Strengthening Smart Contracts to Handle Unexpected Situations

Shuze Liu*, Farhad Mohsin†, Lirong Xia‡ and Oshani Seneviratne§
Department of Computer Science, Rensselaer Polytechnic Institute
Troy, NY 12180, USA
Email: *lius17@rpi.edu, †mohsif@rpi.edu, ‡xial@cs.rpi.edu, §senevo@rpi.edu

***Abstract*—Decentralized application users may face unexpected situations that the smart contract implementing the application should handle, but cannot, because the smart contract cannot be modified once it is deployed. Therefore, we need 'stronger' smart contracts with flexible structures that are resilient in such unexpected situations.**

In this paper, we propose a generic mechanism to strengthen smart contracts and handle possible unexpected situations. Given a smart contract, this mechanism automatically generates an action list which offers actions as interfaces to change parameters of smart contracts and a voting system that utilizes a limited voter group randomly chosen from the peers. Each action in the action list can change a corresponding parameter of smart contracts. The actions, when approved by the majority, are executed to change the parameters.

When users face unexpected situations in a transaction, they choose some actions as the solution and pass them to the voting system. Since a smart contract has finite parameters, there are finite actions. By arranging and combining these actions, our mechanism offers solutions that can handle wide-ranging unexpected situations. Also, to execute a solution, the majority of voters need to approve it, thus not violating the protocol of the original smart contract.

Voters are rewarded based on quadratic rules for peer prediction, which makes telling true preferences the only way to maximize rewards. Using machine learning, we predict users' preferences based on the voting records. The predictions are provided as default values for future votes to avoid users' need to vote manually each time.

***Index Terms*—Blockchain Theory, Decentralized Applications, Smart Contracts, Unexpected Situations, Computational Social Choice, Machine Learning**

I. Introduction

Decentralized applications (DApps), unlike centralized applications, do not require any administrator and are trustworthy due to the immutable ledger behind the scenes that is recording all the transactions originating from the DApp. Thus, many companies venturing their business operations to blockchain systems are deciding to use DApps in place of their traditional systems. For instance, Walmart uses a decentralized system to track the originals of their products [1]. DApps use smart contracts to ensure each transaction is executed properly. Smart contracts are self-executing contracts in which terms of contracts between two parties are directly written into lines of code.

However, once the decentralized applications are deployed to the network, they are hard to modify, since making every peer in the network agree on the modification is almost impossible. A smart contract's response at an unexpected situation is dependent on the designers foresight, and not every possible situation can be covered. Users may face unexpected situations when they are using decentralized applications. For example, in a decentralized medical record sharing application, doctors may need the signature of their patient to access the patient records. However, if a patient is unconscious and cannot give the signature and the authorized doctor is also unreachable, a new doctor may not be able to access the record. If this situation has not been written in smart contracts, it is hard for the doctor to access the record because no administrator can help the doctor pass this requirement. Therefore, decentralization requires us to design stronger smart contracts with flexible structures.

We propose a mechanism to handle existing smart contracts and strengthen them with flexible structures while keeping their original functions unchanged. Our mechanism has three main parts and has been implemented in Hyperledger Composer which is a decentralized application platform developed by the Linux Foundation.

We suggest creating smart contracts equipped with the ability to host a voting mechanism which allow users to vote for a solution to resolve unexpected situations. We target resolving unexpected situations that can be resolved by directly changing relevant parameter values in the smart contract. In our design, we have a preprocessor that takes a smart contracts as input, analyzes the smart contract and adds an action for each parameter in the smart contract. These actions form an action list which provides interfaces for users. Because we have finite parameters for each smart contracts, this action list is finite. However, by arranging them in proper orders, users can deal with wide-ranging unexpected situations. The preprocessor eliminates the need to write these actions manually and provides a comprehensive action list. It also adds in a voting mechanism to vote on a solution of an unexpected situation.

Facing an unexpected situation, users can start a poll through this mechanism along with an action list that will resolve the situation. Peers vote on approving the proposed action list, and based on the peer votes our mechanism may apply the solution to resolve the situation. The preprocessor is not a necessity, because these functions and voting mechanisms may be added in by a developer as well. However, this greatly increases the efficiency and correctness of creating strengthened smart contracts. Finally, we incorporate a learn-

978-1-7281-1264-0/19/$31.00 ©2019 IEEE
DOI 10.1109/DAPPCON.2019.00034

ing mechanism in the smart contract, to learn the preferences of each user to predict how they would vote given a new unexpected situation. For our solution, learning preferences for certain types of situations is made possible. So if a particular unexpected situation comes up many times, the smart contract will eventually learn how the users vote on that topic and can predict their preferences, augmenting the smart contract.

II. Related Work

While there has been work done to strengthen Smart Contracts in various ways, to our knowledge not much work has been focused on unexpected situations. Work has been done to check vulnerabilities of exceptions and ways to work around these vulnerabilities [2]. But we rather focus on increasing flexibility than removing vulnerabilities. There has also been talk of building machine learning systems on top of generic APIs like MedRec to create a health learning system [3], but that is more geared towards learning about health features of users whereas our solution is to have users vote to resolve unexpected situations and in turn learn how the users vote to improve the Smart Contract itself. So, the Smart Contract needs to have some decision making ability. There has been work done on autonomously learning and decision making based on human preference inputs that can be applicable to fields like self-driving cars [4]. Since the innate mechanism of learning from pairwise comparisons to simulate group decision is similar, we have tried to rethink our problem in that way.

III. Motivating Scenario

We implemented a university course selection system in Hyperledger Composer to illustrate our mechanism of strengthening smart contracts. This system is a decentralized application for students to select courses at their university. Students with more earned credits have the priority to select first. Since there is no administrator in this system, we do not need to pay money to administrators and the process is fair. Money and fairness may not have a great importance in course selection, but they could be significant for the election of president.

Suppose developers of this system only implemented adding courses and dropping courses for students. These two transactions seem to satisfy the needs of students. However, an unexpected situation may occur when a course is filled and no new student can be added to the course. In real life, students can get special permissions from professors and the registrar can add them to the course. In this decentralized version, there is no registrar. Clearly, this adding should be done by the smart contract, but it cannot do so because there are no provisions for handling this unexpected situation. Even if this simple case could have been foreseen by the developer, that may not be the case for more complex smart contracts. So, we create a way out for problems that could be solved by changing specific parameters of relevant assets and participants. However, such changes should not be made just because they seem to resolve a particular unexpected situation. We propose a consensus based system to solve this.

A. Drawing Inspiration from the Real World

To get motivation for consensus based real-time change to a system, we look at the system of passing new bills in the US constitution. A simplified explanation of the method is illustrated in Figure 1: Congressional committees and subcommittees are formed that are knowledgeable about a topic, and bills proposed for that topic first goes to the subcommittee. If the subcommittee deems the bill worthy then it goes in front of the rest of the Congress for voting. If we try to just replicate this solution on the blockchain, we see some difficulties. If we consider all users as equivalent as Congress, who do we choose as subcommittee members? How large should a subcommittee be? And if the number of unexpected situations is very high, does the voting process become a nuisance rather than an augmentation?

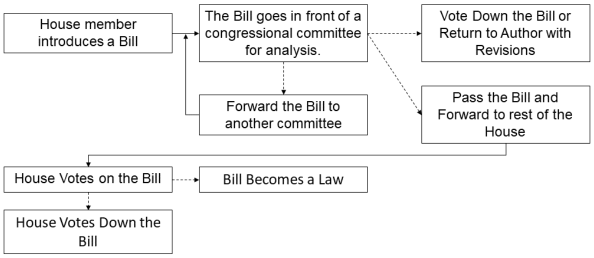
House member introduces a Bill
The Bill goes in front of a congressional committee for analysis.
Vote Down the Bill or Return to Author with Revisions
Forward the Bill to another committee
Pass the Bill and Forward to rest of the House
House Votes on the Bill
Bill Becomes a Law
House Votes Down the Bill

Fig. 1. How congressional bills are passed - simplified

B. Our Proposal

We propose a simple solution of choosing voting groups randomly. For each situation, we select a random group of voters uniformly from the users, rotating the random choices in each round, so that users are not inundated with requests to resolve situations. This is still a preliminary idea, and the randomization may be designed in different ways, e.g. those with higher stake in the system or higher relevance with the situation may have higher probability of being chosen. To emulate the back and forth a proposal may have between revisions, we propose to have multiple rounds of voting. While this system of making a smart contract changeable using a voting mechanism may seem to make it vulnerable, the mechanism is designed in a way such that majority consensus is still mandatory, making it difficult to manipulate the system to make malicious changes. We discuss the voting mechanism in detail in the two following sections.

So in our course selection system, a student facing the mentioned unexpected situation will be handled in this way: a voting poll is created with **targets** and **actions**. In this scenario, *targets* are the course (asset) that this student wants to enroll in and the student themselves (participant) since we also need to change the corresponding parameter. **Actions** are proposed by this student as a solution to the unexpected situation. In this example, the actions will be changing registered students list of the course. However, without this mechanism,

```
Model
   asset A{
      a1
      a2
   }
   participant B{
      b1
      b2

   }

Script{
  transaction t1{
   //...
   }









}
```

```
Model
   asset A{
      a1
      a2
   }
   participant B{
      b1
      b2
      learning_data
   }

Script{
    transaction t1{
    //...
    }
//new transactions
    start_vote{}
    submit_vote{}
    end_vote{}
//action list
    change_a1{}
    change_a2{}
    change_b1{}
    change_b2{}

}
```

Fig. 2. An example of a strengthened smart contract. The LHS shows the assets, participants, and the methods of the generic smart contract, and the RHS shows the strengthened smart contract with additional functions that make the smart contract more robust.

this transaction will fail when the smart contract finds the capacity is full.

In the following sections, we explain our methodology that includes: 1) The preprocessor that examines the smart contract and proposes the action list and the voting mechanism; 2) The voting process; 3) The design and method of learning voter preferences.

IV. Preprocessor

The preprocessor is used to add an action list and a voting system to a given smart contract. Figure 2 is an example of a generic smart contract. The left part is the input to the preprocessor. After being strengthened using the preprocessor which we will discuss later in this section, we get a strong and flexible smart contract with new transactions used for voting and an action list used for changing parameters.

A. Background

Using Hyperledger Composer as the developing environment, smart contracts are composed of two parts: model and script. Model is composed of assets, participants, and transactions. Participants are users. Assets are goods that transactions will modify. Transactions are the only functions which can be directly called by users. We choose the code style of Hyperledger Composer because its good organization of assets and participants lends itself quite well to help illustrate our idea.

B. Parsing

Given an input smart contract, this step deletes annotation and parses the smart contract. A smart contract is composed of Model and Script in Hyperledger Composer. When parsing the given smart contract, this step extracts assets and participants which are two essential elements. It will then extract the parameters of each asset and participant and store their names and types. When generating the action list, the name will be part of the action name which offers a generic API as well as an easy way to change a specific parameter. Types are required because each type of parameter needs to have a different implementation.

C. Action List

We find that an unexpected situation which smart contracts should do but actually cannot do can be solved by changing the parameters of smart contracts. For instance, by directly changing a parameter which is the registered students of a course, a student is successfully added in a filled course. A more generic view is that decentralized applications are running by changing parameters. If users can get direct access to these parameters with proper authorization, they can indeed solve many unexpected situations without administrators.

Thus, this preprocessor uses the result provided by parsing generating an action for each parameter in a smart contract. These **actions** are different from **transactions**. Actions cannot be directly called by users, and transactions are the only functions which can be directly called by users. Actions can only be passed into the voting system as a solution.

The reason that we need this an action list is to provide an interface for users. Without this interface, users can only read the parameters of each smart contracts, but there is no way for users to change them. This step also eliminates the need to write these actions manually and offers a generic API for users. In Figure 2, the preprocessor creates change_Name() for each parameter automatically.

D. Voting System

This step creates three **transactions**: *start_vote, submit_vote, end_vote* as shown in Figure 2. These transactions can be directly called by users. When users are facing an unexpected situation, they start a voting poll with targets and a solution. Others will vote on it. After collecting votes, this poll is ended and smart contracts will execute the solution or not based on voting results.

V. Voting

A. Start Voting

When a user is facing an unexpected situation, this user can create a voting poll with targets and actions. Targets are assets or participants which this user wants to change. Actions are chosen from the action list created by the preprocessor as a solution to this unexpected situation. They also input a target value for the parameter they choose.

```
start_vote()
{
   newPollName: p1
   solution : change_registeredStudents
   targetValue: [...,Alice]
   targetCourse: math
   targetParticipant: Alice
}
```

Fig. 3. An example of starting a poll to add a student to a filled course.

For a concrete example, as shown in Figure 3, A student Alice wants to add her to a filled math course since she gets a special permission. Thus, she can start a new poll with ID ”p1” and choose an action *change_registeredStudents* as the solution. Also, she inputs the target value to add her into the list of registered students. Others will vote on this poll using the poll name.

B. Submit Votes

Peers can submit votes to voting polls created by users. They submit “Agree Value” and “Disagree Value”, which will be normalized to 1. For example, Bob can agree on Alice’s proposal *p1* for 8 and disagree for 2 and the normalized agree value would be 0.8. Peers would be rewarded for participating in a poll, based on their agree and disagree values. Our system gives rewards using quadratic rules of peer prediction, which has been proven to be stable and incentivize honest feedback [5]. This has also been used in cases for incentivizing peer grading to get honest grading [6]. By incentivizing to report true intentions, we confirm that the learning algorithm does not learn flawed or malicious intentions.

In order to ensure high turnout of voters, besides the incentives, we also introduce delegation for the voters. Voters may choose to delegate their votes to others in advance. Delegates get partial rewards from these votes and also enjoy a high probability of being chosen to vote.

C. End Voting

After a round of voting, the poll is finished. Smart contracts will execute the solution in the poll based on voting results. Using the agree value that we have for each user as probabilities which they will vote agree over the proposed action list, we can aggregate agree values for all the users and calculate a joint probability for majority of the voting group agreeing on the proposal. If this probability is high, the actions can be implemented. Afterwards, this step gives rewards to voters and their delegates and updates the learning parameters.

Figure 4 shows the voting and exception handling mechanisms in detail. Users who face an unexpected situation may combine actions from the action list to form a solution and start a voting process. The proposal will go through the voting process, and based on the result of the vote, the actions may be implemented. The learning mechanism which is shown in the figure will be triggered in case of typical unexpected situations, with the purpose of learning the preferences. For example, if students need to get into a filled class, the solution is always to add them to the class, but the smart contract does not know if the voters will vote in favor of that. With the learning mechanism present, it can predict a probability of the users voting for a particular student to be added to the class or not.

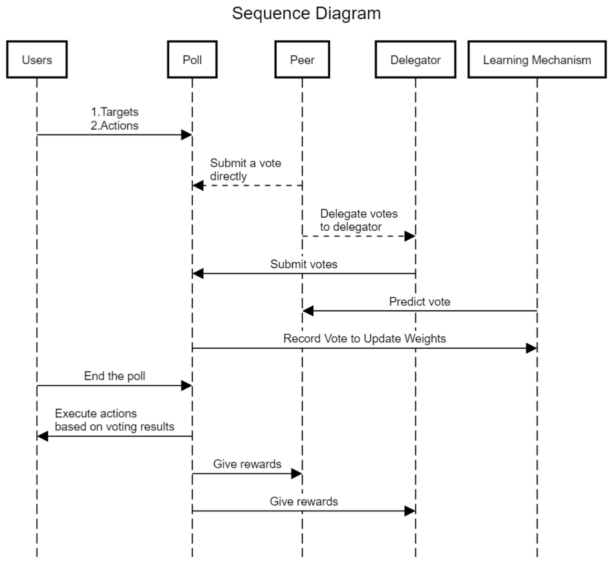

Fig. 4. Sequential flow for complete process

VI. Learning of Preferences

A. Motivation for Learning

As per our mechanism, we would continually need to keep taking the votes, every time an unexpected situation comes up. But if a situation is repetitive, it would make more sense if it could be programmed into the smart contract itself. So, we learn voting preferences of each of the users. At each round of voting, we have the solution and hence the current and changed state of parameters of relevant assets and participants. We assume that for most transactions, the way a user votes shall uniquely be determined by a small number of relevant parameters. We understand that a DApp may be able to implement different types of transactions, affecting different groups of assets and participants. We propose learning the preference of each user for different transactions separately. This learning mechanism brings a level of automation in handling unexpected situations and decreases user disruption. Moreover, if the preference model of a user is learned with high enough confidence level, preferences about a new situation can be generated for all users, not only the subset of voters for a certain round. This gives a better probability of the decision being representative of the user-base.

B. Learning Mechanism

We make use of the Thurstone-Moestler (TM) Model [7], [8] to define the learning process. If we know the relative utilities of alternatives, this model gives us the probability of choosing one alternative over the other. That probability is what we determine from our proposed voting mechanism, and after learning is done, that will be what we predict. The voting mechanism asks each user to give a weight on both agreeing and disagreeing with a proposal. The normalized weight of agreeing is taken to be the probability of the user voting for acceptance of the proposal. This probability will largely depend on how the user sees the utilities for accepting and rejecting the proposals, and with the TM model, we model these utilities in the following way.

$$U_A \sim \mathcal{N}(\mu_A, \frac{1}{2})$$

Where μ_A is the mode utility for some state of the parameters, $\mathbf{x}_A$. That is, if the relevant assets in the smart contract have value $\mathbf{x}_A$, then this mode utility shall be a function of $\mathbf{x}_A.\mu_A$ will differ for the same $\mathbf{x}_A$ for different users but we assume that it is linearly dependent on $\mathbf{x}_A$. We may assume, for user i, $\mu^i_A = \beta^{(i)T}\mathbf{x}_A$. And this $\beta^{(i)T}$ shall completely define user i's preferences over these parameters. Now, faced with a proposal, if the existing and proposed parameter values are $\mathbf{x}_A$ and $\mathbf{x}_B$ respectively, we may say for user i,

$$\begin{aligned} p_i = \mathbb{P}(\mathrm{A} \succ \mathrm{B}) &= \mathbb{P}(U_A > U_B) \\ &= \mathbb{P}(U_A - U_B > 0) \end{aligned}$$

Now, as U_A and U_B are Gaussian distributions, $U_A - U_B$ is also a Gaussian distribution, particularly $U_A - U_B \sim \mathcal{N}(\mu_A - \mu_B, 1)$. Thus

$$\begin{aligned} p_i &= \Phi(\frac{\mu_A - \mu_B}{1}) \\ &= \Phi(\beta^{(i)T}(\mathbf{x}_A - \mathbf{x}_B)) \\ \implies \beta^{(i)T}(\mathbf{x}_A - \mathbf{x}_B) &= \Phi^{-1}(p_i) \end{aligned}$$

Here, p is the normalized approval-weight that we get for user i. Φ indicates the CDF of the standard Gaussian distribution. Since for each round of vote that a particular user participates, we get more input data $(\mathbf{x}_A, \mathbf{x}_B, p_i)$ for that user, and from here we attempt to learn β. And we will see from simulated results in section V that with the assumption of rational user behavior, estimated $\hat{\beta}$ should give accurate predictions with high probability. From the synthetic data simulation done , we can hope to learn d parameters of β in roughly $10d$ comparisons for an user. We also note the case for learning how a group of voters will behave. Based on the results we state that asking $\sqrt{N}$ peers to vote in each round, where N is the number of total users may lead us towards learning the group preference in $(10d \times \sqrt{N})$ rounds.

C. Implementation

We now discuss how we may go implement this in the smart contract. For a particular transaction, let the feature vector of relevant parameters be $\mathbf{x}$, with d features. For example, for the add course process, the unexpected situation comes when the course size is full. Relevant asset parameters would be *Class size*, *Student registered credits*, *Will the student miss a mandatory course* etc. We calculate the difference of these parameters between the current condition and the condition after the action lists were implemented. That is $(\mathbf{x}_A - \mathbf{x}_B)$. Now, a user's preference can simply be modeled with a parameter vector β, β having the same number of parameters as $\mathbf{x}$. β needs to be added to each participant's parameter list, for this learning mechanism to work. Now, for all the different transactions, where we would like to learn user preferences, we would need a new learning parameter array. This is because we would need to predict preferences for different types of transactions. One thing to keep in mind is that the learning needs to be done on the blockchain and in real-time. Because of the lack of support for high-end machine learning tools in Hyperledger Composer, which was used to develop this particular example, we chose a simple Recursive Linear Squares (RLS) estimator [9]. This approach only takes $O(d^3)$ time in updating the weights for each user, while we need to store $O(d^2)$ in weights for each user for each type of transaction that we want to learn as well. So there is a price to pay in storage.

```
concept Learning{
   o Integer type
   o Double[] P
   o Double[] beta
}
participant Student{
    o String Name
    /...other parameters/
    o Learning[] learning
}
```

Fig. 5. An example of added learning structures. Beta is the learning parameter for the probability model, P is a helper parameter for the RLS algorithm

D. Evaluation of Learning Preferences

1) Setup: For lack of actual data, we opt to create synthetic data based on a generic case. The dimension of the feature vector is d ($d = 10$ for Figure 6 case), with both boolean and real numbered variables. The real numbered inputs are taken from $\mathcal{N}(0, 1)$ and the boolean variables have a uniform distribution. To indicate preferences, we sample β from $\mathcal{N}(\mathbf{u}, \mathbf{I})$ where $\mathbf{u}$, each mean, u_i being sample uniformly from $\mathbf{unif}(-1, 1)$. Then we randomly pick pairs of inputs from the set of generated inputs and calculate the preference using the TM model, with some added noise.

2) Results: For each user we attempt to learn the preference model, that can predict their probability of agreeing or disagreeing with a proposal. With each vote they cast, we update the values of $\beta^{(i)T}$ for them. And we note that $\beta^{(i)T}$ converges towards the real value of β within $10d$ votes.

We repeat this process by generating synthetic preference data for a large number of users, randomly picking users for each round of voting and then estimating the group preference. For the graph in Figure 6, we have simulated with 1000 users and 20 voters per each round and we see that the average predicted group preference also converges towards the average real group preference.

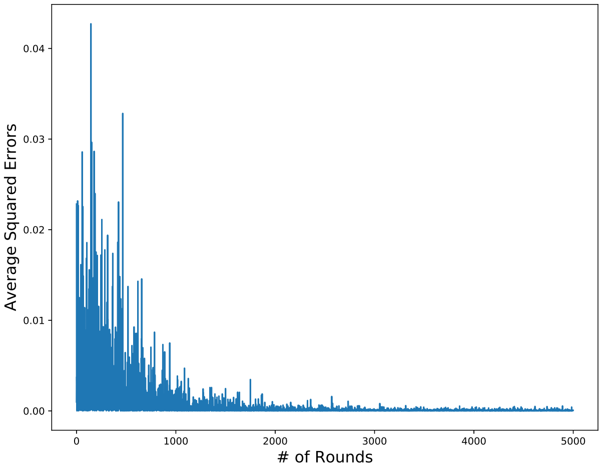

Fig. 6. Convergence towards learning group preference

We propose having $\sqrt{N}$ voters for each round because that way we would converge towards good estimates for the group preference by $O(d\sqrt{N})$ rounds. This simulation indicates that this method would be able to model preferences in smart contracts, where the users' preference primarily depend on the difference between the alternatives. However, as the TM model can deal with multiple alternatives instead of one as well as shown in [4], a similar preference learning and aggregation technique may be applied there as well, with proper ranking or scoring rules.

VII. Future Work

Now users need to propose solutions by themselves. We could use semantic techniques to automatically generate solutions for unexpected situations. There is also the fact that the learning mechanism only works for the pre-programmed voting polls, which makes it kind of limited in its implementation. Making the system more robust and generic is what we are looking towards in this aspect. For the contents of solutions, we also want to make sure the contents are proper. Like a peer cannot propose a solution to use others money to buy goods. Also, a technique for spam detection and penalties may also be deployed to penalize malicious users who may want to hinder operation by flooding the system with unexpected situations.

In the learning methodology, the advantage of the TM model is that the argument can readily be increased to multiple options as well. So when multiple proposals are at hand, we may have the users rank the alternate proposals and use this same learning mechanism to learn of their preferences based on some rank aggregation rule.

VIII. Conclusion

Even though smart contracts add much functionality to DApps in terms of automated transaction processes, most smart contracts are not well-equipped to handle unexpected situations. We design a preprocessor that works on Hyperledger Composer models and chain-code to augment the code with new basic action lists to change parameters and also adds a generic voting mechanism. The preprocessor automatically generates a standard and correct action list based on types of parameters. It also generates three important voting transactions to build the voting mechanism and ensures that the idea can be implemented into a given smart contract efficiently. To implement the necessary changes of parameters to resolve an unexpected situation, we make use of a consensus-based protocol using voting. The protocol is inspired by the system of passing US Congressional bills, and majority agreement is still enforced so as not to open up vulnerabilities in the smart contract.

Our proposed voting mechanism is augmenting the smart contract, as is learning parameters introduced to help a preference learning model. The strengthened smart contract supports a mechanism to learn preferences that voters show while voting for unexpected situations based on specific transaction types. We model each user's preferences based on relevant parameters using the Thurstone-Moesteller probability model. We theoretically show that this should eventually lead towards being able to predict all user's preferences, which we can aggregate. This effectively adds a decision-making capability to the smart contract. We have implemented this learning mechanism in a decentralized course selection system that we designed.

Acknowledgements

This work is supported by the IBM-RPI Artificial Intelligence Research Collaboration (a member of the IBM AI Horizons Network). We thank our colleagues James A. Hendler and Geeth De Mel for their insight and expertise that greatly assisted the research, and also the reviewers for helpful comments and suggestions.

References

[1] R. Miller. (2018) Walmart is betting on the blockchain to improve food safety. [Online]. Available: https://techcrunch.com/2018/09/24/walmart-is-betting-on-the-blockchain-to-improve-food-safety/

[2] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making smart contracts smarter," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security - CCS'16*. ACM Press, 2016. [Online]. Available: https://doi.org/10.1145/2976749.2978309

[3] A. Azaria, A. Ekblaw, T. Vieira, and A. Lippman, "MedRec: Using blockchain for medical data access and permission management," in *2016 2nd International Conference on Open and Big Data (OBD)*. IEEE, aug 2016. [Online]. Available: https://doi.org/10.1109/obd.2016.11

[4] R. Noothigattu, S. N. S. Gaikwad, E. Awad, S. Dsouza, I. Rahwan, P. Ravikumar, and A. D. Procaccia, "A voting-based system for ethical decision making," in *Proceedings of the 2018 AAAI Conference on Artificial Intelligence*. AAAI Press, 2018.

[5] D. Friedman, "Effective scoring rules for probabilistic forecasts," *Management Science*, vol. 29, no. 4, pp. 447–454, apr 1983.

[6] L. de Alfaro, M. Shavlovsky, and V. Polychronopoulos, "Incentives for truthful peer grading," *CoRR*, vol. abs/1604.03178, 2016. [Online]. Available: http://arxiv.org/abs/1604.03178

[7] L. L. Thurstone, "A law of comparative judgment." *Psychological Review*, vol. 34, no. 4, pp. 273–286, 1927.

[8] F. Mosteller, "Remarks on the method of paired comparisons: I. the least squares solution assuming equal standard deviations and equal correlations," in *Springer Series in Statistics*. Springer New York, 1951, pp. 157–162. [Online]. Available: https://doi.org/10.1007/978-0-387-44956-2_8

[9] R. L. Plackett, "Some theorems in least squares," *Biometrika*, vol. 37, no. 1/2, p. 149, jun 1950.