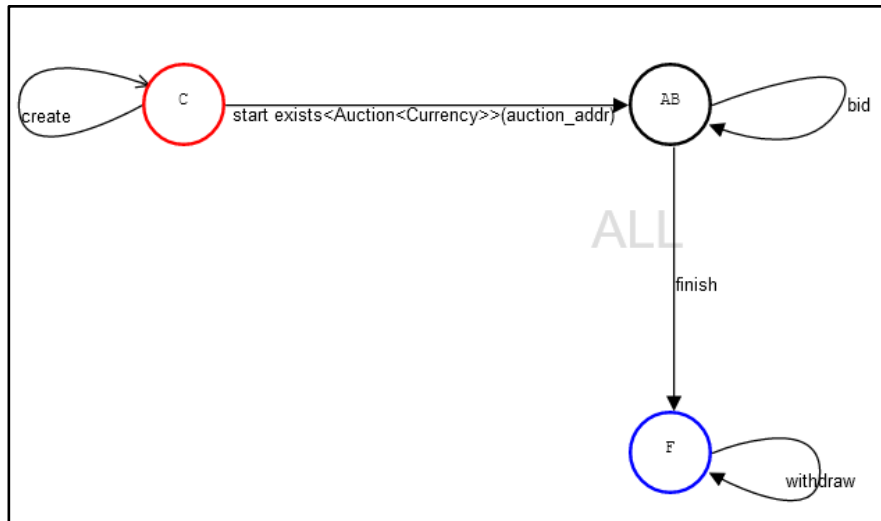


Figure 14: Auction contract model used for testing verification flow.



Existing Properties	
Type 1	
start cannot happen after create	
Type 2	
Type 3	
Type 4	
withdraw will eventually happen after create happens	

Figure 15: Specification of one liveness and one safety property for the Auction contract.



```

-- (start) cannot happen after (create) --
-- specification AG (BAUC_acreate_guard -> AG !(BAUC_astype_guard)) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
0: BAUC_acreate_guard
1: BAUC_astart
2: BAUC_astart
3: BAUC_astype_guard

-- (withdraw) will eventually happen after (withdraw) --
-- specification AG (BAUC_acreate_guard -> AF BAUC_astart_guard) is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Loop starts here:
0: BAUC_acreate_guard
1: BAUC_astart
2: BAUC_astart
3: BAUC_acreate_guard

```

Figure 16: NuSMV verification output for the Auction contract.

### 3.3 Move Code (Input and Output) (author: P. Patel)

The requirement of being able to enter Move code as well as get Move code output is easy to test. Through the code editor, the user can first enter Move code as seen in Figure 17. After clicking “Save Code”, the code should persist. Figure 18 shows the code being persisted as an attribute called “customMoveCode” which previously did not exist. In terms of getting Move code output, the user can click “View Generated Code” to see code previously generated by the application. Appendix D shows the Move code generated for the Auction contract mentioned earlier in the document.

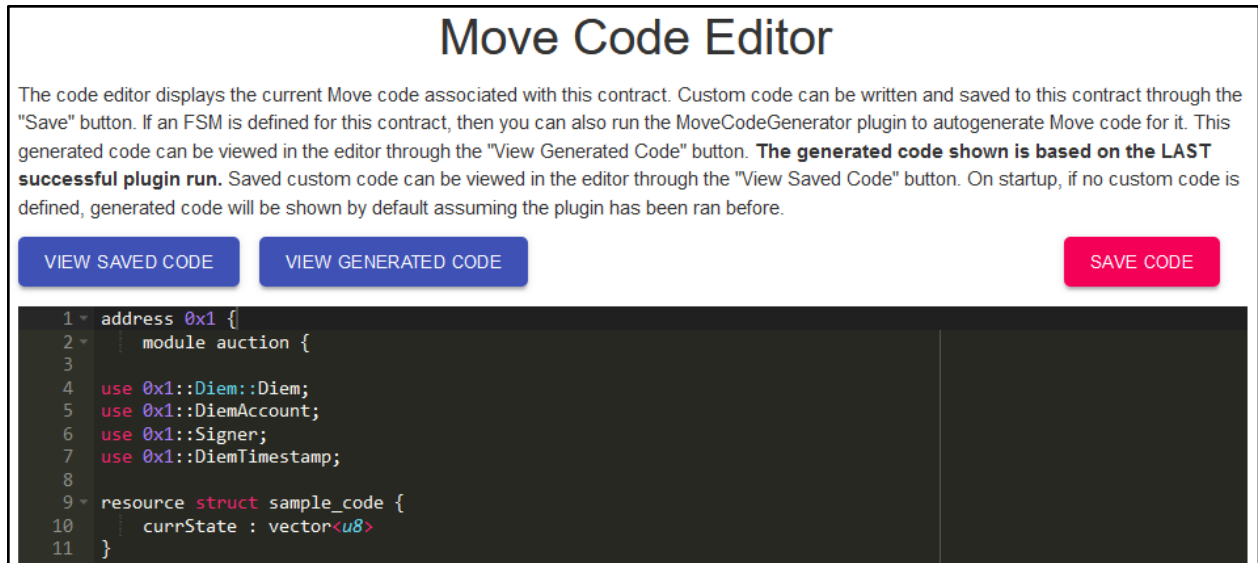


Figure 17: User enters sample code into the code editor and clicks “Save Code”.

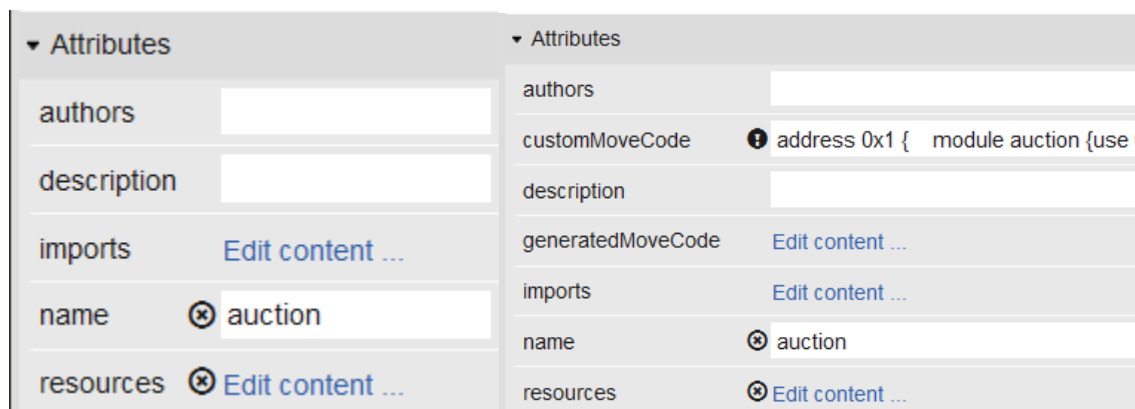


Figure 18: The left image is the attributes before saving, the right image is after saving.

### 3.3.1 Move Statement Support (author: P. Patel)

Although the user can enter Move code, an important aspect is the supported statement types. The application currently accepts expressions, while loops, and control flow statements. This covers the core language constructs and meets the specified requirement. As the primary user of imputed Move code, the augment module enforces these types. The testing and verification of the supported types are factored into the augment module tests seen in Figure 13. There are defined tests for an expression, an if condition, and a while loop which all pass. In addition, the test for detecting unsupported statements also passes. This proves the requirement is met.

### 3.3.2 Generated Code (author: P. Patel)

An additional sub-requirement is that the generated code must be valid. Ideally, this is tested by seeing if the official Move code compiler can compile the generated code. At the time of the testing, this was unfeasible and an alternative option was to confirm the Move parser could parse the output. If parsing is successful, the code can be considered valid. Using the Auction contract as an example, the generated code was entered as Move code input, and FSM generation was performed. If the application had trouble parsing the Move code, FSM generation would fail. As seen in Figure 19, FSM generation was successful and thus confirming the generated code is at least syntactically correct. The generation templating was based on VeriSolid's code generation, which through similarity provides some verification that the logic is correct. Testing is overall difficult as Move is new and there are no existing simple contracts to work with.



Figure 19: Successful run of FSM generation using previously generated code as input.

### 3.4 Model Definition Capabilities (author: P. Patel)

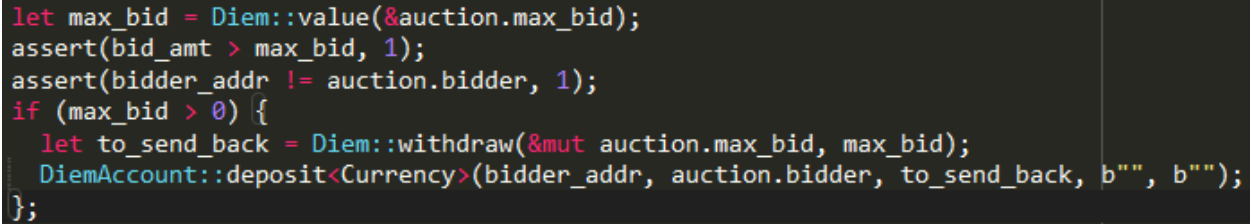
Model definition capabilities are provided out-of-box by WebGME. Figure 14 is an example of model specification working as expected, where states can be created, with transitions between them. Through the application, a user can specify a Move contract as a model, in this case, an Auction contract.

### 3.5 User Interface Accessibility (author: P. Patel)

The WebGME interface was leveraged for the application. Other than the custom Verification Properties visualizer and the Code Editor visualizer, the user interface is out-of-box. As a result, the accessibility of the application was not measured. Accessibility is an important objective that should be considered in future iterations of the project. However, for an initial product, it does not directly impact the project goal.

### 3.6 Code Editor Syntax Highlighting (author: P. Patel)

The current code editor uses Rust syntax highlighting. Move syntax highlighting has not yet been built by the Diem team and would be out of scope to build ourselves. As the Diem team also recommends Rust syntax highlighting, this requirement is considered to be met. As seen in Figure 20, constructs such as declaration types and if conditions are still highlighted and is helpful to the user.



```
let max_bid = Diem::value(&auction.max_bid);
assert(bid_amt > max_bid, 1);
assert(bidder_addr != auction.bidder, 1);
if (max_bid > 0) {
    let to_send_back = Diem::withdraw(&mut auction.max_bid, max_bid);
    DiemAccount::deposit<Currency>(bidder_addr, auction.bidder, to_send_back, b"", b"");
};
```

Figure 20: Rust syntax highlighting on Move code.

### 3.7 Reverse Code Flow - FSM Generation (author: R. Arora)

Although not explicitly stated on Table 6, this feature was important for the robustness of the project. The Reverse Code Flow is the process of creating the FSM from Move code. The verification of this feature working correctly consists of three steps: (1) verifying the Move code entered is correct; (2) verifying that the nodes are created on the Model Editor; (3) verifying that the newly created contract/FSM can be verified for BIP properties using the verification plugin. Firstly, verifying that the Move code entered is correct is done to ensure that the FSM generated is capable of being used in the real world. This is verified by running through the Move syntax tree and checks if the code entered is parsable and essentially, “compilable.” The code for this functionality can be found in Appendix F, where it shows how violations in the contracts syntax is found. Secondly, checking that the nodes are created onto the Model Editor is a manual process. After running the generation, the Model Editor panel/page needs to be checked to ensure that the model is visible to the user. A functioning and correct output example can be seen from Figure 9. Finally, to verify that the new FSM can run the verification plugin is also a manual test, i.e. going into the GUI and running the VerifyContract command. Figure 21 shows a successful run of verifying the newly created model, where the green message can be seen in the top right hand corner after running the VerifyContract plugin. This was the approach taken to completely verify this feature. The feature is unable to have more automated unit tests due to the WebGME

interface chosen to carry out the project, where node creation is based on the assumption of a correctly running interface.

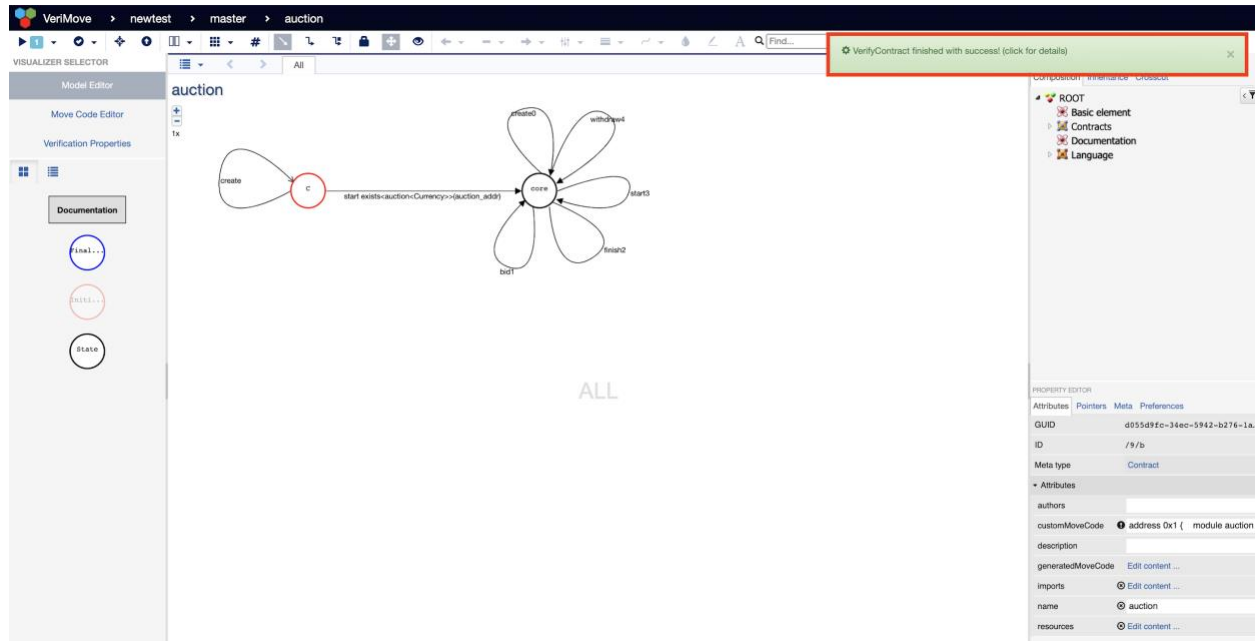


Figure 21: Functioning verification on the newly created model

## **4.0 - Summary & Conclusions** (author: K. Raihanizadeh)

### **Summary** (author: K. Raihanizadeh)

In conclusion, the goal of the project is to create an application to support correct-by-design development of Move smart contracts on the Diem blockchain. With Move being a language that lacks semantic definition and developer familiarity, it is highly susceptible to bugs that can potentially lead to security risks. A tool accomplishing the task of formal verification is currently unavailable for Move and will benefit the Diem network. The design of this project is a graphical interface for user interaction, a verification system leveraging BIP and NuSMV for model validation and a Move language parser powering a code editor. Through the model editor found in the graphical interface, a user can define a transition system modeling a smart contract that is passed through the verification system ensuring correctness and create a complete smart contract. A user can also input an already created contract into the code editor where the Move language parser helps to convert it into a transition system where it can be verified for expected functionality. This project utilizes automated and manual tests as well as using algorithms that are already proven to work to guarantee all functionality works as expected.

### **How the project meets its goals** (author: R. Arora)

The project does meet the initial goal as it was to help developers write Move smart contracts and formally verify them. This is accomplished through defining the contracts as transition systems which abstract away details of the Move language and leveraging tools such as BIP and NuSMV for model verification that perform formal verification. This is proved through testing as the team tested models that should fail and should pass verification and they receive the expected outcome. Contracts created through the verification system are also passed through a Move language parser which acts as a compiler to determine they are syntactically correct.

### **Limitation of Testing** (author: K. Raihanizadeh)

Although testing was done to the best of the team's ability, due to the infancy of Move it is difficult to perform very rigorous tests. This is largely due to the lack of an official compiler being released for Move as the language is still under development. This gap was filled using a Move language parser which parses statements and checks for syntactical errors, though effective it is hard to determine whether it is sufficiently robust and will always guarantee the contract is compilable.

**Applications of the project** (author: R. Arora)

Due to the generality of smart contracts this project is applicable to any Move contract planning on being deployed onto the Diem blockchain. The biggest threat to smart contracts is that they cannot easily be modified once deployed. This application eases this threat by validating the functionality of a contract before deployment to allow developers to be sure it is safe to deploy the contract on the Diem network. The other potential application of the project is to help new Move developers write contracts using transition systems rather than just through coding. The application can help visualize the contract as well as simplify the creation due to semantics of the Move language being abstracted from the user.

**How technology involved in this project can be applied today** (author: K. Raihanizadeh)

The work done in this project is widely applicable to the society with the emergence of blockchain technology. Since smart contracts are simply executable code on the blockchain, they have a wide variety of applications such as facilitating transactions securely. A tangible example of which would be orchestrating the purchasing of a home [7]. Using a smart contract loans, mortgages and even the ownership of a home can be programmed inside the contract, where payments are tracked and automatically release the property when the loan is paid [7]. The smart contract would automate the process, as well guarantee security as it is an innate property of the blockchain. This example also illustrates the importance of verifying the functional correctness of a contract before deployment, a task this project tackles.

**Future work** (author: K. Raihanizadeh)

Potential future work for the project includes modifying it to work with additional future languages designed for the blockchain. In this way a variety of developers would be able to seamlessly change between various projects involved with different networks. Another extension of the project would be to incorporate the official Move compiler when it is released. This would allow more thorough testing to ensure created contract correctness as well as outputting a compiled contract that is ready to be deployed on the Diem network.

## 5.0 - References

- [1] “Welcome · Libra,” *Libra*. [Online]. Available: <https://developers.libra.org/docs/welcome-to-libra>. [Accessed: 30-Mar-2021].
- [2] Blackshear, S., Cheng, E., Dill, D., Gao, V., Maurer, B., Nowacki, T., Pott, A., Qadeer, S., Russi, D., Sezer, S., Zakian, T., & Zhou, R. (2019). Move: A Language With Programmable Resources. [Online]. Available: <https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources/2020-05-26.pdf>. [Accessed: 30-Mar-2021].
- [3] Mavridou A., Laszka A., Stachtari E., Dubey A. (2019) VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In: Goldberg I., Moore T. (eds) Financial Cryptography and Data Security. FC 2019. Lecture Notes in Computer Science, vol 11598. Springer, Cham. [https://doi.org/10.1007/978-3-030-32101-7\\_27](https://doi.org/10.1007/978-3-030-32101-7_27). [Accessed: 30-Mar-2021].
- [4] M. Nygaard and E. M. Schmidt, *Transition systems*. Denmark: Department of Mathematics, Aarhus University, Department of Computer Science, 2004. [Online]. Available: <https://www.cs.au.dk/~gerth/dADS1-12/daimi-fn64.pdf>. [Accessed: 30-Mar-2021].
- [5] Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., Jurácz, L., Levendovszky, T., & Lédeczi, Á. (2014). Next Generation (Meta)Modeling: Web- and Cloud-based Collaborative Tool Infrastructure. MPM@MoDELS. [Online]. Available: <https://webgme.org/WebGMEWhitePaper.pdf>. [Accessed: 30-Mar-2021].
- [6] K. Nelaturu, A. Mavridou, A. Veneris, and A. Laszka. "Verified Development and Deployment of Multiple Interacting Smart Contracts with VeriSolid," in Proceedings of the 2nd IEEE International Conference on Blockchain and Cryptocurrency (ICBC 2020), May 2020. Available: <https://aronlaszka.com/publication/nelaturu2020verified/>. [Accessed: 30-Mar-2021].
- [7] D. Geroni, “Top 12 Smart Contract Use Cases,” *101 Blockchains*, 09-Jan-2021. [Online]. Available: <https://101blockchains.com/smart-contract-use-cases/>. [Accessed: 30-Mar-2021].



## 6.0 Appendices

### Appendix A: Gantt Chart History (author: R. Arora)

#### Iteration 1:

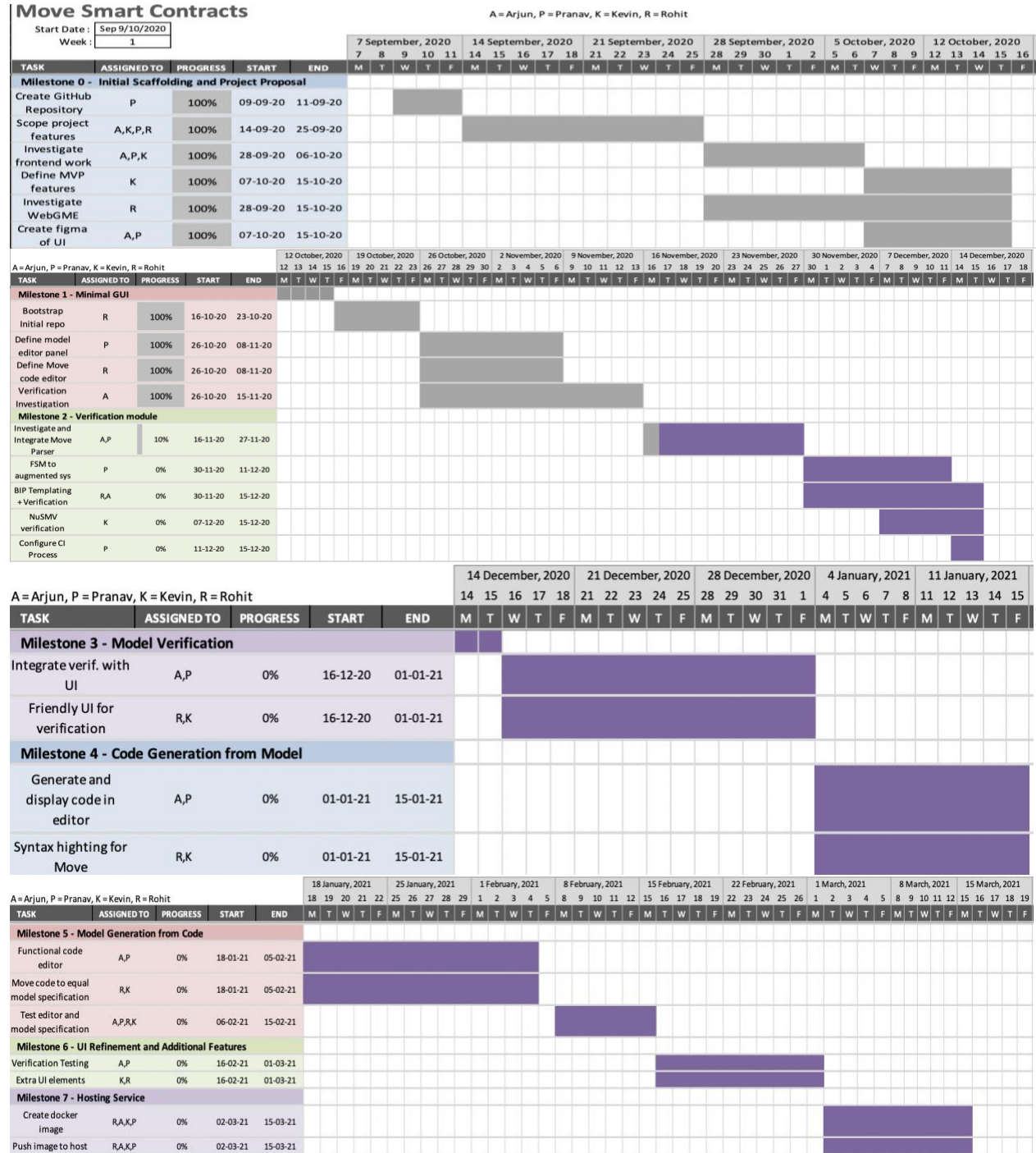
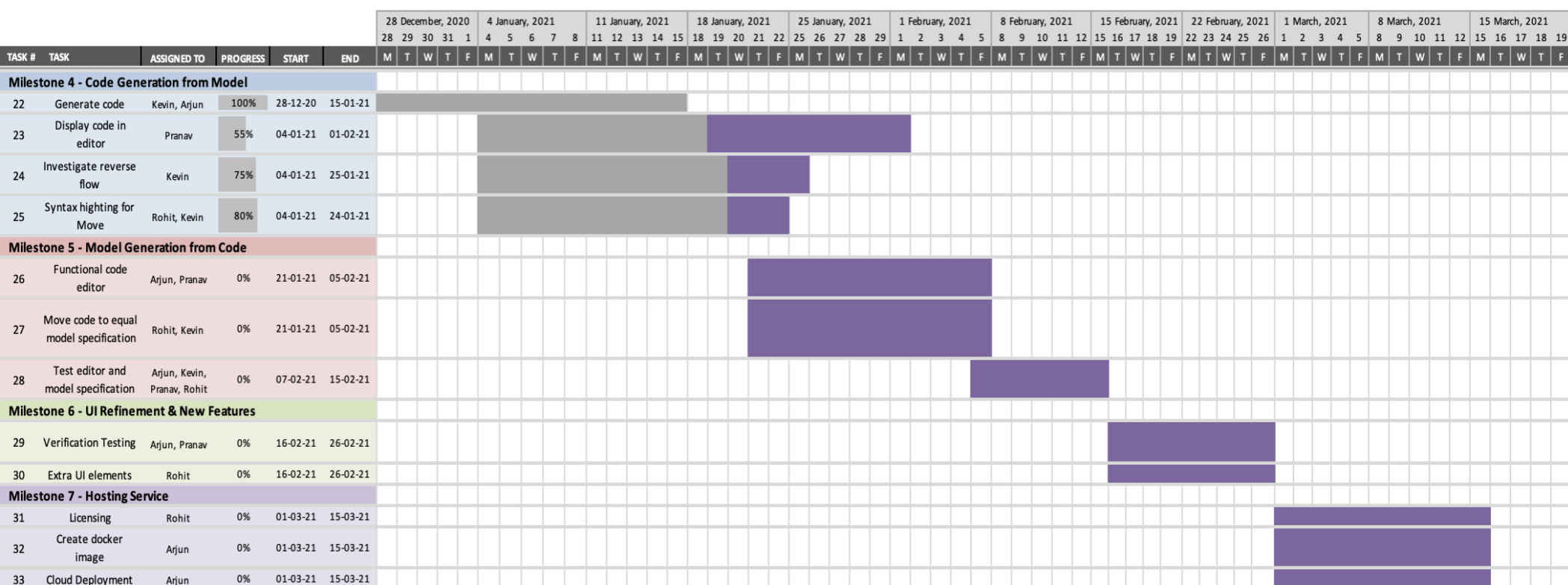


Figure 22: First iteration of Gantt chart





## Move Smart Contracts

Start Date : Sep 9/10/2020

Week : 1

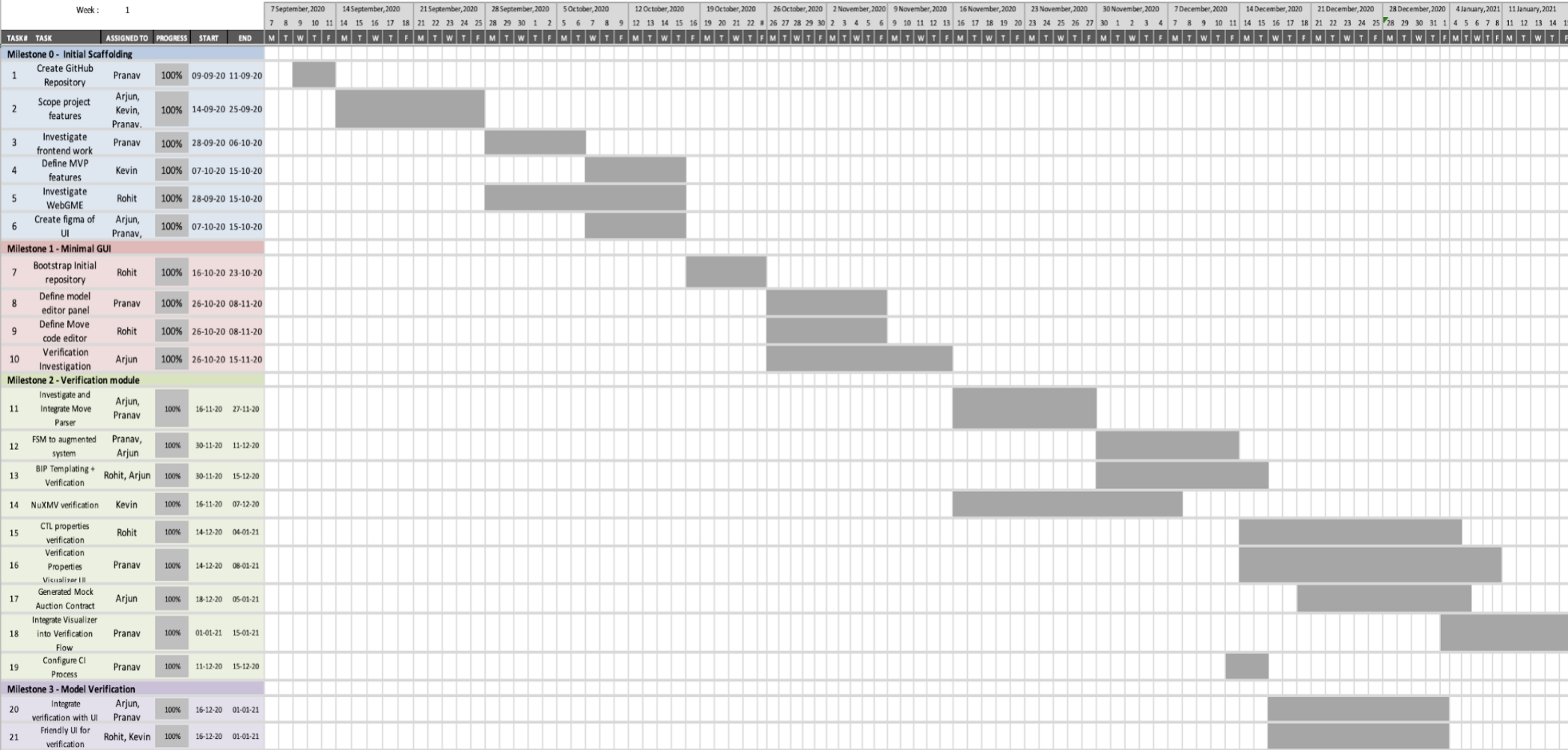


Figure 25: Final Gantt chart for milestones 0 - 3

## Move Smart Contracts

Start Date : Sep 9/10/2020

Week : 17

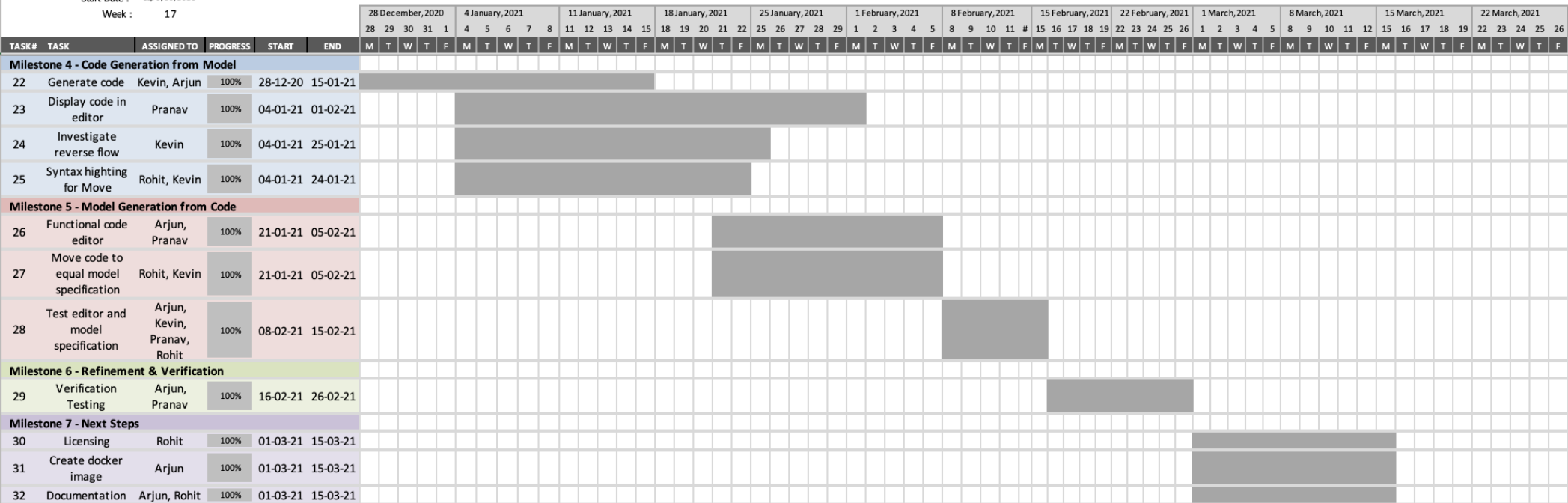


Figure 26: Final Gantt chart for milestones 4 - 7

## Appendix B: Financial Plan (author: R. Arora)

The cost of this project consists of consumables & services, and the cost of student labour. The project comes to a grand total of 24K CAD which is specifically due to the cost of the student labour. The team has found resources online to mitigate the costs of the consumables. The team has room for 400 CAD in funding but the team does not see the need to use this amount. This is summarized below in Table 7.

Table 7: The Budget Table

Consumables & Services					
Item	Priority	Cost/Unit	Quantity (# or hours)	Total Cost	Requires Funding
Domain name	1	\$0	12 months	\$0	n
Hosting service	1	\$0	12 months	\$0	n
CD/CI software	1	\$0	12 months	\$0	n
<b>Total Cost of items</b>				<b>\$0</b>	
Student Labour				Funding	
Item	Cost/unit	Quantity (# or hours)	Total cost	Students (\$100 each)	
Student 1	\$30	200	\$6,000	Supervisor	\$100
Student 2	\$30	200	\$6,000	Request from Design Center	\$0
Student 3	\$30	200	\$6,000	<b>Total Potential Funding</b>	<b>\$400</b>
Student 4	\$30	200	\$6,000		
<b>Total Student Labour (unfunded)</b>			<b>\$24,000</b>		
<b>Total Cost of Project</b>			<b>\$24,000</b>		
<b>Total Cost Requiring Funding</b>			<b>\$0</b>		

There are 3 items in the consumables & services section that have resources online that can be accessed and used for free. Although domain name and hosting services are no longer necessary, consumables in the current project iteration, an analysis is conducted for completeness. Domain names ending in “.tk” and “.ml” are free for the first year from specific domain buying websites. For example, the team can gain access to the URL “moveverification.tk” free of charge during the period of this project. Therefore, the cost for a domain name is \$0. For hosting services, Amazon Web Services offers a free tiered service that can be used along with the custom domain to host the application for this project. Choosing a basic non-free option on Amazon Web Services costs ~858.48 USD. For CI/CD software, GitHub Actions is leveraged free of charge as usage is not expected to exceed free tier limits.

## **Appendix C: Glossary** (author: All)

**Behavioural correctness** - Refers to the input-output behaviour of an algorithm. Ie. For a given input, it produces the expected output.

**BIP Tool (Library)** - Verifies behavioural correctness by checking to see if there is any point in which the program is unable to run. For example, if a deadlock occurs.

**Blockchain** - A distributed and decentralized ledger made up of three main components, including data, a hash, and the hash of the previous block.

**Block Expressions** - Code statements in the body of a block { }, in the context of our application, we always group multiple statements as a block expression. The body of for loops, if conditions, etc all count as block expressions.

**Correct-by-design methodology** - Synthesis of Move code based on a verified model.

**Deadlock** - Occurs when a process is unable to change its state indefinitely because the resources requested by it are being used by another waiting process.

**Decentralized** - Functions are not carried out at a central location.

**Ethereum network** - A decentralized open-source blockchain featuring smart contract functionality.

**Figma** - A UI and UX design application used to create designs and prototype websites, apps, or smaller user interface components that can be integrated into other projects.

**Formal semantic definition** - A framework that offers a theoretical account of how sentences' meanings are derived from the meanings of their parts.

**Formal verification** - Verification through mathematical models that guarantees from all possible inputs any output is not vulnerable.

**GUI** - Graphical User Interface.

**Language agnostic** - A software development paradigm where a particular language is chosen because of its appropriateness for a particular task.

**Diem** - A permissioned blockchain-based payment system proposed by Facebook, previously known as Libra.

**Liveness property** - All states are at some point reachable. If it is not possible to reach a given state within a transition system during the entire lifecycle of the application then the following design contains a violation.

**Move** - A programming language used for developing customizable transaction logic and smart contracts for the Diem digital currency.

**NuSMV** - A symbolic model checking tool. Used to verify user defined properties.

**Safety property** - Checks for the non-reachability for a set of erroneous states.

**Solidity** - A statically-typed programming language designed for developing smart contracts that run on the Ethereum Virtual Machine.

**Smart contract** - A transaction protocol which is intended to automatically execute, control or document legally relevant events and actions according to the terms of a contract or an agreement.

**Smart contract synthesis** - Creation of smart contracts.

**Transition model** - A state space representation of a model showing the transitions between states.

**Variable Declaration** - A code statement defining a new variable.

**VeriSolid** - A program that verifies smart contracts made in Solidarity using the Correct-by-design methodology.

**Visualizer** - A term specific to WebGME, refers to a screen visible in the main display box

**Vulnerability** - A weakness which can be exploited by a threat.

**WebGME** - A web-based, collaborative meta-modeling environment with a centralized version controlled model storage. WebGME comes with a generic GUI.

**WebGME-Engine** - The server component of WebGME. Developers can build their own GUI by interacting directly with WebGME-Engine.



## Appendix D: Generated Auction Move Code (author: R. Arora)

```
1 address 0x1 {
2   module auction {
3     use 0x1::Diem::Diem;
4     use 0x1::DiemAccount;
5     use 0x1::Signer;
6     use 0x1::DiemTimestamp;
7     resource struct auction_res { currState: vector < u8 > }
8     resource struct Auction < Currency > { max_bid: Diem < Currency > , bidder: address, start_at: u64 }
9     fun create(ownerAddr: & signer) {
10      move_to < auction_res > (ownerAddr, auction_res {
11        currState: b"AB"});
12    }
13    fun bid(ownerAddr: address, bidder_addr: address, auction_owner_addr: address, bid: Diem < Currency > ) acquires {
14      let baseResource = borrow_global_mut < auction_res > (ownerAddr);
15      assert(baseResource.currState == AB);
16      * baseResource.currState = b"InTransition";
17      let auction = borrow_global_mut < Auction < Currency >> (auction_owner_addr);
18      let bid_amt = Diem::value( & bid);
19      let max_bid = Diem::value( & auction.max_bid);
20      assert(bid_amt > max_bid, 1);
21      assert(bidder_addr != auction.bidder, 1);
22      if (max_bid > 0) {
23        let to_send_back = Diem::withdraw( &mut auction.max_bid, max_bid);
24        DiemAccount::deposit < Currency > (bidder_addr, auction.bidder, to_send_back, b"", b "");
25      };
26      Diem::deposit( &mut auction.max_bid, bid);
27      * auction.bidder = bidder_addr;
28      * baseResource.currState = b"AB";
29    }
30    fun finish(ownerAddr: address, auction_owner: address) acquires {
31      let baseResource = borrow_global_mut < auction_res > (ownerAddr);
32      assert(baseResource.currState == AB);
33      * baseResource.currState = b"InTransition";
34      let auction = borrow_global_mut < Auction < Currency >> (auction_owner_addr);
35      assert(auction.auction_start + 432000 == DiemTimestamp::now_seconds());
36      * baseResource.currState = b"F";
37    }
38  }
39  fun start(ownerAddr: address, auction_addr: address) acquires {
40    let baseResource = borrow_global_mut < auction_res > (ownerAddr);
41    assert(baseResource.currState == C);
42    assert(exists < Auction < Currency >> (auction_addr));
43    * baseResource.currState = b"AB";
44  }
45  fun withdraw(ownerAddr: address, auction_owner_addr: address, bidder_addr: address) acquires {
46    let baseResource = borrow_global_mut < auction_res > (ownerAddr);
47    assert(baseResource.currState == F);
48    * baseResource.currState = b"InTransition";
49    let Auction {
50      bid,
51      bidder: _,
52      start_at: _,
53    } = move_from < Auction < Currency >> (auction_owner_addr);
54    let bid_amount = Diem::value( & bid);
55    DiemAccount::deposit(bidder_addr, auction_owner_addr, bid, b"", b "");
56    * baseResource.currState = b"F";
57  }
58 }
59 }
60 }
```

Figure 27: The auction code made by the Move Code Generator function displayed within the Move Code editor of the application

## Appendix E: Figma Schematic (author: R. Arora)

At the beginning of the project, Arjun, Pranav, and Rohit created a UI prototype using the Figma program. 3 pages were created in accordance to how the team wanted the front end of the application to look early on in the implementation stages. The idea of using this UI was cancelled because rebuilding the WebGME interface was not in the scope of the project. It became unfeasible due to the integration between front end components and backend functionality and so the alternative to use the WebGME front end as is instead of rebuilding it from scratch was chosen.

### Task # : Create Figma of UI - CANCELLED

**Responsibility:** Rohit Arora (33%), Arjun Mittal (33%), Pranav Patel (33%)

#### Actions:

- Similar interface to the basis of WebGME because of familiarity for verification tools.
- Rohit had created the login page, while Pranav created the project selection page and Arjun had created the landing page.

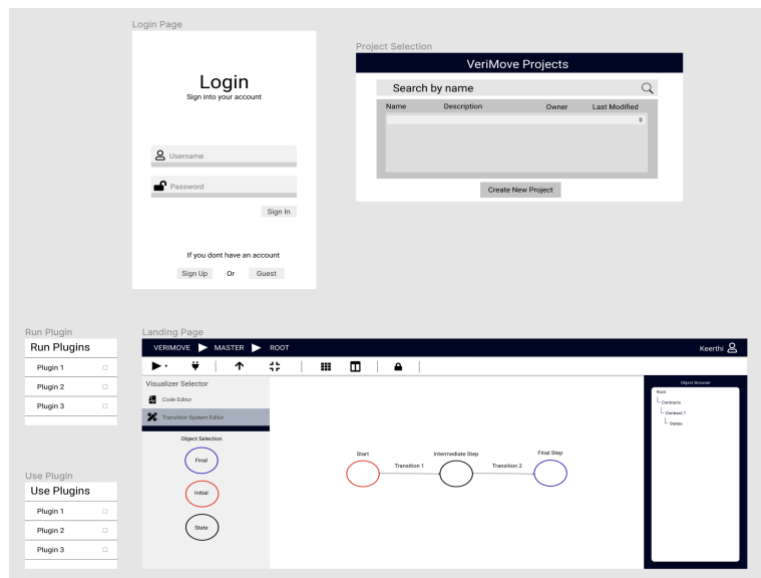


Figure 28: schematics of UI created in Figma

#### Decisions:

- Prototype was scrapped because changes to WebGME UI were not feasible.

#### Testing & Verification:

- To test the completion of the schematics the team would have compared each page created in VeriSolid to have a basis.

## Appendix F: Verifying Move syntax (author: R. Arora)

```
FSMGenerator.prototype.parseResult = function (contractNode, code) {
  const self = this
  const tree = self.parser.parse(code)
  const violations = []

  const cursor = tree.walk()
  let moreSiblings = true
  let moreChildren = true
  let depth = 0
  cursor.gotoFirstChild()

  // DFS - Visit all children, and backtrack based on depth
  while (moreSiblings || depth > 0) {
    while (moreChildren) {
      if (cursor.nodeType === 'ERROR') {
        violations.push({
          node: contractNode,
          message: 'Unexpected : ' + code.slice(cursor.startIndex, cursor.endIndex)
        })
      }
      moreChildren = cursor.gotoFirstChild()
      // Only increment if I went deeper
      if (moreChildren) {
        depth += 1
      }
    }

    // move to available sibling
    moreSiblings = cursor.gotoNextSibling()
    if (moreSiblings) {
      moreChildren = true
    } else {
      // no more siblings backtrack
      cursor.gotoParent()
      depth -= 1
    }
  }
  return violations
}
```

Figure 29: The code snippet that finds violations in the contract code