NANYANG TECHNOLOGICAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

SCSE18-0141

Visual Formulation of Smart Contracts on Blockchains

Sean Tan Jun Yu

for the degree of Bachelor of Computer Science

Abstract

This project focuses on making it easy for both developers and non-developers to develop smart contracts for solidity on the Ethereum blockchain. Our tool, the Smart Contract Builder, visually represents and simplifies the concepts in blockchain, which is still in its infancy. Graphical objects and diagrams are used to represent code and logic, which will allow quicker prototyping of smart contracts. The Smart Contract Builder also allows easy deployment of smart contract code on to the blockchain with the click of a button. We also examine the performance and gas usage of a smart contract built using this tool and compare it to an example smart contract. This document summarises all design issues, challenges faced and the steps taken to complete the project.

Acknowledgements

Thank you to A/P Sourav Saha Bhowmick for his much appreciated guidance and mentorship.
Thank you to my friends and family for their constant support and encouragement, and thank you to the school for providing me a solid foundation.

Contents Page

List of Figures

1. Introduction

1.1 Background

A blockchain is an open-source distributed database using state-of-the-art cryptography that aims to facilitate collaboration and tracking of all kinds of transactions and interactions. Ever since its conceptualisation in 2008, many organisations have been undertaking heavy research in order to unlock its potential. Despite the many performance related concerns that it has, it is widely viewed as an important part of the future of commercial transactions.

Because blockchain technology is still in its infancy, there are many different implementations, and the technology is still evolving rapidly. As a result, blockchain technology is still a very niche field that is intimidating to developers and non-developers alike. In order to simplify development on existing blockchains, we propose an application called Smart Contract Builder. The Smart Contract Builder application targets the Ethereum implementation of smart contracts, and uses the Solidity language. We believe that this application will simplify and encourage development on the blockchain, even by non-developers.

1.2 Objectives and Scope

The objective of this project is to create a graphical user interface tool that makes it easy for both developers and non-developers to create and deploy smart contracts. As blockchain and smart contracts are still new concepts, most people will not be able to develop smart contracts effectively due to the steep learning curve and relatively small community. By abstracting the code layer from the user as much as possible, we believe that this tool will encourage users to build on the blockchain, which will also increase the enthusiasm and competence around this revolutionary technology.

The scope of the project will not only cover the understanding of how smart contracts work but also how to write smart contracts. Because this tool is only a prototype, we will only focus on one language.In the future, further work can be done to extend the implementation to other languages and other blockchains.

2. Literature Review

2.1 Blockchain

A blockchain is a growing list of records, called blocks, which are linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data (generally represented as a merkle tree root hash). By design, a blockchain is resistant to modification of the data. It is "an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way". For use as a distributed ledger, a blockchain is typically managed by a peer-to-peer network collectively adhering to a protocol for inter-node communication and validating new blocks. Once recorded, the data in any given block cannot be altered retroactively without alteration of all subsequent blocks, which requires consensus of the network majority. Although blockchain records are not unalterable, blockchains may be considered secure by design and exemplify a distributed computing system with high Byzantine fault tolerance. Decentralized consensus has therefore been claimed with a blockchain. [1]
Blockchain was initially invented by a person going by the pseudonym Satoshi Nakamoto in 2008. The invention of blockchain for bitcoin made it the first digital currency to solve the double spending problem without needing central servers. [2]
When entrepreneurs understood the power of blockchain, there was a surge of investment and discovery to see how blockchain could impact supply chains, healthcare, insurance, transportation, voting, contract management and more. Nearly 15% of financial institutions are currently using blockchain technology. In 2013, Vitalik Buterin, who was an initial contributor to the Bitcoin codebase, became frustrated with its programming limitations and pushed for a malleable blockchain. Met with resistance from the Bitcoin community, Buterin set out to build the second public blockchain called Ethereum. The largest difference between the two is that Ethereum can record other assets such as loans or contracts, not just currency.
There are 3 types of blockchain: public, private and consortium blockchains. [3]

2.1.1 Public Blockchains

Public blockchains (the likes of Bitcoin and Ethereum) are essentially everyone’s to control. They put out no entry restrictions. They permit everybody, save for those not connected to the web, to access and even manage the networks, provided that the validators deposit some internal or external resources into securing them.

What public chains prohibit, however, is one’s complete authority. No single entity can write to such a network’s history unless the nodes, who participate in the consensus process, decide the entry is valid.

The security, which public chains are most praised for, is achieved by clever application of crypto economics. They use algorithms such as Proof of Work and Proof of Stake to prevent malicious activities and offer financial incentives for miners (validators) willing to establish the protection.

They concern themselves above all with providing transparency and anonymity. They regard efficiency (and scalability) as features of secondary importance.

2.1.2 Private and Consortium Blockchains

Consortium and private blockchains have a slightly different focus. They’re designed not to expose to the whole world the record of transactions which they store. And they are managed, much more effectively than their public counterparts, by a limited number of nodes.

Consortium and private blockchains are only different in one way. The former are governed by a group of corporations (say a consortium of banks), while the latter are maintained by a single firm. The purpose of these chains, unlike that of public ones, is not to reinvent the existing business processes, but to complement them.

Financial institutions and large-scale corporations alike can exchange assets using the blockchains technology, thus not having to pay an intermediary and having these transactions settled within seconds. They also might monitor the private peer-to-peer networks in real time, whenever they need to.

2.2 Smart Contracts

A smart contract is computer code that verifies and ensures that the terms of a contract are carried out to the satisfaction of all parties. While a standard contract outlines the terms of a relationship (usually one enforceable by law), a smart contract enforces a relationship with cryptographic code. First conceived in 1993, the idea was originally described by computer scientist and cryptographer Nick Szabo as a kind of digital vending machine. In his famous example, he described how users could input data or value, and receive a finite item from a machine, in this case a real-world snack or a soft drink. [4]

It’s worth noting that bitcoin was the first to support basic smart contracts in the sense that the network can transfer value from one person to another. The network of nodes will only validate transactions if certain conditions are met. But bitcoin is limited to the currency use case. By contrast, ethereum replaces bitcoin’s more restrictive language (a scripting language of a hundred or so scripts) and replaces it with a language that allows developers to write their own programs. Ethereum allows developers to program their own smart contracts, or ‘autonomous agents’, as the ethereum white paper calls them. The language is ‘Turing-complete’. [5]

Smart contracts can:

- Function as ‘multi-signature’ accounts, so that funds are spent only when a required percentage of people agree
- Manage agreements between users, say, if one buys insurance from the other
- Provide utility to other contracts (similar to how a software library works)
- Store information about an application, such as domain registration information or membership records.

2.3 Solidity Programming Language

Solidity is an object-oriented, high-level language for implementing smart contracts. Smart contracts are programs which govern the behaviour of accounts within the Ethereum state.

Solidity was influenced by C++, Python and JavaScript and is designed to target the Ethereum Virtual Machine (EVM).

Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features. [6]

The Ethereum Virtual machine essentially creates a level of abstraction between the executing code and the executing machine. This layer is needed to improve the portability of software, as well as to make sure applications are separated from each other, and separated from their host. Smart contract languages like Solidity cannot be executed by the EVM directly. Instead, they are compiled to low-level machine instructions (called opcodes). Under the hood, the EVM uses a set of instructions (called opcodes) to execute specific tasks. At the time of writing, there are 140 unique opcodes. Together, these opcodes allow the EVM to be Turing-complete. [7]

2.3 Terminology used

The following section explains some important Solidity, Ethereum or smart contract related terms that will be used later in this document.

2.3.1 Events

Solidity events give an abstraction on top of the EVM's logging functionality. Applications can subscribe and listen to these events through the RPC interface of an Ethereum client.

Events are inheritable members of contracts. When blockchain nodes emit events, they cause the arguments to be stored in the transaction's log - a special data structure in the blockchain. These logs are associated with the address of the contract, are incorporated into the blockchain, and stay there as long as a block is accessible. [8]

2.3.2 Gas

Gas is a unit that measures the amount of computational effort that it will take to execute certain operations. Every single operation that takes part in Ethereum, be it a simple transaction, or a smart contract, or even an ICO takes some amount of gas.

Gas is what is used to calculate the amount of fees that need to be paid to the network in order to execute an operation. [9]

2.3.3 Application Binary Interface

The Contract Application Binary Interface (ABI) is the standard way to interact with contracts in the Ethereum ecosystem, both from outside the blockchain and for contract-to-contract interaction. [10] The solc compiler will compile Solidity code into ABI which can then be used directly with the Ethereum blockchain.

2.3.4 ERC-20 Token

Tokens, in the context of cryptocurrency, represent digital assets that can have a variety of values attached. They can represent assets as diverse as vouchers, IOUs, or even objects in the real world. In this way, tokens are essentially smart contracts that make use of the Ethereum blockchain. One of the most significant token standards of all for Ethereum is called ERC-20.

The ERC-20 defines a common list of rules for all Ethereum tokens to follow, meaning that this particular token empowers developers of all types to accurately predict how new tokens will function within the larger Ethereum system. The impact that ERC-20 therefore has on developers is massive, as projects do not need to be redone each time a new token is released. Rather, they are designed to be compatible with new tokens, provided those tokens adhere to the rules. Developers of new tokens have by-and-large observed the ERC-20 rules, meaning that most of the tokens released through Ethereum initial coin offerings are ERC-20 compliant.

ERC-20 defines six different functions for the benefit of other tokens within the Ethereum system. These are generally basic functionality issues, including how tokens are transferred and how users can access data about a token. ERC-20 also prescribes two different signals that each token takes on and which other tokens are attuned to.

Put together, this set of functions and signals ensures that Ethereum tokens of different types will typically work the same in any place within the Ethereum system.

This means that almost all of the wallets which support the ether currency also support ERC-20 compliant tokens. [11]

2.4 Related Work

At the moment, there are no existing popular graphical user interface applications catering to new developers or non-developers. Command line tools such as Truffle exist to help developers create and deploy smart contracts. However, these are targeted at experienced developers who are comfortable with the command line. Furthermore, with these tools only aiding with the structure, deployment and testing of smart contract code, the user is still required to write a significant amount of code on their own. There are also tools such as MetaMask which allow users to interact with smart contracts and manage their wallets, but this does not conflict with the objective of our application. There are similar tools to the Smart Contract Builder for other fields such as Tableau for Big Data Visualisation and SAS for data science and business analytics. However, since Ethereum and blockchain in general are still very new, most open source effort is going into making blockchain more viable and efficient rather than making development accessible to non-technical users.

3. Development and Implementation

3.1 Software And Tools Used

3.1.1 React and Electron

In order to build the Smart Contract Builder, the Electron and React frameworks were chosen.

Electron is an increasingly popular framework due to its compatibility across platforms, and enables developers to create their user interfaces with HTML and JavaScript. Well known examples of other Electron applications include Slack, GitHub and Microsoft Visual Studio Code. [12]

React is a popular front-end framework developed by Facebook to allow for responsive web interfaces through the Virtual DOM. Well known examples of React users are: Facebook, Instagram, Netflix, Whatsapp, Salesforce, Uber, The New York Times, CNN, Dropbox, DailyMotion, IMDB, Venmo, and Reddit. [13]

Electron React Boilerplate was used to set up the initial packages for the project. It uses Electron, React, Redux, React Router, Webpack and React Hot Loader for rapid application development. [14]

3.1.2 Additional packages

In addition, the solc compiler, Ganache CLI and web3 packages are used. Solc, the Solidity compiler, is used to compile Solidity smart contract code into machine readable code. Ganache CLI is used to simulate full client blockchain behaviour to enable quicker prototyping of the application, and Web3 is used to interact with the blockchain in order to deploy the smart contract and call its functions. Ganache CLI runs on http://localhost:8545 by default.

Material-UI, a React library that follows Google’s material design, is also used to create a simple yet powerful graphical user interface.

The Storm React Diagrams library was also used to provide the drag and drop diagramming functionality needed by the Graphical User Interface. This library is relatively simple to use and includes important features such as serialization and deserialization of the diagrams, as well as extensibility of the nodes to enable custom features.

3.1.3 Remix IDE

Remix IDE allows compilation and testing of smart contracts through the browser. We use Remix IDE's static analysis tool to obtain the gas usage for smart contracts in order to compare their performance and efficiency in a fair manner.

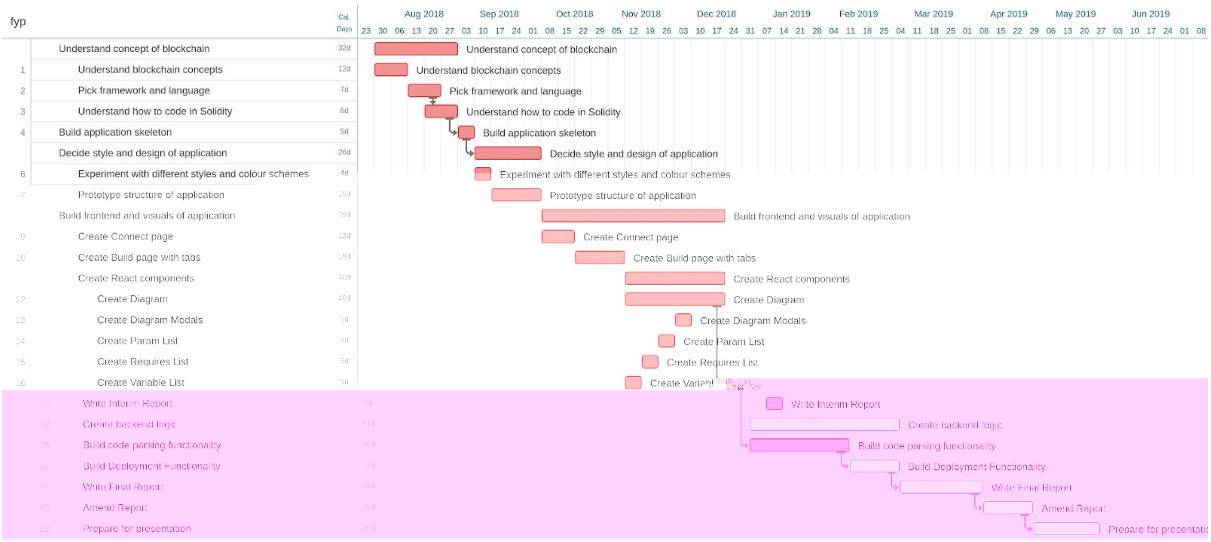
3.1.4 Hardware

The application was built on a personal computer running 64 bit Linux Ubuntu 18.04, with 8GB of RAM and an Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz processor. It has also been tested on a Windows machine with 8GB of RAM and an Intel Core i5 (4th Gen) 4210U @ 1.7 GHz processor.

3.1.5 Dependencies

Appendix A shows the dependencies in the package.json of the project, including the above packages and other packages that were used.

3.2 Project Schedule



	fyp	Cal. Days
	Understand concept of blockchain	32d
1	Understand blockchain concepts	12d
2	Pick framework and language	7d
3	Understand how to code in Solidity	6d
4	Build application skeleton	5d
	Decide style and design of application	26d
6	Experiment with different styles and colour schemes	4d
7	Prototype structure of application	19d
	Build frontend and visuals of application	75d
9	Create Connect page	12d
10	Create Build page with tabs	19d
	Create React components	40d
12	Create Diagram	40d
13	Create Diagram Modals	5d
14	Create Param List	5d
15	Create Requires List	5d
16	Create Variable List	5d
17	Write Interim Report	5d
18	Create backend logic	51d
19	Build code parsing functionality	40d
20	Build Deployment Functionality	19d
21	Write Final Report	33d
22	Amend Report	19d
23	Prepare for presentation	20d

Figure 3.1: Project schedule

The work was planned in this manner to allow for rapid prototyping of the application. By quickly creating a rough skeleton of the application, we were able to iterate and improve on our designs. We built the frontend first as we felt that the main challenge of the project is making the app usable and attractive to use. Once the frontend was completed, we created the backend logic which includes the parser, code generation and code deployment. Once completed, we used reference contracts from Solidity documentation and source code available on GitHub to model using the Smart Contract Builder and we were able to test the contracts generated on Remix IDE and compare the results.

3.3 Implementation

In this section, a full account of how the project work was carried out is given.

3.3.1 Targeted Feature Scope

In order to prevent over cluttering and over complication of the user interface, we chose to only implement the basic functionality of Solidity contracts, as our target audience, consisting largely of non-developers or developers new to blockchain and

Solidity, are not advanced users and are unlikely to require the overly advanced features of Solidity.

The following is a list of Solidity features or constructs that have been implemented and included in the Smart Contract Builder:

- Variable assignment
- Basic variable types (integer, string, address, boolean)
- Mapping variable type and nested mapping
- Logical if/else and while loop constructs
- Events
- Structs (known as “Entities” in the Smart Contract Builder)
- Require statements
- Transfer function

This list is not exhaustive but covers the core functionality of the Smart Contract Builder. The implementation of these features will be covered in the next section, Section 3.3.2.

The following list shows some Solidity features that will not be included in the Smart Contract Builder:

- For loops
- Bit operations
- Arrays
- Modifiers
- Enumerators
- Self Destruct
- Libraries

These are some features that we think will create too much complexity for the Smart Contract Builder. Some of these features are replaceable with some of the features that we have implemented. For example, an array of strings can be modeled as a mapping of index to the strings and modifiers can be modelled as require statements. By not adding these features, we avoid creating unnecessary confusion and clutter for the user.

3.3.2 Design

The design of the application was intended to be as simple and non-intimidating as possible. In the initial stages, we rapidly prototyped the application and tested different styles and appearances so as to find out which design would be the most appealing and least intimidating to the user. We settled on a white and blue colour scheme as the colour blue is associated with calmness while the colour white is associated with cleanliness and simplicity. [15]

The Smart Contract Builder has 3 main components that the user should take note of: the connection page, the Global State Tab and the Build Tabs. The Build page contains the Global State Tab and the Build Tabs. It also has the following buttons at the bottom of the page:

- Back button to return the user to the connection page
- Save button which saves the user's current progress
- Load button which loads the user's previously saved progress
- Generate contract button which generates and saves the generated Solidity contract
- Deploy button which generates the smart contract and deploys it to the connected blockchain

The Smart Contract Builder also heavily uses tooltips to guide and help the user.

3.3.2.1 Connection Page

The smart contract builder’s landing page is a connection screen. Users will enter the address and port of the blockchain in order to connect to it. For example, to connect to Ganache CLI, the address should be localhost and the port should be 8545, based on default Ganache CLI settings. Connecting to a valid blockchain will bring the user to the second page, the Build page, which contains the Global State Tab and the Build Tabs.

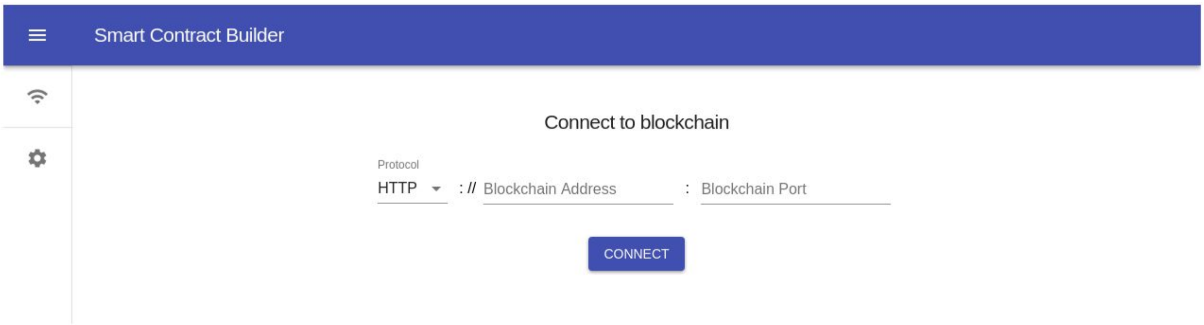


Figure 3.2: Screenshot of Connection Page

3.3.2.2 Global State Tab

The Global State Tab represents details of the smart contract that are not tied to any functions, namely, events and constructor variable declarations.

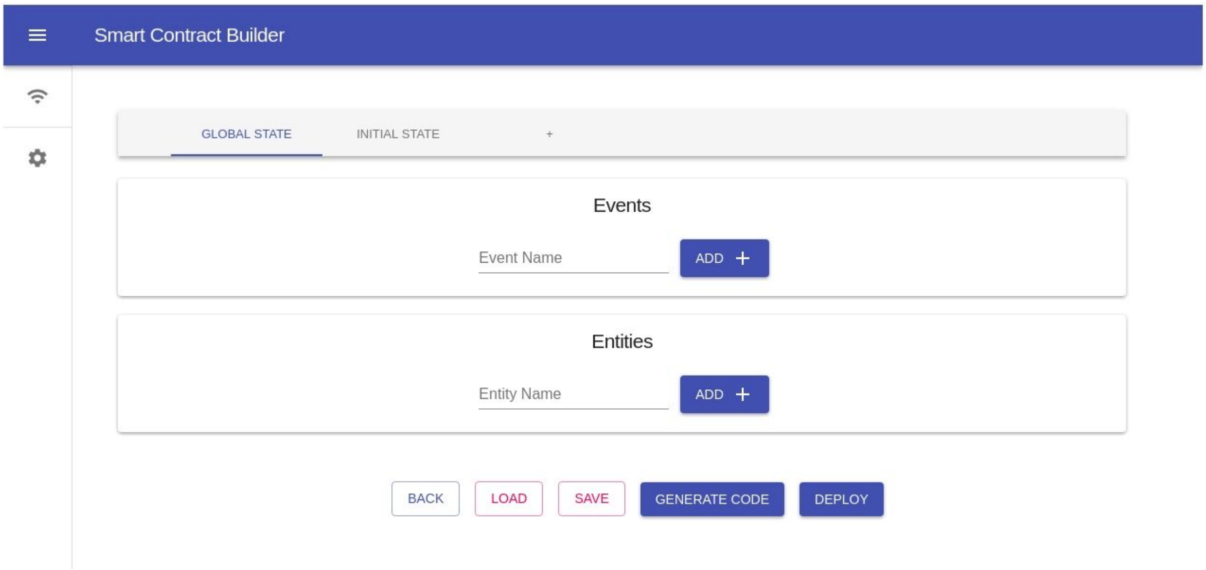
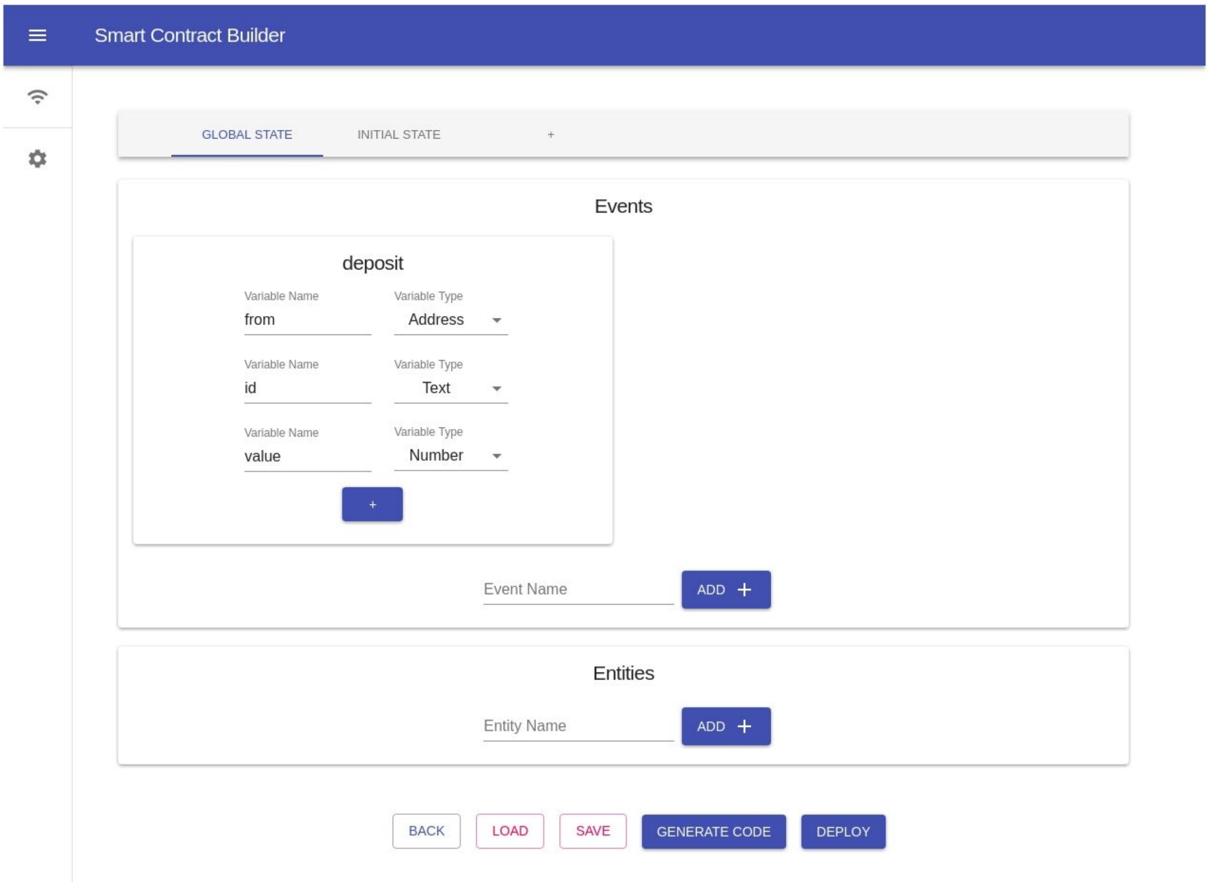


Figure 3.3: Screenshot of Global State Tab

3.3.2.2.1 Events Box

In the Smart Contract Builder, users declare events by typing the event name and clicking the add button. The events appear above the box and the user can add and customise the event's parameters. These events can be emitted in the Build Tabs. An elaboration of emitting events will be covered in Section 3.3.2.3.3.



deposit

Variable Name: from — Variable Type: Address

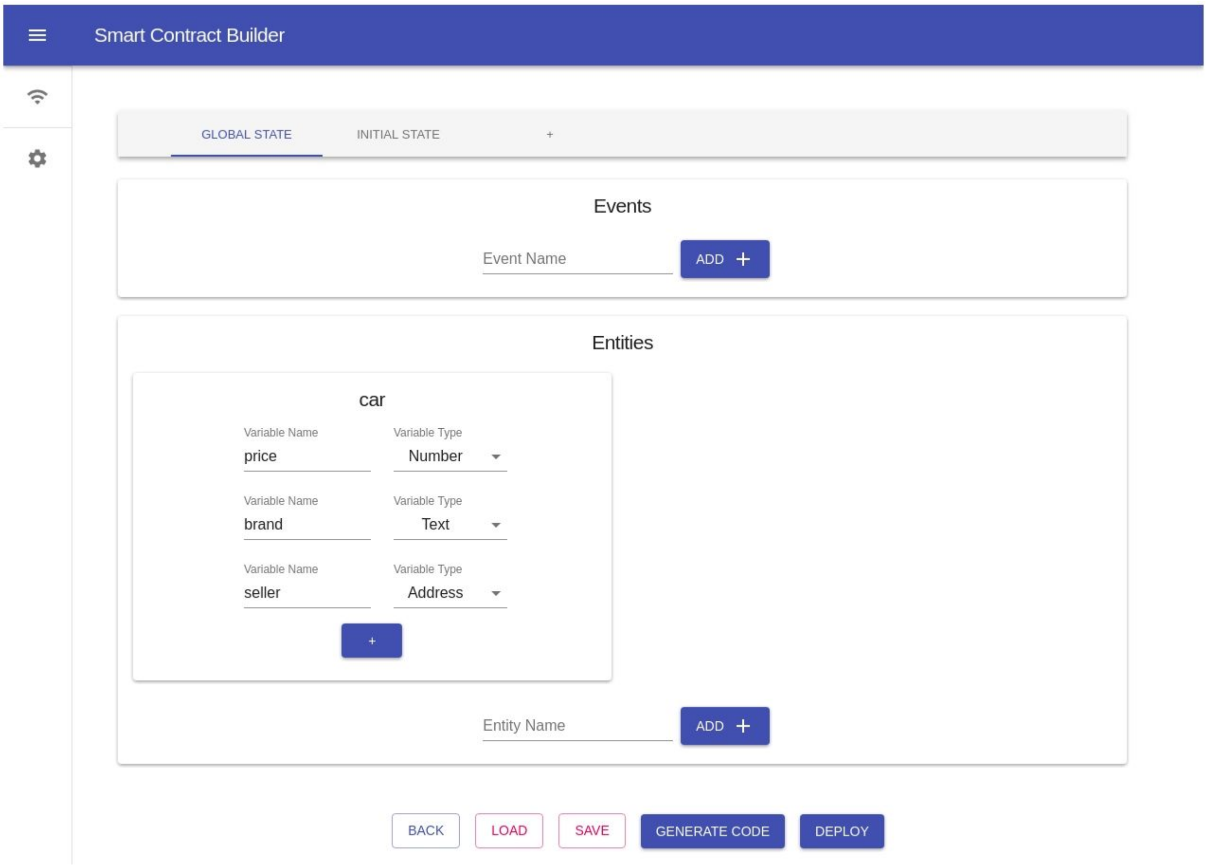
Variable Name: id — Variable Type: Text

Variable Name: value — Variable Type: Number

Figure 3.4: Screenshot of an event

3.3.2.2.2 Entities Box

In the Smart Contract Builder, the concept of an entity is equivalent to a struct in Solidity or other languages like C. Entities encapsulate information into an object which allows cleaner and more logical code. Similar to events, entities are declared in the Global State Tab so that they can be used later in the diagrams in the Build Tabs.



car

Variable Name: price — Variable Type: Number

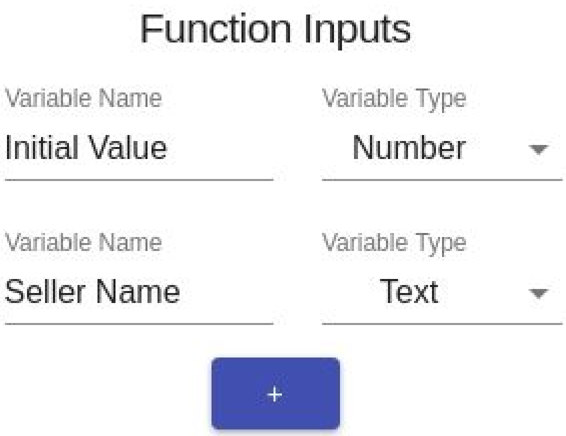
Variable Name: brand — Variable Type: Text

Variable Name: seller — Variable Type: Address

Figure 3.5: Screenshot of an entity

3.3.2.2.3 Constructor Parameters Box

The constructor parameters are obtained from the Initial State Tab, and will need to be filled on the Global State Tab. This gives the values for the smart contract to be initialised if the constructor has any parameters. If the constructor does not have any parameters, the constructor parameters field will be omitted. For example, for the constructor input box in the Initial State Tab in Figure 3.6, the Constructor Parameter box will appear as in Figure 3.7:



Function Inputs

Variable Name: Initial Value
Variable Type: Number

Variable Name: Seller Name
Variable Type: Text

Figure 3.6: Initial State Tab’s example inputs

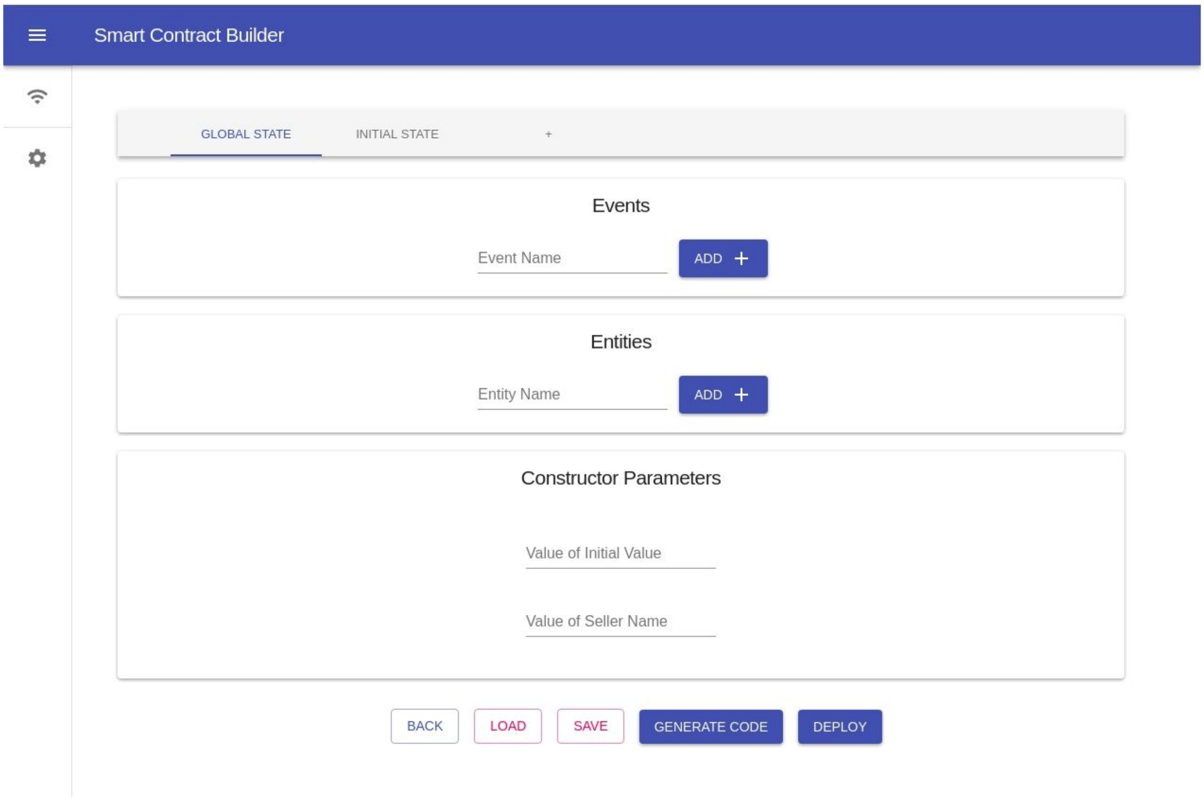


Figure 3.7: Global State Tab’s Constructor Parameters Corresponding to Figure 3.6

3.3.2.3 Build Tabs

The Build Tabs represent the functions of the smart contract. Each function corresponds to a single Build Tab, and the function name is the name of the Build Tab. The Initial State Tab represents the constructor function, while additional functions can be added by clicking on the plus sign and entering a non-duplicate name.

There are 3 boxes in a Build Tab: the function input box, the checking phase and the action phase.

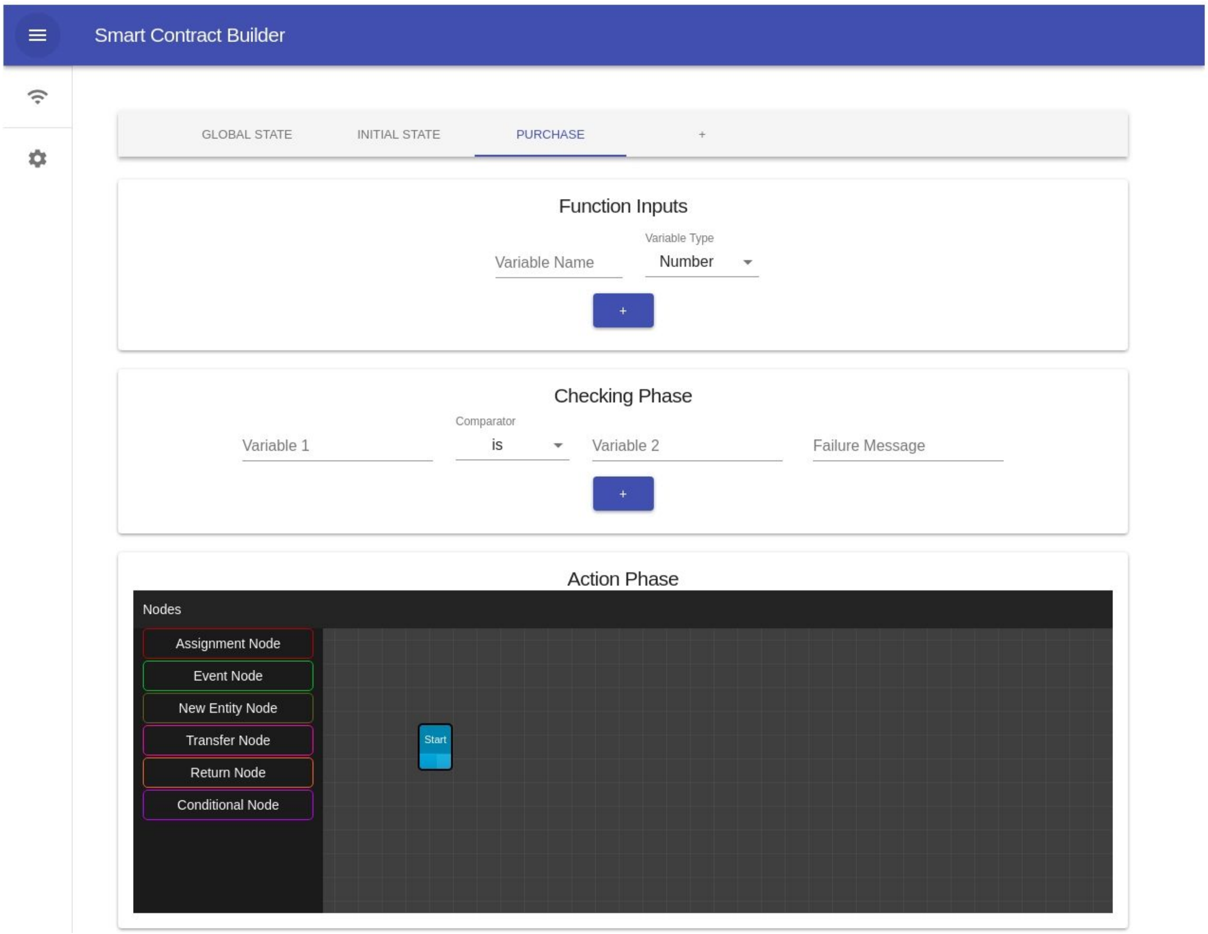


Figure 3.8: Example “Purchase” Build Tab

3.3.2.3.1 Function Input Box

The function input box allows the user to specify function parameters for each function as well as their types. Clicking on the plus button will add a new row for the user to add another function parameter.

Users will have to give each parameter a name in order to identify it in the checking and action phases, and also the type of the parameter. The Smart Contract Builder requires the user to declare the type of each parameter as it is impossible to infer the type of the function parameters during code generation.

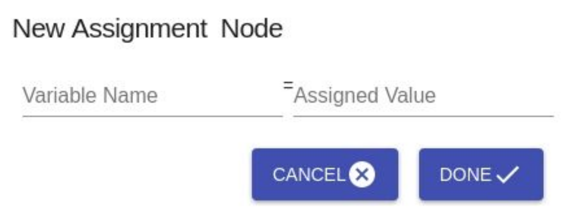
3.3.2.3.2 Checking Phase Box

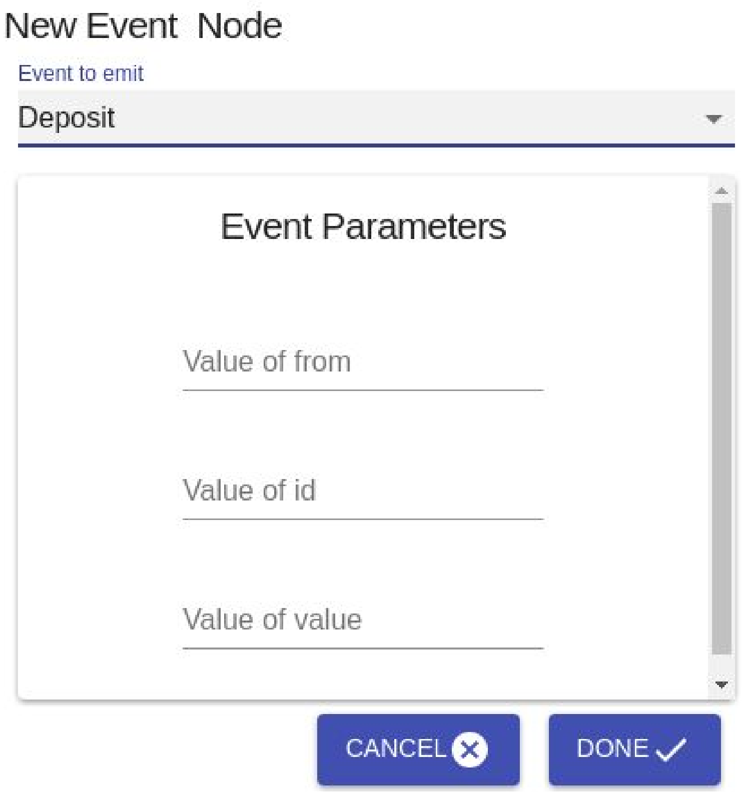
The checking phase ensures the validity of inputs before a function is run. If the user specified conditions are not met, the transaction and blockchain will be reverted to before the function was run. The failure message will be displayed by the smart contract when the condition is not met. Each condition will be converted to a require statement when the code is generated.

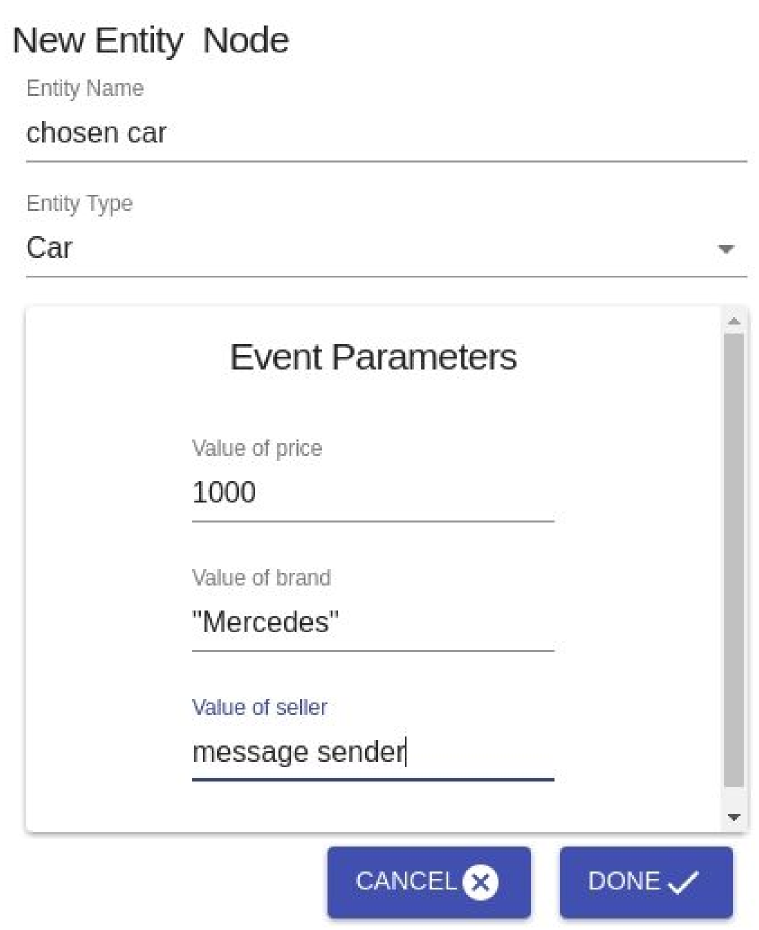
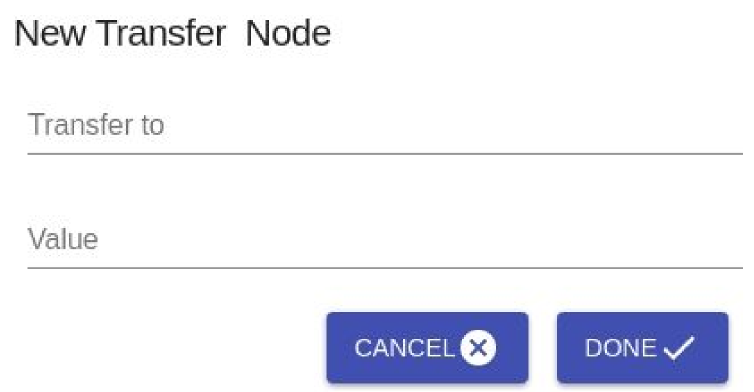
3.3.2.3.3 Action Phase Box

The action phase represents the actual logic that will be translated into code, and is visualised as a drag and drop flow diagram. The user will choose a type of node that he wants to add to the diagram from the left panel, and drags and drops it on to the diagram. A modal will be opened to ask the user for more details and once the user fills in the modal and clicks the done button, the node will be added to the diagram. The user will then connect the nodes in the logical order of execution.

The following table shows the modal for each type of node:

Node	Modal
Assignment Node	 New Assignment Node Variable Name = Assigned Value CANCEL ⊗ DONE ✓

Event Node	 New Event Node Event to emit Deposit Event Parameters Value of from Value of id Value of value

Entity Node	 New Entity Node Entity Name chosen car Entity Type Car Event Parameters Value of price 1000 Value of brand "Mercedes" Value of seller message sender
Transfer Node	 New Transfer Node Transfer to Value

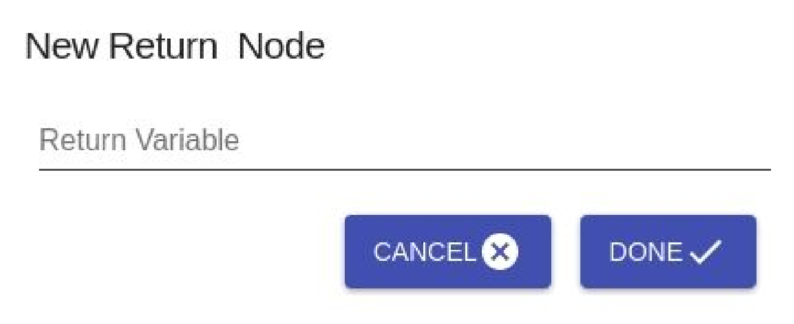
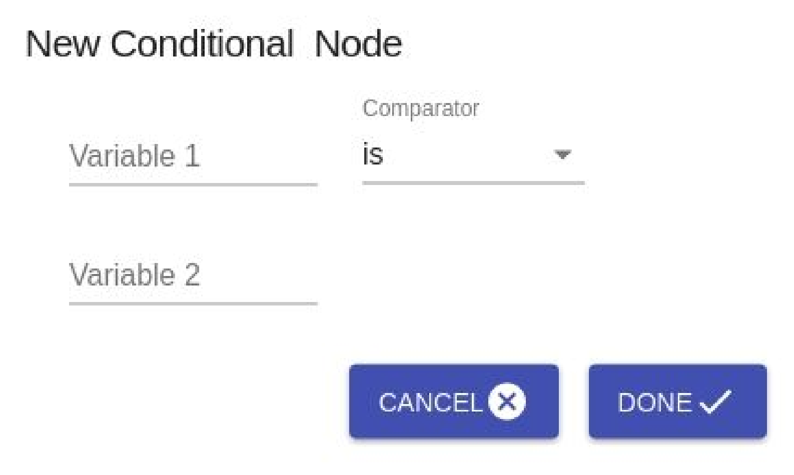
Return Node	 New Return Node Return Variable
Conditional Node	 New Conditional Node Variable 1 Comparator is Variable 2

Table 3.1: Modal appearance and options for all node types

Note that for the event and new entity type nodes, users will have to fill in different parameters depending on how the event or entity was defined in the Global State Tab. In Table 3.1, the example used for the entity node follows the entity shown in Figure 3.4.

All variable fields are free text. This is to allow the user more freedom in creating variables and not have to worry about declaring them, as the backend logic will handle this (more on the implementation of this in section 3.4). Our initial designs had the user declaring all variables that they wanted to use in the Global State Tab, but we decided to make it more intuitive and abstract the concept of variables from the user as much as possible.

3.3.3 Backend Logic

There are 2 main components responsible for the backend logic: the BuildParser class and the BuildOptions component. There also exists a saving and loading mechanism, allowing users to save their progress and continue from where they left off.

3.3.3.1 BuildParser

The BuildParser is responsible for parsing the diagrams and generating the code for each function. An instance of the BuildParser class is attached to each instance of the Action Phase diagram. A linkUpdated listener is attached to each diagram, and the BuildParser's parse method is called when a link is created between 2 nodes. The BuildParser keeps track of the code generated and the return type, which is needed in the function declaration. The BuildParser parses the diagram in 2 loops. The first loop looks for variables, infers their types and adds their names and types to a variable lookup table. It checks Assignment, New Entity and Transfer nodes in the first step as these nodes provide the most information about variable types. In the second loop, the BuildParser traverses each node recursively from the start node of the diagram, and visits each neighbour. At each node, it parses all variables involved in that node using the parseVariable method and generates the code for that node using the parseNode method. It uses the variable lookup table from the first loop that keeps track of all variables in the application and their types, allowing it to check for typing problems. If the node is a return type node, it will also update the return variable type in the function declaration.

When it encounters a conditional node, in which there are 2 connected neighbours for the true and false conditions, it will traverse each path separately and fill in the if/else conditions. If there is a loop, as shown in Figure 3.8, it will detect a cycle in the diagram and append a while loop to the smart contract code instead of an if statement. Also, if it detects an intersection later in the diagram, it will exit the if/else condition and resume normal single path traversal.

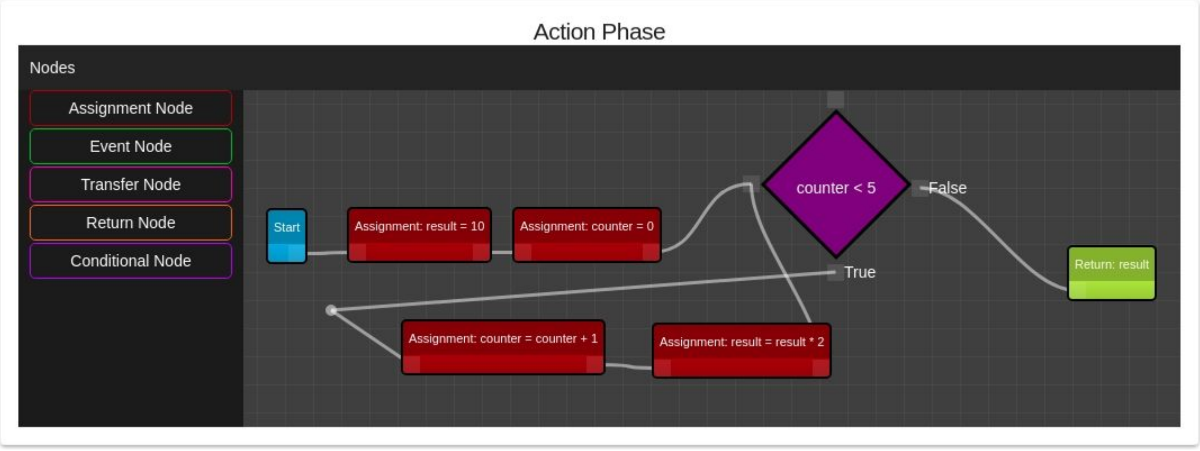


Figure 3.9: Example of while loop detected due to cycle

The initial algorithm used a single iteration to loop through the diagram, but unfortunately this led to many bugs as the algorithm needed more variable typing information about nodes that it had not visited yet. It also attempted to infer variable types and generate code for the same node at the same time, which led to more unexpected errors being thrown. Although the new algorithm is slower due to using 2 loops instead of 1, it is a more reliable parsing algorithm as it splits the two steps of inferring variable types and generating code, giving it some lookahead.

Since the algorithm runs in 2 iterations for parsing variables and generating code and during each iteration it visits every node exactly once, the algorithm has a time complexity of O(2n) = O(n) where n is the number of nodes in the diagram. Thus it runs in linear time to the number of nodes in the diagram that are connected to the start node.

3.3.3.2 BuildOptions

The BuildOptions React component is responsible for generating the code and deploying it to the blockchain. It is triggered when the user clicks on the Build button at the bottom of the Build page. It takes the details and code generated from the Initial State Tab and Build Tabs, and organises it to a syntactically correct Solidity smart contract. It then uses web3 to attempt to deploy this smart contract. If an error is raised, the user will be notified. If deployment is successful, the transaction hash and contract address of the smart contract will be shown. This transaction hash can be used to check the status of the transaction on the blockchain.

3.3.3.2.1 Code Generation

Each solidity function can be broken down into segments, which correspond to each piece of information input by the user. Appendix B illustrates the structure of a typical solidity contract and its components.

By identifying the components of a smart contract, we are able to place the information given by the user at the correct points of the code. This allows us to formulate a general formula for getting a smart contract. Appendix B shows the general anatomy of a Solidity smart contract.

When the generate code button is clicked, the file will be saved to a user-defined Solidity file in the saved_contracts directory. The saving mechanism is described later in Section 3.3.2.2.3.

3.3.3.2.2 Contract Compilation and Deployment

Because Electron works by having a single backend main process and multiple renderer windows, the solc compilation must be done on the backend, as it is an operating system process. By using the ipcRenderer and ipcMain classes, we are able to send a request to the main thread to compile the Solidity code into the machine readable application binary interface (ABI). After compilation, web3 will be used to deploy the contract onto the blockchain.

3.3.3.2.3 Saving and Loading Mechanism

The Smart Contract Builder is able to save the current progress of the user and load it at a later time. This is achieved using Electron’s readdir, readFile and writeFile functions in the filesystem module (fs). The files are stored in the saved_data directory. The readdir function reads all json files from the saved_data directory and keeps it in the state. When the save button is clicked, a popover appears prompting the user to input a name for the file. After the user enters a name, the entire state of the Build component, that stores and controls the state of the Initial State Tab and the Build Tabs, will be stringified and dumped into a JSON file with the given filename in the saved_data directory. When the load button is clicked, a popover appears for the user to select which JSON file to load. The JSON file will be read and parsed, and the state of the Build component will be reverted to the contents of the JSON file.

4. Evaluation of Results

4.1 Methodology of evaluation

We identify and use 2 key metrics for evaluating the usefulness of the Smart Contract Builder, flexibility and performance, which inform us on its usability and efficiency respectively.

We use 7 smart contracts that are available as examples on the official Solidity documentations [16] and the official ERC20 token implementation from Open Zeppelin [17] as reference material, so that we can implement them using the Smart Contract Builder and compare them. For convenience, they can be found in the Appendix (Appendices C to I).

To measure the flexibility of the application, we attempt to implement the reference contracts using the Smart Contract Builder. If the reference contract uses Solidity features that the Smart Contract Builder does not implement, we attempt to find a substitute for the feature using other currently implemented features. The flexibility of the application is the complexity of contracts that the Smart Contract Builder is able to fully implement.

To measure the performance and efficiency of the application, we will use the amount of gas consumed in the deployment and execution of the smart contracts. We model the reference contracts using the Smart Contract Builder to achieve the same functionality. We then ran both the original and generated smart contracts through Remix IDE, which is a powerful, open source tool that allows the writing of Solidity contracts straight from the browser. Remix provides the user with gas usage numbers, which we need to make a comparison. For all contracts, the default Javascript VM environment was used.

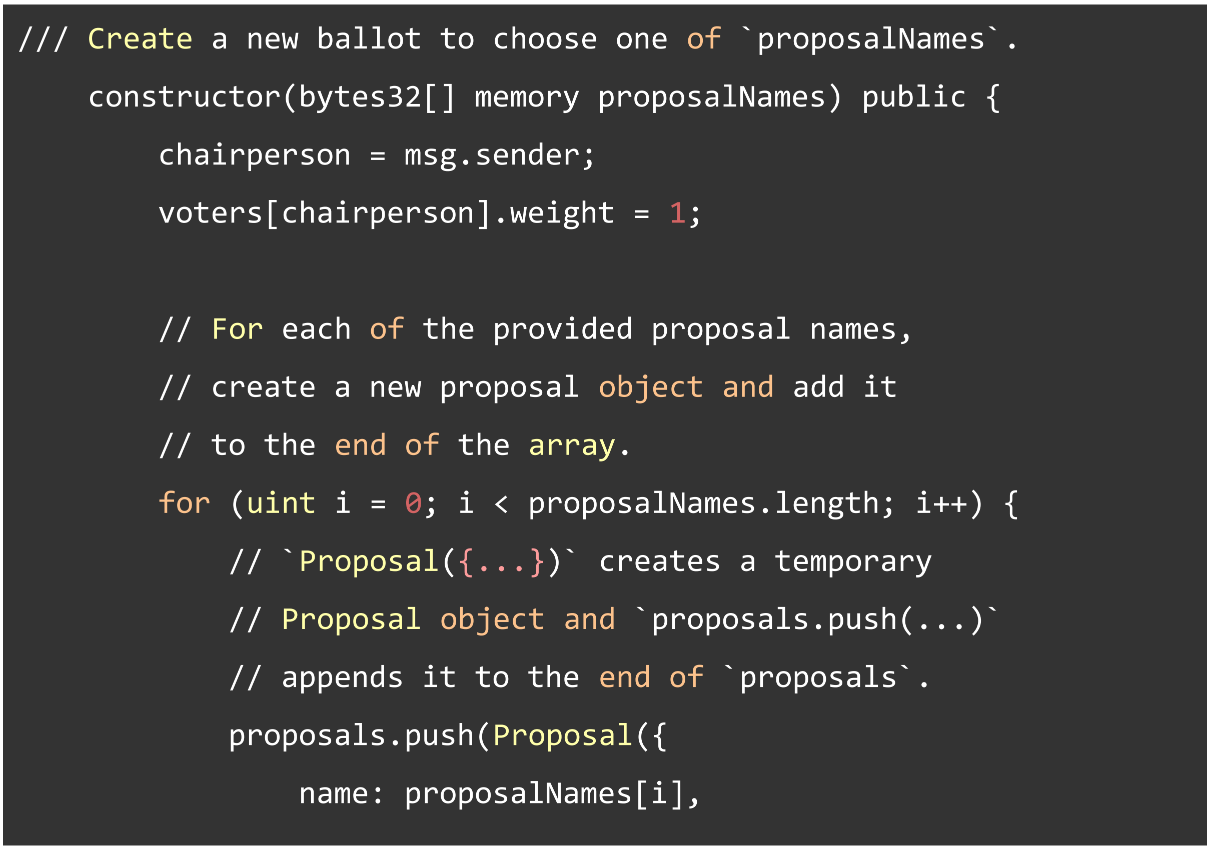
4.2 Results

4.2.1 Flexibility

In this section, we attempt to implement the 8 reference contracts using the Smart Contract Builder.

4.2.1.1 Voting

The voting contract can be fully implemented on the Smart Contract Builder. However, because we did not implement arrays functionality into the Smart Contract Builder, the proposals array in the code, which contains Proposal structs, has to be replaced by a mapping of proposal name to vote counts. This is functionally the same as an array of Proposal objects. We also separated the constructor function so that it does not have an array argument, but instead the chairperson will have to call the addProposal function for each proposal that they would like to add instead. Figures 4.1 and 4.2 show the changes in the constructor functions.



```
/// Create a new ballot to choose one of `proposalNames`.
    constructor(bytes32[] memory proposalNames) public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;

        // For each of the provided proposal names,
        // create a new proposal object and add it
        // to the end of the array.
        for (uint i = 0; i < proposalNames.length; i++) {
            // `Proposal({...})` creates a temporary
            // Proposal object and `proposals.push(...)`
            // appends it to the end of `proposals`.
            proposals.push(Proposal({
                name: proposalNames[i],
                voteCount: 0
            }));
        }
    }
```

Figure 4.1: Initial Voting Constructor

```
/// Create a new ballot to choose one of `proposalNames`.
    constructor() public {
        chairperson = msg.sender;
        voters[chairperson].weight = 1;
    }

    function addProposal(bytes32 proposalName) public {
        require(
            msg.sender == chairperson,
            "Only chairperson can add a new proposal."
        );
        proposals[proposalName] = 0;
    }
```

Figure 4.2: Changed Voting Constructor with addProposal Function

The reasoning behind not implementing arrays into the Smart Contract Builder was that it would require an additional node type. Currently there are already 6 node types, and increasing the number of node types any further may confuse users. In most cases, it is also possible to replace arrays with mappings, as shown above. Therefore we chose to omit arrays from the Smart Contract Builder.

Appendix J show the Build Tabs of the Smart Contract Builder for the Voting contract.

4.2.1.2 Open Auction

We were able to fully implement this contract using the Smart Contract Builder. Appendix K shows the Build Tabs of the Smart Contract Builder for the Open Auction contract.

4.2.1.3 Blind Auction

The Blind Auction contract is an extension of the Open Auction contract. We were not able to implement the Blind Auction contract as it uses hashing and encoding functions in the reveal function, which we did not implement as we feel that those functions may be too complex for basic users.

4.2.1.4 Safe Remote Purchase

The contract was successfully implemented, but modifiers and enumerators were used in the contract, which we had to replace with require statements and using integers to replace the enumerators. In this case, 0 represented the “Created” enum, 1 represented the “Locked” enum and 2 represented the “Inactive” enum. As a result, the generated code still functions as expected, but is less readable.
Appendix L shows the Build Tabs of the Smart Contract Builder for the Safe Remote Purchase contract.

4.2.1.5 Micropayments Receiver and Payment Channel

We were unable to implement these complex contracts as they heavily use self-destruct, hashing and encoding functions, and even write assembly code directly. The self-destruct operation removes the smart contract code from the blockchain. The remaining Ether stored at that address is sent to a designated target and then the storage and code is removed from the state. It is unlikely that a basic user will require such functionality.

4.2.1.6 Modular Contracts

Libraries are similar to contracts, but their purpose is that they are deployed only once at a specific address and their code is reused. Libraries can be seen as implicit

base contracts of the contracts that use them. They will not be explicitly visible in the inheritance hierarchy, but calls to library functions look just like calls to functions of explicit base contracts.
We did not implement libraries for the Smart Contract Builder as it is unlikely that users will require the modularity that it provides.

4.2.1.7 ERC20 Token

We were able to implement this contract. We had to replace the function calls to the internal functions directly with the code of the internal function as the Smart Contract Builder does not support internal function calling.
Appendix M shows the Build Tabs of the Smart Contract Builder for the ERC20 Token.

4.2.2 Performance

The following 4 sections, from 4.2.2.1 and 4.2.2.4, cover the results for Open Auction, Safe Remote Purchase, Voting and ERC20 Token respectively.
Deposit costs are based on the cost of sending data to the blockchain. There are 4 items which make up the full transaction cost:

- the base cost of a transaction (21000 gas)
- the cost of a contract deployment (32000 gas)
- the cost for every zero byte of data or code for a transaction.
- the cost of every non-zero byte of data or code for a transaction.

Execution costs are based on the cost of computational operations which are executed as a result of the transaction.
Unfortunately, Remix IDE’s static analysis tool is unable to obtain the gas usage for some functions because the function modifies large areas of storage, and so it returns an infinite or undefined value for gas usage.

4.2.2.1 Results for Open Auction Smart Contract

The general idea of the Open Auction contract is that everyone can send their bids during a bidding period. The bids already include sending money / ether in order to bind the bidders to their bid. If the highest bid is raised, the previously highest bidder gets their money back.

The results are as follows:

	Open Auction from Documentations	Smart Contract Builder	Performance Difference
Deposit cost	400000 gas	422800 gas	-5.7%
Constructor cost	440866 gas	463667 gas	-5.17%
Bid cost	63208 gas	63543 gas	-0.53%
Withdraw cost	infinite	infinite	-
Auction ended cost	infinite	infinite	-

Table 4.1: Comparison of performance with reference contract for Open Auction

The code generated for Open Auction by the Smart Contract Builder can be found in Appendix N.

4.2.2.2 Results for Safe Remote Purchase Contract

The Safe Remote Purchase contract allows 2 parties, a buyer and a seller to transact without the risk of fraud. The money from the buyer will be released to the seller only when the buyer confirms that he has received the item.

The results are as follows:

	Safe Remote Contract From Documentations	Smart Contract Builder	Performance Difference
Deposit cost	437000 gas	425000 gas	+2.82%
Constructor cost	477996 gas	465813 gas	+2.61%
Abort cost	infinite	infinite	-

Confirm purchase cost	42184 gas	42036 gas	+0.35%
Confirm received cost	infinite	infinite	-

Table 4.2: Comparison of performance with reference contract for Safe Remote Purchase

The code generated for Safe Remote Purchase using the Smart Contract Builder can be found in Appendix O.

4.2.2.3 Results for Voting Smart Contract

The Voting smart contract allows users to vote for proposals that have been submitted by a chairperson. It is significantly more complex than the Open Auction contract, and showcases Solidity's and the Smart Contract Builder's capabilities to a greater extent.

The results are as follows:

	Voting from Documentations	Smart Contract Builder	Performance Difference
Deposit cost	446400 gas	540600 gas	-21.1%
Constructor cost	487492 gas	709092 gas	-45.5%
Add proposal cost	infinite	infinite	-
Give right to vote cost	infinite	infinite	-
Vote cost	102527 gas	184327 gas	-79.8%
Winning proposal cost	418 gas	394 gas	+6.1%

Table 4.3: Comparison of performance with reference contract for Voting

The code generated for Voting using the Smart Contract Builder can be found in Appendix P.

4.2.2.4 Results for ERC20 Token

This contract implements the basic functionality of all ERC20 tokens.

The results are as follows:

audience consisting of developers new to blockchain or non-technical users are unlikely to require these advanced features.

For the Open Auction contract, the deposit costs and execution costs have a less than 6% drop in performance, as shown in Table 4.1. For the Safe Remote Purchase contract, the generated code even slightly outperforms the reference contract because it does not use the modifier and enum code abstractions. The performance for these small, simple contracts does not deviate significantly from the reference contracts.

However, the ERC20 and Voting contracts are far more complex and with more logical branches and constructs. As a result, the deviation in performance is 79.8% for the Voting contract's vote function because of loop and mapping usage. The cost difference for the constructor function of the ERC20 token, 63.88%, is very significant as well, likely because of the heavy use of nested mapping structures. This shows that the code generated by the Smart Contract Builder does not scale well to complex contracts, which limits its usability for industrial and professional use cases. The Smart Contract Builder performs almost optimally for smaller contracts, but is unable to handle complexity well. There is significant room for improvements through enhancement of the parsing algorithm. Users should use the Smart Contract Builder for learning purposes or for creating smaller contracts only. For larger contracts, they may use the Smart Contract Builder to create the skeleton for their smart contract code, but it is not recommended to use it directly in production due to large drops in performance.

5. Conclusions

In this project, the goal was to create a simple, user-friendly application that allowed both developers and non-developers to create smart contracts and deploy them easily. The main focus was on creating a user interface that was clean and simple, while the backend logic would have to be more complex in order to attempt to infer what the user wants to do, such as the types of their variables and the flow of their diagrams.

From the results, we can tell that the Smart Contract Builder is able to implement simple contracts and most of the functionality in advanced contracts, but it is unable to implement highly complex contracts. In terms of performance, the Smart Contract Builder is able to generate a decent performing smart contract for small contracts, but is unable to produce a well performing smart contract for a larger contract. More work will have to be done on the algorithms used to generate the smart contract from the diagrams before it is ready for commercial or industrial use. Furthermore, the Smart Contract Builder will have to ensure the security and safety of its generated code for usage in a production environment.

Nevertheless, the Smart Contract Builder is not focused on achieving optimal performance, but to simplify and encourage development on the blockchain. Developers may use the Smart Contract Builder to generate the skeleton of a smart contract, then further refine it and optimise it manually. Users who are interested in blockchain but do not have the technical know-how can use the Smart Contract Builder as a first step to learning how smart contracts work in general.

6. Recommendations

There is still much room for improvement in terms of usability, such as further abstracting the code layer from the user, which can be done by introducing an even greater amount of natural language processing so that the user can describe what the smart contract should do instead of its low level logic. The diagrams can be generated from their descriptions, which they may further refine manually. Furthermore, with the pace of development for blockchains and Ethereum, it is likely that due to rapid version changes, the application may become outdated when new updates for Solidity are introduced. Constant development is required to keep the application up to date with Solidity standards. It may even be possible that Solidity becomes obsolete as the Ethereum network may decide to move to Vyper, a newer smart contract language with a focus on readability. [18] Future work should be done to ensure that the application keeps up with ongoing developments. Compatibility with other smart contract languages like Vyper can be introduced to allow greater flexibility and future proofing. In addition, compatibility with permissioned Blockchain networks like Hyperledger Fabric can also be introduced so that a larger corporate audience can be reached.

7. End Section

7.1 Appendices

Appendix A: package.json dependencies

```
"devDependencies": {
   "babel-core": "^6.26.3",
   "babel-eslint": "^8.2.6",
   "babel-jest": "^23.4.2",
   "babel-loader": "^7.1.5",
   "babel-plugin-add-module-exports": "^0.2.1",
   "babel-plugin-dev-expression": "^0.2.1",
   "babel-plugin-flow-runtime": "^0.17.0",
   "babel-plugin-transform-class-properties": "^6.24.1",
   "babel-plugin-transform-es2015-classes": "^6.24.1",
   "babel-preset-env": "^1.7.0",
   "babel-preset-react": "^6.24.1",
   "babel-preset-react-optimize": "^1.0.1",
   "babel-preset-stage-0": "^6.24.1",
   "babel-register": "^6.26.0",
   "chalk": "^2.4.1",
   "concurrently": "^3.6.1",
   "cross-env": "^5.2.0",
   "cross-spawn": "^6.0.5",
   "css-loader": "^1.0.0",
   "detect-port": "^1.2.3",
   "electron": "^2.0.6",
   "electron-builder": "^20.26.0",
   "electron-devtools-installer": "^2.2.4",
   "electron-rebuild": "^1.8.2",
   "enzyme": "^3.3.0",
   "enzyme-adapter-react-16": "^1.1.1",
   "enzyme-to-json": "^3.3.4",
   "eslint": "^5.2.0",
   "eslint-config-airbnb": "^17.0.0",
   "eslint-config-prettier": "^2.9.0",
   "eslint-formatter-pretty": "^1.3.0",
   "eslint-import-resolver-webpack": "^0.10.1",
   "eslint-plugin-compat": "^2.5.1",
   "eslint-plugin-flowtype": "^2.50.0",
   "eslint-plugin-import": "^2.13.0",
   "eslint-plugin-jest": "^21.18.0",
   "eslint-plugin-jsx-a11y": "6.1.1",
   "eslint-plugin-promise": "^3.8.0",
   "eslint-plugin-react": "^7.10.0",
   "express": "^4.16.3",
   "fbjs-scripts": "^0.8.3",
   "file-loader": "^1.1.11",
   "flow-bin": "^0.77.0",
   "flow-runtime": "^0.17.0",
   "flow-typed": "^2.5.1",
   "husky": "^0.14.3",
   "identity-obj-proxy": "^3.0.0",
   "jest": "^23.4.2",
   "lint-staged": "^7.2.0",
   "mini-css-extract-plugin": "^0.4.1",
   "minimist": "^1.2.0",
   "node-sass": "^4.9.2",
   "npm-logical-tree": "^1.2.1",
   "optimize-css-assets-webpack-plugin": "^5.0.0",
   "prettier": "^1.14.0",
   "react-test-renderer": "^16.4.1",
   "redux-logger": "^3.0.6",
   "rimraf": "^2.6.2",
   "sass-loader": "^7.0.3",
   "sinon": "^6.1.4",
   "spectron": "^3.8.0",
   "storm-react-diagrams": "^5.2.1",
   "style-loader": "^0.21.0",
   "stylelint": "^9.4.0",
   "stylelint-config-standard": "^18.2.0",
   "uglifyjs-webpack-plugin": "1.2.7",
   "url-loader": "^1.0.1",
   "webpack": "^4.16.3",
   "webpack-bundle-analyzer": "^2.13.1",
   "webpack-cli": "^3.1.0",
   "webpack-dev-server": "^3.1.5",
   "webpack-merge": "^4.1.3",
   "yarn": "^1.9.2"
 },
 "dependencies": {
   "@fortawesome/fontawesome-free": "^5.2.0",
   "@material-ui/core": "^3.0.1",
   "@material-ui/icons": "^3.0.1",
   "devtron": "^1.4.0",
   "electron-debug": "^2.0.0",
   "history": "^4.7.2",
   "react": "^16.4.1",
   "react-dom": "^16.4.1",
   "react-hot-loader": "^4.3.4",
   "react-redux": "^5.0.7",
   "react-router": "^4.3.1",
   "react-router-dom": "^4.3.1",
   "react-router-redux": "^5.0.0-alpha.6",
   "redux": "^4.0.0",
   "redux-thunk": "^2.3.0",
   "source-map-support": "^0.5.6",
   "typeface-roboto": "0.0.54",
   "web3": "^1.0.0-beta.35"
 },
 "devEngines": {
   "node": ">=7.x",
   "npm": ">=4.x",
   "yarn": ">=0.21.3"
 }
```

Appendix B: Structure of a Solidity Smart Contract

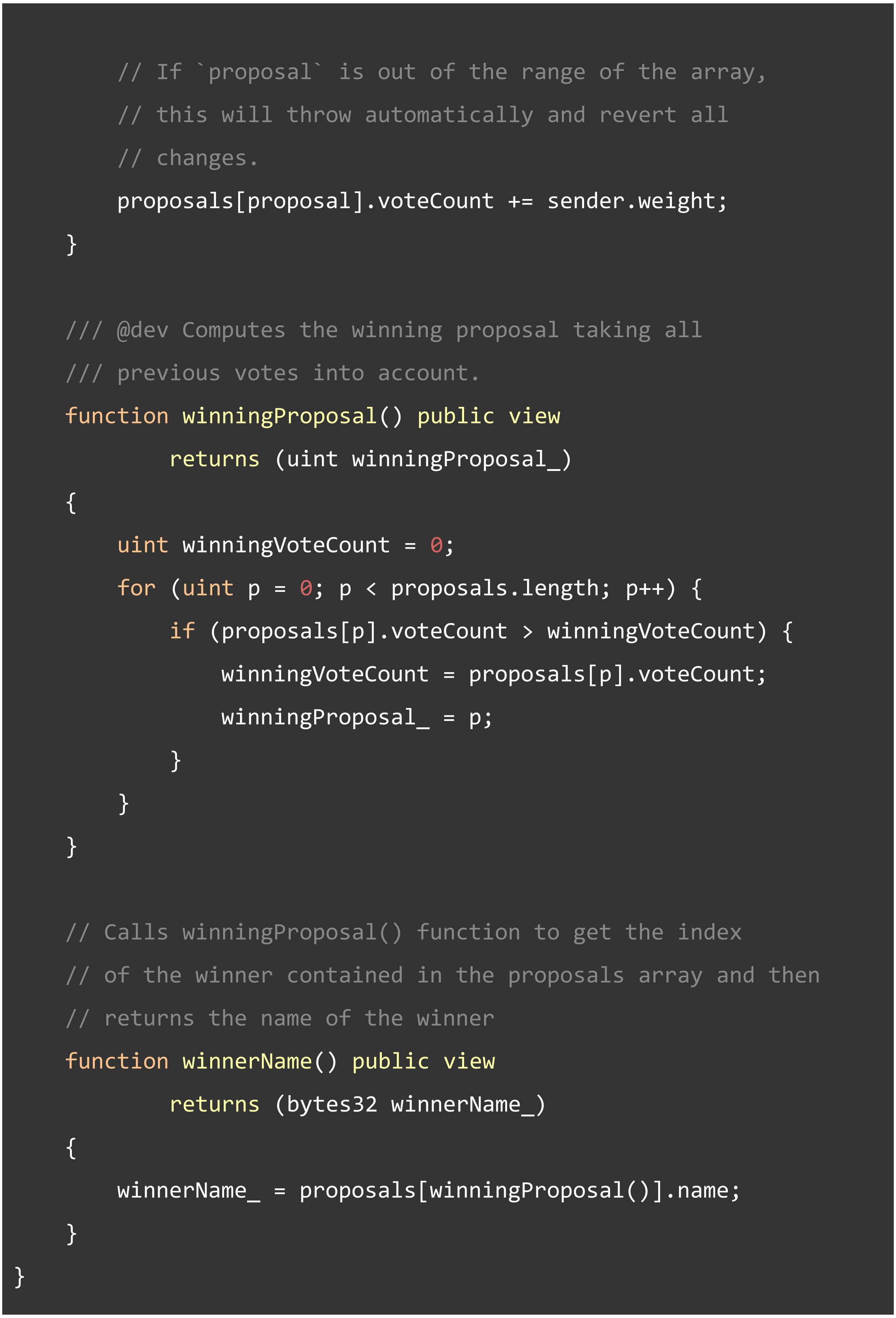
```
pragma solidity ^0.5.4;

contract {contract name} {
    /// variable declarations
    {variable_type} public {variable_name}
    uint public value;
    address public seller;
    address public buyer;

    constructor() public payable {
       {constructor code}
    }

    event {event_name} ({parameter_type} {parameter_name})
    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();

    /// returns does not appear if there is no return statement
function {function_name}({function_params}) public returns (int) {
        {require_statements}
        {function_code}
    }

    function abort() public
    {
        emit Aborted();
        seller.transfer(address(this).balance);
    }
    function confirmPurchase() public payable
    {
        emit PurchaseConfirmed();
        buyer = msg.sender;
    }

    function confirmReceived() public
    {
        emit ItemReceived();
        buyer.transfer(value);
        seller.transfer(address(this).balance);
    }
}
```

Appendix C: Voting from Solidity documentation

```
pragma solidity >=0.4.22 <0.7.0;

/// @title Voting with delegation.
contract Ballot {
    // This declares a new complex type which will
    // be used for variables later.
    // It will represent a single voter.
    struct Voter {
        uint weight; // weight is accumulated by delegation
        bool voted;  // if true, that person already voted
        address delegate; // person delegated to
        uint vote;   // index of the voted proposal
    }

    // This is a type for a single proposal.
    struct Proposal {
        bytes32 name;   // short name (up to 32 bytes)
        uint voteCount; // number of accumulated votes
    }

    address public chairperson;

    // This declares a state variable that
    // stores a `Voter` struct for each possible address.
    mapping(address => Voter) public voters;

    // A dynamically-sized array of `Proposal` structs.
    Proposal[] public proposals;
```

```
/// Create a new ballot to choose one of `proposalNames`.
constructor(bytes32[] memory proposalNames) public {
    chairperson = msg.sender;
    voters[chairperson].weight = 1;

    // For each of the provided proposal names,
    // create a new proposal object and add it
    // to the end of the array.
    for (uint i = 0; i < proposalNames.length; i++) {
        // `Proposal({...})` creates a temporary
        // Proposal object and `proposals.push(...)`
        // appends it to the end of `proposals`.
        proposals.push(Proposal({
            name: proposalNames[i],
            voteCount: 0
        }));
    }
}

// Give `voter` the right to vote on this ballot.
// May only be called by `chairperson`.
function giveRightToVote(address voter) public {
    // If the first argument of `require` evaluates
    // to `false`, execution terminates and all
    // changes to the state and to Ether balances
    // are reverted.
    // This used to consume all gas in old EVM versions, but
    // not anymore.
    // It is often a good idea to use `require` to check if
    // functions are called correctly.
    // As a second argument, you can also provide an
```

```
        // explanation about what went wrong.
        require(
            msg.sender == chairperson,
            "Only chairperson can give right to vote."
        );
        require(
            !voters[voter].voted,
            "The voter already voted."
        );
        require(voters[voter].weight == 0);
        voters[voter].weight = 1;
    }

    /// Delegate your vote to the voter `to`.
    function delegate(address to) public {
        // assigns reference
        Voter storage sender = voters[msg.sender];
        require(!sender.voted, "You already voted.");

        require(to != msg.sender, "Self-delegation is
disallowed.");

        // Forward the delegation as long as
        // `to` also delegated.
        // In general, such loops are very dangerous,
        // because if they run too long, they might
        // need more gas than is available in a block.
        // In this case, the delegation will not be executed,
        // but in other situations, such loops might
        // cause a contract to get "stuck" completely.
        while (voters[to].delegate != address(0)) {
            to = voters[to].delegate;

            // We found a loop in the delegation, not allowed.
            require(to != msg.sender, "Found loop in
delegation.");
        }

        // Since `sender` is a reference, this
        // modifies `voters[msg.sender].voted`
        sender.voted = true;
        sender.delegate = to;
        Voter storage delegate_ = voters[to];
        if (delegate_.voted) {
            // If the delegate already voted,
            // directly add to the number of votes
            proposals[delegate_.vote].voteCount += sender.weight;
        } else {
            // If the delegate did not vote yet,
            // add to her weight.
            delegate_.weight += sender.weight;
        }
    }

    /// Give your vote (including votes delegated to you)
    /// to proposal `proposals[proposal].name`.
    function vote(uint proposal) public {
        Voter storage sender = voters[msg.sender];
        require(sender.weight != 0, "Has no right to vote");
        require(!sender.voted, "Already voted.");
        sender.voted = true;
        sender.vote = proposal;
```



```
        // If `proposal` is out of the range of the array,
        // this will throw automatically and revert all
        // changes.
        proposals[proposal].voteCount += sender.weight;
    }

    /// @dev Computes the winning proposal taking all
    /// previous votes into account.
    function winningProposal() public view
            returns (uint winningProposal_)
    {
        uint winningVoteCount = 0;
        for (uint p = 0; p < proposals.length; p++) {
            if (proposals[p].voteCount > winningVoteCount) {
                winningVoteCount = proposals[p].voteCount;
                winningProposal_ = p;
            }
        }
    }

    // Calls winningProposal() function to get the index
    // of the winner contained in the proposals array and then
    // returns the name of the winner
    function winnerName() public view
            returns (bytes32 winnerName_)
    {
        winnerName_ = proposals[winningProposal()].name;
    }
}
```

Appendix D: Simple Open Auction from Solidity documentation

```
pragma solidity >=0.4.22 <0.6.0;

contract SimpleAuction {
    // Parameters of the auction. Times are either
    // absolute unix timestamps (seconds since 1970-01-01)
    // or time periods in seconds.
    address payable public beneficiary;
    uint public auctionEndTime;

    // Current state of the auction.
    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    // Set to true at the end, disallows any change.
    // By default initialized to `false`.
    bool ended;

    // Events that will be emitted on changes.
    event HighestBidIncreased(address bidder, uint amount);
    event AuctionEnded(address winner, uint amount);

    // The following is a so-called natspec comment,
    // recognizable by the three slashes.
    // It will be shown when the user is asked to
    // confirm a transaction.
```

```
/// Create a simple auction with `_biddingTime`
/// seconds bidding time on behalf of the
/// beneficiary address `_beneficiary`.
constructor(
    uint _biddingTime,
    address payable _beneficiary
) public {
    beneficiary = _beneficiary;
    auctionEndTime = now + _biddingTime;
}

/// Bid on the auction with the value sent
/// together with this transaction.
/// The value will only be refunded if the
/// auction is not won.
function bid() public payable {
    // No arguments are necessary, all
    // information is already part of
    // the transaction. The keyword payable
    // is required for the function to
    // be able to receive Ether.

    // Revert the call if the bidding
    // period is over.
    require(
        now <= auctionEndTime,
        "Auction already ended."
    );

    // If the bid is not higher, send the
    // money back.
```

```
        require(
            msg.value > highestBid,
            "There already is a higher bid."
        );

        if (highestBid != 0) {
            // Sending back the money by simply using
            // highestBidder.send(highestBid) is a security risk
            // because it could execute an untrusted contract.
            // It is always safer to let the recipients
            // withdraw their money themselves.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBidder = msg.sender;
        highestBid = msg.value;
        emit HighestBidIncreased(msg.sender, msg.value);
    }

    /// Withdraw a bid that was overbid.
    function withdraw() public returns (bool) {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the
recipient
            // can call this function again as part of the
receiving call
            // before `send` returns.
            pendingReturns[msg.sender] = 0;

            if (!msg.sender.send(amount)) {
                // No need to call throw here, just reset the
 amount owing
                pendingReturns[msg.sender] = amount;
                return false;
            }
        }
        return true;
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd() public {
        // It is a good guideline to structure functions that
 interact
        // with other contracts (i.e. they call functions or send
 Ether)
        // into three phases:
        // 1. checking conditions
        // 2. performing actions (potentially changing conditions)
        // 3. interacting with other contracts
        // If these phases are mixed up, the other contract could
 call
        // back into the current contract and modify the state or
 cause
        // effects (ether payout) to be performed multiple times.
        // If functions called internally include interaction with
 external
        // contracts, they also have to be considered interaction
 with
        // external contracts.

        // 1. Conditions
```

```
        require(now >= auctionEndTime, "Auction not yet ended.");
        require(!ended, "auctionEnd has already been called.");

        // 2. Effects
        ended = true;
        emit AuctionEnded(highestBidder, highestBid);

        // 3. Interaction
        beneficiary.transfer(highestBid);
    }
}
```

Appendix E: Blind Auction from Solidity documentation

```
pragma solidity >0.4.23 <0.7.0;

contract BlindAuction {
    struct Bid {
        bytes32 blindedBid;
        uint deposit;
    }

    address payable public beneficiary;
    uint public biddingEnd;
    uint public revealEnd;
    bool public ended;

    mapping(address => Bid[]) public bids;

    address public highestBidder;
    uint public highestBid;

    // Allowed withdrawals of previous bids
    mapping(address => uint) pendingReturns;

    event AuctionEnded(address winner, uint highestBid);

    /// Modifiers are a convenient way to validate inputs to
    /// functions. `onlyBefore` is applied to `bid` below:
    /// The new function body is the modifier's body where
    /// `_` is replaced by the old function body.
    modifier onlyBefore(uint _time) { require(now < _time); _; }
    modifier onlyAfter(uint _time) { require(now > _time); _; }
```

```
constructor(
    uint _biddingTime,
    uint _revealTime,
    address payable _beneficiary
) public {
    beneficiary = _beneficiary;
    biddingEnd = now + _biddingTime;
    revealEnd = biddingEnd + _revealTime;
}

/// Place a blinded bid with `_blindedBid` =
/// keccak256(abi.encodePacked(value, fake, secret)).
/// The sent ether is only refunded if the bid is correctly
/// revealed in the revealing phase. The bid is valid if the
/// ether sent together with the bid is at least "value" and
/// "fake" is not true. Setting "fake" to true and sending
/// not the exact amount are ways to hide the real bid but
/// still make the required deposit. The same address can
/// place multiple bids.
function bid(bytes32 _blindedBid)
    public
    payable
    onlyBefore(biddingEnd)
{
    bids[msg.sender].push(Bid({
        blindedBid: _blindedBid,
        deposit: msg.value
    }));
}
```

```
    /// Reveal your blinded bids. You will get a refund for all
    /// correctly blinded invalid bids and for all bids except for
    /// the totally highest.
    function reveal(
        uint[] memory _values,
        bool[] memory _fake,
        bytes32[] memory _secret
    )
        public
        onlyAfter(biddingEnd)
        onlyBefore(revealEnd)
    {
        uint length = bids[msg.sender].length;
        require(_values.length == length);
        require(_fake.length == length);
        require(_secret.length == length);

        uint refund;
        for (uint i = 0; i < length; i++) {
            Bid storage bidToCheck = bids[msg.sender][i];
            (uint value, bool fake, bytes32 secret) =
                    (_values[i], _fake[i], _secret[i]);
            if (bidToCheck.blindedBid !=
keccak256(abi.encodePacked(value, fake, secret))) {
                // Bid was not actually revealed.
                // Do not refund deposit.
                continue;
            }
            refund += bidToCheck.deposit;
            if (!fake && bidToCheck.deposit >= value) {
                if (placeBid(msg.sender, value))
                    refund -= value;
            }
            // Make it impossible for the sender to re-claim
            // the same deposit.
            bidToCheck.blindedBid = bytes32(0);
        }
        msg.sender.transfer(refund);
    }

    // This is an "internal" function which means that it
    // can only be called from the contract itself (or from
    // derived contracts).
    function placeBid(address bidder, uint value) internal
            returns (bool success)
    {
        if (value <= highestBid) {
            return false;
        }
        if (highestBidder != address(0)) {
            // Refund the previously highest bidder.
            pendingReturns[highestBidder] += highestBid;
        }
        highestBid = value;
        highestBidder = bidder;
        return true;
    }

    /// Withdraw a bid that was overbid.
    function withdraw() public {
        uint amount = pendingReturns[msg.sender];
        if (amount > 0) {
            // It is important to set this to zero because the
recipient
            // can call this function again as part of the
receiving call
            // before `transfer` returns (see the remark above
about
            // conditions -> effects -> interaction).
            pendingReturns[msg.sender] = 0;

            msg.sender.transfer(amount);
        }
    }

    /// End the auction and send the highest bid
    /// to the beneficiary.
    function auctionEnd()
        public
        onlyAfter(revealEnd)
    {
        require(!ended);
        emit AuctionEnded(highestBidder, highestBid);
        ended = true;
        beneficiary.transfer(highestBid);
    }
}
```

Appendix F: Safe Remote Purchase from Solidity documentation

```
pragma solidity >=0.4.22 <0.7.0;

contract Purchase {
    uint public value;
    address payable public seller;
    address payable public buyer;
    enum State { Created, Locked, Inactive }
    State public state;

    // Ensure that `msg.value` is an even number.
    // Division will truncate if it is an odd number.
    // Check via multiplication that it wasn't an odd number.
    constructor() public payable {
        seller = msg.sender;
        value = msg.value / 2;
        require((2 * value) == msg.value, "Value has to be
even.");
    }

    modifier condition(bool _condition) {
        require(_condition);
        _;
    }

    modifier onlyBuyer() {
        require(
            msg.sender == buyer,
            "Only buyer can call this."
        );
        _;
    }

    modifier onlySeller() {
        require(
            msg.sender == seller,
            "Only seller can call this."
        );
        _;
    }

    modifier inState(State _state) {
        require(
            state == _state,
            "Invalid state."
        );
        _;
    }

    event Aborted();
    event PurchaseConfirmed();
    event ItemReceived();

    /// Abort the purchase and reclaim the ether.
    /// Can only be called by the seller before
    /// the contract is locked.
    function abort()
        public
        onlySeller
        inState(State.Created)
    {
        emit Aborted();
        state = State.Inactive;
        seller.transfer(address(this).balance);
    }

    /// Confirm the purchase as buyer.
    /// Transaction has to include `2 * value` ether.
    /// The ether will be locked until confirmReceived
    /// is called.
    function confirmPurchase()
        public
        inState(State.Created)
        condition(msg.value == (2 * value))
        payable
    {
        emit PurchaseConfirmed();
        buyer = msg.sender;
        state = State.Locked;
    }

    /// Confirm that you (the buyer) received the item.
    /// This will release the locked ether.
    function confirmReceived()
        public
        onlyBuyer
        inState(State.Locked)
    {
        emit ItemReceived();
        // It is important to change the state first because
        // otherwise, the contracts called using `send` below
        // can call in again here.
        state = State.Inactive;

        // NOTE: This actually allows both the buyer and the
seller to
        // block the refund - the withdraw pattern should be used.

        buyer.transfer(value);
        seller.transfer(address(this).balance);
    }
}
```

Appendix G: Micropayments Channel Receiver from Solidity documentations

```
pragma solidity >=0.4.24 <0.7.0;

contract ReceiverPays {
    address owner = msg.sender;

    mapping(uint256 => bool) usedNonces;

    constructor() public payable {}

    function claimPayment(uint256 amount, uint256 nonce, bytes
memory signature) public {
        require(!usedNonces[nonce]);
        usedNonces[nonce] = true;

        // this recreates the message that was signed on the
client
        bytes32 message =
prefixed(keccak256(abi.encodePacked(msg.sender, amount, nonce,
this)));

        require(recoverSigner(message, signature) == owner);

        msg.sender.transfer(amount);
    }

    /// destroy the contract and reclaim the leftover funds.
    function kill() public {
        require(msg.sender == owner);
        selfdestruct(msg.sender);
    }

    /// signature methods.
    function splitSignature(bytes memory sig)
        internal
        pure
        returns (uint8 v, bytes32 r, bytes32 s)
    {
        require(sig.length == 65);

        assembly {
            // first 32 bytes, after the length prefix.
            r := mload(add(sig, 32))
            // second 32 bytes.
            s := mload(add(sig, 64))
            // final byte (first byte of the next 32 bytes).
            v := byte(0, mload(add(sig, 96)))
        }

        return (v, r, s);
    }

    function recoverSigner(bytes32 message, bytes memory sig)
        internal
        pure
        returns (address)
    {
        (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);

        return ecrecover(message, v, r, s);
    }

    /// builds a prefixed hash to mimic the behavior of eth_sign.
    function prefixed(bytes32 hash) internal pure returns
(bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed
Message:\n32", hash));
    }
}
```

Appendix H: Simple Payments Channel from Solidity documentations

```
pragma solidity >=0.4.24 <0.7.0;

contract SimplePaymentChannel {
    address payable public sender;      // The account sending
payments.
    address payable public recipient;   // The account receiving
the payments.
    uint256 public expiration;  // Timeout in case the recipient
never closes.

    constructor (address payable _recipient, uint256 duration)
        public
        payable
    {
        sender = msg.sender;
        recipient = _recipient;
        expiration = now + duration;
    }

    function isValidSignature(uint256 amount, bytes memory
signature)
        internal
        view
        returns (bool)
    {
        bytes32 message =
prefixed(keccak256(abi.encodePacked(this, amount)));

        // check that the signature is from the payment sender
        return recoverSigner(message, signature) == sender;
    }

    /// the recipient can close the channel at any time by
presenting a
    /// signed amount from the sender. the recipient will be sent
that amount,
    /// and the remainder will go back to the sender
    function close(uint256 amount, bytes memory signature) public
{
        require(msg.sender == recipient);
        require(isValidSignature(amount, signature));

        recipient.transfer(amount);
        selfdestruct(sender);
    }

    /// the sender can extend the expiration at any time
    function extend(uint256 newExpiration) public {
        require(msg.sender == sender);
        require(newExpiration > expiration);

        expiration = newExpiration;
    }

    /// if the timeout is reached without the recipient closing
the channel,
    /// then the Ether is released back to the sender.
    function claimTimeout() public {
        require(now >= expiration);
        selfdestruct(sender);
```

```
}

/// All functions below this are just taken from the chapter
/// 'creating and verifying signatures' chapter.

function splitSignature(bytes memory sig)
    internal
    pure
    returns (uint8 v, bytes32 r, bytes32 s)
{
    require(sig.length == 65);

    assembly {
        // first 32 bytes, after the length prefix
        r := mload(add(sig, 32))
        // second 32 bytes
        s := mload(add(sig, 64))
        // final byte (first byte of the next 32 bytes)
        v := byte(0, mload(add(sig, 96)))
    }

    return (v, r, s);
}

function recoverSigner(bytes32 message, bytes memory sig)
    internal
    pure
    returns (address)
{
    (uint8 v, bytes32 r, bytes32 s) = splitSignature(sig);
```

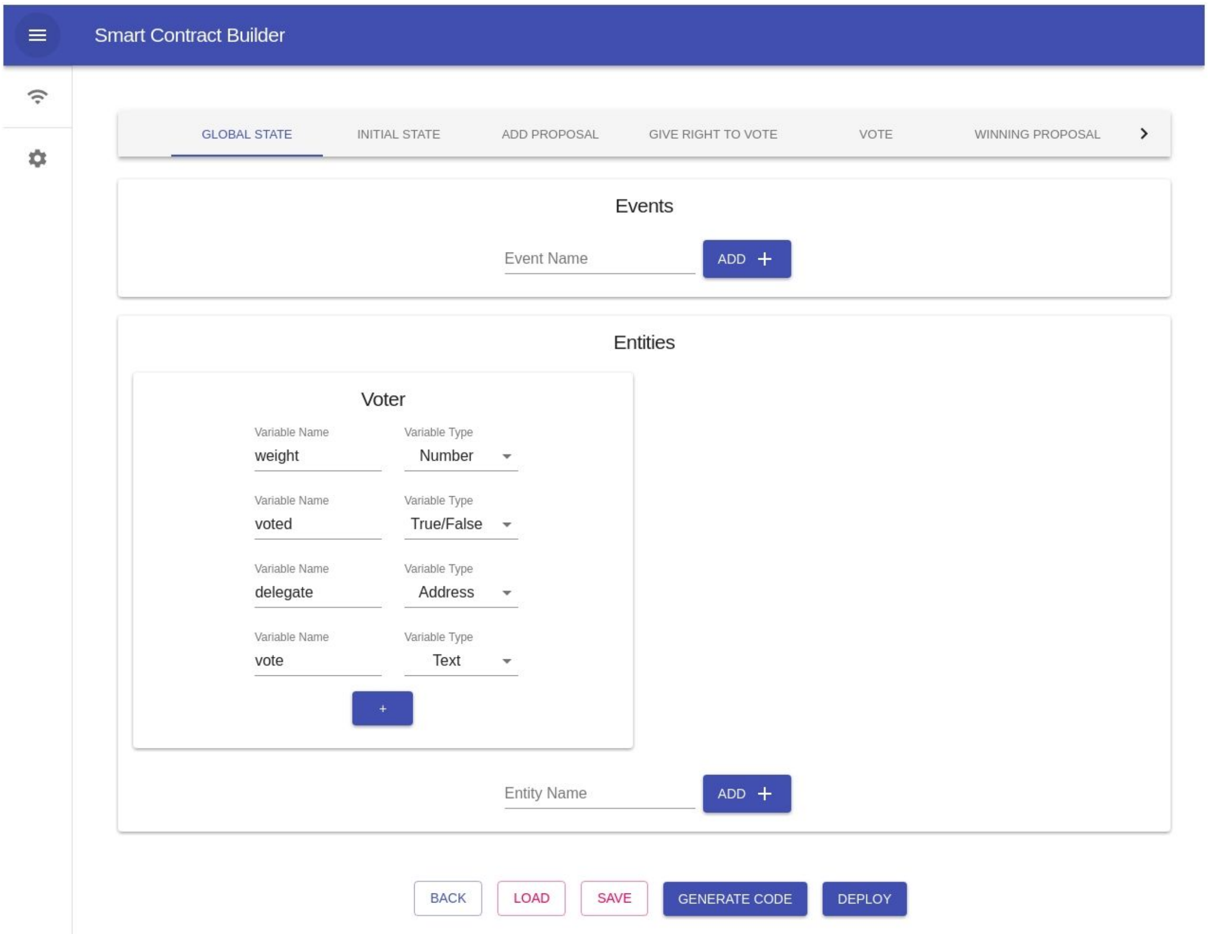
```
        return ecrecover(message, v, r, s);
    }

    /// builds a prefixed hash to mimic the behavior of eth_sign.
    function prefixed(bytes32 hash) internal pure returns
(bytes32) {
        return keccak256(abi.encodePacked("\x19Ethereum Signed
Message:\n32", hash));
    }
}
```

Appendix I: Modular Contracts from Solidity documentations

```
pragma solidity >=0.4.22 <0.7.0;

library Balances {
    function move(mapping(address => uint256) storage balances,
address from, address to, uint amount) internal {
        require(balances[from] >= amount);
        require(balances[to] + amount >= balances[to]);
        balances[from] -= amount;
        balances[to] += amount;
    }
}

contract Token {
    mapping(address => uint256) balances;
    using Balances for *;
    mapping(address => mapping (address => uint256)) allowed;

    event Transfer(address from, address to, uint amount);
    event Approval(address owner, address spender, uint amount);

    function balanceOf(address tokenOwner) public view returns
(uint balance) {
        return balances[tokenOwner];
    }
    function transfer(address to, uint amount) public returns
(bool success) {
        balances.move(msg.sender, to, amount);
        emit Transfer(msg.sender, to, amount);
        return true;
    }

    function transferFrom(address from, address to, uint amount)
public returns (bool success) {
        require(allowed[from][msg.sender] >= amount);
        allowed[from][msg.sender] -= amount;
        balances.move(from, to, amount);
        emit Transfer(from, to, amount);
        return true;
    }

    function approve(address spender, uint tokens) public returns
(bool success) {
        require(allowed[msg.sender][spender] == 0, "");
        allowed[msg.sender][spender] = tokens;
        emit Approval(msg.sender, spender, tokens);
        return true;
    }
}
```

Appendix J: Voting Build Tabs

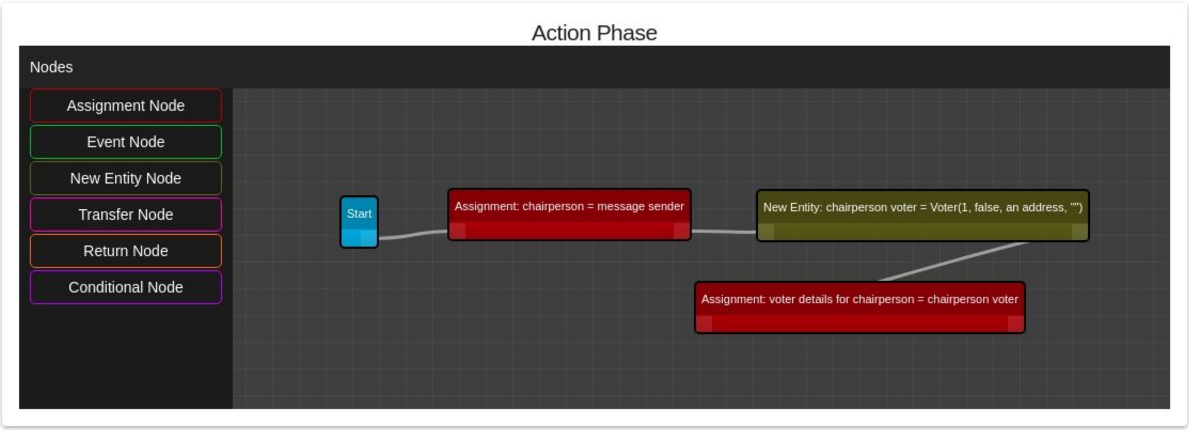
Global State Tab



Voter

Variable Name	Variable Type
weight	Number
voted	True/False
delegate	Address
vote	Text

Constructor Function

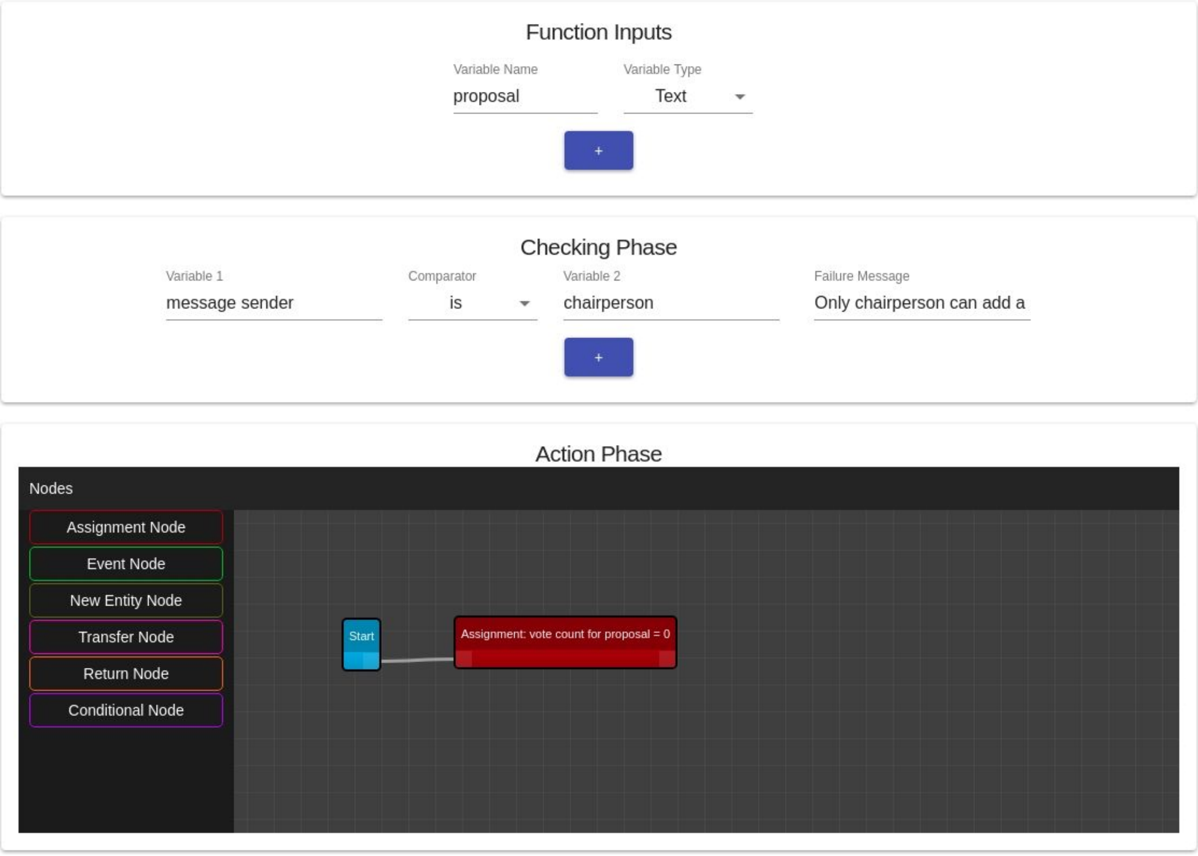


Assignment: chairperson = message sender

New Entity: chairperson voter = Voter(1, false, an address, "")

Assignment: voter details for chairperson = chairperson voter

Add Proposal Function



Function Inputs

Variable Name: proposal

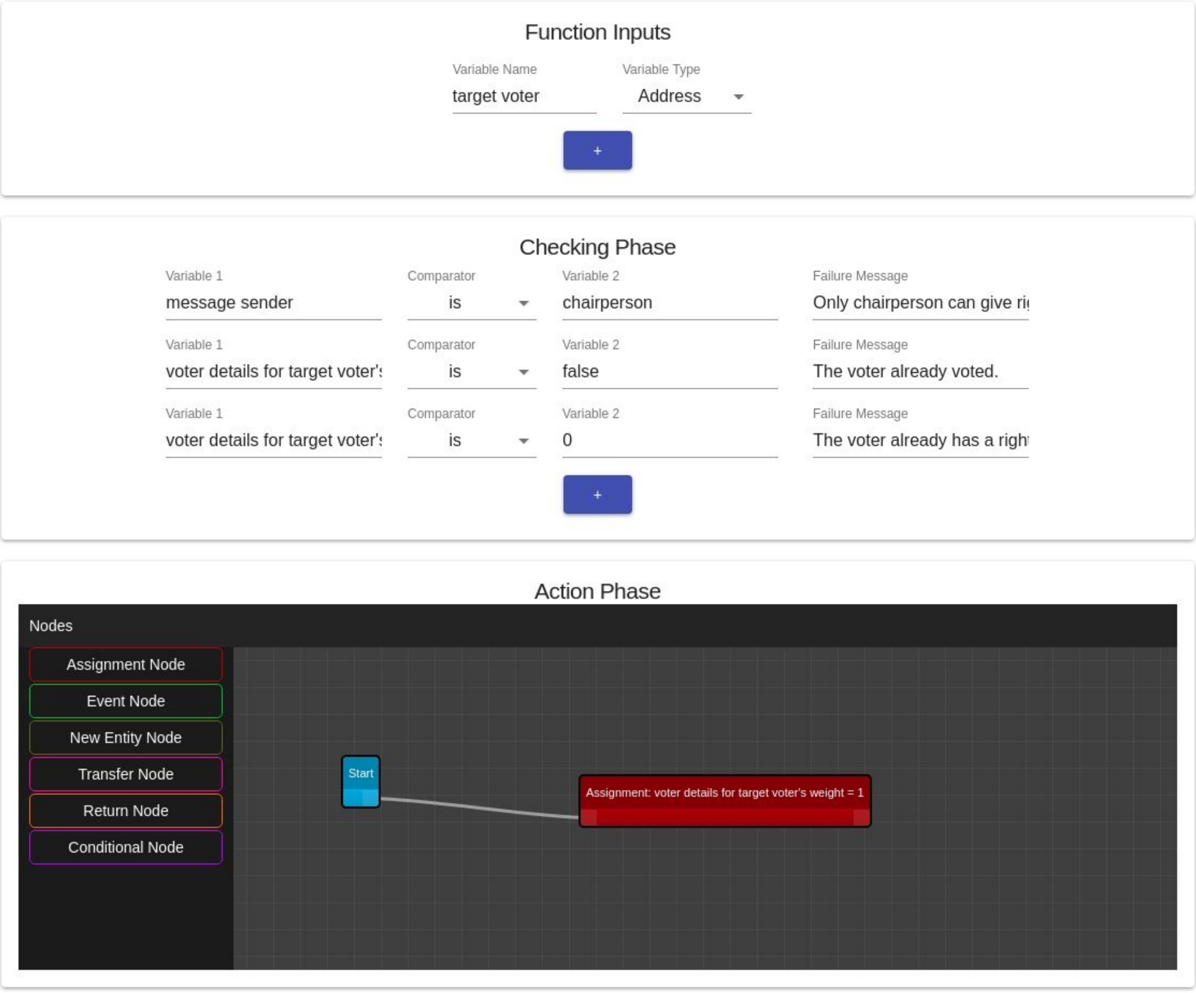
Variable Type: Text

Checking Phase

Variable 1	Comparator	Variable 2	Failure Message
message sender	is	chairperson	Only chairperson can add a

Action Phase

Give Right To Vote Function



Function Inputs

Variable Name	Variable Type
target voter	Address

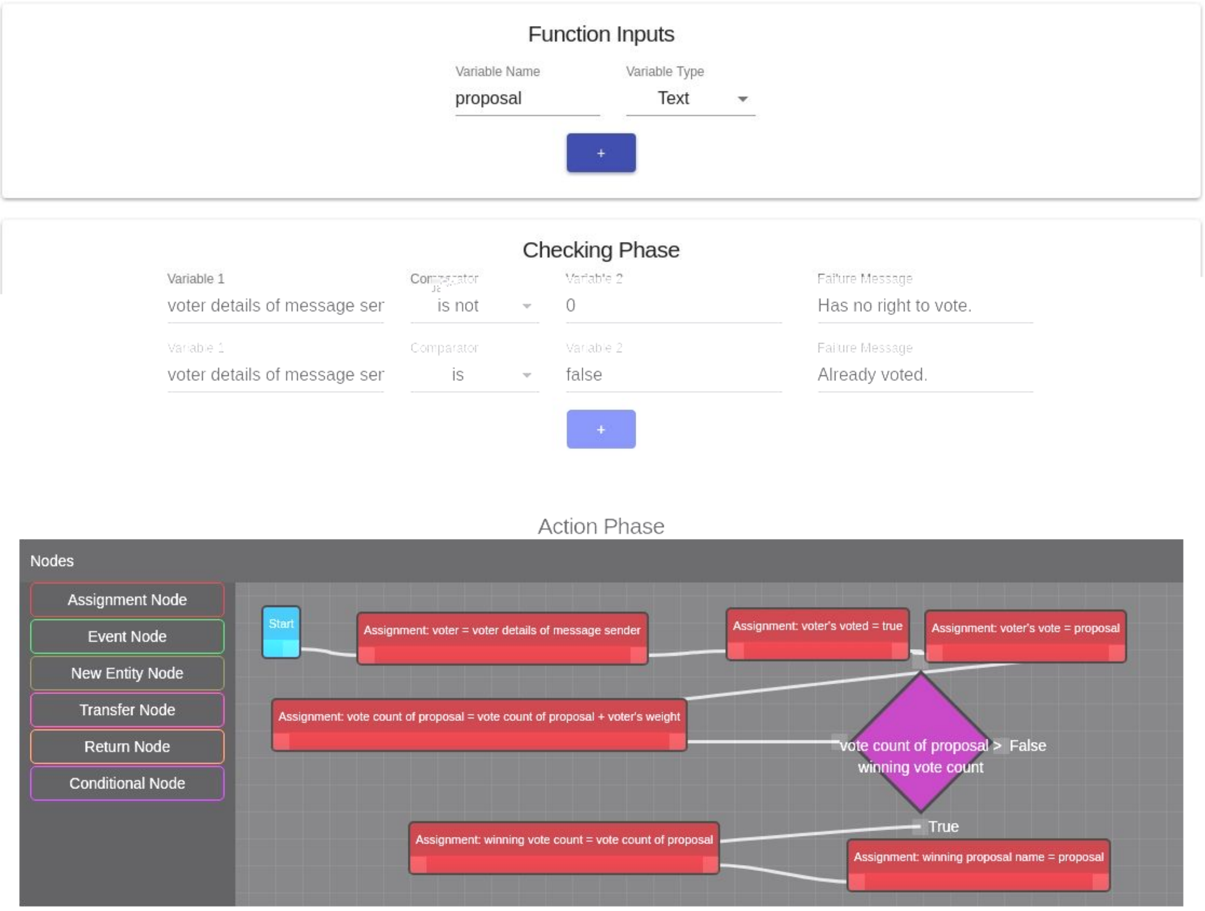
Checking Phase

Variable 1	Comparator	Variable 2	Failure Message
message sender	is	chairperson	Only chairperson can give ri
voter details for target voter's	is	false	The voter already voted.
voter details for target voter's	is	0	The voter already has a righ

Action Phase

Assignment: voter details for target voter's weight = 1

Vote Function



Function Inputs

Variable Name: proposal

Variable Type: Text

Checking Phase

Variable 1	Comparator	Variable 2	Failure Message
voter details of message ser	is not	0	Has no right to vote.
voter details of message ser	is	false	Already voted.

Action Phase

Start

Assignment: voter = voter details of message sender

Assignment: voter's voted = true

Assignment: voter's vote = proposal

Assignment: vote count of proposal = vote count of proposal + voter's weight

vote count of proposal > winning vote count

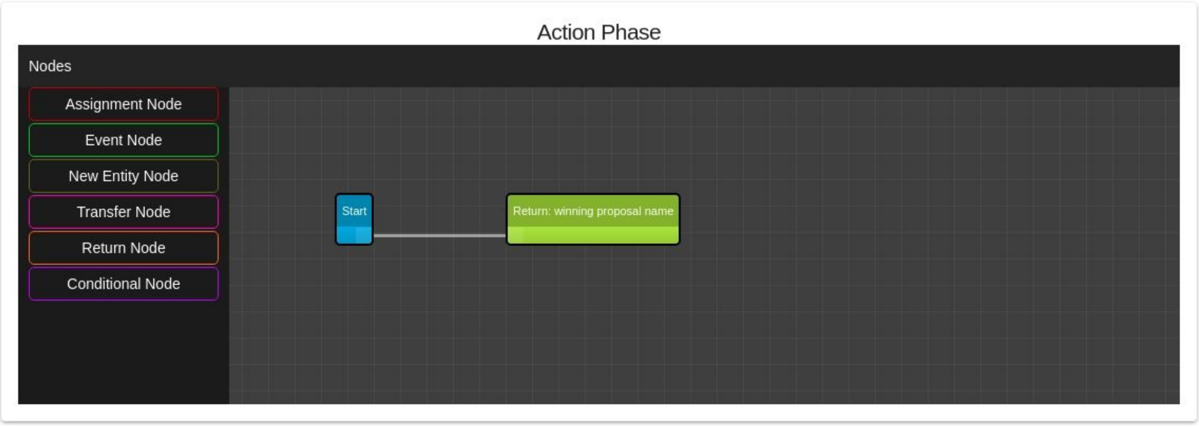
False

True

Assignment: winning vote count = vote count of proposal

Assignment: winning proposal name = proposal

Winning Proposal Function



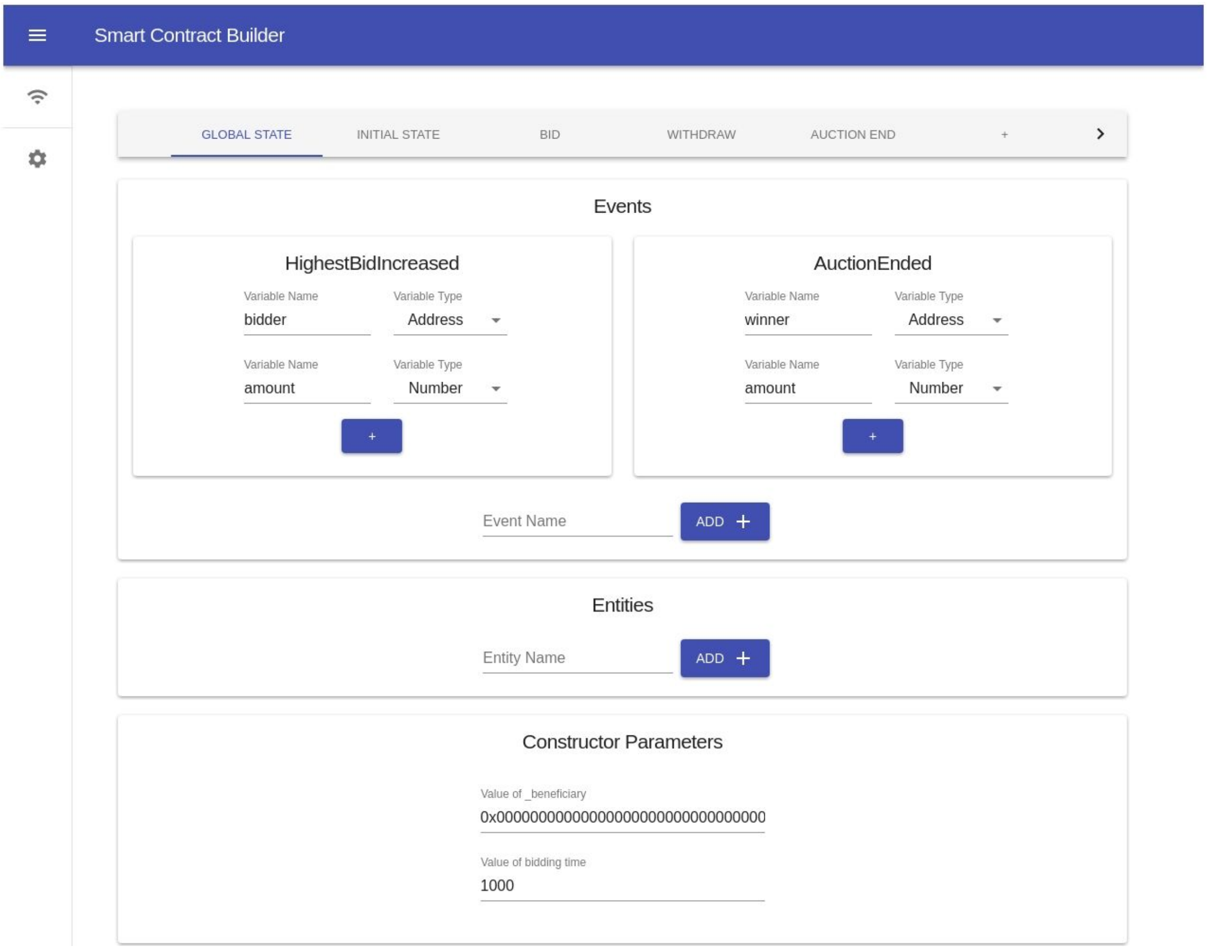
Action Phase

Start

Return: winning proposal name

Appendix K: Open Auction Build Tabs

Global State Tab



Events

HighestBidIncreased

Variable Name: bidder
Variable Type: Address

Variable Name: amount
Variable Type: Number

AuctionEnded

Variable Name: winner
Variable Type: Address

Variable Name: amount
Variable Type: Number

Event Name

Entities

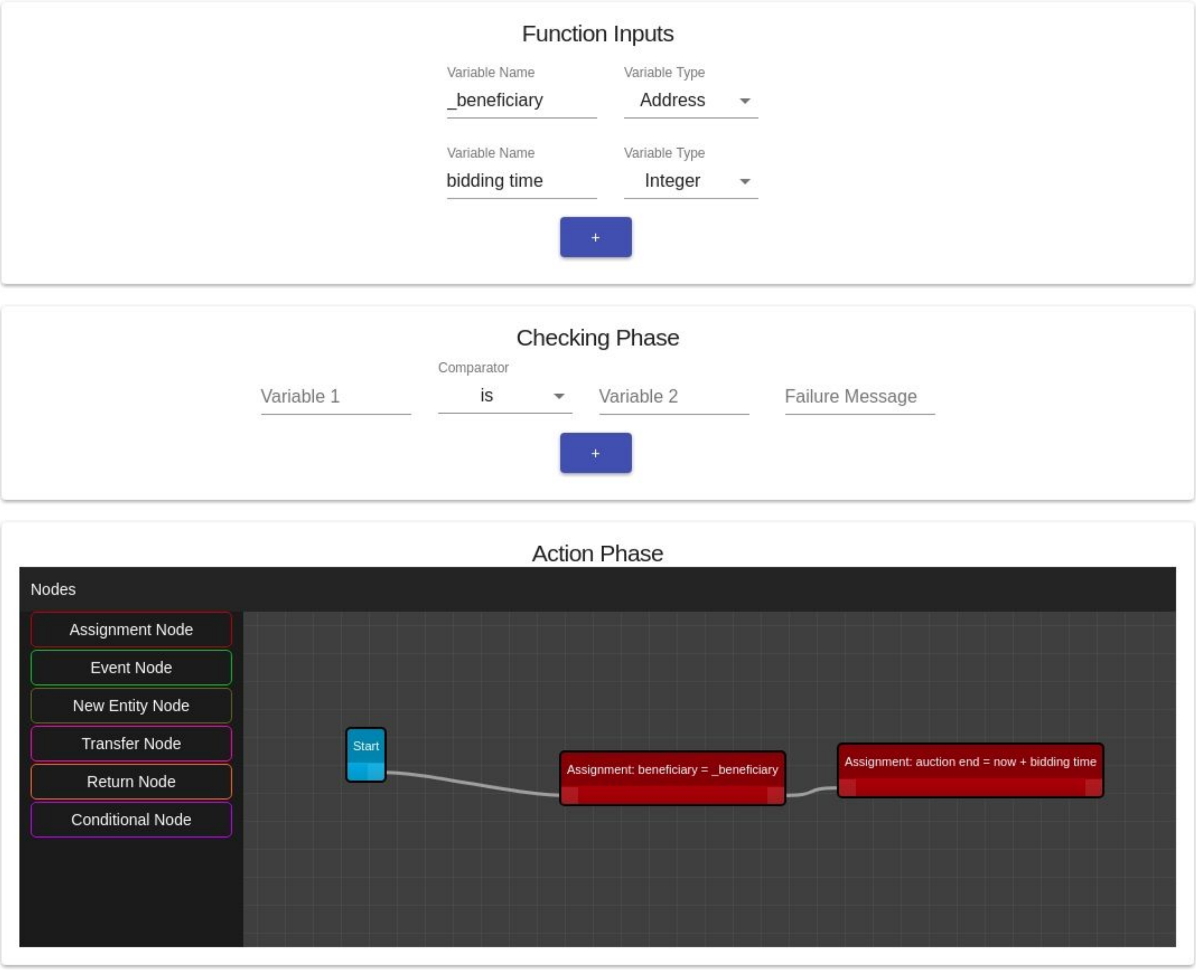
Entity Name

Constructor Parameters

Value of _beneficiary: 0x00000000000000000000000000000000

Value of bidding time: 1000

Constructor Function



Function Inputs

Variable Name: _beneficiary
Variable Type: Address

Variable Name: bidding time
Variable Type: Integer

Checking Phase

Variable 1
Comparator: is
Variable 2
Failure Message

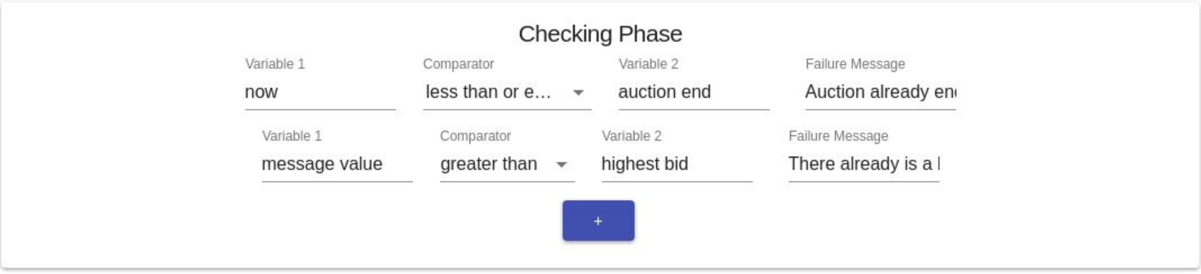
Action Phase

Start

Assignment: beneficiary = _beneficiary

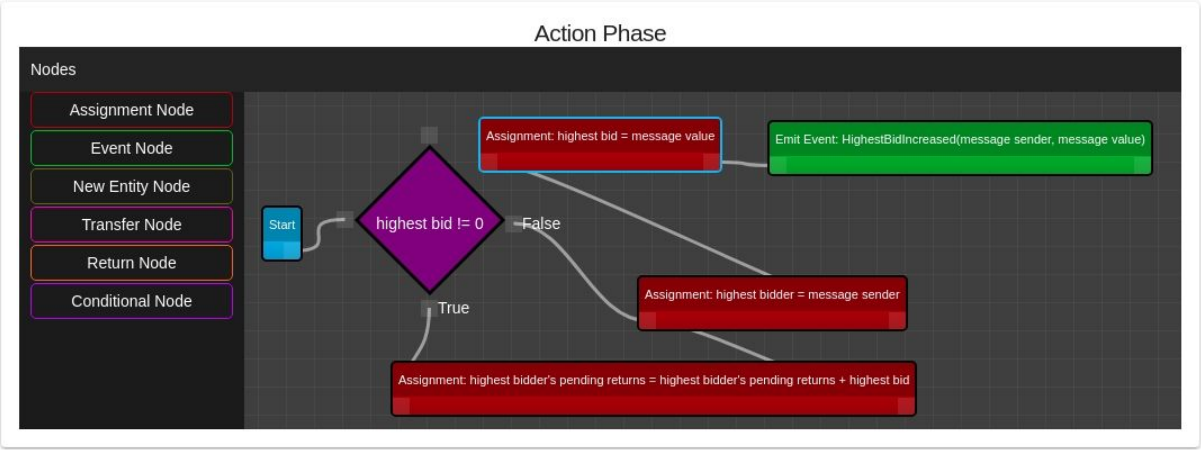
Assignment: auction end = now + bidding time

Bid Function



Checking Phase

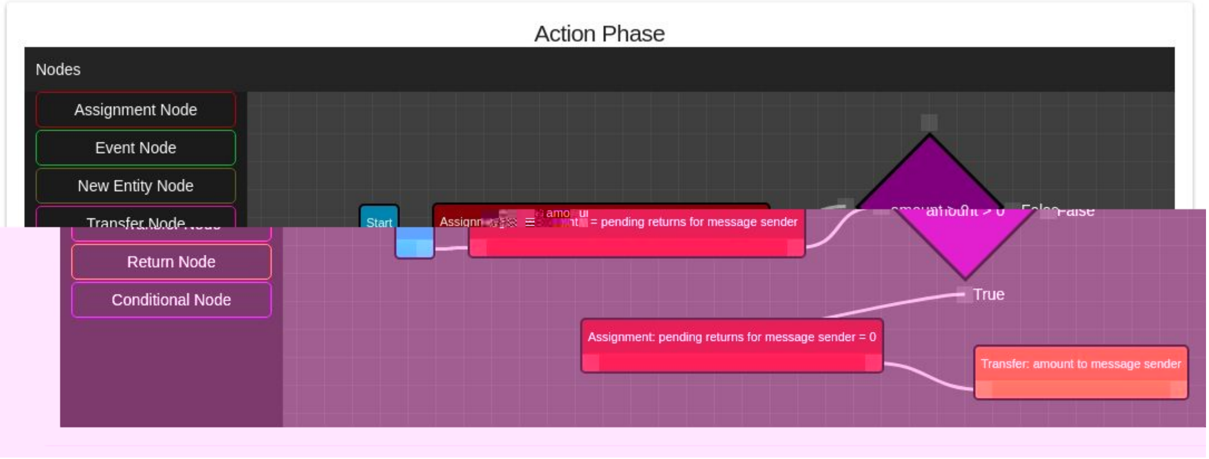
Variable 1	Comparator	Variable 2	Failure Message
now	less than or e…	auction end	Auction already en
message value	greater than	highest bid	There already is a l



Action Phase

Nodes
Assignment Node
Event Node
New Entity Node
Transfer Node
Return Node
Conditional Node
Start
highest bid != 0
False
True
Assignment: highest bid = message value
Emit Event: HighestBidIncreased(message sender, message value)
Assignment: highest bidder = message sender
Assignment: highest bidder's pending returns = highest bidder's pending returns + highest bid

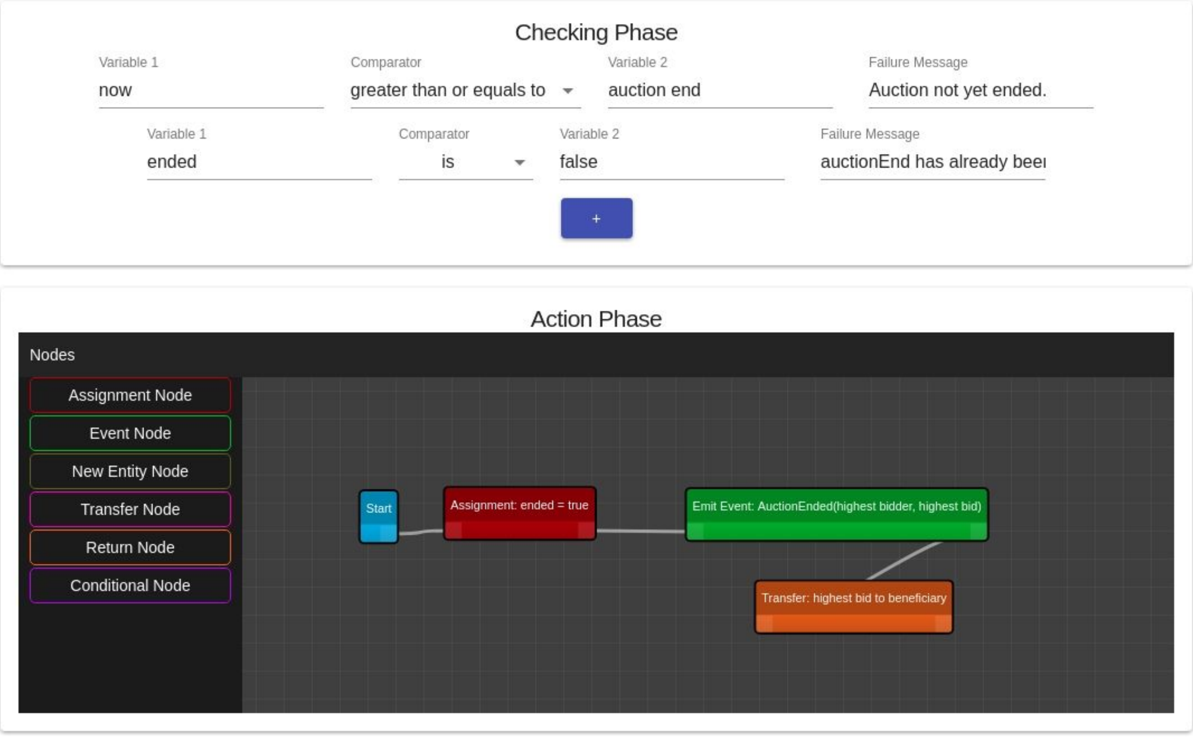
Withdraw Function



Action Phase

Nodes
Assignment Node
Event Node
New Entity Node
Transfer Node
Return Node
Conditional Node
Start
= pending returns for message sender
amount > 0
False
True
Assignment: pending returns for message sender = 0
Transfer: amount to message sender

Auction End Function



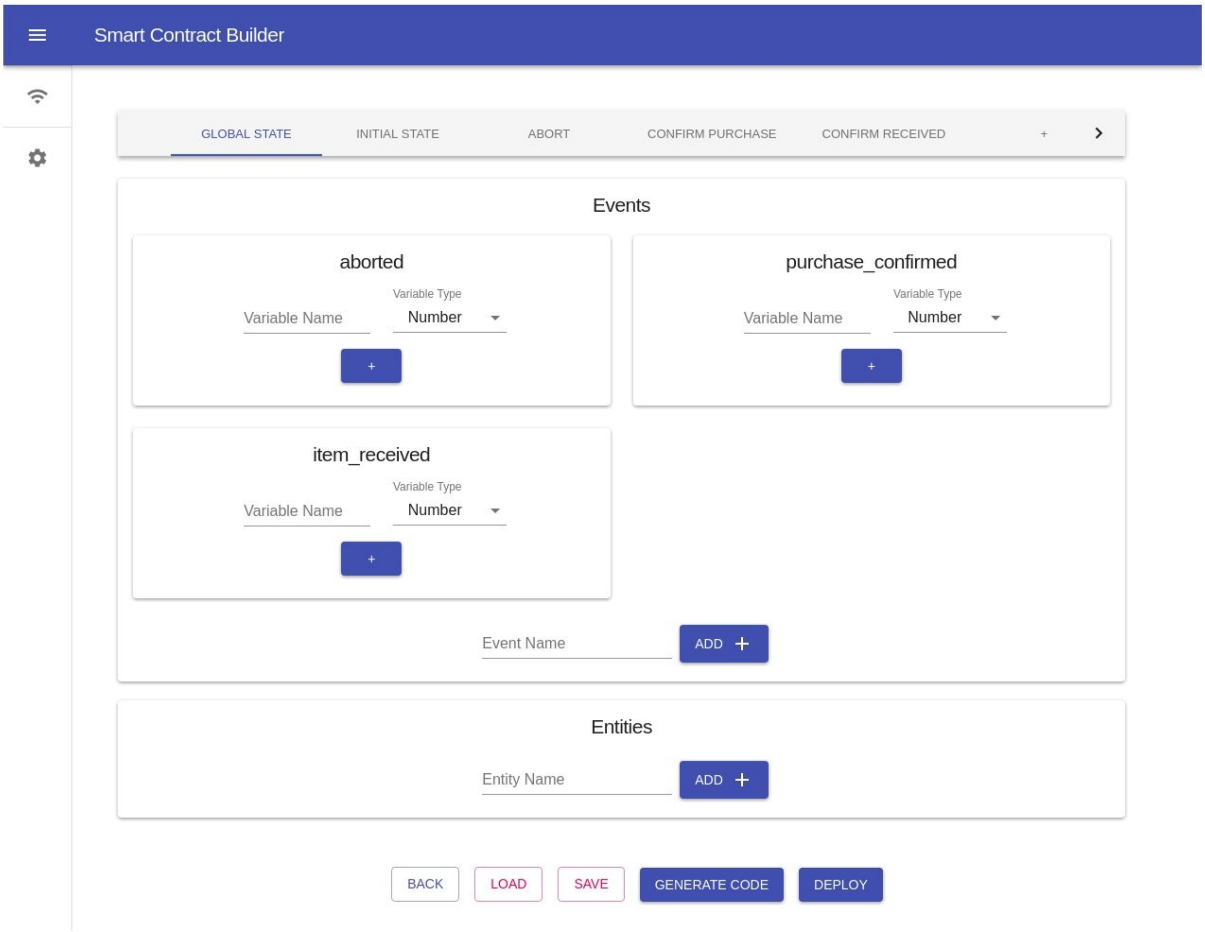
Checking Phase

Variable 1	Comparator	Variable 2	Failure Message
now	greater than or equals to	auction end	Auction not yet ended.
ended	is	false	auctionEnd has already bee

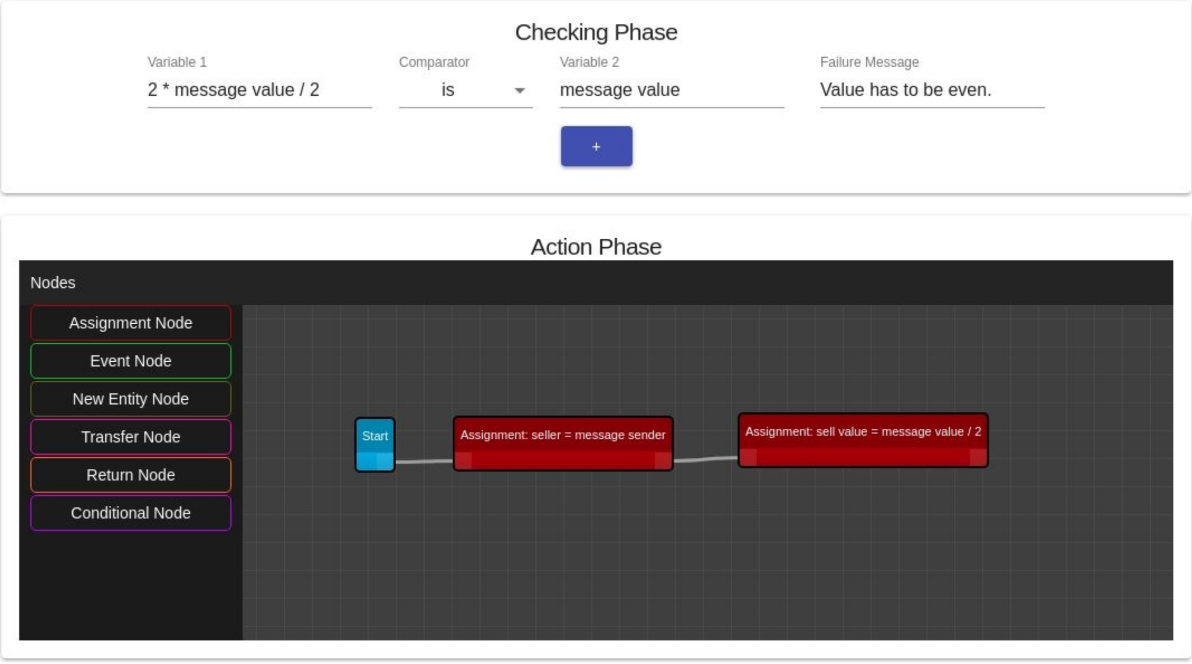
Action Phase

Appendix L: Safe Remote Purchase

Global State Tab



Constructor Function



Checking Phase

Variable 1	Comparator	Variable 2	Failure Message
2 * message value / 2	is	message value	Value has to be even.

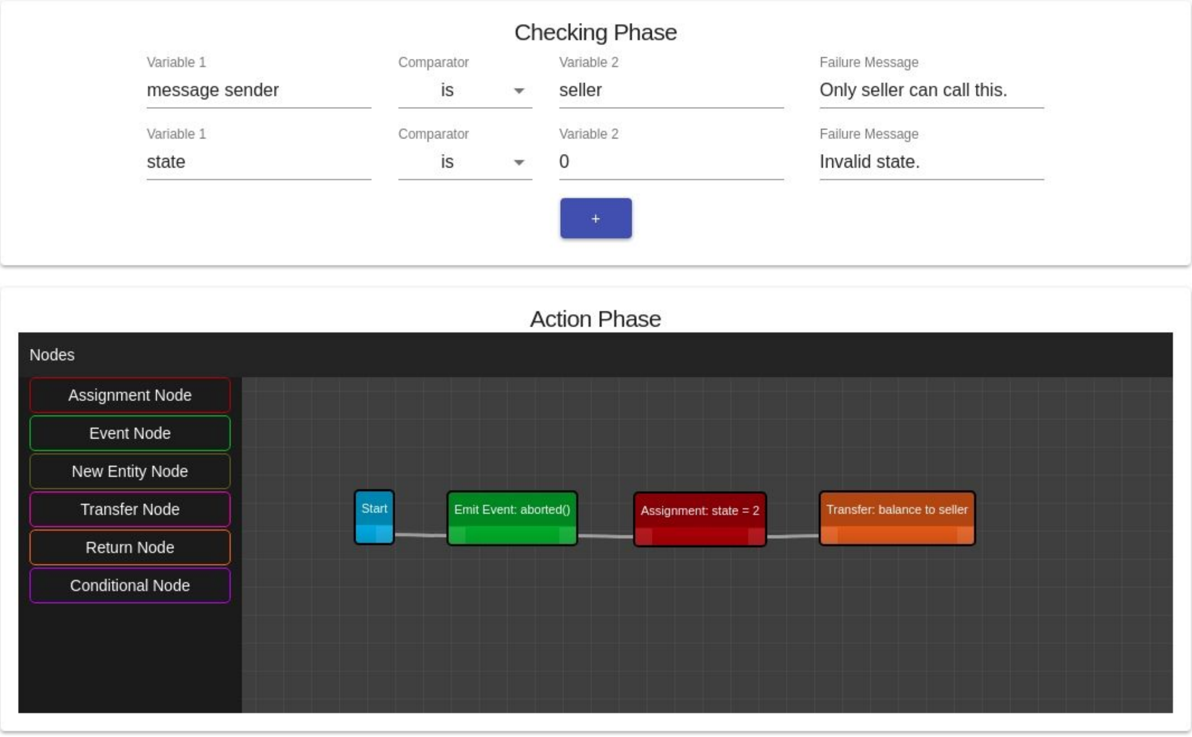
Action Phase

Start

Assignment: seller = message sender

Assignment: sell value = message value / 2

Abort Function



Checking Phase

Variable 1	Comparator	Variable 2	Failure Message
message sender	is	seller	Only seller can call this.
state	is	0	Invalid state.

Action Phase

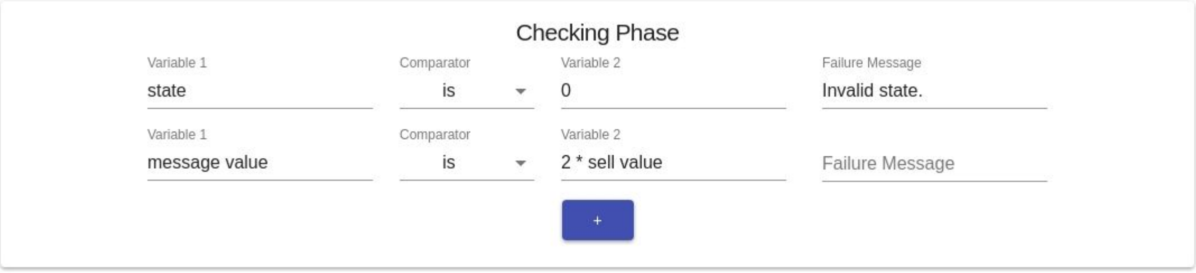
Start

Emit Event: aborted()

Assignment: state = 2

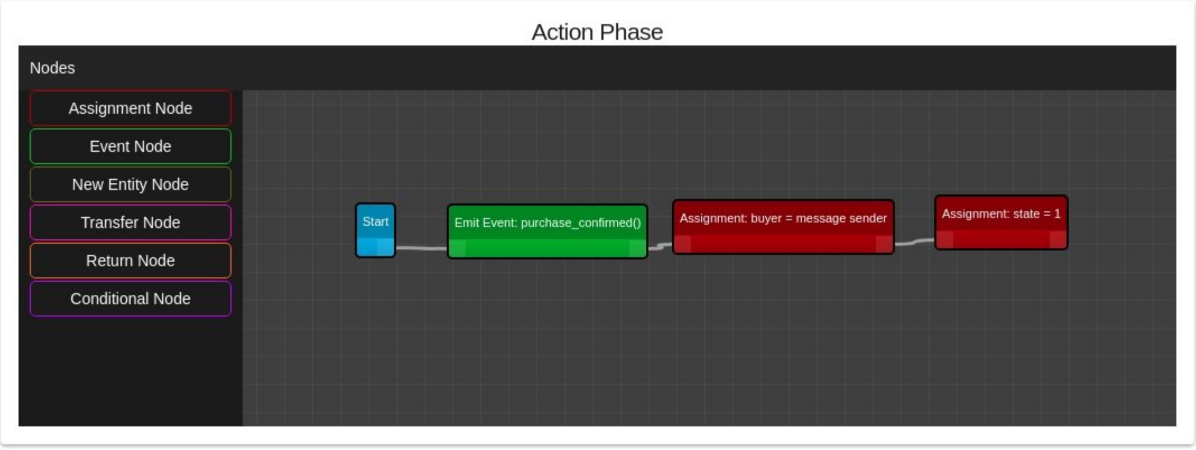
Transfer: balance to seller

Confirm Purchase Function



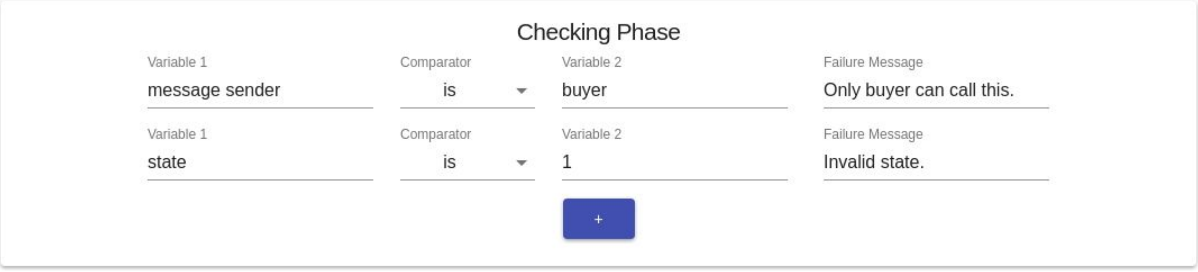
Checking Phase

Variable 1	Comparator	Variable 2	Failure Message
state	is	0	Invalid state.
message value	is	2 * sell value	Failure Message



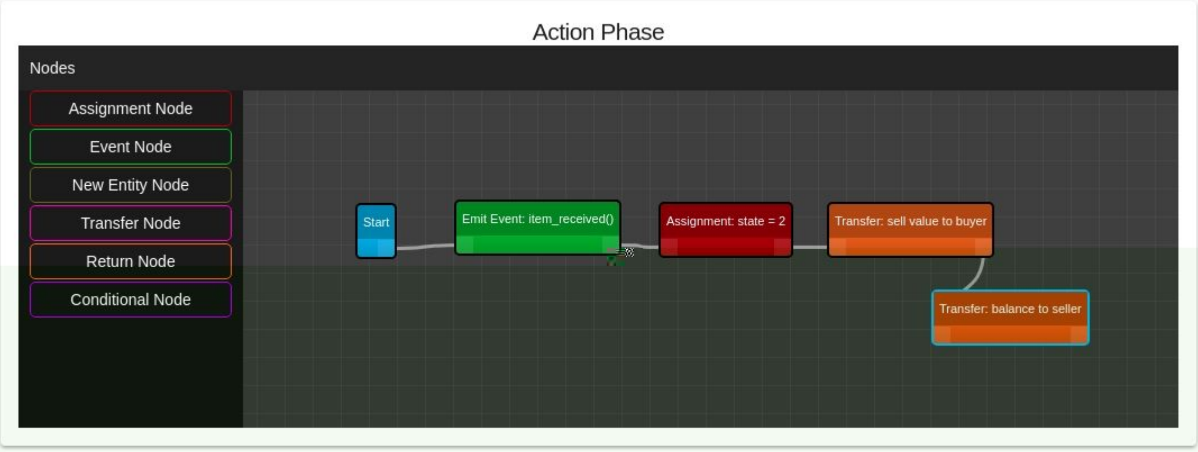
Action Phase

Confirm Received Function



Checking Phase

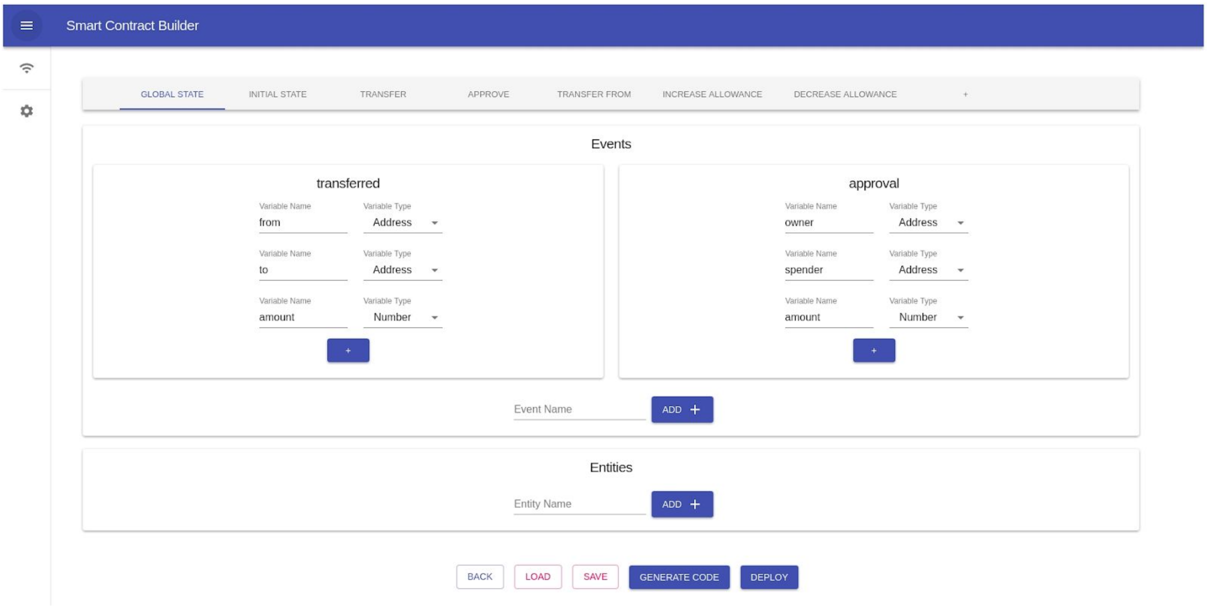
Variable 1	Comparator	Variable 2	Failure Message
message sender	is	buyer	Only buyer can call this.
state	is	1	Invalid state.



Action Phase

Appendix M: ERC20 Token

Global State Tab for ERC20



Events

transferred

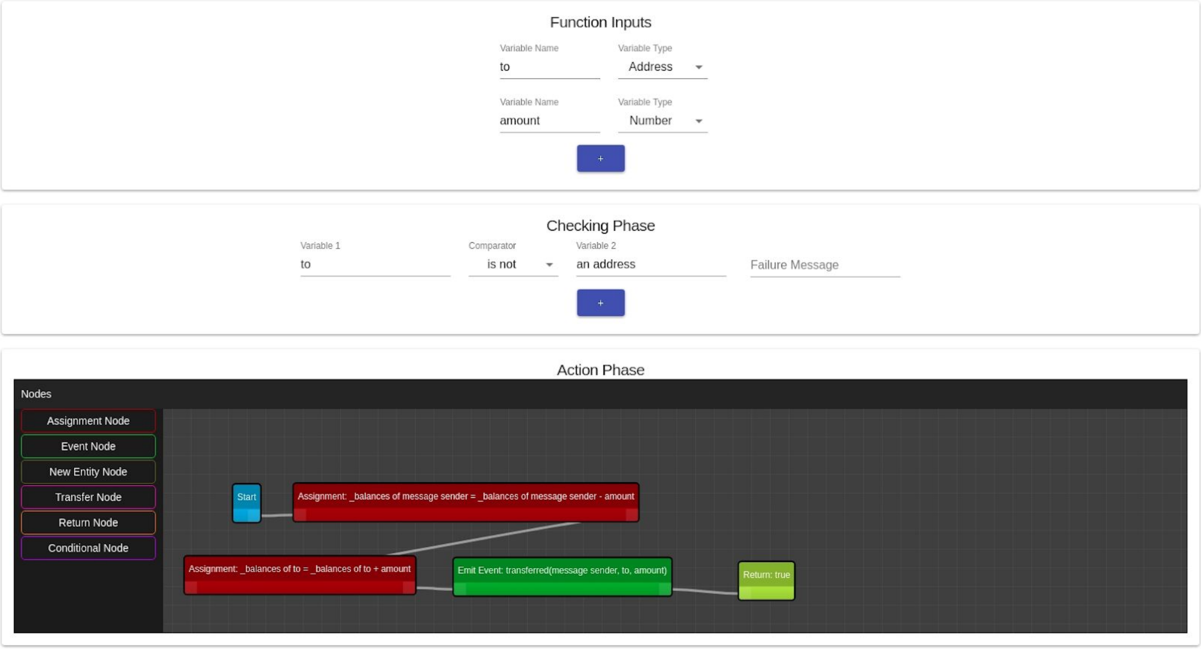
Variable Name	Variable Type
from	Address
to	Address
amount	Number

approval

Variable Name	Variable Type
owner	Address
spender	Address
amount	Number

Entities

Transfer Function



Function Inputs

Variable Name	Variable Type
to	Address
amount	Number

Checking Phase

Variable 1	Comparator	Variable 2	Failure Message
to	is not	an address	

Action Phase

Start

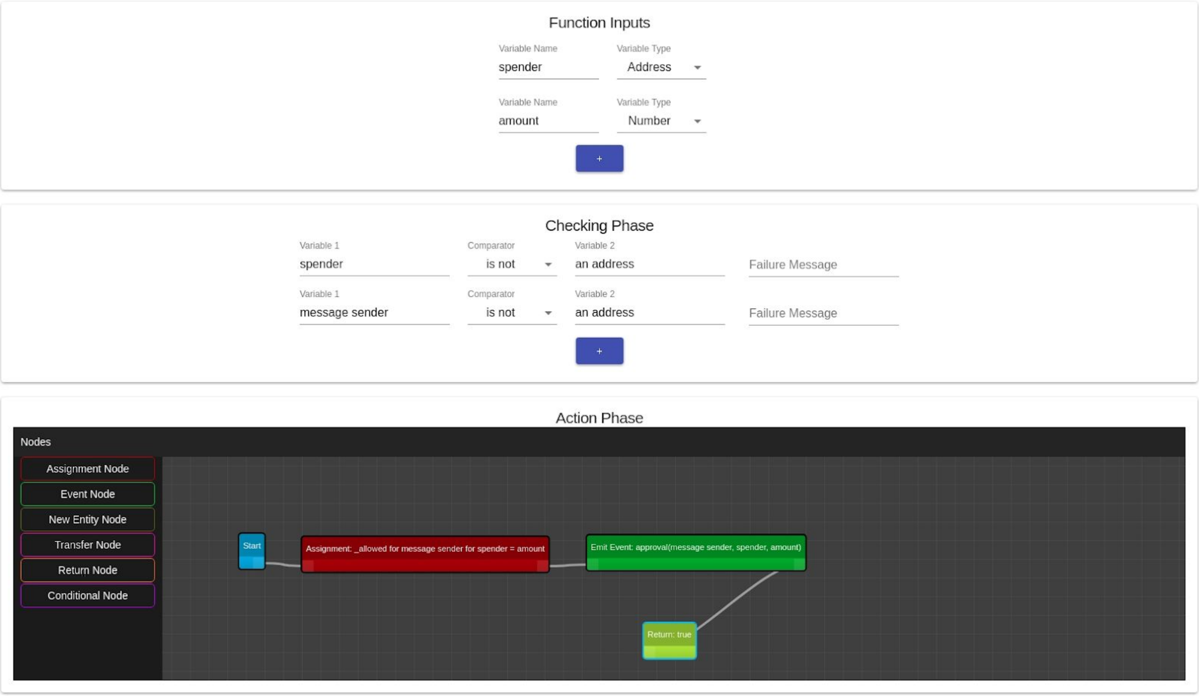
Assignment: _balances of message sender = _balances of message sender - amount

Assignment: _balances of to = _balances of to + amount

Emit Event: transferred(message sender, to, amount)

Return: true

Approve Function



Function Inputs

Variable Name	Variable Type
spender	Address
amount	Number

Checking Phase

Variable 1	Comparator	Variable 2	
spender	is not	an address	Failure Message
message sender	is not	an address	Failure Message

Action Phase

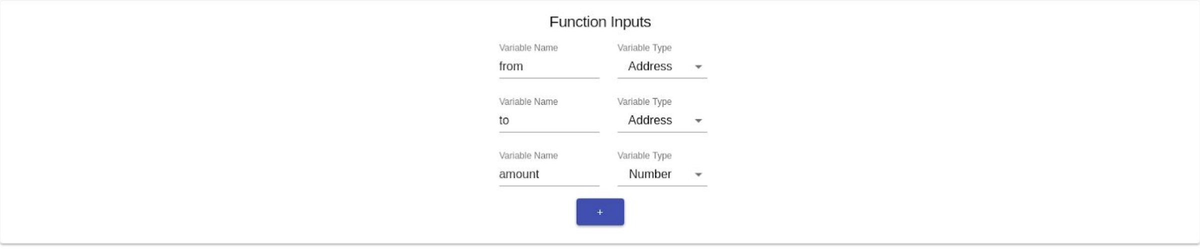
Start

Assignment: _allowed for message sender for spender = amount

Emit Event: approval(message sender, spender, amount)

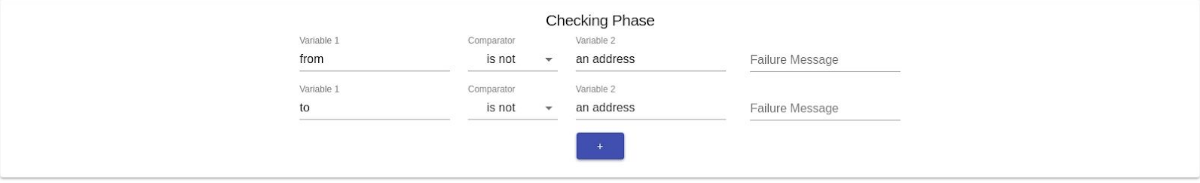
Return: true

Transfer From Function



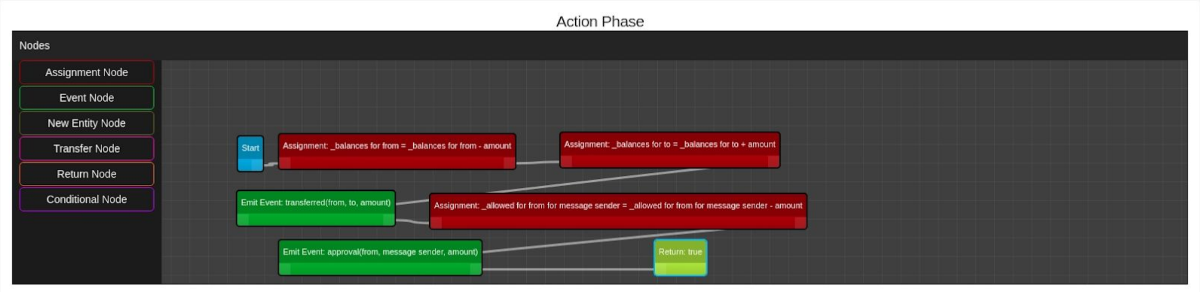
Function Inputs

Variable Name	Variable Type
from	Address
to	Address
amount	Number



Checking Phase

Variable 1	Comparator	Variable 2	
from	is not	an address	Failure Message
to	is not	an address	Failure Message



Action Phase

Start

Assignment: _balances for from = _balances for from - amount

Assignment: _balances for to = _balances for to + amount

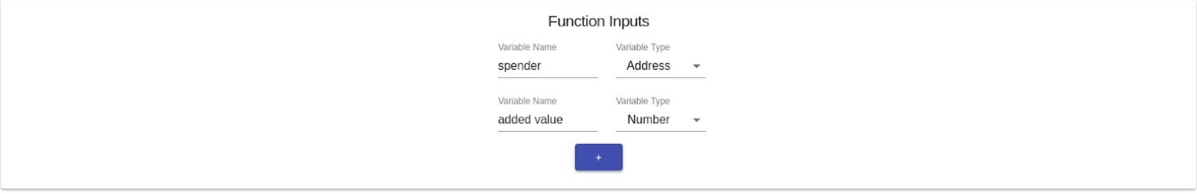
Emit Event: transferred(from, to, amount)

Assignment: _allowed for from for message sender = _allowed for from for message sender - amount

Emit Event: approval(from, message sender, amount)

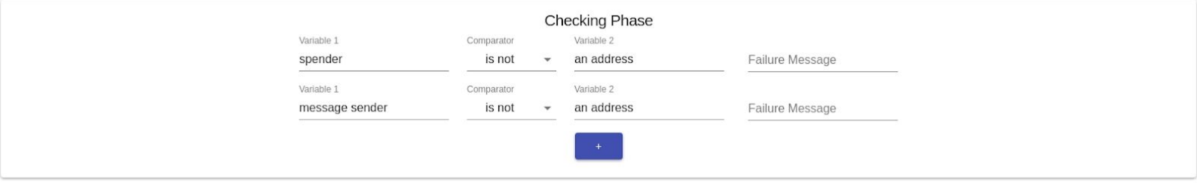
Return: true

Increase Allowance Function



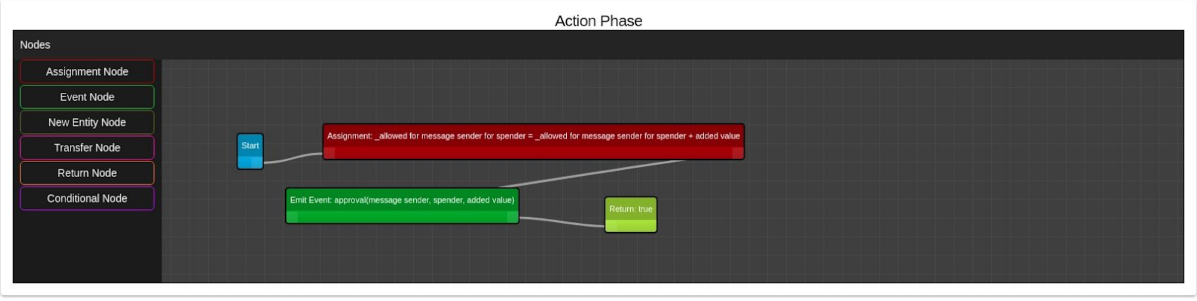
Function Inputs

Variable Name	Variable Type
spender	Address
added value	Number



Checking Phase

Variable 1	Comparator	Variable 2	
spender	is not	an address	Failure Message
message sender	is not	an address	Failure Message



Action Phase

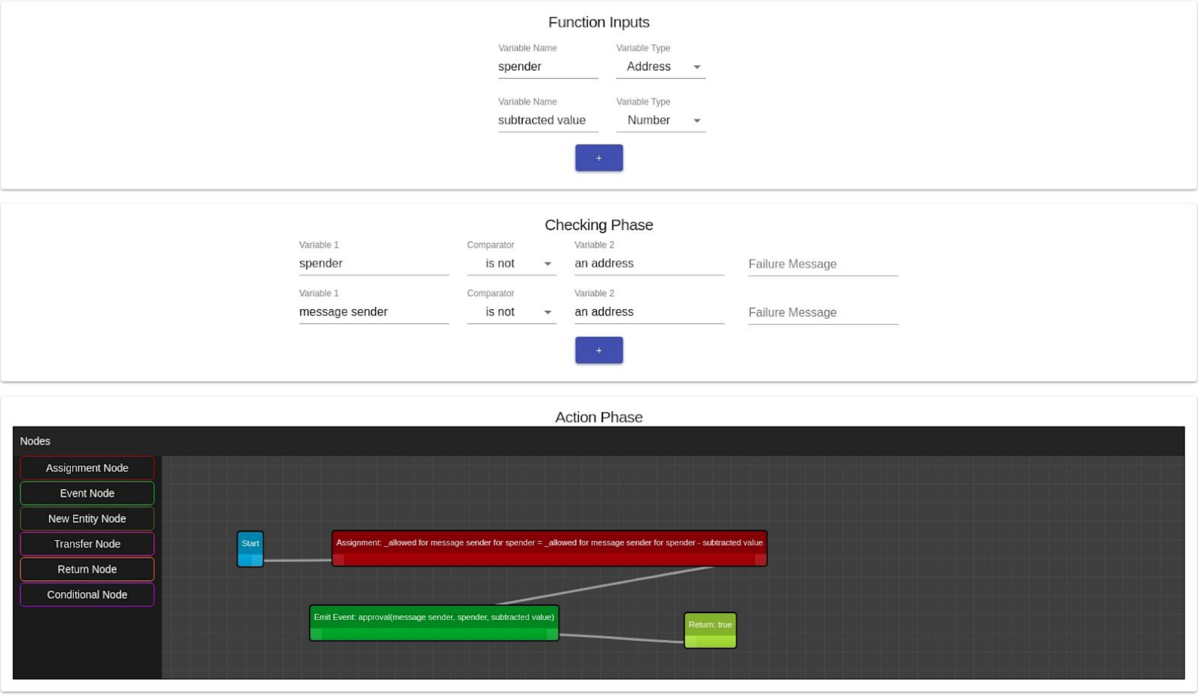
Start

Assignment: _allowed for message sender for spender = _allowed for message sender for spender + added value

Emit Event: approval(message sender, spender, added value)

Return: true

Decrease Allowance Function



Function Inputs

Variable Name	Variable Type
spender	Address
subtracted value	Number

Checking Phase

Variable 1	Comparator	Variable 2	
spender	is not	an address	Failure Message
message sender	is not	an address	Failure Message

Action Phase

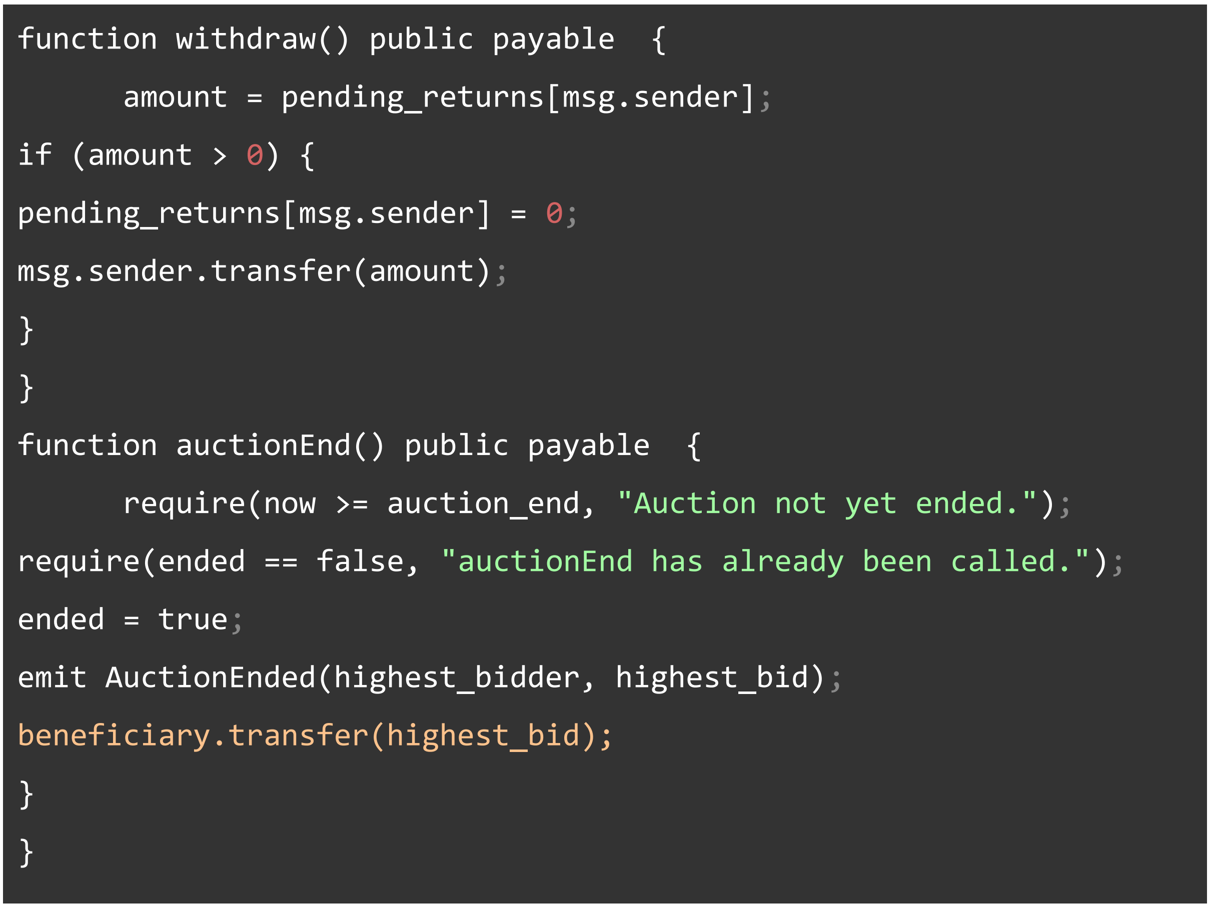
Assignment: _allowed for message sender for spender = _allowed for message sender for spender - subtracted value

Emit Event: approval(message sender, spender, subtracted value)

Return: true

Appendix N: Code Generated by Smart Contract Builder for Open Auction

```
pragma solidity ^0.5.4;
contract Code {
bool public ended;
uint public amount;
mapping(address => uint) pending_returns;
address payable public highest_bidder;
uint public highest_bid;
address payable public beneficiary;
uint public auction_end;
event HighestBidIncreased (address payable bidder, uint amount);
event AuctionEnded (address payable winner, uint amount);
constructor(address payable _beneficiary, uint bidding_time)
public payable  {
      beneficiary = _beneficiary;
auction_end = now + bidding_time;
}
function bid() public payable  {
      require(now <= auction_end, "Auction already ended.");
require(msg.value > highest_bid, "There already is a higher
bid.");
if (highest_bid != 0) {
pending_returns[highest_bidder] = pending_returns[highest_bidder]
+ highest_bid;
}
highest_bidder = msg.sender;
highest_bid = msg.value;
emit HighestBidIncreased(msg.sender, msg.value);
}
```



```
function withdraw() public payable  {
      amount = pending_returns[msg.sender];
if (amount > 0) {
pending_returns[msg.sender] = 0;
msg.sender.transfer(amount);
}
}
function auctionEnd() public payable  {
      require(now >= auction_end, "Auction not yet ended.");
require(ended == false, "auctionEnd has already been called.");
ended = true;
emit AuctionEnded(highest_bidder, highest_bid);
beneficiary.transfer(highest_bid);
}
}
```

Appendix O: Code Generated by Smart Contract Builder for Safe Remote Purchase

```
pragma solidity ^0.5.4;
contract Code {
uint public state;
address payable public buyer;
address payable public seller;
uint public sell_value;
event aborted();
event purchase_confirmed();
event item_received();
constructor() public payable  {
      require(2 * msg.value / 2 == msg.value, "Value has to be
even.");
seller = msg.sender;
sell_value = msg.value / 2;
}
function abort() public payable  {
      require(msg.sender == seller, "Only seller can call this.");
require(state == 0, "Invalid state.");
emit aborted();
state = 2;
seller.transfer(address(this).balance);
}
function confirmPurchase() public payable  {
      require(state == 0, "Invalid state.");
require(msg.value == 2 * sell_value, "");
emit purchase_confirmed();
buyer = msg.sender;
state = 1;
}
function confirmReceived() public payable  {
      require(msg.sender == buyer, "Only buyer can call this.");
require(state == 1, "Invalid state.");
emit item_received();
state = 2;
buyer.transfer(sell_value);
seller.transfer(address(this).balance);
}
}
```

Appendix P: Code Generated by Smart Contract Builder for Voting

```
pragma solidity ^0.5.4;
contract Code {
struct Voter {
uint weight;
bool voted;
address payable delegate;
string vote;
}
mapping(string => uint) vote_count;
address payable public chairperson;
Voter public chairperson_voter;
mapping(address => Voter) voter_details;
Voter public voter;
uint public winning_vote_count;
string public winning_proposal_name;
constructor() public payable  {
     chairperson = msg.sender;
chairperson_voter = Voter(1, false, address(uint160(0)), "");
voter_details[chairperson] = chairperson_voter;
}
function addProposal(string memory proposal) public payable  {
     require(msg.sender == chairperson, "Only chairperson can add a new proposal.");
vote_count[proposal] = 0;
}
function giveRightToVote(address payable target_voter) public payable  {
     require(msg.sender == chairperson, "Only chairperson can give right to vote.");
require(voter_details[target_voter].voted == false, "The voter
already voted.");
require(voter_details[target_voter].weight == 0, "The voter
already has a right to vote.");
voter_details[target_voter].weight = 1;
}
function vote(string memory proposal) public payable  {
     require(voter_details[msg.sender].weight != 0, "Has no right
to vote.");
require(voter_details[msg.sender].voted == false, "Already
voted.");
voter = voter_details[msg.sender];
voter.voted = true;
voter.vote = proposal;
vote_count[proposal] = vote_count[proposal] + voter.weight;
if (vote_count[proposal] > winning_vote_count) {
winning_vote_count = vote_count[proposal];
winning_proposal_name = proposal;
}
}
function winningProposal() public payable returns (string memory)
{
     return winning_proposal_name;
}
}
```

Appendix Q: Code Generated by Smart Contract Builder for ERC20 Token

```
pragma solidity ^0.5.4;
contract Code {
mapping(address => mapping(address => uint)) _allowed;
mapping(address => uint) _balances;
event transferred(address payable from, address payable to, uint amount);
event approval(address payable owner, address payable spender, uint amount);
constructor() public payable  {
     }
function transfer(address payable to, uint amount) public payable returns (bool) {
     require(to != address(uint160(0)), "");
_balances[msg.sender] = _balances[msg.sender] - amount;
_balances[to] = _balances[to] + amount;
emit transferred(msg.sender, to, amount);
return true;
}
function approve(address payable spender, uint amount) public payable returns (bool) {
     require(spender != address(uint160(0)), "");
require(msg.sender != address(uint160(0)), "");
_allowed[msg.sender][spender] = amount;
emit approval(msg.sender, spender, amount);
return true;
}
function transferFrom(address payable from, address payable to, uint amount) public payable returns (bool) {
     require(from != address(uint160(0)), "");
require(to != address(uint160(0)), "");
_balances[from] = _balances[from] - amount;
_balances[to] = _balances[to] + amount;
emit transferred(from, to, amount);
_allowed[from][msg.sender] = _allowed[from][msg.sender] - amount;
emit approval(from, msg.sender, amount);
return true;
}
function increaseAllowance(address payable spender, uint
added_value) public payable returns (bool) {
     require(spender != address(uint160(0)), "");
require(msg.sender != address(uint160(0)), "");
_allowed[msg.sender][spender] = _allowed[msg.sender][spender] +
added_value;
emit approval(msg.sender, spender, added_value);
return true;
}
function decreaseAllowance(address payable spender, uint
subtracted_value) public payable returns (bool) {
     require(spender != address(uint160(0)), "");
require(msg.sender != address(uint160(0)), "");
_allowed[msg.sender][spender] = _allowed[msg.sender][spender] -
subtracted_value;
emit approval(msg.sender, spender, subtracted_value);
return true;
}
}
```

7.2 References

[1] Raval, Siraj (2016). "What Is a Decentralized Application?". Decentralized Applications: Harnessing Bitcoin's Blockchain Technology. O'Reilly Media, Inc. pp. 1–2. ISBN 978-1-4919-2452-5. OCLC 968277125.[Accessed 19 Mar. 2019].

[2] Forbes.com. (2019). A Very Brief History Of Blockchain Technology Everyone Should Read. [online] Available at: https://www.forbes.com/sites/bernardmarr/2018/02/16/a-very-brief-history-of-blockchain-technology-everyone-should-read/#79c719f97bc4 [Accessed 18 Feb. 2019].

[3] Demush, R. (2019). How Companies Can Leverage Private Blockchains. [online] Perfectial. Available at: https://perfectial.com/blog/leveraging-private-blockchains/ [Accessed 19 Mar. 2019].

[4] CoinDesk. (2019). How Do Ethereum Smart Contracts Work? - CoinDesk. [online] Available at: https://www.coindesk.com/information/ethereum-smart-contracts-work [Accessed 19 Mar. 2019].

[5] GitHub. (2019). Ethereum White Paper. [online] Available at: https://github.com/ethereum/wiki/wiki/White-Paper#ethereum [Accessed 19 Mar. 2019].

[6] Solidity.readthedocs.io. (2019). Solidity — Solidity 0.5.3 documentation. [online] Available at: https://solidity.readthedocs.io/en/v0.5.3/ [Accessed 19 Mar. 2019].

[7] Medium. (2019). The Ethereum Virtual Machine — How does it work?. [online] Available at: https://medium.com/mycrypto/the-ethereum-virtual-machine-how-does-it-work-9abac2b7c9e [Accessed 19 Mar. 2019].

[8] Solidity.readthedocs.io. (2019). Contracts — Solidity 0.5.3 documentation. [online] Available at: https://solidity.readthedocs.io/en/v0.5.3/contracts.html [Accessed 18 Feb. 2019].

[9] Blockgeeks.com. (2019). [online] Available at: https://blockgeeks.com/guides/ethereum-gas-step-by-step-guide/ [Accessed 18 Feb. 2019].

[10] Solidity.readthedocs.io. (2019). Contract ABI Specification — Solidity 0.5.3 documentation. [online] Available at: https://solidity.readthedocs.io/en/v0.5.3/abi-spec.html [Accessed 18 Feb. 2019].

[11] Investopedia. (2019). What is ERC-20 and What Does it Mean for Ethereum?. [online] Available at: https://www.investopedia.com/news/what-erc20-and-what-does-it-mean-ethereum/ [Accessed 19 Mar. 2019].

[12] How-To Geek. (2019). What Are Electron Apps, and Why Have They Become So Common?. [online] Available at: https://www.howtogeek.com/330493/what-are-electron-apps-and-why-have-they-become-so-common/ [Accessed 18 Feb. 2019].

[13] Medium. (2019). Advantages of Developing Modern Web apps with React.js. [online] Available at: https://medium.com/@hamzamahmood/advantages-of-developing-modern-web-apps-with-react-js-8504c571db71 [Accessed 18 Feb. 2019].

[14] GitHub. (2019). electron-react-boilerplate/electron-react-boilerplate. [online] Available at: https://github.com/electron-react-boilerplate/electron-react-boilerplate [Accessed 18 Feb. 2019].

[15] Cao, J. (2019). Web design color theory: how to create the right emotions with color in web design. [online] The Next Web. Available at: https://thenextweb.com/dd/2015/04/07/how-to-create-the-right-emotions-with-color-in-web-design/ [Accessed 18 Feb. 2019].

[16] Solidity.readthedocs.io. (2019). Solidity by Example — Solidity 0.5.5 documentation. [online] Available at: https://solidity.readthedocs.io/en/latest/solidity-by-example.html#simple-open-auction [Accessed 1 Mar. 2019].

[17] OpenZeppelin/openzeppelin-solidity. (2019). Retrieved from https://github.com/OpenZeppelin/openzeppelin-solidity/blob/master/contracts/token/ERC20/ERC20.sol

[18] Blockgeeks.com. (2019). [online] Available at: https://blockgeeks.com/guides/vyper-plutus/ [Accessed 18 Feb. 2019].