LATTE: Visual Construction of Smart Contracts

Sean Tan[‡] Sourav S Bhowmick[‡] Huey Eng Chua[‡] Xiaokui Xiao[§]

[‡]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[§]School of Computing, National University of Singapore, Singapore
assourav|hechua@ntu.edu.sg,xkxiao@nus.edu.sg

ABSTRACT

Smart contracts enable developers to run instructions on blockchains (*e.g.,* Ethereum) and have broad range of real-world applications. Solidity is the most popular high-level smart contract programming language on Ethereum. Coding in such language, however, demands a user to be proficient in contract programming and debugging to construct smart contracts correctly. In practice, such expectation makes it harder for non-programmers to take advantage of smart contracts. In this demonstration, we present a novel *visual smart contract construction system* on Ethereum called LATTE to make smart contract development accessible to non-programmers. Specifically, it allows a user to construct a contract without writing Solidity code by manipulating visual objects in a *direct manipulation-based* interface. Furthermore, LATTE interactively guides users and makes them aware of the cost (in units of *Gas*) of visual actions undertaken by them during contract construction.

ACM Reference Format:
Sean Tan, Sourav S Bhowmick, Huey Eng Chua, Xiaokui Xiao. 2020. LATTE: Visual Construction of Smart Contracts. In 2020 International Conference on Management of Data (SIGMOD '20), June 14–June 19, 2020, Portland, OR, USA. ACM, NewYork, NY, USA, 4 pages. https://doi.org/10.1145/3318464.3384687

1 INTRODUCTION

With the increasing commercial and research interests on blockchains, several blockchain-based technologies have emerged in recent times such as *smart contracts*. Specifically, a *smart contract* (*contract* for brevity) enables developers to run instructions on blockchains. A contract can encode any set of rules using a programming language. Accordingly, it can carry out and conditionally transfer digital assets or tokens between parties in a transparent, conflict-free way while avoiding the services of a middleman. Ethereum is a popular example of a public, blockchain-based decentralized platform that provides a low-level bytecode language called Ethereum virtual machine (EVM) to execute smart contracts. Each Ethereum contract holds some virtual coins (Ether), has its own private storage, and is associated with a predefined executable code. Recently, Ethereum's smart contract framework has seen steady adoption supporting thousands of contracts [2]. The most popular smart contract programming language on Ethereum is Solidity (a Javascript-like language), which is compiled into EVM code. Some examples of smart contracts written in Solidity are given in [7].

Solidity has stringent coding requirements for contract programming. Consequently, it demands a user to have proficiency in contract programming to construct smart contracts correctly. In practice, such expectation makes it challenging for a non-programmer to design a usable smart contract. Many blockchain platforms are geared toward expert users and lack the support needed for easier use by non-experts.

In this demonstration, we present a novel tool called LATTE (VisuaL SmArt ConTracT BuildEr) to visually construct a smart contract in Ethereum using a *direct-manipulation* interface [5]. LATTE facilitates formulation of simple smart contracts by manipulating visual objects instead of writing Solidity code. Specifically, it automatically generates the Solidity code from a visually-constructed contract. Furthermore, the unique characteristics of the (Ethereum) blockchain strongly influence the way a smart contract has to be developed compared to common developmental approaches. Accordingly, LATTE is *Gas-aware* and guides users on the Gas usage of a contract during its construction. Intuitively, each instruction executed by EVM has a predefined cost, which is expressed in units of *Gas*. Hence, Gas can be considered as the cost to run smart contracts on Ethereum. Note that this feature is paramount in smart contract development as execution of contracts cost real money to users.

In our demonstration, we shall highlight various innovative features of LATTE and show how it can be used to visually construct a Solidity contract. We believe that LATTE

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06…$15.00
https://doi.org/10.1145/3318464.3384687

will make smart contract development accessible to non-programmers. Nevertheless, it can also be used by contract programmers to develop complex contracts by first generating the Solidity code of the core features of a contract using LATTE and then augmenting it by coding complex features (*e.g.,* bit operations, hashing). Note that LATTE is designed to make contract construction easier. Issues related to security and testing of contracts [2] are orthogonal to this tool.

2 DESIGN PHILOSOPHY

The design of LATTE is based on the following key principles.

Construct core features without coding. LATTE should enable users to effortlessly create a single smart contract without writing Solidity code. To this end, it should allow visual construction of following Solidity features: basic variable types, structs (known as *entities* in LATTE), functions, variable assignments, mapping variable types and nested mappings, conditional statements and loops, events, require statements, and transfer functions. Note that these are core features of Solidity and can be used to create a variety of contracts. Bit operations, assembly functions, hashing, encoding and decoding are not supported as these are designed for programmers. Also, modifiers (to improve readability) and self destruct are not supported as they are not required to create a functioning smart contract. Note that these advanced features can be added by programmers by first transforming a visually formulated contract to Solidity code in LATTE and then augmenting it with relevant codes. Lastly, since we aim to create a single contract instead of multiple modular contracts, the support of libraries is orthogonal to our goal.

Towards Gas-aware formulation. Since execution of a contract in Solidity costs real money, LATTE should support *Gas-aware* interactions. First, it should provide fine-grained variable storage choices (*e.g.,* declaring a variable as 8 or 256 bits have substantial impact on Gas usage). Second, it should explicitly warn users on the implication of assigning a variable to *Storage* instead of *Memory*. The former is written permanently to a blockchain and incurs significant Gas compared to the latter, which will disappear when a function call ends. Hence, users need to carefully consider the usage of storage or memory keyword when declaring variables. Third, a unique feature of Solidity is that it differentiates between functions that write to a blockchain and those that only perform read tasks. A function can be declared as read-only by adding a view keyword. A view function can optimize the Gas usage since it does not incur any Gas when it is invoked. Hence, LATTE should detect and assign a function as view if it does not write to a blockchain. Finally, users are able to view the total Gas usage by a contract at any time during formulation so that relevant modifications can be performed (if necessary).

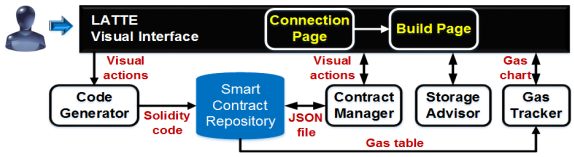

Figure 1: Functional architecture of LATTE.

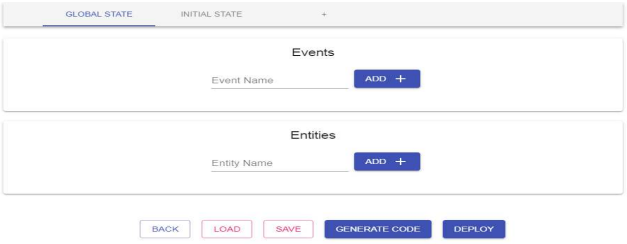

Figure 2: Global state tab.

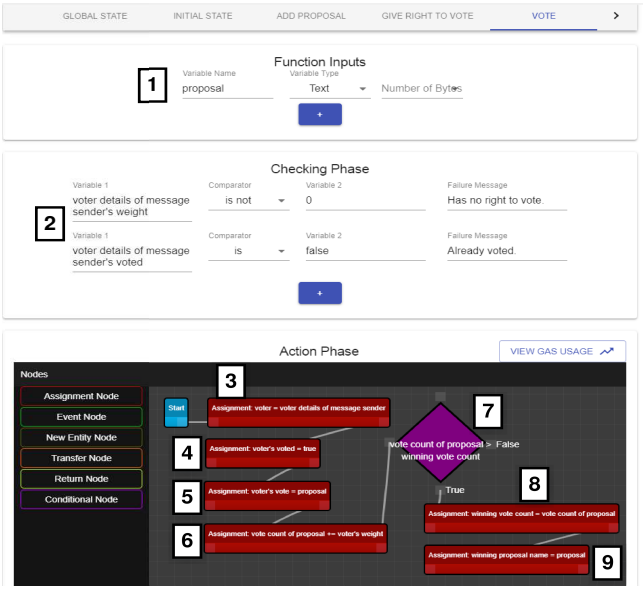

Figure 3: Build tab.

3 SYSTEM OVERVIEW

Figure 1 depicts the architecture of LATTE. It consists of the following components.

The Visual Interface Module. The visual interface of LATTE consists of two key pages, namely *connection* and *build*. The *connection* page is the landing page when a user invokes LATTE. Once a user has connected to a valid blockchain through the *connection* page, she will be transferred to the *build* page. The *build* page consists of two components, the *global state tab* (GST) and the *initial state tab* (IST).

The *global state tab* enables formulation of details of a smart contract that are not tied to any functions such as events and entities. Figure 2 shows a screenshot of the GST component. The *Events box* is used to declare events by specifying the event names. One can add and customise an event's

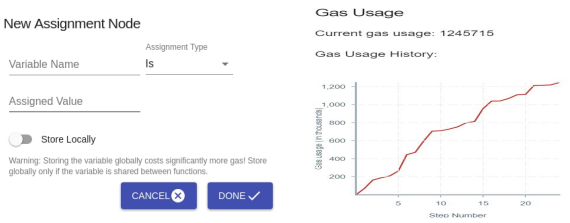

Figure 4: Storage Advisor and Gas Tracker modules.

parameters. The *Entities box* enables formulation of struct data structure by simply declaring the attributes and types.

The IST enables one to visually construct the functions of a smart contract. Each function is associated with a tab (referred to as *build tab*) having name corresponding to the function name. Each *build tab* (*i.e.,* function) consists of the *function input*, the *checking phase*, and the *action phase* to visually specify details related to a Solidity function. Figure 3 depicts an example of the *build tab*.

The *function input* allows one to specify function parameters for each function as well as their types. The *checking phase* allows a user to specify necessary conditions that need to be satisfied before a function is executed. If these conditions are not met, a user-specified failure message is returned. The *action phase* allows a user to specify the content of a function by drawing an *action graph* in the *Action panel* (bottom right panel). The panel has a default start node to indicate start of the function body. A user may choose a *type* of node she wants to add to an action graph from the left panel and drag and drop it on the *Action panel*. The node types supported by LATTE and their correspondence to Solidity code are given in Table 1. A modal box will appear that takes specific input from the user before adding the node to an action graph. Each node has ports. The *incoming port* is on the left and the *outgoing port* is on the right. One needs to connect an outgoing port of a node to an incoming port of another node. This enables us to specify a sequence of statements in a function body. Note that a Conditional Node has 4 ports to capture an if/else statement. The input ports are at the left and top and the output ports are at the right and bottom. Figure 3 shows an action graph. Note that LATTE supports fine-grained choices of variable types and all variable fields are free text to allow a user to create variables using natural language. A *variable mapping* is maintained to map a free text field to an internal variable representation. The *Code Generator Module* ensures the correct translation of user-defined variables to Solidity code by using the mapping.

The Contract Manager Module. This module allows a user to save the current progress of smart contract formulation at any time and load it again later. When the Save button is clicked, the entire state of the *Build* page is stringified and dumped into a JSON file. When the Load button is clicked, the state of the *Build* page is reverted to the saved content.

Table 1: Node types in an action graph.

Name	Function in Solidity	Color
Assignment Node	Assignment of value to a variable.	Red
Event Node	Emit a previously declared event.	Green
Entity Node	Create an instance of a previously declared struct and assign it to a variable.	Dark green
Transfer Node	Transfer an amount of ether to an address.	Pink
Return Node	Complete the function and return the value provided.	Orange
Conditional Node	Branching functionality with true and false conditions.	Purple

This module is also responsible for deploying a smart contract to the blockchain. When a user clicks on the Deploy button, it compiles the smart contract code and leverages web3 to deploy it onto the blockchain. If the deployment is successful, the transaction hash and address of the smart contract will be shown.

The Code Generator Module. This module generates the Solidity code of the visual formulation actions undertaken by a user. It comprises of two components, *BuildParser* and *CodeBuilder*.

The *BuildParser* submodule is responsible for parsing the action graphs to generate the code of the functions in a smart contract. It parses an action graph using two loops. The first loop looks for variables, infers their types, and adds their names and types to a *lookup table*. In particular, it checks the Assignment, Entity and Transfer Nodes as they provide information about variable types. In the second loop, it traverses the action graph from the start node. At each node, it parses all variables involved in it and generates the code by utilizing the lookup table. If the node is a Return Node, it updates the return variable type in the function declaration. If it encounters a Conditional Node, it will traverse each path associated with it separately and fill in the `if/else` conditions. If there is a loop (cycle in the action graph), it appends a `while` loop to the code. Note that the *BuildParser* also identifies certain natural language keywords in a user's input and convert them to Solidity code. For example, "message sender", "msg sender", "sender", "function caller" keywords are converted to `msg.sender` in Solidity code (*i.e.,* address of the person that called the function). Similarly, "message value", "msg value", "value" keywords are converted to `msg.value` (*i.e.,* incoming amount of ether sent to the function).

The *CodeBuilder* submodule is responsible for generating the complete code. It is invoked when a user clicks on the Generate Code button. LATTE leverages a *contract code template* and places the variables, structs, events, parameters and the code formed by the *BuildParser* into it to generate a syntactically correct Solidity smart contract. Note that LATTE automatically detects view functions by analyzing the code. Lastly, all state variables are transformed to private instead of public so that sensitive information is not unintentionally revealed to the blockchain users. To allow public access, one needs to create a function to expose a state variable.

The Storage Advisor Module. This module advises users on storage of variables that may lead to reduction in Gas usage. It allows a user to define how many bytes or bits will be used to store a variable. Also, when initialising a new Entity Node or using the Assignment Node, it prompts a user to decide if she wishes to store a variable locally or globally. If she chooses the latter, it warns her that the Gas usage is significantly higher. Figure 4 (left) depicts an example.

The Gas Tracker Module. This module tracks the potential Gas usage of a contract during its visual formulation. It is invoked by clicking the View Gas Usage tab in the *Action panel*, and displays a plot of the total Gas used to visual action increase during contract formulation (Figure 4 (right)). A *Gas table*, which contains the amount of Gas that are required in order to execute several basic computation operations (*e.g.,* addition) on the EVM, is utilized to this end. The current implementation of LATTE utilizes the estimated values provided by web3 via the estimateGas function.

4 RELATED SYSTEMS

There are visual programming editors such as *Google Blocky* [1] to facilitate interactive programming. However, these editors are not designed specifically for smart contracts. Weingärtner *et al.* [6] develop a mapping process to transform the graphical representation of *Blocky* to Solidity. Mao *et al.* [4] automatically generate the basic functions of specialized templates for contract coding using machine learning and provides a visual page editor based on *Google Blocky* [1] to help users write smart contracts. None of these tools are Gas-aware. The DB community has shown increasing interest in blockchain technologies [3] but to the best of our knowledge there is no effort towards visual construction of contracts.

5 DEMONSTRATION OVERVIEW

LATTE is implemented using the *Electron* and *React* frameworks. In addition, the *solc* compiler, *Ganache CLI* (to simulate full client blockchain behaviour) and web3 packages are used. Our demonstration will be loaded with a few real smart contracts (*e.g.,* [7]). Users can also visually formulate their own ad-hoc contracts through our GUI.

The key objective of the demonstration is to enable the audience to interactively formulate a contract without writing Solidity code. An audience may formulate a preloaded or ad-hoc contract using the visual interface of LATTE. She can also experience first-hand how LATTE guides users toward Gas-aware formulation of contracts through the *Storage Advisor* and *Gas Tracker* modules. We illustrate this experience by visually creating a *voting* contract using LATTE.

Voting is a smart contract written to ensure transparency and fairness of an election. It contains a Voter struct to capture voter objects. A *chairperson* will deploy the smart contract and is in charge of overseeing the voting process. The chairperson is assigned in the constructor of the contract (*i.e.,* constructor()), which is invoked when she deploys the smart contract onto a blockchain node. She can add proposals to the contract and give others the right to vote by invoking the addProposal and giveRightToVote functions, respectively. The vote function can be called by anyone with the right to vote. The delegate function allows a voter to give her vote to someone else to vote in place of her. The winningProposal (a view function) function can be invoked by anyone to find out the winning proposal. A **video** of the formulation of this contract in LATTE is available at https://youtu.be/2aD37KTM80Q.

The GST in Figure 2 is used to declare a Voter entity. Specifically, four attributes are formulated: (a) `weight`: weight of the vote of the voter; (b) `voted`: whether the voter has already voted; (c) `delegate`: who the voter delegated her vote to; (d) `vote`: the name of the proposal the voter voted for. Next, the IST (Figure 3) is invoked to create the constructor and other functions of the smart contract. Due to space constraints, we describe the formulation of the vote function.

A user has to provide the proposal name she is voting for when she invokes the vote function. Hence, (1) a proposal is added in *function input*. (2) In the *checking phase*, specify that the voter has the right to vote and has not voted yet. Next, in the *action phase*, (3) the voter details of the message sender is assigned to a voter variable. (4) Set the `voted` attribute to True. (5) Assign the `vote` attribute to the proposal she is voting for. (6) Update the `voteCount` by adding the vote `weight` to the proposal she is voting for. (7) Drag a Conditional Node to the *Action panel* to check if the vote count for this proposal is the highest. (8-9) If it is, then update winning vote count and declare this proposal as the current winning proposal. Figure 3 depicts these steps. Observe that Steps 3-6, 8, and 9 are realized using the Assignment Node. LATTE allows a user to set a new variable as local or global when it is configured. Also, a user can track Gas usage of the formulated contract any time by clicking on the View Gas Usage tab in the *Action panel*.

Once the construction is completed, clicking on the Generate Code button generates the Solidity code. The audience may then compare the generated code with the original Solidity code to appreciate the benefits of using our tool.

REFERENCES

[1] I. Culic, et al. Auto-generating Google Blockly visual programming elements for peripheral hardware. *In RoEduNet Int. Conf.-Netw. Educ. Res.*, 2015.
[2] L. Luu, D.-H. Chu, et al. Making Smart Contracts Smarter. In *ACM CCS*, 2016.
[3] S. Maiyya, V. Zakhary, et al. Database and Distributed Computing Foundations of Blockchains. *In SIGMOD*, 2019.
[4] D. Mao, F. Wang, Y. Wang, Z. Hao. Visual and User-Defined Smart Contract Designing System Based on Automatic Coding. *IEEE Access*, 7: 73131-73143, 2019.
[5] B. Shneiderman, C. Plaisant. Desigining the User Interface: Strategies for Effective Human-Computer Interaction. 5th Ed., Addison-Wesley, 2010.
[6] T. Weingärtner, et al. Smart Contracts Using Blockly: Representing a Purchase Agreement Using a Graphical Programming Language. *In CVCBT*, 2018.
[7] Solidity by Example - Solidity 0.5.5 documentation. Available at: https://solidity.readthedocs.io/en/latest/solidity-by-example.html#simple-open-auction.