The Knowledge Engineering Review, Vol. 00:0, 1–24. © 2004, Cambridge University Press
DOI: 10.1017/S0000000000000000 Printed in the United Kingdom

A Blockchain Based Decentralised Booking System

Naipeng Dong[1], Guangdong Bai[2], Lung-Chen Huang[3], Edmund Kok Heng Lim[4] and Jin Song Dong[5]

[1,3,4,5]*School of Computing, National University of Singapore, COM*1*,* 13 *Computing Drive,* 117417*, Singapore*
[2,5]*School of Information and Communication Technology, Griffith University, N*44 2.28*,* 170 *Kessels Road Nathan, QLD,* 4111*, Australia*
E-mail: [1]*dcsdn@nus.edu.sg,* [2]*g.bai@griffith.edu.au,* [3]*lungchenhuang@u.nus.edu,* [4]*e0335737@u.nus.edu,* [5]*dcsdjs@nus.edu.sg*

Abstract

Blockchain technology has rapidly emerged as a decentralised trusted network to replace the traditional centralised intermediator. Especially, the smart contracts that are based on blockchain allow users to define the agreed behaviour among them, the execution of which will be enforced by the smart contracts. Based on this, we propose a decentralised booking system that uses the blockchain as the intermediator between hoteliers and travellers. The system enjoys the trustworthiness of blockchain, improves efficiency and reduces the cost of the traditional booking agencies. The design of the system has been formally modelled using the CSP# formal language and has been formally verified using the model checker PAT. We have implemented a prototype decentralised booking system based on the Ethereum ecosystem.

1 Introduction

The blockchain technology has rapidly emerged in recent years (Nakamoto (2019), Buterin (2019), Swan (2015)), especially when the concept of smart contract was first introduced to and later relied on the technology (Morris (2019), Cachin (2019), Schwartz *et al.* (2014)). Once the smart contract code is deployed to the blockchain, it can be executed by any computer node that keeps the same historical record of transactions as other nodes. This makes it difficult to be compromised with a single node on the network, unlike centralized platforms that could be easily breached and prone to the failure of single point. However, decentralized platforms are still in its infancy. Although some communities and companies have proposed some use cases, for example for the logistics (DHL (2019), Marr (2019)) and insurance industry (CBInsights (2019), Sarasola (2019)), its use cases are lacking in different domains. To give a more concrete example of using the smart contract, we design a fundamental architecture of a decentralized hotel booking system.

Most travelers would have hard times finding a hotel room for their next journey. They would possibly browse through pages of entries on online travel agencies such as *Booking.com* or *Agoda.com*. As we can imagine, it is time-consuming that each time a search result appears on the screen, they have to delve into the deals one by one, which could be overlapping in the previous search results. In order to alleviate this monotonous experience, we shall leverage the blockchain technology and allow our search requests to be deployed as smart contracts such that the process of discovery is left to the decentralized system, where hoteliers can easily match their rooms with the criteria required by a traveler.

Our approach is to provide users with an interface where they can draft their booking request with the requirement of a hotel room in a domain-specific language, which is similar to a real contract in a human-readable form. Thereafter, the interface compiles the request into a machine-readable form and injects some predefined functions. Later, the user can deploy the request onto the blockchain. Once the request is visible to other nodes, hoteliers can propose offers by invoking a function in the request. When the user is notified of new proposals, the interface automatically starts a selection process of the proposals depending on the request criteria from the user, showing the matched results. At the end, the user and one of the hoteliers seal a deal.

To ensure the correctness of the design, we model the system using the CSP# (Communicating Sequential Programs sharp) formal language (Sun *et al.* (1999)), as CSP# integrates the high-level modeling operators of CSP with low-level procedural codes in C# language and supports custom data structures, which will be convenient to represent the behaviors of the blocks and the blockchain. We verify the system design using the model checker PAT (Process Analysis Toolkit, available at pat.comp.nus.edu.sg), which supports the CSP# models as input. The formal model is comprehensive, containing not only the behaviour of the smart contract, the hoteliers and the travellers, but also contains the underlying blockchain mechanism.

In addition, we have implemented a prototype booking system based on the Ethereum ecosystem (Ethereum (2019)), and tested the prototype system on a Ethereum Virtual Machine (EVM). The booking system contains three components: the hotelier, the traveler and the smart contract. The EVM mainly contains the EVM nodes, the miner nodes. We use *Truffle* to compiler the smart code and deploy the compiled bytecode to the EVM nodes. In addition, we installed a web server to test the hotelier and traveller web pages.

2 Background

We use the CSP# as the language to formally model the system design. This language is supported by the automatic model checker PAT. The syntax of this language and the model checker PAT are briefly introduced in the first sub-section. After verifying the correctness of the design, we implemented the design based on EVM, which is described in the second part.

2.1 CSP# and PAT

The CSP# is a rich modelling language that contains both high level modelling operators in the traditional CSP language and programmer-favored low-level constructs like variables, arrays, if-then-else, while, etc.. It offers great flexibility to model systems with complicated structures, like blockchain. A system is modelled as a process in CSP#, where the operators in CSP# is defined as follows (in Figure reffig:syntax).

$$
\begin{array}{lll}
\text{Process } P,\ Q ::= & & \\
& \mathit{Stop} & \text{– deadlock} \\
& \mid \mathit{Skip} & \text{– termination} \\
& \mid [b]P & \text{– state guard} \\
& \mid e \rightarrow P & \text{– event prefixing} \\
& \mid e\{\mathit{program}\} \rightarrow P & \text{– data operation prefixing} \\
& \mid c?d \rightarrow P(d) & \text{– channel input} \\
& \mid c!d \rightarrow P & \text{– channel output} \\
& \mid P;\ Q & \text{– sequence} \\
& \mid P \sqcap Q & \text{– internal choice} \\
& \mid P \Box Q & \text{– external choice} \\
& \mid \mathit{if}\ b\ \mathit{then}\ P\ \mathit{else}\ Q & \text{– conditional branch} \\
& \mid P \mid\mid Q & \text{– synchronous} \\
& \mid P \mid\mid\mid Q & \text{– asynchronous}
\end{array}
$$

where P and Q are *processes*, b is a condition, e is a simple event, *program* is a block of code that is atomically executed and c is a synchronized communication channel.

Figure 1 CSP# Syntax

The deadlock process is *Stop*, meaning that the process does absolutely nothing. Process *Skip* means that the process termites immediately. Process $[b]P$ is a guarded process - the process behaves as P if b is satisfied. Process $e \rightarrow P$ is event prefixing - the process performs event e (a simple event is a name for representing a observation) and then behaves as P. Process $e\{\mathit{program}\} \rightarrow P$ is similar to event prefixing. The difference is that the $\{\mathit{program}\}$ attached with event e allows us to write assignments which update global variables. Process $c?d \rightarrow P(d)$ is channel input - the process reads a value d from channel c, thus d is known in the subsequent process P. Process $c!d \rightarrow P$ is channel output - the process outputs value d in channel c. Process $P;\ Q$ concatenates two processes P and Q sequentially. Process $P \sqcap Q$ is internal choice - the process either chooses P or Q to execute. If P performs an event first, then P takes control; Otherwise Q takes control. Process $P \Box Q$ is external choice. Differing from the internal choice, the external choice is resolved by the environment. Process $\mathit{if}\ b\ \mathit{then}\ P\ \mathit{else}\ Q$ is straight forward: if b is true, the process behaves as P; Otherwise behaves as Q. Both $P \mid\mid Q$ and $P \mid\mid\mid Q$ model that processes P and Q run in parallel. The difference is that the form one requires P and Q to synchronise on the shared events, whereas in the later process, P and Q interleave.

PAT is a self-contained framework which supports simulating, reasoning and verifying concurrent systems. It has user friendly interfaces, a featured model editor and an animated simulator. Most importantly, PAT implements various model checking techniques catering for different properties such as deadlock-freeness, reachability and LTL (Linear Temporal Logic) properties that are useful for our system verification. To achieve good performance, advanced optimization techniques are implemented in PAT, e.g. partial order reduction, symmetry reduction, process counter abstraction, parallel model checking. So far, PAT has 4370+ registered users from 1341+ organizations in 150 countries and regions (PAT (2019)).

2.2 Ethereum

The Ethereum ecosystem serves as a runtime environment for smart contracts. It provides a language *Solidity* to write smart contracts. The smart contracts need *Gas* to be executed. And Gas can only be bought using the cryptocucrency *Ether* Provided by Ethereum. It mainly involves the following parts (Ethereum (2019), Modi (2019), Buterin (2019)):

- EVM nodes: The EVM runs smart contract bytecode. Each node runs an instance of the EVM. Once written, the contracts have to be deployed on the EVM. Users can interact with the contracts that are on the EVM. The EVM node is a host for the Ethereum network. There can be many EVM nodes serving one network, in that sense the Ethereum network is decentralised.
- Mining nodes: The mining nodes (a.k.a. miners) form the basic blockchain. Each block contains data and smart contract code, and holds a copy of the blockchain. The mining nodes are required to verify and add blocks to the chain and in the process get paid in Ether. Every transaction requires Gas to run. Gas is the measure of the amount of Ether that is required to run.
- Externally owned account: It represents a user. When a new account is created, a private public key pair is created. The externally owned account is the public key of the account.
- Wallet: It is a software that is associated to user accounts. One user may have more than one account in one wallet. The wallet is an important component as it is the software that tracks the amount of Ether that one user has. Mining nodes also require a wallet to store the Ether earned. Most wallets will show transactions made; these transactions are identified by their hash values.
- Smart contracts: Smart contracts are programs that exist on the EVM. They can accept inputs and, based on the inputs, can produce an action whether it is to read a value from the blockchain or to write a value to the blockchain. Smart contracts are the key to interacting with the blockchain programmically.
- Uer interface: It refers to any interface that is interacting with a smart contract on the EVM. User interfaces together with smart contracts are known as DApps or decentralised apps.

3 System design

Using the Ethereum environment, we will not need to consider the blockchain in the system design, as the EVM provides interfaces to communicate with the underlying blockchain. Thus, what we need in the system design is mainly three parts: the smart contract, the traveler behaviour and the hotelier behaviour. To more clearly illustrate the workings of our system, the system sequence diagram in Figure 2 describes how travelers and hoteliers interact on the decentralized booking platform.

Once the smart contract is deployed on the EVM, a traveler can put a request to the smart contract and then listens to the contract for any response. A Hotelier listens to the contract and once discovers a request he proposes his offer. After that the hotelier listens to the contract for response. When the traveler accepts the offer, he settles the request and then pay the deposit.

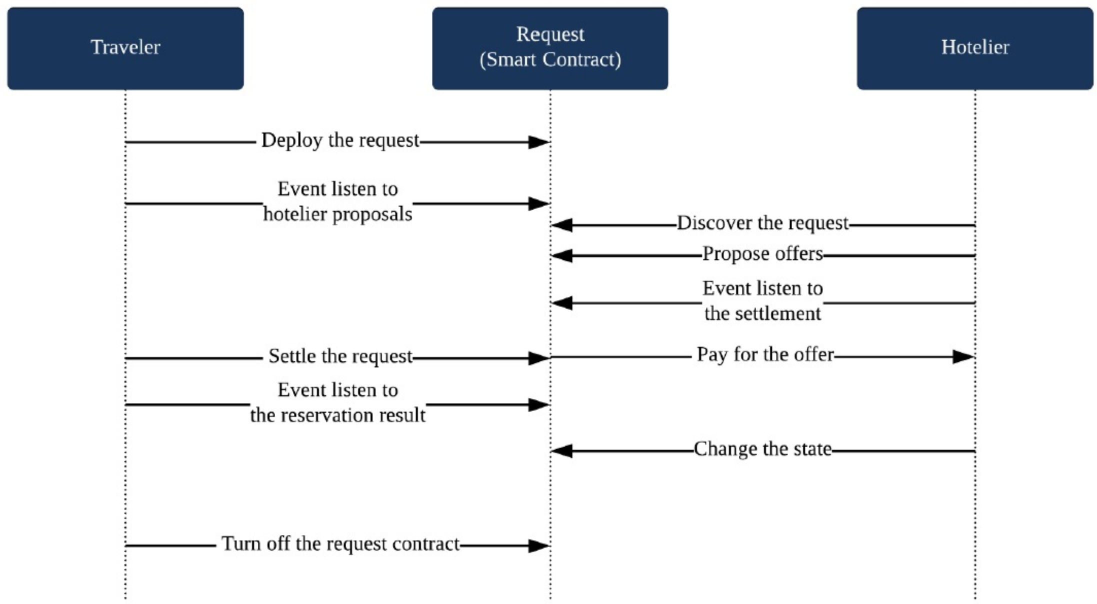

Figure 2 Overview of the system design

On receiving the deposit payment, the hotelier change the room into unavailable and informs the smart contract. Finally, the traveler disables the request.

Note that for the smart contract to run, we need to additionally consider the underlying blockchain. Thus, the entire system contains three roles *user*, *miner*, and the *smart contract*. In more details,

- A *user* could be a traveler or hotelier who sends a transaction by invoking a function that includes the address of the user, function name, gas, and gas price. A traveler can invoke *Fetch* to acquire the latest set of proposals submitted by hoteliers, or to *Settle* for a specific proposal. A hotelier can invoke *Propose* to send their proposals to the smart contract.
- *Miners* form the basic blockchain network. Each miner has its own transaction pool *TxPool* that continuously receives a new transaction with gas greater than zero. Moreover, *TxBlock*, to be executed by a miner, also finds the new transactions with more than or equal to a specified transaction *gasPrice* from the pool.
- The *smart contract* is named as the *Request*. The request deployed on the blockchain is a smart contract that accepts proposals and can be invoked by the users.

We use model the entire system in the formal verification as it is more comprehensive.

4 System modelling

First of all, we define a list of constants that are used in the model in Figure 3. Then we divide the model into two parts: the *user* behaviour in Section 4.1 and the *Miner* behaviour in Section 4.2. The miner part includes the behaviour of the EVM nodes, the mining nodes, and the smart contract. Since these behaviours are closely connected, we do not separate them.

```
enum {off, on, switchOp, proposeOp, fetchOp, settleOp, ProposeFunction, na,
      SwitchFunction, FetchFunction, SettleFunction, newcomer};
```

where *off* and *on* are used to indicate whether the smart contract is still available; the *switchOp*, *proposeOp*, *fetchOp*, and *settleOp* are the 4 operations that the users can perform; *na* indicates the empty data; *SwitchFunction*, *FetchFunction*, and *SettleFunction* are the names of the 3 functions provided by the smart contract; and *newcomer* is used to indicate a new block in the model.

Figure 3 Constants declaration

4.1 user

A user, with an address *addr*, would like to execute *function* and would like to pay a fund *val* plus some gas amount *gas* with price *gasPrice*. To do so, the user broadcasts the transaction to all miners. This is modelled as a process *User*(*addr*, *val*, *gas*, *gasPrice*, *function*) in Figure 4. The reason that the user needs to specify the gas amount and gas price is that the gas price is changing according to the demand and supply; to be clear on how much the user pays the miner for executing the function, the user specifies the current gas price and the amount of gas the user would like to pay.

When the user sends broadcasts transactions, for simplicity, we assume the user sends the transactions to miners in order, as this will not affect the results we would like to check. The process of sending a transaction to miner i is modelled as *TxBrdcst*(i, *addr*, *val*, *gas*, *gasPrice*, *function*) in Figure 4. When the miner index i is correct (i.e., i is smaller than the maximum number of miners), this process sends a transaction to the i-th miner with the parameters *addr*, *val*, *gas*, *gasPrice*, *function*, and then continues to sends the transaction to the next miner. If the miner index reaches the top, meaning that the user has sends to every miner, then the process terminates with *Skip*.

```
User(addr, val, gas, gasPrice, function) =
            TxBrdcst(0, addr, val, gas, gasPrice, function);

TxBrdcst(iter, addr, val, gas, gasPrice, function) =
  atomic{
    if (iter < M) {
      mnet[iter]!addr.val.gas.gasPrice.function ->
      TxBrdcst(iter+1, addr, val, gas, gasPrice, function) }
    else {
      Skip } };
```

Figure 4 The user process

4.2 Miner

The miner i has two processes running in parallel: process *TxPool*(i) which receives transactions and puts them into a pool, and process *TxBlock*(i) which includes the transactions in the pool into a block and initiates the mining.

4.2.1 Process TxPool

The miner i receives the transactions broadcast by the users in the process *TxPool*(i). On receiving a transaction with parameters, the miner first checks whether the sender i.e., the user with address *addr* has enough funds and gas, if so, the miner inserts the transaction into his transaction pool *TxInsert*$(i$, *addr*, *val*, *gas*, *gasPrice*, *function*$)$ defined in Figure 6, otherwise, the miner keeps waiting for another transaction, i.e., returning back to process *TxPool*(i) (see Figure 5).

```
TxPool(i) =
  mnet[i]?addr.val.gas.gasPrice.function ->
  if (userWallet[i][addr] >= (val + (gas * gasPrice)) && gas > 0)
       {TxInsert(i, addr, val, gas, gasPrice, function) }
  else {TxPool(i) };
```

Figure 5 The transaction pool process

```
TxInsert(i, addr, val, gas, gasPrice, function) =
  if (poolPtr[i] < poolSize) {
    if (poolTxGas[i][poolPtr[i]] == 0) {
      txReceive{
        var ptr = poolPtr[i];
        poolTxAddr[i][ptr] = addr;
        poolTxVal[i][ptr] = val;
        poolTxGas[i][ptr] = gas;
        poolTxGasPrice[i][ptr] = gasPrice;
        poolTxFunction[i][ptr] = function;
        poolPtr[i]++;} ->
      datach[i]!newcomer ->
      TxPool(i) }
    else {
      next{poolPtr[i]++;} ->
      TxInsert(i, addr, val, gas, gasPrice, function) } }
  else {
    next{poolPtr[i] = 0;} ->
    TxInsert(i, addr, val, gas, gasPrice, function)};
```

Figure 6 The transaction insertion process

When inserting a transaction into a pool of miner i, the miner first checks whether the pool is full, i.e., the current transactions in i (*PoolPtr*$[i]$ is smaller than the max *poolSize*). If not, the miner receives the transaction in the pool by updating the address, the value, the gas, the gasPrice and the function to the pool of miner i. At the end, we update the pointer of the transaction pool to the next empty position. After updating the transaction pool, the miner sends a single of *newcomer* to the channel *datach* for miner i to initiate the block forming. Finally the process goes back to *TxPool*(i) to listen to a new transaction. Note that before updating the transaction pool, we double check whether the current empty pool position is really empty by checking whether the gas for the current position is 0. If the current position is not empty, the process moves to the next position and tries to insert the transaction to the pool. If the transaction pool is full, the transaction pointer is reset to 0 and then the transaction is inserted into the transaction pool, i.e., to form transactions for the next block. We notice that if there are more than the maximum amount of transactions received in the pool before a block is formed, the pool will be reset to new ones. In our experiment, the total transactions are less than the maximum transitions in a pool, so this will not happen.

4.2.2 Process TxBlock

```
TxBlock(i) =
  if (inPoolPtr[i] < poolSize) {
    if (!call(cempty, datach[i])) {
      datach[i]?newcomer ->
      if (poolTxGas[i][inPoolPtr[i]] == 0) {
        if (blkSize[i] >= 1) {
          Miner(i, 0)}
        else { TxBlock(i) } }
      else {
        if (blkTotalGas[i] + poolTxGas[i][inPoolPtr[i]] <= blkGasLimit) {
          if (poolTxGasPrice[i][inPoolPtr[i]] >= gasPriceCondition) {
            txInclude{
              var pIter = inPoolPtr[i];
              var bIter = blkSize[i];
              blkTxAddr[i][bIter] = poolTxAddr[i][pIter];
              blkTxVal[i][bIter] = poolTxVal[i][pIter];
              blkTxGas[i][bIter] = poolTxGas[i][pIter];
              blkTxGasPrice[i][bIter] = poolTxGasPrice[i][pIter];
              blkTxFunction[i][bIter] = poolTxFunction[i][pIter];
              blkTotalGas[i] = blkTotalGas[i] + poolTxGas[i][pIter];
              poolTxAddr[i][pIter] = 0;
              poolTxVal[i][pIter] = 0;
              poolTxGas[i][pIter] = 0;
              poolTxGasPrice[i][pIter] = 0;
              poolTxFunction[i][pIter] = 0;
              inPoolPtr[i]++;
              blkSize[i]++;} ->
            TxBlock(i)}
          else {nextTx{inPoolPtr[i]++;} ->
        TxBlock(i)}}
      else if (blkSize[i] >= 1) {Miner(i, 0)}
      else {nextTx{inPoolPtr[i]++;}->
      TxBlock(i)}}}
    else if (blkSize[i] >= 1) {Miner(i, 0)}
    else { TxBlock(i) }}
  else {nextTx{inPoolPtr[i] = 0;} -> TxBlock(i)};
```

Figure 7 The process of including transactions into a block

The process $TxBlock(i)$ (Figure 7) includes available transitions into a block. When the number of transactions in the pool is smaller than the pool size and the channel for receiving transactions is not empty, the process receives a signal that a new transaction has been inserted into the transaction pool. Otherwise the process sets the pointer $inPoolPtr(i)$ to 0 and goes back to the process $TxBlock(i)$.

On receiving the signal, the miner checks whether the gas for this transaction is 0, which indicates there is no transaction in the pool.

- If there is no transaction in the pool and the block size is bigger than 1, meaning that all the transactions have been included in the block, the miner starts mining the block by calling $Miner(i, 0)$. If there is no transaction in the pool and the block size is smaller than 1, meaning that there is no transaction in the block, then the process goes back to $TxBlock(i)$.
- If there are transactions in the pool, the miner checks whether the total gas for the current block reaches the top limit and checks whether the current transaction in the pool provides enough gas to execute the corresponding function in the transaction. If so, the miner includes the transaction in the block, by copying the address, value, gas, gas price and function into the block. In addition, the miner updates the total gas of the block and sets the position of the transaction in the pool to be 0 to indicate that the transaction in the pool has been included into the block. Furthermore, the pointer in the pool will be moved to the next and the size of the block is increased by 1. After updating the current transaction in the pool, the process goes back to $TxBlock(i)$ to include the next transaction into the pool, until all transactions have been included, i.e., the pointer $inPoolPtr[i]$ is bigger than the $poolSize$. If the current transaction in the pool fails to provide enough gas, the miner simply ignores the transaction and moves to the next one. If the total gas has reached the top limit and there are transactions in the block, the miner starts mining. If the total gas reaches the top, but somehow there is no transaction in the block, the process goes back.

The mining process $Miner(i, iter)$ in Figure 8 starts with a checking on whether the block index is smaller than the block size of the miner i. If so, the miner locks up the total value and executes the functions in the block.

```
Miner(i, iter) = if (iter < blkSize[i]) {LockUp(i, iter)}
                 else { BlkUpdate(i, 0)};
```

Figure 8 The miner process

To lock up the total value, the miner i calculates the amount to lock by adding the fund value and the gas value as the locked total amount (Figure 9). The remaining amount for the user with address $addr$ is also calculated by deducting the locked amount. Once the amount for a transaction $iter$ is locked, the miner executes the transaction by calling $TxExec(i, iter)$.

To execute a transaction $iter$ (Figure 10), the miner reads in the function in the transaction. There are four functions provided in the smart contract: *SwitchFunction*, *ProposeFunction*, *SettleFunction* and *FetchFunction*, which are defined as constants.

```
LockUp(i, iter) =
  weiLockUp{
    var addr = blkTxAddr[i][iter];
    var val = blkTxVal[i][iter];
    var gas = blkTxGas[i][iter];
    var gasPrice = blkTxGasPrice[i][iter];
    var lockedTotal = val + (gas * gasPrice);
    userWallet[i][addr] = userWallet[i][addr] - lockedTotal;
    lockedWallet[i] = lockedTotal;} ->
  TxExec(i, iter);
```

Figure 9 The lock up process

```
TxExec(i, iter) =
  case {
    blkTxFunction[i][iter] == SwitchFunction:
      if (contractOwner == blkTxAddr[i][iter]) {
        GasConsume(i, iter, UPDATE) || Execution(i, iter)}
      else { LockedReturn(i, iter, false) }
    blkTxFunction[i][iter] == ProposeFunction:
      if (contractSwitch[i] == on) {
        GasConsume(i, iter, UPDATE) || Execution(i, iter)}
      else {LockedReturn(i, iter, false)}
    blkTxFunction[i][iter] == SettleFunction:
      if (contractOwner == blkTxAddr[i][iter]) {
        GasConsume(i, iter, UPDATE + TRANSFER) || Execution(i, iter)}
      else {LockedReturn(i, iter, false)}
    blkTxFunction[i][iter] == FetchFunction:
      if (contractOwner == blkTxAddr[i][iter]) {
        GasConsume(i, iter, FETCH) || Execution(i, iter)}
      else {LockedReturn(i, iter, false)}};
```

Figure 10 The overall process of transaction execution

- When *SwitchFunction* is called, the miner checks whether the user who calls the function is the contract owner. If so, the miner executes the transaction and updates the gas with price of *UPDATE*; Otherwise returns false to indicate that the function is not successfully executed.
- When *PropsoeFucntion* is called, the miner checks whether the contract is switched on. If so, the miner updates the gas with price of *UPDATE* and executes the transaction; Otherwise returns false.
- When *SettelFunction* is called, the miner checks whether the user who calls the function is the contract owner. If so, the miner executes the transaction and update the gas with price of *UPDATE* + *TRANSFER*; Otherwise returns false. Differing from the previous two functions, this function costs more, including the updating fee and the transferring fee.

```
GasConsume(i, iter, opcode) =
  if (estimatedGas[i][iter] + opcode > blkTxGas[i][iter]) {
ExecFail(i, iter)}
  else {
    consuming{
      var gasPrice = blkTxGasPrice[i][iter];
      var price = opcode * gasPrice;
      minerCoinbase[i] = minerCoinbase[i] + price;
      lockedWallet[i] = lockedWallet[i] - price;
      estimatedGas[i][iter] = estimatedGas[i][iter] + opcode;} ->
      consumed ->
      Skip};

ExecFail(i, iter) =
  lockedReset{ lockedWallet[i] = 0;} ->
  BlkDetect(i, iter, false);
```

Figure 11 The gas update process

```
Execution(i, iter) = consumed ->
                     execution{
                       pendUid[i][iter] = txUid;
                       pendFromAddr[i][iter] = blkTxAddr[i][iter];
                       pendToAddr[i][iter] = contractAddr;
                       pendOp[i][iter] = blkTxFunction[i][iter];
                       pendField[i][iter] = na;
                       pendState[i][iter] = 1;
                       pendValue[i][iter] = blkTxVal[i][iter];
                       txUid++;} ->
                     LockedReturn(i, iter, true);

LockedReturn(i, iter, success) =
  returnGas{
    var addr = blkTxAddr[i][iter];
    var return = (blkTxGas[i][iter] - estimatedGas[i][iter]) * blkTxGasPrice[i][iter];
    userWallet[i][addr] = userWallet[i][addr] + return;
    lockedWallet[i] = 0; } ->
  BlkDetect(i, iter, success);

BlkDetect(i, iter, success) =
  if (call(cempty, bnet[i])) {Miner(i, iter+1)}
  else {bnet[i]?j.newBlockNum.blockid -> BlkAppend(i, j, newBlockNum, blockid)};
```

Figure 12 The actual execution process

- When *FetchFunction* is called, the miner checks whether the the user who calls the function is the contract owner. If so, the miner executes the transaction and updates the gas with the price of *FETCH*.

Before executing the functions in the transaction, the miner first checks whether the transaction has enough gas. If not, the execution fails, modelled in the process *ExecFail*(i, *iter*). If there is enough gas, the miner updates the gas by deducting the amount (specified in the parameter according to different functions) from the locked wallet and adding them to the miner's wallet. Then the miner imitates the execution, modelled by the synchronised event *consumed* (see Figure 11).

Once the event *consumed* is synced, the miner executes the function. We model the execution of a function *Execution*(i, *iter*) by giving the transaction a unique ID and storing the transaction owner, contract address, transaction function and transaction fee in a set of arrays to indicate that the function of the smart contract in that transaction has been executed (Figure 12). Thus, the process returns *true* by calling process *LockedReturn*(i, *iter*, *true*). Then the miner returns the remaining locked gas to the user. Before the miner executes the next transaction, she checks whether there is any block from another miner. If so, the miner chooses to execute the first block she receives, modelled in process *BlkDetect*(i, *iter*, *success*).

```
BlkUpdate(i, iter) =
  if (iter < blkSize[i]) {
    update{
      block[blockUid][iter] = pendUid[i][iter];
      txFromAddr[pendUid[i][iter]] = pendFromAddr[i][iter];
      txToAddr[pendUid[i][iter]] = pendToAddr[i][iter];
      txOp[pendUid[i][iter]] = pendOp[i][iter];
      txField[pendUid[i][iter]] = pendField[i][iter];
      txState[pendUid[i][iter]] = pendState[i][iter];} ->
    BlkUpdate(i, iter+1)}
  else {reward{minerCoinbase[i] = minerCoinbase[i] +
        succAppendPrice; rewardCount++;} ->
  BlkBrdcst(i, 0, blockNum[i], blockUid)};
```

Figure 13 The block update process

```
BlkBrdcst(i, iter, newBlockNum, blockid) =
  if (i == iter && iter < M) {BlkBrdcst(i, iter+1, newBlockNum, blockid)}
  else if (iter < M) {
    bnet[iter]!i.newBlockNum.blockid ->
    BlkBrdcst(i, iter+1, newBlockNum, blockid)}
  else {
    updateBlockUid{blockUid++;} ->
    BlkAppend(i, i, newBlockNum, blockid)};
```

Figure 14 The block broadcasting process

Recall that the miner process starts with a check on whether the block index is smaller than the block size of the miner i. If not, i.e., the transaction pool for the miner is full, the miner updates the block by rewarding the miner and broadcasting the block as in process *BlkUpdate*(i, *iter*) defined in Figure 13. To broadcast a block, the miner sends the block number and block ID

```
BlkAppend(i, j, newBlockNum, blockid) =
  append{chain[i][newBlockNum] = blockid;
         blockNum[i]++;
         blkSize[i] = blkSize[j];} ->
  ChainUpdate(i, 0, blockid);
```

Figure 15 The block appending process

to other miners (Figure 14). Once this has been done, the miner increases the block ID by 1 and appends the block to her chain (process *BlkAppend*(i, i, *newBlockNum*, *blockid*)). In process *BlkAppend*(i, i, *newBlockNum*, *blockid*) defined in Figure 15, the minder appends the latest block and updates the state in the blockchain.

```
ChainUpdate(i, iter, blockid) =
  if (iter < blkSize[i]) {
    case {
      txOp[block[blockid][iter]] == FetchFunction:
        fetch ->
        ChainUpdate(i, iter+1, blockid)
      txOp[block[blockid][iter]] == SwitchFunction:
        switch{var id = block[blockid][iter];
               if (contractSwitch[i] == off) {contractSwitch[i] = on;}
               else {contractSwitch[i] = off;}} ->
        ChainUpdate(i, iter+1, blockid)
      txOp[block[blockid][iter]] == ProposeFunction:
        propose{var id = block[blockid][iter];
               var addr = txFromAddr[id];
               var data = txState[id];
               var ptr = proposalPtr[i];
               proposalFrom[i][ptr] = addr;
               proposal[i][ptr] = data;
               proposalPtr[i]++; } ->
        ChainUpdate(i, iter+1, blockid)
      txOp[block[blockid][iter]] == SettleFunction:
        settle{var id = block[blockid][iter];
              var addr = txFromAddr[id];
              var data = txState[id];
              settledWith[i] = data;
              userWallet[i][data] = userWallet[i][data] + txValue[id]; } ->
        ChainUpdate(i, iter+1, blockid)}}
  else {Reset(i, 0) };
```

Figure 16 The chain updating process

The process of updating the blockchain actually updates the results of the functions as shown in Figure 16. When the operation is *FetchFuction*, we add a *fetch* event to denote it; When the operation is *SiwtchFunction*, the process switches the smart contract from the current state *on*/*off* to the alternative sate *off*/*on*; When the operation is *ProposeFunction*, the process executes a *propose* event, where the proposal initiator and the proposal state are recorded; When the operation is *SettleFunction*, the process executes a *settle* event, which records the one who settle the deal and the corresponding transaction state. Once an operation is finished, the process goes back to itself for the operation in the next transaction. Note that the process first checks whether all the transaction has been updated. If so, the process resets the transaction container in process $Reset(i, 0)$.

In the reset process, the miner clears each transaction container by setting the values into 0 shown as follows in Figure 17. Once all the transactions in a block is cleared, the miner clears the block by setting the gas and size into 0.

```
Reset(i, iter) =
  if (iter < blkSize[i]) {clearTx{
                                pendUid[i][iter] = 0;
                                pendFromAddr[i][iter] = 0;
                                pendToAddr[i][iter] = 0;
                                pendOp[i][iter] = 0;
                                pendField[i][iter] = 0;
                                pendState[i][iter] = 0;
                                pendValue[i][iter] = 0;
                                blkTxAddr[i][iter] = 0;
                                blkTxVal[i][iter] = 0;
                                blkTxGas[i][iter] = 0;
                                blkTxGasPrice[i][iter] = 0;
                                blkTxFunction[i][iter] = 0;
                                estimatedGas[i][iter] = 0;} ->
                        Reset(i, iter+1)}
  else {clearBlk{
              blkTotalGas[i] = 0;
              blkSize[i] = 0;} ->
        TxBlock(i)};
```

Figure 17 The reset process

4.3 The overall process

In summary, the proposed blockchain based booking system can be modelled as the *user* process and the miner process running in parallel. And the miner process is the transaction pool process *TxPool* and the blockchain process *TxBlock* running in parallel.

```
ProposerExecution = User(1, 0, gasLimit, 5, ProposeFunction) |||
                    (|||i:{0..M-1} @ (TxPool(i) ||| TxBlock(i)));
```

Figure 18 An overall system model with only one user

5 System verification

In this section, we discuss the properties that the booking system aims to achieve, define the executions for verifying each property, and finally present the verification results.

5.1 Properties

We verified 5 properties that are defined as follows.

- deadlockfree: No deadlock situation occurs in the system.
- GasRunOut: Each transaction in a block has enough gas to be executed to completion by miners.
- sameBlockNumEventually: Each miner reaches the same block number eventually.
- receiveSettlement: The request (smart contract) owner has settled with some proposer, and thus each miner receives the same transaction.
- proposalReceived: Proposals have been accepted by the request.

Each property is formalised as an assertion in PAT for automatic formal verification as shown in Table 1. Note that the symbol && is logical 'and'. Thus, the formula $\&\&i : \{0..M\}$ @ $blockNum[i] == 1$ means $blockNum[0] == 1 \wedge blockNum[1] == 1 \wedge \ldots \wedge blockNum[M] == 1$. Similarly, the formula $\&\&i : \{0..M\}$ @ $settledWith[i] == 1$ means that $\forall\, i \in \{0..M\} : settledWith[i] == 1$.

Property	**Formalisation in PAT**
deadlockfree	*deadlockfree*
GasRunOut	$estimatedGas[0][0] > gasLimit$
sameBlockNumEventually	$\&\&i : \{0..M\}$ @ $blockNum[i] == 1$
receiveSettlement	$\&\&i : \{0..M\}$ @ $settledWith[i] == 1$
proposalReceived	$proposalPtr[0] >= 1$

Table 1 Formalisation of the property

For each property, we match one execution. And the 5 executions are listed as follows:

- ProposerExecution: A user submits proposals to the request (smart contract).
- ListnerExecution: A user submits proposals to the request, and the request owner listens to the "proposal" event from the request.
- UnavailabeExecution: The request owner turns off the request, and no more proposals are allowed to be accepted by the request.
- NotOwnerExecution: A user who is not the request owner fetches the proposals from the request.
- MultipleUsersExecution: Many users submit their proposals at the same time.
- SettlementExecution: The request owner seals a deal.

The formalisation of the executions are defined in Table 2. Note that the 'ProposeExecution' differs from the 'MultipleUsersExecution' in the number of users; the executions 'Propose-Execution', 'UnavialableExecution', 'NotOwnerExecution' and 'SettlementExecution' differ in

ProposeExecution	$User(1, 0, gasLimit, 5, ProposeFunction)$ ||| $(\|\|\| i : \{0..M-1\}$ @ $(TxPool(i)$ ||| $TxBlock(i)))$;
ListenerExecution	$User(1, 0, gasLimit, 5, ProposeFunction)$ ||| $Listener(contractOwner)$ ||| $(\|\|\| i : \{0..M-1\}$ @ $(TxPool(i)$ ||| $TxBlock(i)))$;
UnavailabeExecution	$User(contractOwner, 0, gasLimit, 5, SwitchFunction)$ ||| $(\|\|\| i : \{0..M-1\}$ @ $(TxPool(i)$ ||| $TxBlock(i)))$;
NotOwnerExecution	$User(0, 0, gasLimit, 5, FetchFunction)$ ||| $(\|\|\| i : \{0..M-1\}$ @ $(TxPool(i)$ ||| $TxBlock(i)))$;
MultipleUsersExecution	$(\|\|\| i : \{0..C-1\}$@$User(i, 0, gasLimit, 5, ProposeFunction))$ ||| $(\|\|\| i : \{0..M-1\}$ @ $(TxPool(i)$ ||| $TxBlock(i)))$;
SettlementExecution	$User(contractOwner, 200, gasLimit, 5, SettleFunction)$ ||| $(\|\|\| i : \{0..M-1\}$ @ $(TxPool(i)$ ||| $TxBlock(i)))$;

Table 2 Caption

the function parameters (*ProposeFunction*, *SwithchFunction*, *FetchFunction* and *SettelFunction* respectively). The 'ListenerExecution' adds one more process to the 'ProposeExeution'. The additonal process *Listener*(*contractOwner*) is defined as follows in Figure 19, which defines an user interface that listens to the state change.

```
Listener(addr) = listen[addr]?function -> listenerReceiving ->
                 User(addr, 0, gasLimit, 5, function);
```

Figure 19 The process Listener

The assertions that match the property with the executions are defined as follows:

- Assertion 1: *ProposerExecution deadlockfree*
- Assertion 2: *ProposerExecution* $\models$ []!*GasRunOut*
- Assertion 3: *ProposerExecution reaches sameBlockNumEventually*
- Assertion 4: *ListenerExecution* $\models$ *proposalReceived* $\rightarrow$ *listenerReceiving*
- Assertion 5: *UnavailableExecution* $\models$ []!*proposalReceived*
- Assertion 6: *NotOwnerExecution* $\models$ []!*fetch*
- Assertion 7: *MultipleUsersExecution reaches sameBlockNumEventually*
- Assertion 8: *SettlementExecution reaches receiveSettlement*

The Assertion 1 states that the execution *ProposerExecution* is deadlockfree. The Assertion 2 ensures that each transaction has enough gas to be executed. The symbol '|=' reads as "satisfies", which states that the execution satisfies the claimed LTL property on the right hand side. The symbol "[]" reads as "globally" meaning that every state in the system execution shall satisfy the subsequent LTL formula. The Assertion 3 states that in the execution *ProposerExecution*, each miner reaches the same block. The Assertion 4 ensures that if the proposal is received by the smart contract, then the listener must have received the proposal. The Assertion 5 says that if the smart contract is turned off, then the proposal will not be received. The Assertion 6 states that when a user, who is not the smart contract owner, fetches the proposal, she will never succeed. The Assertion 7 ensures that when there are multiple users, the execution can still reach the same block eventually. The Assertion 8 means that when the smart contract owner seals a deal, the deal must have been settled.

5.2 Experiment settings

For the simplicity of verification, we assume the following settings: there are 5 hoteliers or travelers and 2 miners; and the length of blockchain is 5 with the maximally 20 transactions in a block. The maximum gas limit for a block is set to be a large number 20000, which is the gas limit for a transaction times the maximum number of transaction in a block. We set the reward for successfully appending the valid block to be a large number as well, so that there is enough Ether and Gas to perform the functions. We assume the gas consumption of each functions (*FETCH*, *TRANSFER* and *UPDATE*) is an integer. We define asynchronised channels with buffer size of 5 to model the network delays. In addition, we set smart contract address and the smart contract's owner address as integers as well. Furthermore, we set the gas consumption limit, miner's selection of gas price per transaction, the pool size to receive transactions for each miner and pool size for received proposal, and the data size as shown in Figure 20.

```
#define C 5; // # of hoteliers or travelers
#define M 2; // # of miners
#define chainSize 5; // The length of each blockchain
#define maxTx 20; // The maximum number of transaction for each block
#define blkGasLimit 20000; // The maximum gas limit for each block, which is
calculated from a bunch of transactions with gas limit
#define channelBufferSize 5;
#define succAppendPrice 1000000; // reward for successfully appending the
valid block
#define contractAddr 9494; // smart contract address
#define FETCH 10; // gas consumption in terms of opcode
#define TRANSFER 500; // gas consumption in terms of opcode
#define UPDATE 100; // gas consumption in terms of opcode
#define gasLimit 1000; // gas consumption limit per transaction
#define gasPriceCondition 3; // miner's selection of gas price per transaction
#define contractOwner 2; // the smart contract's owner address
#define poolSize 100; // pool size to receive transactions for each miner
#define proposalPoolSize 10; // pool size for received proposals
#define dataSize 100;
```

Figure 20 The experiment settings

5.3 Verification results

Our experiment shows that the outcome of the assertions from PAT with respect to different configurations of C users and M miners. We can see from Table 3 that the properties are all satisfied when there are 5 users and 2 miners. However, when C and M is set to be larger, the state space is too large to be verified. The states and transactions increase exponentially as shown in Table 4 and Table 5.

Assertion	**States**	**Transitions**	**Time**(s)	**Result**
Assertion 1	8328	18823	1.76	Valid
Assertion 2	13413	30563	2.68	Valid
Assertion 3	285	336	0.04	Valid
Assertion 4	13413	30563	2.76	Valid
Assertion 5	13413	30563	2.73	Valid
Assertion 6	4537	11940	0.78	Valid
Assertion 7	504	521	0.05	Valid
Assertion 8	362	605	0.06	Valid

Table 3 The assertion results with $C = 5$ and $M = 2$

Assertion	States	Transitions	Time(s)	Result
Assertion 1	6774622	68280939	3647.43	Incomplete
Assertion 2	169626	2165266	112.52	Incomplete
Assertion 3	297717	415836	21.45	Valid
Assertion 4	230685	2897974	415.428	Incomplete
Assertion 5	169929	2168902	111.65	Incomplete
Assertion 6	317081	1511802	189.67	Incomplete
Assertion 7	10312	10425	0.98	Valid
Assertion 8	3313604	32332261	1607.98	Incomplete

Table 4 The assertion results with $C = 5$ and $M = 10$

Assertion	States	Transitions	Time(s)	Result
Assertion 1	512033	523985	201.99	Incomplete
Assertion 2	12140	1240978	194.12	Incomplete
Assertion 3	1005567	1034176	659.31	Incomplete
Assertion 4	15906	1500982	124.36	Incomplete
Assertion 5	15972	1507186	130.03	Incomplete
Assertion 6	181892	907271	510.04	Incomplete
Assertion 7	239832	240425	102.25	Valid
Assertion 8	650388	667096	224.03	Incomplete

Table 5 The assertion results with $C = 5$ and $M = 50$

6 System implementation

We have implemented a prototype system (available at https://github.com/naipengdong/Knowledge_engineering_Review). It contains 3 main components:

- Hotel HTML Page (Hotel_Proposals.html)
- Customer HTML Page (requester.html)
- Hotel Smart Contract (Hotel.sol)

Hotel and Customer HTML Page contains web3.js javascript that interacts with the Hotel Smart Contract. Hotel and Customer HTML Page will each have their own account - the Hotel has an account: $0xfdd7aa0b1bcbb77df50b9ef2a5559808d07396d7$ and the customer has an account the account: $0x1c5f708395f2c13cb3471ea3e4332b1b3a6408ca$.

The implementation of the Hotel Contract DApp demonstrates the use case where

1. A traveler is able to set up a contract.
2. A Hotelier is able to view the contract.
3. A Hotelier is able to propose an offer.
4. The Customer is able to pay with Fiat money or Ether.
5. Payment will be withheld in Contract.
6. Contract will release payment to Hotel once Customer checks out.
7. Hotel and Customer are both able to view past transactions.

7 Conclusions

In this work, we designed a decentralised booking system that links the hoteliers and the travellers, based on the blockchain techniques. The blockchain network replaces the traditional centralised travel agencies that are often inefficient and costly. In addition, it makes the hotel booking business more transparent and avoids the price tricks that can be played by the booking agencies. We have formally modelled the system design including the blockchain using a formal language and verified a set of properties of the system using a model checker. We implemented a prototype system based on Ethereum and tested its feasibility.

References

Buterin, V.: Ethereum white paper. Available at https://github.com/ethereum/wiki/wiki/White-Paper/f18902f4e7fb21dc92b37e8a0963eec4b3f4793a, visited at 15 June 2019.

CBInsights: How Blockchain Could Disrupt Insurance. Available at https://www.cbinsights.com/research/blockchain-insurance-disruption/, visited at 15 June 2019.

Cachin, C: Architecture of the Hyperledger Blockchain Fabric. Available at https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf, visited at 15 June 2019.

DHL Trend Research: Blockchain in Logistics. Available at https://www.logistics.dhl/content/dam/dhl/global/core/documents/pdf/glo-core-blockchain-trend-report.pdf, visited at 15 June 2019.

Ethereum. Available at https://www.ethereum.org/, visited at 14 June 2019.

Marr. B: How Blockchain Will Transform The Supply Chain And Logistics Industry. Available at https://www.forbes.com/sites/bernardmarr/2018/03/23/how-blockchain-will-transform-the-supply-chain-and-logistics-industry/#50e51e6d5fec, visited at 15 June 2019.

Morris, D. Z: Bitcoin is not just digital currency. It's Napster for finance. Available at http://fortune.com/2014/01/21/bitcoin-is-not-just-digital-currency-its-napster-for-finance/, visited at 15 June 2019.

Modi, R: Introduction to Blockchain, Ethererum and Samrt Contracts. Available at https://medium.com/coinmonks/https-medium-com-ritesh-modi-solidity-chapter1-63dfaff08a11, visited at 14 June 2019.

Nakamoto. S.: Bitcoin: A Peer-to-Peer Electronic Cash System. Available at https://bitcoin.org/bitcoin.pdf, visited at 15 June 2019.

PAT. Available at pat.comp.nus.edu.sg, visited at 15 June 2019.

Sun, J., Liu, Y., Dong, J.S., Pang, J.: Pat: Towards flexible verification under fairness. In: International Conference on Computer Aided Verification. pp. 709 714. Springer (2009).

Sarasola. M.R.: So maybe you figured out what blockchain is but what can you do with it? Available at https://www.willistowerswatson.com/en-SG/insights/2018/06/emphasis-blockchain-use-in-insurance-from-theory-to-reality, visited at 15 June 2019.

Schwartz, D., Youngs, N., Britto, A.: The ripple protocol consensus algorithm. Ripple Labs Inc White Paper 5 (2014).

Swan, M.: Blockchain: Blueprint for a new economy. OReilly Media, Inc. (2015)