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 and challenges faced and the steps taken to complete the project.

Acknowledgements

Thank you to A/P Sourav Saha Bhowmick for your 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.

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. [1] 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 around 2013 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.

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.

A smart contract is computer code that verifies and ensure that the terms of a contract are carried out to the satisfaction of all parties. 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 cover understanding how smart contracts work and how to write smart contracts as well. Because this tool is only a prototype, we will only focus on one language, but work can be done in future to extend the implementation to other languages and other blockchains.

1.3 Existing Solutions

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 to create and deploy smart contracts, but these are targeted to experienced developers who are comfortable with the command line. Furthermore, these tools still require the user to write a significant amount of code on their own, as these tools only help with the structure, deployment and testing of smart contract code. 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, but due to the fact that Ethereum and blockchain in general

are still very new, it is likely that most open source effort will go into making blockchain more viable and efficient than to make development accessible to non-technical users.

2. Development and Implementation

2.1 Software And Tools Used

2.1.1 React and Electron

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

Electron is an increasingly popular framework due to its cross-compatibility between platforms, and enables developers to use HTML and JavaScript code to write their user interfaces. Well known examples of other Electron applications include Slack, GitHub and Microsoft Visual Studio Code. [2]

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. [3]

Electron React Boilerplate was used to setup the initial packages for the project. It uses Electron, React, Redux, React Router, Webpack and React Hot Loader for rapid application development (HMR). [4]

2.1.2 Additional packages

In addition, the solc compiler, testRPC and web3 packages are used. Solc, the Solidity compiler, is used to compile Solidity smart contract code into machine readable code. TestRPC 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. TestRPC runs on localhost:8545 by default.

Material-UI, a React library that follows Google's material design, is also used to create a simple, 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, and extensibility of the nodes to enable custom features.

2.1.3 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.

2.1.4 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.

2.1.5 Dependencies

Appendix 1 shows the dependencies in the package.json of the project.

2.2 Project Schedule

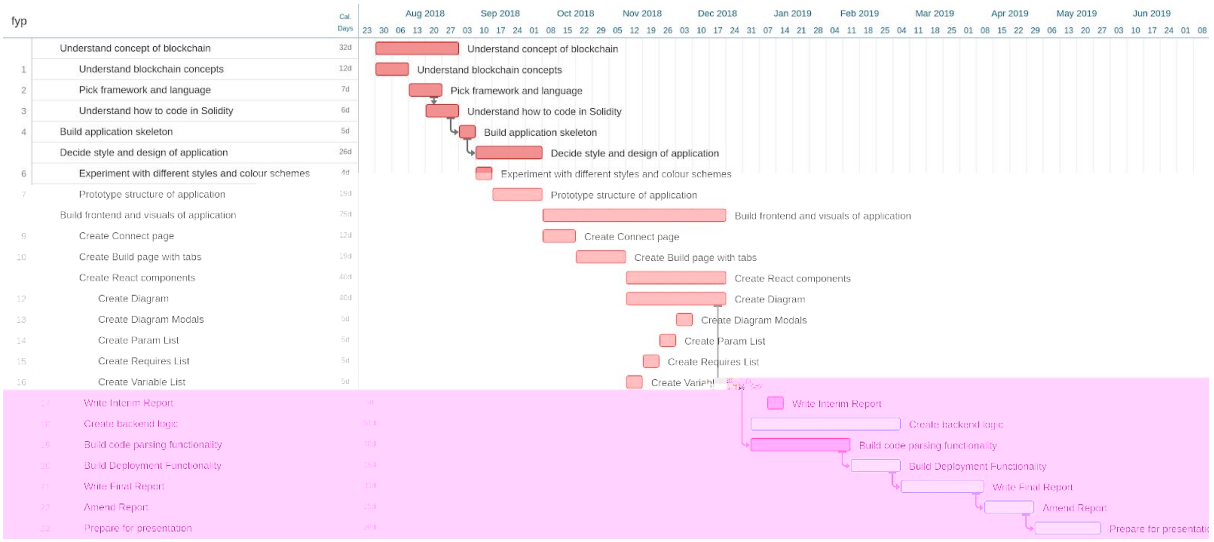


Figure 1.1: Project schedule

The work was planned in this manner as we wanted to rapidly prototype 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 in 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 were able to test the contracts generated on Remix IDE and compare the results.

2.3 Implementation

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

2.3.1 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, and which would be the most appealing and least

intimidating to the user. We settled on a white and blue color scheme as the colour blue is associated with calmness and the colour white is associated with cleanliness and simplicity. [5]

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 that contains the Global State Tab and the Builds tabs will have a back button which returns the user to the connection page, as well as a build button, which generates the solidity smart contract code and deploys it.

2.3.1.1 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 testRPC, the address should be localhost and the port should be 8545, based on default testRPC 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.

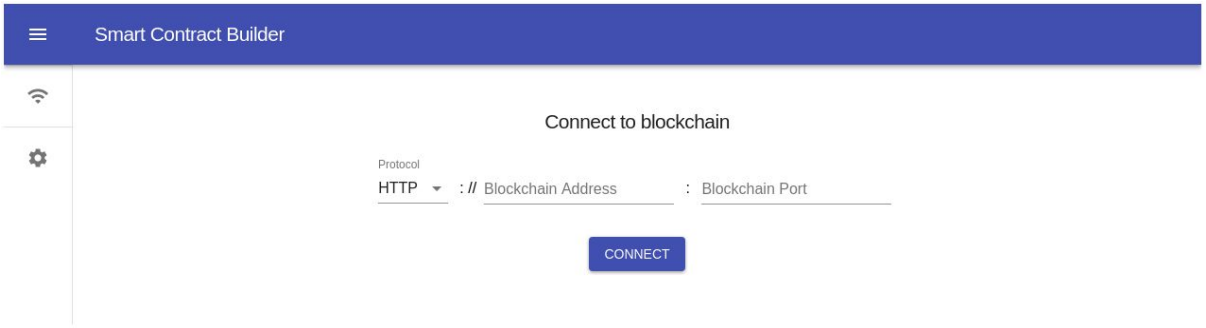


Figure 2.1: Screenshot of Connection Page

2.3.1.2 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.

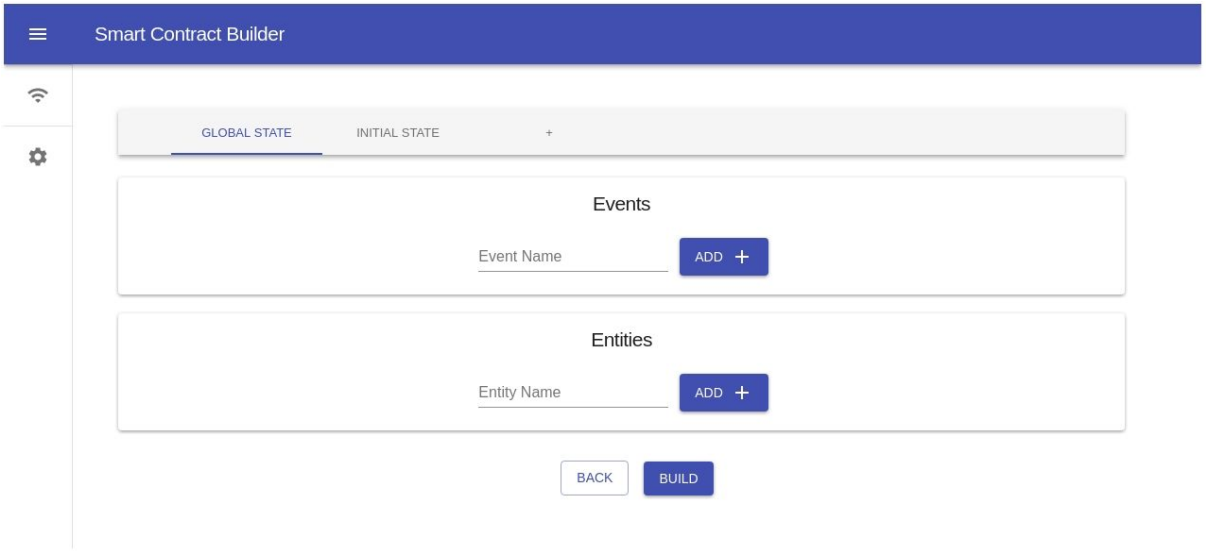


Figure 2.2: Screenshot of Global State Tab

2.3.1.2.1 Events Box

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 you call them, 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. [6]

In the Smart Contract Builder, users add 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. (more on this in Section 2.3.1.3.3)

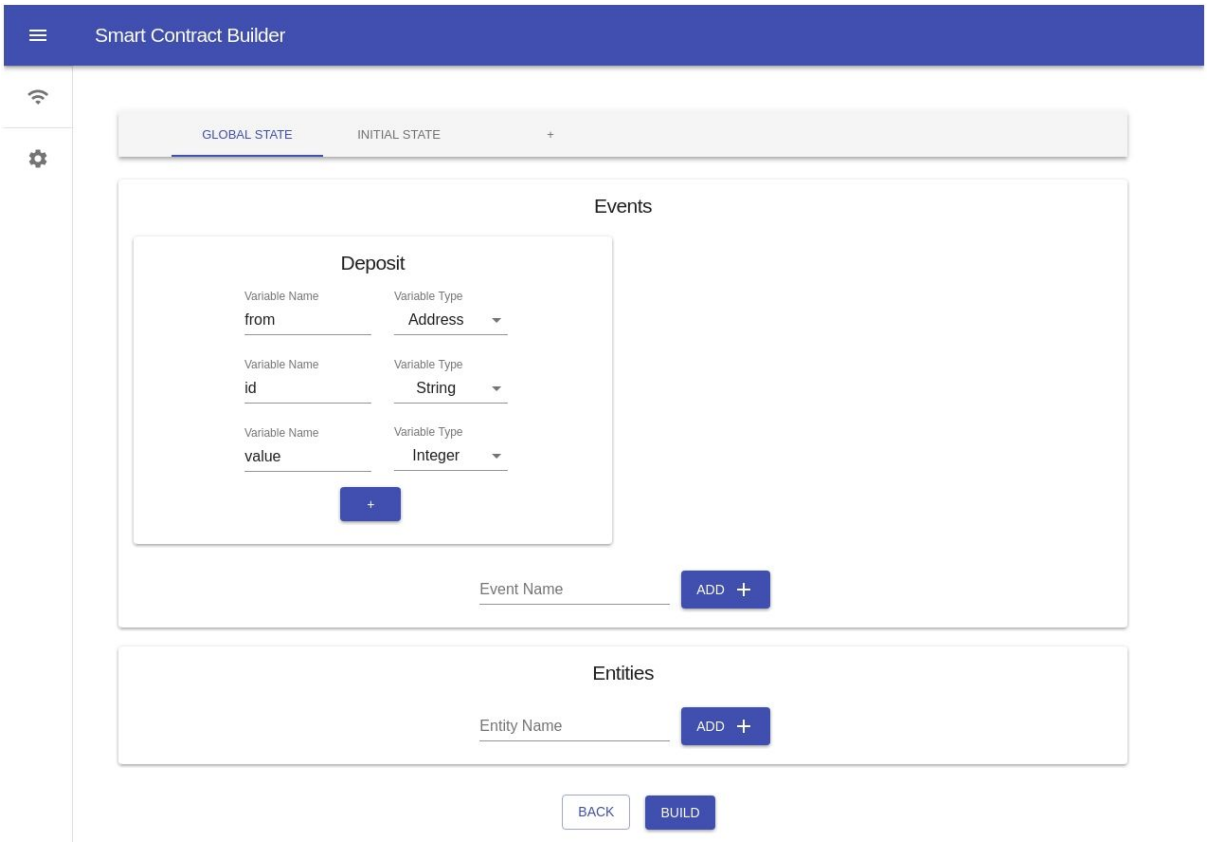


Figure 2.3: Screenshot of an event

2.3.1.2.2 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, 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.

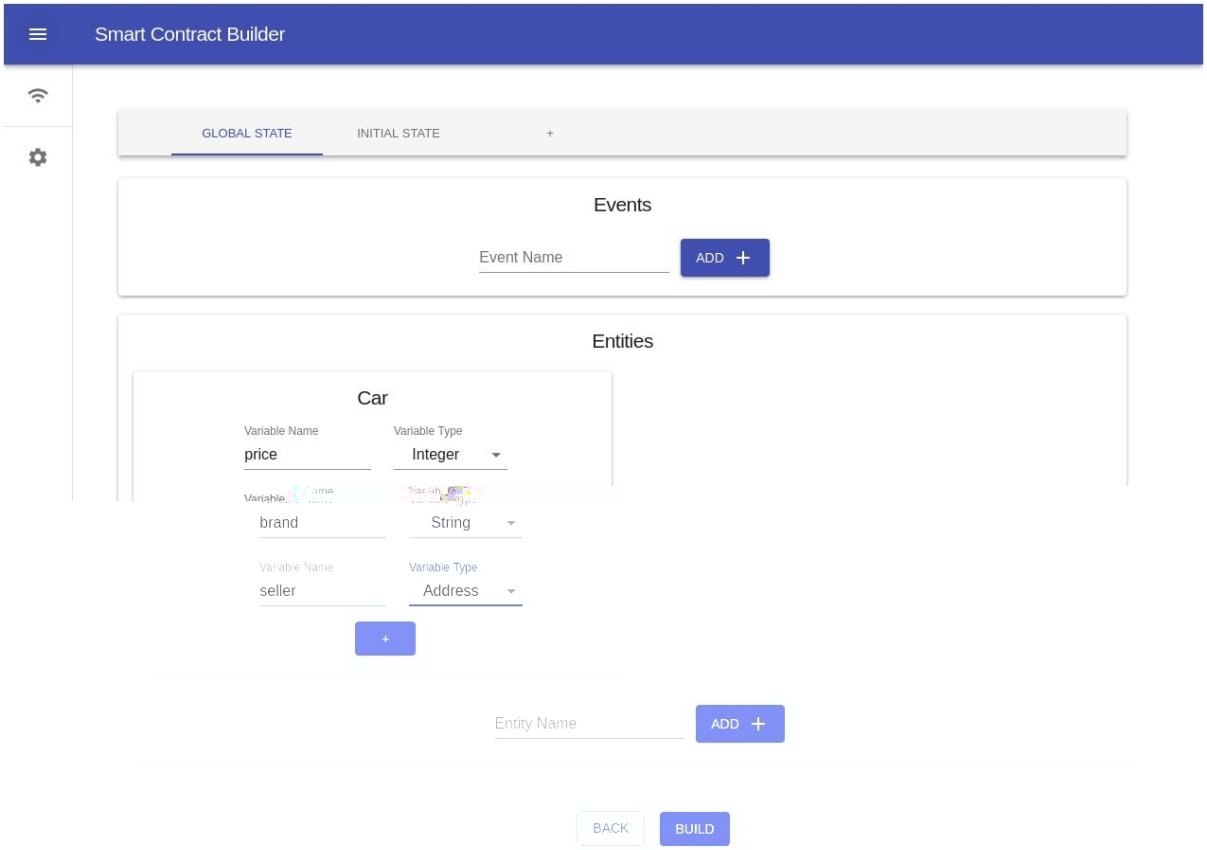


Figure 2.4: Screenshot of an entity

2.3.1.2.3 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 2.4, the Constructor Parameter box will appear as in Figure 2.5:

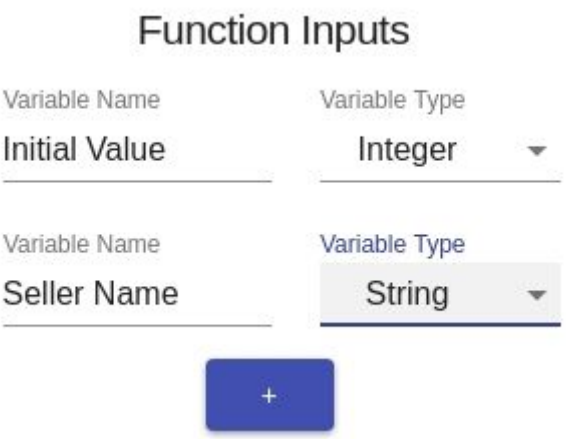


Figure 2.5: Initial State Tab's example inputs

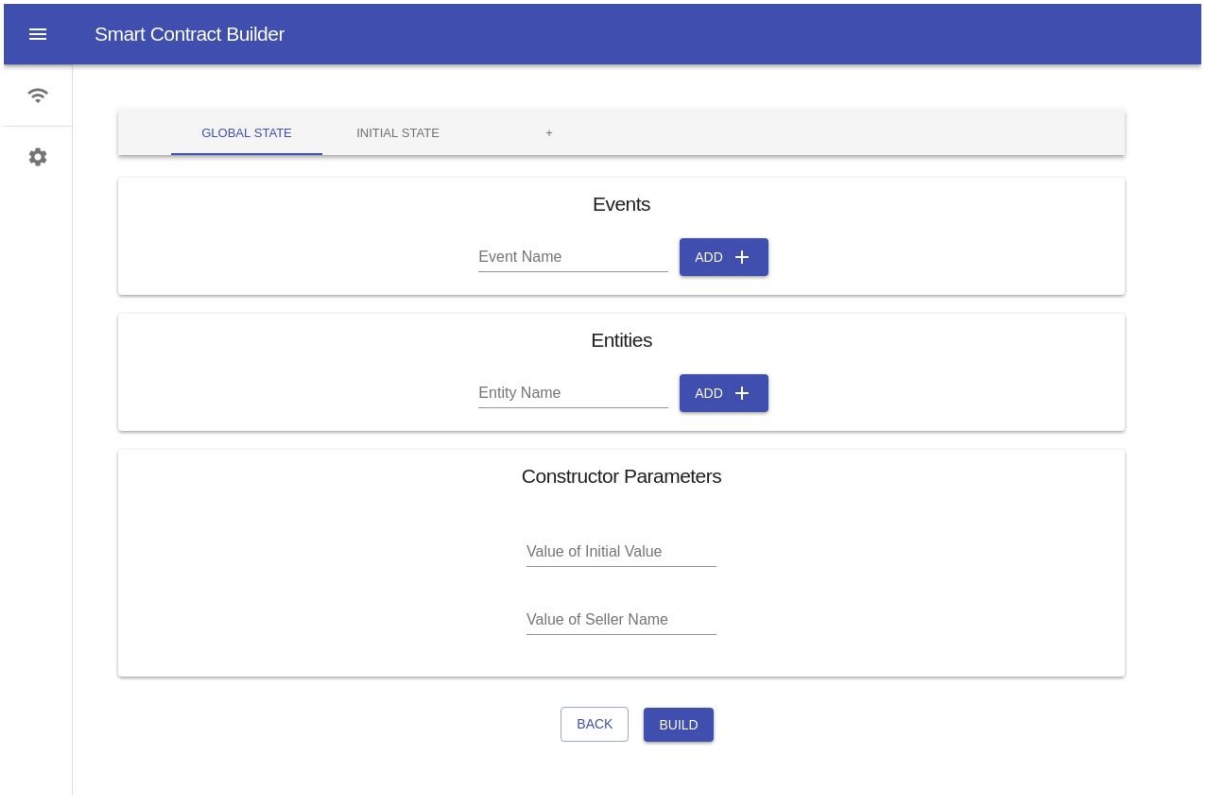


Figure 2.6: Global State Tab's Constructor Parameters Corresponding to Figure 2.4

2.3.1.3 Build Tabs

The Build Tabs represent the functions of the smart contract. Each function corresponds to one 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.

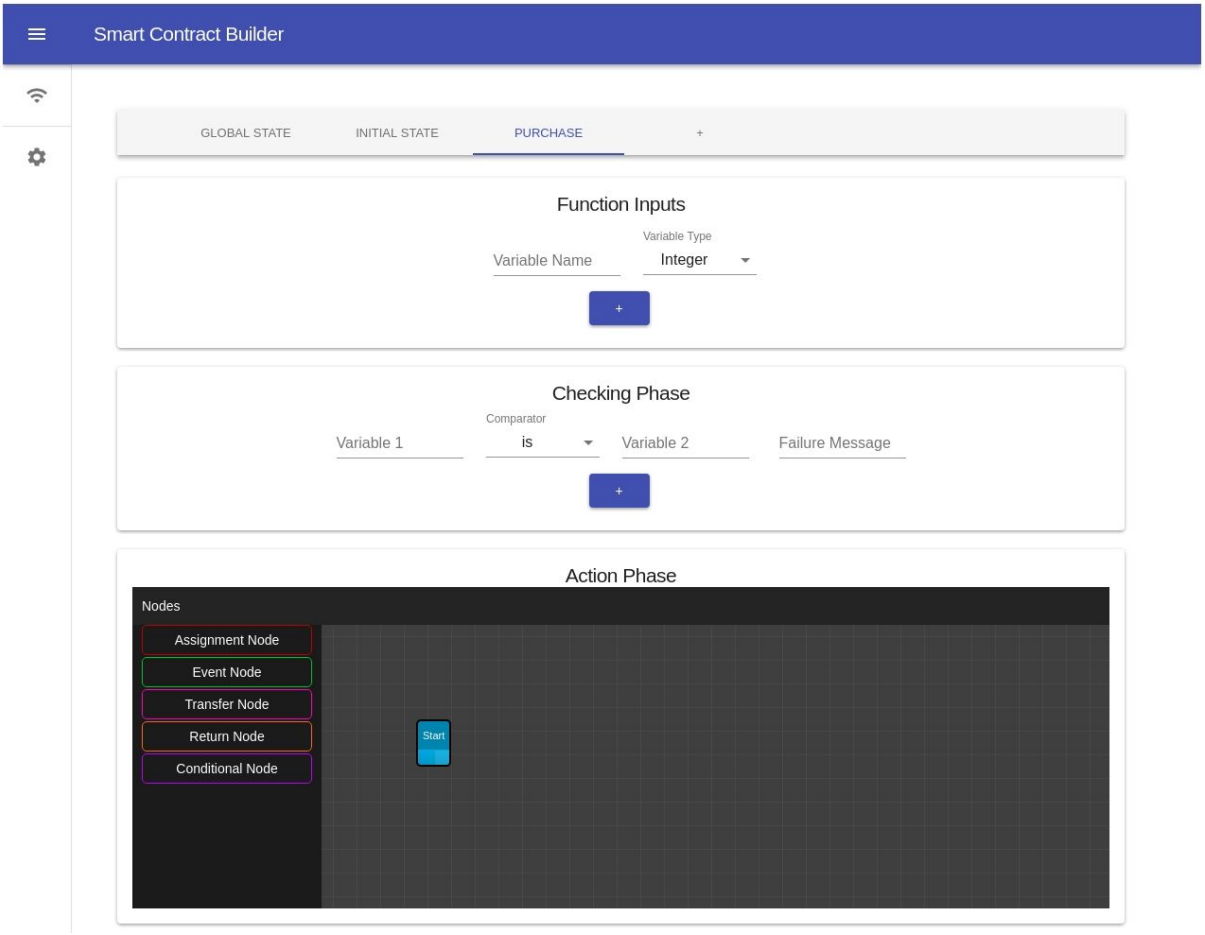


Figure 2.7: Example “Purchase” Build Tab

2.3.1.3.1 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.

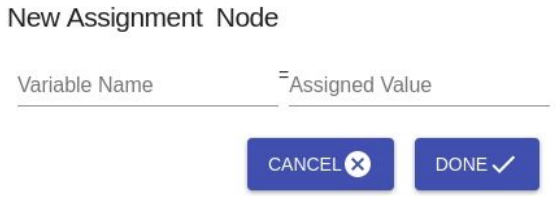
2.3.1.3.2 Checking Phase Box

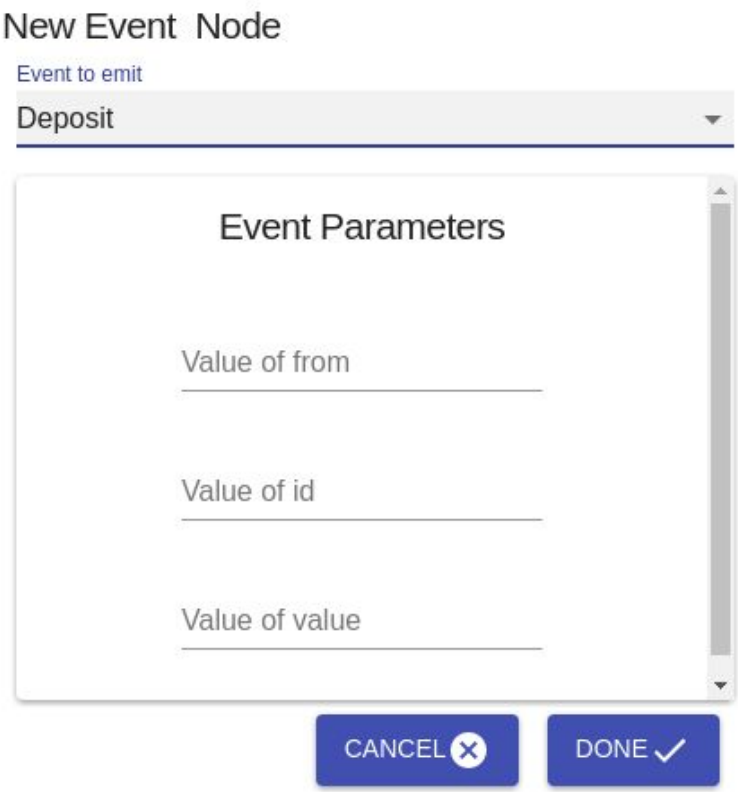
The checking phase ensures the validity of inputs before a function is run. If the user specified conditions are not met, the transaction and blockchain will be reverted to

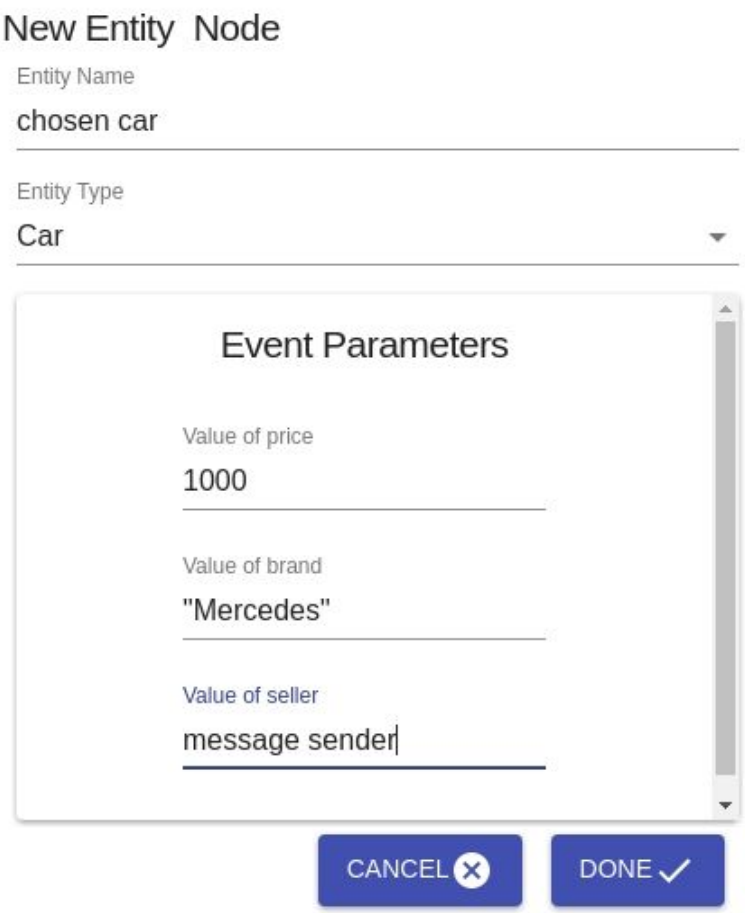
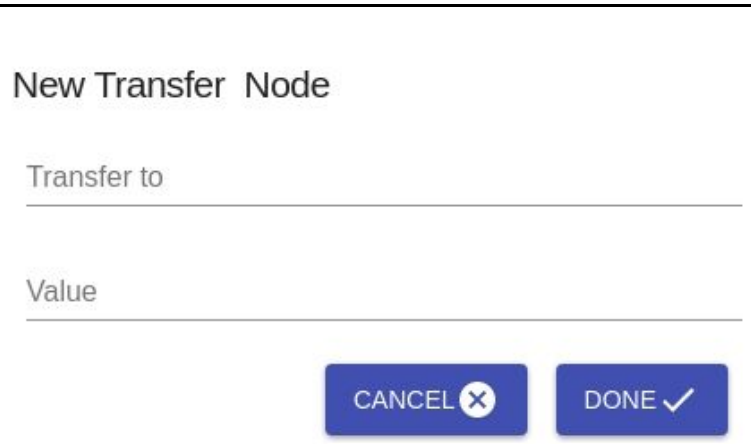
before the function was run. The failure message will be displayed by the smart contract when the condition is not met.

2.3.1.3.3 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	

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

Entity Node	
Transfer Node	

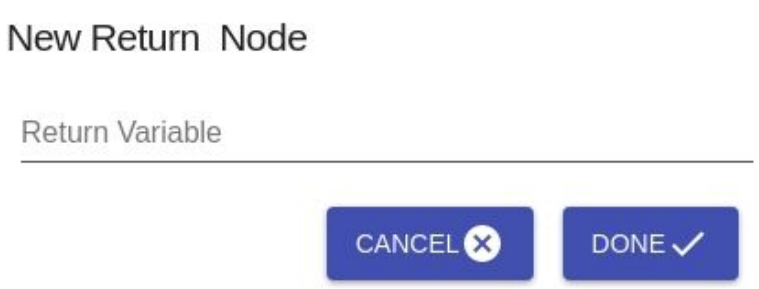
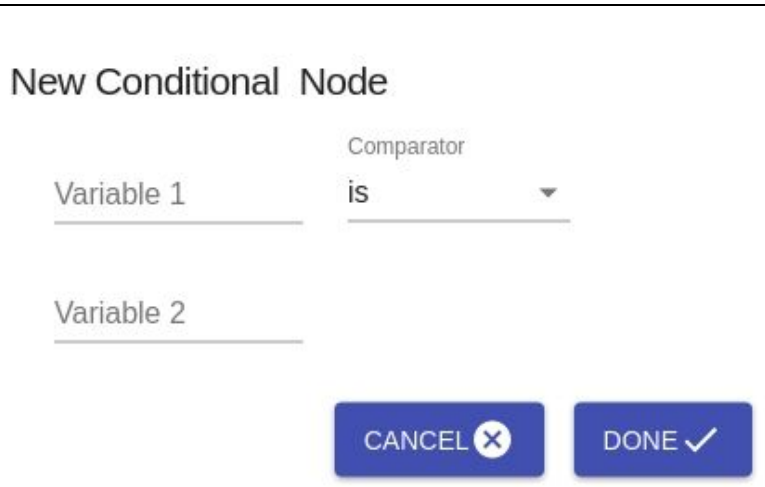
Return Node	
Conditional Node	

Table 2.1: Modal appearance and options for all node types

Note that for the event and new entity type nodes, users will have to fill up different parameters depending on how the event or entity was defined in the Global State Tab. In Table 2.1, the example used follows the event shown in Figure 2.3.
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 2.4). 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.

2.3.2 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 where they left off.

2.3.2.1 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 when a link is created between 2 nodes, the BuildParser's parse method is called. 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 neighbor. 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 checks for typing problems. If the node is a return type node, it was also update the return variable type in the function declaration.
When it encounters a conditional node, in which there are 2 connected neighbours for true and false condition, it will traverse each path separately and fill in the if else conditions. If there is a loop, it will detect a cycle in the diagram and append a while loop to the smart contract code instead of an if statement.

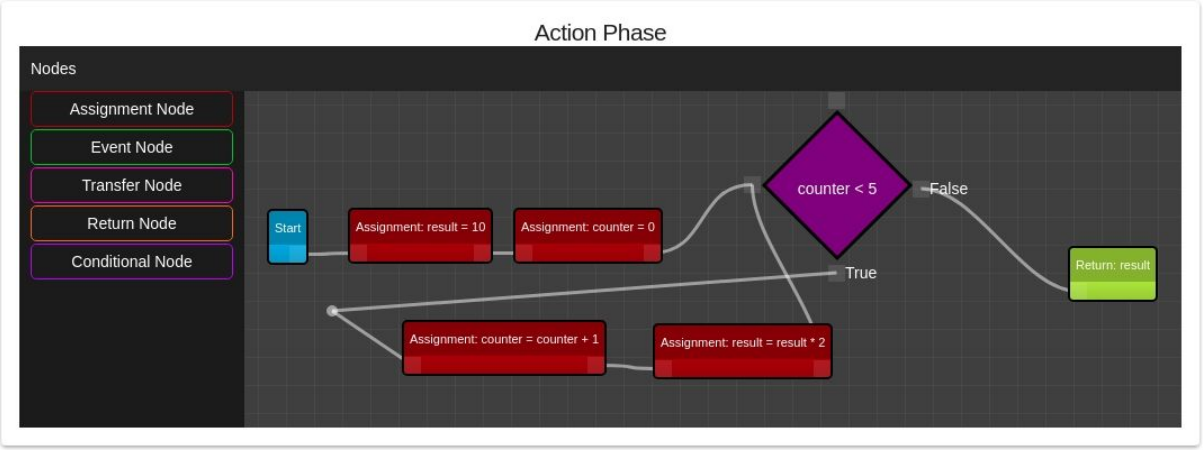


Figure 2.8: Example of while loop detected due to cycle

Appendices 2, 3 and 4 contain code snippets for the BuildParser.
The original 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.

2.3.2.2 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 of the smart contract will be shown. This transaction hash can be used to check the status of the transaction on the blockchain.

2.3.2.2.1 Code Generation

Each solidity function can be broken down into segments, which correspond to each piece of information input by the user. Appendix 2 illustrates a 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. This allows us to formulate a general formula for getting a smart contract.

2.3.2.2.2 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). 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. [7] After compilation, web3 will be used to deploy the contract onto the blockchain.

2.3.2.2.3 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 readFile and writeFile functions in the filesystem module (fs). When the save button is clicked, 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 in the root of the application directory. When the load button is clicked, 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.
The data is currently stored in a single file called data.json. This means that there can only be one save at a time, and any saving will override the previous saved

state. This is because this project is a proof of concept and this single file saving mechanism is sufficient for our use case. However, future modifications can be made to allow multiple saves. For example, Electron can connect with MongoDB and store the data as documents, which allows the user to choose what to label the data when saving, and which data to load.

3. Evaluation of Results

3.1 Methodology of evaluation

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. 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. [8]
We obtained an example of a Solidity smart contract from the official Solidity documentation [9] and modeled it using the Smart Contract Builder to achieve the same functionality. We then ran both the original and generated smart contract 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 both contracts, the default Javascript VM environment was used.

3.2 Results

The results are as follows:

	Open_auction.sol 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 3.1: Comparison of performance with reference contract

The code for open_auction.sol and the code generated using the smart contract builder can be found in appendix 3 and 4 respectively. Figures 3.1 and 3.2 show the Build Tabs for the constructor and bid function respectively.

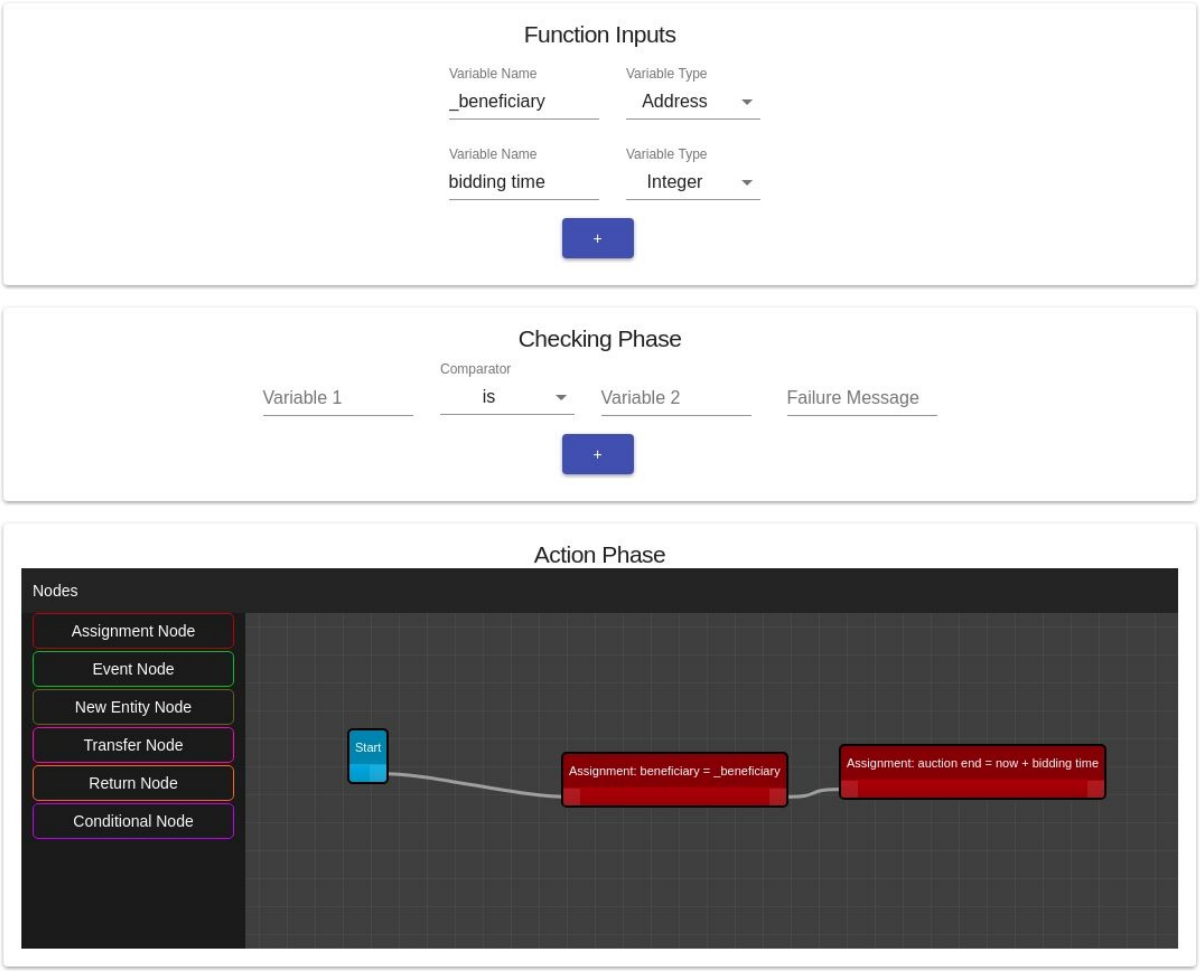


Figure 3.1: Constructor Function

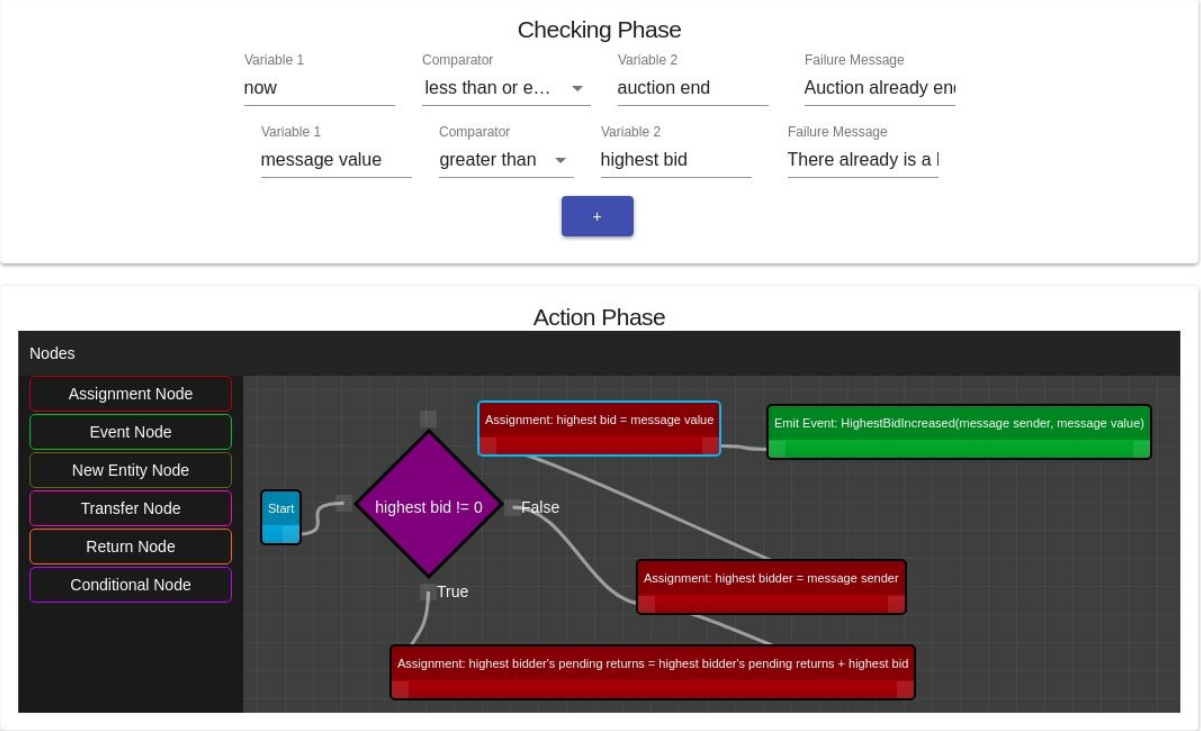


Figure 3.2: Bid Function

Unfortunately, Remix IDE's static analysis tool is unable to obtain the gas usage for the withdraw and auction ended functions, and so it returns an infinite value for both the reference contract and the generated contract.

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.

From table 3.1, we can see that for the deposit cost is 5.7% worse for the generated contract, the execution cost of the constructor function is 5.17% worse for the generated contract, and the execution cost of the bid function is 0.53% worse. The performance does not deviate significantly from the reference contract, but this is likely because it is a relatively small, simple contract. For a larger contract, the 5%

difference is likely to be far more significant. However, since the code is algorithmically generated, the deviation is acceptable and decent enough for most use cases.

4. 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 focus was heavily 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 in terms of performance, the Smart Contract Builder is able to generate a decent performing smart contract, but it still underperforms compared to actual human-written code, especially the cost for constructing and deploying the contract, as seen in Table 3.1. This test was only run on a fairly simple smart contract from the documentations, so we are likely to see even greater performance deviations for a highly complex smart contract. More work will likely 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 to use in a production environment.

However, with that said, the purpose of the Smart Contract Builder is not for 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.

5. Recommendations

There is still much room for improvement in terms of usability, such as abstracting the code layer from the user to a greater extent, such as 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 Solidity introduces new updates. 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. [10] Future work should be done to ensure that the application keeps up with developments. Compatibility with other smart contract languages like Vyper can be introduced to allow greatly flexibility and future proofing, and compatibility with permissioned Blockchain networks like Hyperledger Fabric can also be introduced so that a larger corporate audience can be reached.

6. End Section

6.1 Appendices

```
"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 1: package.json dependencies

```
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 2: Structure of a Solidity smart contract

```
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 3: open_auction.sol from Solidity documentation

```
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 4: Code generated by Smart Contract Builder

6.2 References

[1] 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].

[2] 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].

[3] 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].

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

[5] 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].

[6] 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].
[7] 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].
[8] Blockgeeks.com. (2019). [online] Available at: https://blockgeeks.com/guides/ethereum-gas-step-by-step-guide/ [Accessed 18 Feb. 2019].
[9] 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].
[10] Blockgeeks.com. (2019). [online] Available at: https://blockgeeks.com/guides/vyper-plutus/ [Accessed 18 Feb. 2019].