The Knowledge Engineering Review, Vol. 35, e17, 1 of 21.
doi:10.1017/S0269888920000260

A blockchain-based decentralized booking system

NAIPENG DONG[1,2], GUANGDONG BAI[1], LUNG-CHEN HUANG[2], EDMUND KOK HENG LIM[2] and JIN SONG DONG[2,3]

[1]*School of Information Technology and Electrical Engineering, University of Queensland, General Purpose South Building (Building* 78*), St Lucia Campus, University of Queensland, Brisbane QLD,* 4072*, Australia*
e-mails: n.dong@uq.edu.au, g.bai@uq.edu.au
[2]*School of Computing, National University of Singapore, COM1,* 13 *Computing Drive,* 117417*, Singapore*
e-mails: lungchenhuang@u.nus.edu, e0335737@u.nus.edu
[3]*School of Information and Communication Technology, Griffith University, N*44 2.28*,* 170 *Kessels Road Nathan, QLD,* 4111*, Australia*
e-mail: dcsdjs@nus.edu.sg

Abstract

Blockchain technology has rapidly emerged as a decentralized trusted network to replace the traditional centralized 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 decentralized 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# language and verified using the model checker Process Analysis Toolkit. We have implemented a prototype decentralized 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 applications to use them, for example, in the logistics (DHL, 2019; Marr, 2019) and insurance industry (CBInsights, 2019; Sarasola, 2019), its full potential is yet to be exploited in other domains. To give a more concrete example of using the smart contract, we design a fundamental architecture of a decentralized hotel booking system.

Most travellers would usually spear no effort in booking for a suitable hotel room by browsing through pages of entries on online travel agencies such as *Booking.com* or *Agoda.com*. This can be both tedious and 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. 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 the traveller.

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

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

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

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 and Chen *et al*., 2009), as CSP# integrates the high-level modelling operators of CSP with low-level procedural codes in C# language and supports custom data structures, which will be convenient to represent the behaviours 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) (Sun and Pang *et al*., 2009), 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 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 an Ethereum virtual machine (EVM). The booking system contains three components: the hotelier, the traveller and the smart contract. The EVM mainly contains the EVM nodes and the miner nodes. We use *Truffle* to compile 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

The system design is formally modelled in the CSP# language, and its verification is supported by the automatic model checker PAT. The syntax of this language and the model checker PAT are briefly introduced in the first subsection. Following that, we briefly introduce some concepts in blockchain and smart contract. The EVM, which our implementation is based on, is described in the final subsection.

2.1 CSP# and process analysis toolkit

The CSP# is a rich modelling language that contains both high-level modelling operators in the traditional CSP language and programmer-favoured 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# are defined as follows (in Figure 1).

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

The deadlock process is *Stop*, meaning that the process does absolutely nothing. Process *Skip* means that the process terminates 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 an observation) and then behaves as P. Process $e\{program\} \rightarrow P$ is similar to event prefixing. The difference is that the $\{program\}$ 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 chooses either 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 *if b then P else Q* is straightforward: if b is true, the process behaves as P; otherwise behaves as Q. Both $P||Q$ and $P|||Q$ model that processes P and Q run in parallel. The difference is that the former one requires P and Q to synchronize on the shared events, whereas in the latter 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, for example partial order reduction, symmetry reduction, process counter abstraction and parallel model checking (PAT, 2019).

2.2 Blockchain and smart contract

Proposed in the digital currency—bitcoin, blockchain technique is initially used as a way to replace the central bank to mint digital coins with the ability to prevent double spending (as long as the majority of participants called miners are honest). Following bitcoin, a large number of decentralized digital currencies have been invented. Essentially, blockchain relies on a peer-to-peer network with distributed nodes. Each node holds a copy of the ledger of the currency transactions, and thus even when some nodes (minority) malfunction, the ledger is still safe, since majority of nodes keep the correct ledger. The blockchain mechanism also uses cryptography to ensure its tamper-resistance, for example using digital signature to ensure that every transaction has its owner, and using hash pointer to ensure no transaction is tampered (for more details, see Narayanan *et al*., 2016).

With the popularity of blockchain, it has been discovered that blockchain can be useful in a wide range of applications other than digital currency. To facilitate building applications on top of blockchain, a few platforms have been developed. The most well-known platform is Ethereum, which provides a language *Solidity* to allow developers to program the agreements between participants on the blockchain. These agreements are called smart contracts. The execution of the smart contracts that are deployed on the blockchain will be enforced, as the execution (presented as state change of the smart contract) is logged and accepted by the majority of the nodes. Thus, whenever a condition/state of the contract is reached, the corresponding actions will be enforced, and no one can modify it (for details, see Chinchilla, 2019).

2.3 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 cryptocurrency *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 decentralized.

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

- 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 with 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.
- User 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 decentralized apps.

The underlying blockchain consensus was initially proof-of-work (PoW) protocols using a memory-bound puzzle. Due to the criticism of PoW, for example energy waste and mining centralization, Ethereum is moving from PoW to proof-of-stake (PoS). Before moving to pure PoS, the two consensus PoW and PoS both exist in Ethereum. In this work, the PoW concept is used to illustrate how the blockchain works. Which consensus is used in EVM will not influence the booking system, which is on a higher abstract level. We highlight that the modelling of blockchain is also general and can be adopted to PoS.

3 System design

Compared to some existing blockchain-based booking solutions, such as Locktrip (2020), Winding Tree (2020) and GOeureka (2020), which build their own blockchain and sell their own tokens, this work chooses to build the system based on existing platform, aiming only to illustrate the design. Among all the platforms that support Decentralized applications, Ethereum is the most well-known one with convenient development language and tool support.

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 traveller behaviour and the hotelier behaviour. To more clearly illustrate the workings of our system, the system sequence diagram in Figure 2 describes how travellers and hoteliers interact on the decentralized booking platform.

Once the smart contract is deployed on the EVM, a traveller 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 traveller accepts the offer, he settles the request and then pays the deposit. On receiving the deposit payment, the hotelier change the room into unavailable and informs the smart contract. Finally, the traveller will then complete and close 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 traveller or hotelier who sends a transaction by invoking a function that includes the address of the user, function name, gas and gas price. A traveller 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.

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

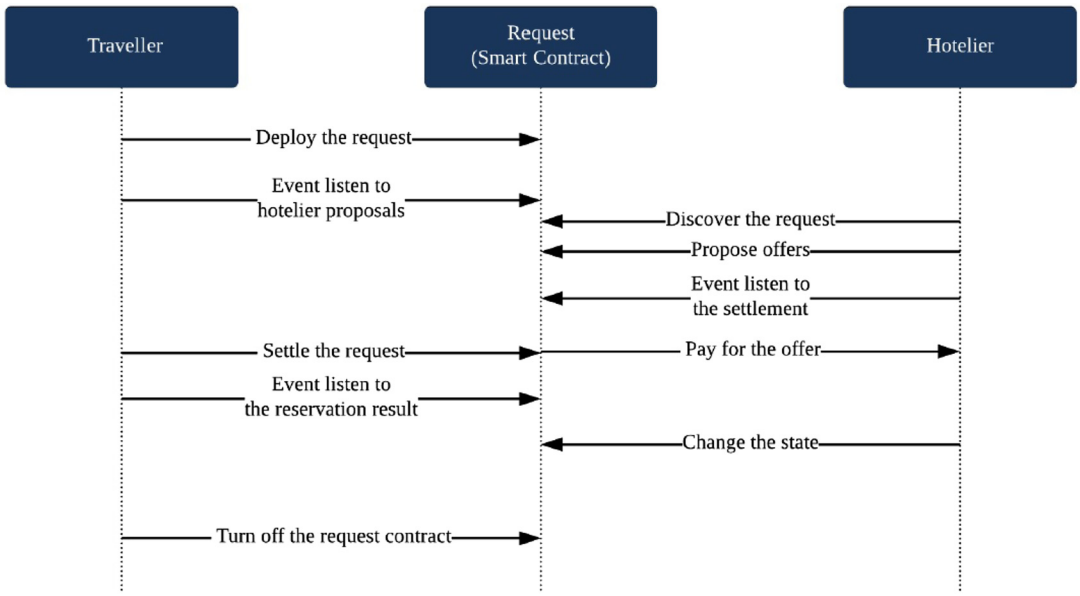

Figure 2 Overview of the system design

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

Figure 3 Constants declaration

- 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.

4 System modelling

We model the entire system in the formal language CSP# as it is more comprehensive and precise.

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.

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

4.1 User

A user, with an address *addr*, executes *function* and pays a fund *val* plus some gas amount *gas* with price *gasPrice*. This is modelled as a process *User*(*addr, val, gas, gasPrice, function*) (*u*1 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.

To call the function, the user broadcasts the transaction to miners. For simplicity, we assume the user sends the transactions to all miners in order starting from miner 0 (*u*2 in Figure 4), 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*) (*u*3 in Figure 4). When the miner index *i* is correct (i.e. *i* is smaller than the maximum number of miners—*u*5 in Figure 4), this process sends a transaction to

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

```
u1 User(addr, val, gas, gasPrice, function) =
u2            TxBrdcst(0, addr, val, gas, gasPrice, function);

u3 TxBrdcst(i, addr, val, gas, gasPrice, function) =
u4  atomic{
u5    if (i < M) {
u6      mnet[i]!addr.val.gas.gasPrice.function ->
u7      TxBrdcst(i+1, addr, val, gas, gasPrice, function) }
u8    else {
u9      Skip } };
```

Figure 4 The user process

```
p1 TxPool(i) =
p2  mnet[i]?addr.val.gas.gasPrice.function ->
p3  if (userWallet[i][addr] >= (val + (gas * gasPrice)) && gas > 0)
p4        {TxInsert(i, addr, val, gas, gasPrice, function) }
p5  else {TxPool(i) };
```

Figure 5 The transaction pool process

the i-th miner with the parameters *addr, val, gas, gasPrice, function* ($u6$), and then continues to send the transaction to the next miner $i + 1$ ($u7$). If the miner index reaches the top M ($u8$), meaning that the user has sent to every miner, then the process terminates with *Skip* ($u9$). The key word *atomic* ($u4$) prevents interleaving actions from the other processes and ensures that the sending action is atomic.

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) ($p1$ in Figure 5). On receiving a transaction with parameters, the miner first checks whether the sender, that is the user with address addr, has enough funds and *gas* ($p2$); if so, the miner inserts the transaction into his transaction pool ($p4$), which is modelled by the process *TxInsert*(*i, addr, val, gas, gasPrice, function*) defined in Figure 6); otherwise, the miner keeps waiting for another transaction, that is returning to process *TxPool*(i) (p5 in Figure 5).

When inserting a transaction into a pool of miner i (Figure 6), the miner first checks whether the pool is full ($t2$), that is 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 ($t6$), the value ($t7$), the gas ($t8$), the gasPrice ($t9$) and the function ($t10$) to the pool of miner i. At the end, we update the pointer of the transaction pool to the next empty position ($t11$). After updating the transaction pool, the miner sends a signal of *newcomer* to the channel *datach* for miner i to activate the block forming ($t12$). Finally the process goes back to *TxPool*(i) ($t13$) 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 ($t3$). If the current position is not empty ($t14$), the process moves to the next position ($t15$) and tries to insert the transaction to the pool ($t16$). If the transaction pool is full ($t17$), the transaction pointer is reset to 0 ($t18$) and then the transaction is inserted into the transaction pool for the next block ($t19$). 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. However, in our experiment, the total transactions are less than the maximum transitions in a pool, so this will not happen.

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

```
t1 TxInsert(i, addr, val, gas, gasPrice, function) =
t2  if (poolPtr[i] < poolSize) {
t3    if (poolTxGas[i][poolPtr[i]] == 0) {
t4      txReceive{
t5        var ptr = poolPtr[i];
t6        poolTxAddr[i][ptr] = addr;
t7        poolTxVal[i][ptr] = val;
t8        poolTxGas[i][ptr] = gas;
t9        poolTxGasPrice[i][ptr] = gasPrice;
t10        poolTxFunction[i][ptr] = function;
t11        poolPtr[i]++;} ->
t12      datach[i]!newcomer ->
t13      TxPool(i) }
t14    else {
t15      next{poolPtr[i]++;} ->
t16      TxInsert(i, addr, val, gas, gasPrice, function) } }
t17  else {
t18    next{poolPtr[i] = 0;} ->
t19    TxInsert(i, addr, val, gas, gasPrice, function)};
```

Figure 6 The transaction insertion process

4.2.2 Process TxBlock

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 ($b2$) and the channel for receiving transactions is not empty ($b3$), the process receives a signal that a new transaction has been inserted into the transaction pool ($b4 - b35$). If the pool is full, the process sets the pointer *inPoolPtr*(i) to 0 and goes back to the process *TxBlock*(i) ($b36$). When the pool is not full yet but there is no further transactions from the channel, as long as there are some transactions, the process starts mining ($b34$); Otherwise, the process goes back to *TxBlock*(i) and wait for transactions to be included ($b35$).

On receiving the signal ($b4$), the miner checks whether the gas for this transaction is 0 ($b5$), 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 ($b6$), meaning that all the transactions have been included in the block, the miner starts mining the block by calling Miner(i, 0) ($b7$). 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) ($b8$).
- If there are transactions in the pool ($b9$), the miner checks whether the total gas for the current block reaches the top limit ($b10$) and checks whether the current transaction in the pool provides enough gas to execute the corresponding function in the transaction ($b11$). If so, the miner includes the transaction in the block ($b12 - b27$), by copying the address ($b15$), value ($b16$), gas ($b17$), gas price ($b18$) and function ($b19$) into the block. In addition, the miner updates the total gas of the block ($b20$) 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 ($b21 - b25$). Furthermore, the pointer in the pool will be moved to the next and the size of the block is increased by 1 ($b26 - b27$). After updating the current transaction in the pool, the process goes back to *TxBlock*(i) ($b28$) to include the next transaction into the pool, until all transactions have been included, that is 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 ($b29 - b30$). If the total gas has reached the top limit and there are transactions in the block, the miner starts mining ($b31$). If the total gas reaches the top, but somehow there is no transaction in the block, the process goes back ($b33$).

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

```
b1 TxBlock(i) =
b2  if (inPoolPtr[i] < poolSize) {
b3    if (!call(cempty, datach[i])) {
b4      datach[i]?newcomer ->
b5      if (poolTxGas[i][inPoolPtr[i]] == 0) {
b6        if (blkSize[i] >= 1) {
b7          Miner(i, 0)}
b8        else { TxBlock(i) } }
b9      else {
b10        if (blkTotalGas[i] + poolTxGas[i][inPoolPtr[i]] <= blkGasLimit) {
b11          if (poolTxGasPrice[i][inPoolPtr[i]] >= gasPriceCondition) {
b12            txInclude{
b13              var pIter = inPoolPtr[i];
b14              var bIter = blkSize[i];
b15              blkTxAddr[i][bIter] = poolTxAddr[i][pIter];
b16              blkTxVal[i][bIter] = poolTxVal[i][pIter];
b17              blkTxGas[i][bIter] = poolTxGas[i][pIter];
b18              blkTxGasPrice[i][bIter] = poolTxGasPrice[i][pIter];
b19              blkTxFunction[i][bIter] = poolTxFunction[i][pIter];
b20              blkTotalGas[i] = blkTotalGas[i] + poolTxGas[i][pIter];
b21              poolTxAddr[i][pIter] = 0;
b22              poolTxVal[i][pIter] = 0;
b23              poolTxGas[i][pIter] = 0;
b24              poolTxGasPrice[i][pIter] = 0;
b25              poolTxFunction[i][pIter] = 0;
b26              inPoolPtr[i]++;
b27              blkSize[i]++;} ->
b28            TxBlock(i)}
b29          else {nextTx{inPoolPtr[i]++;} ->
b30         TxBlock(i)}}
b31       else if (blkSize[i] >= 1) {Miner(i, 0)}
b32       else {nextTx{inPoolPtr[i]++;}->
b33       TxBlock(i)}}}
b34     else if (blkSize[i] >= 1) {Miner(i, 0)}
b35     else { TxBlock(i) }}
b36  else {nextTx{inPoolPtr[i] = 0;} -> TxBlock(i)};
```

Figure 7 The process of including transactions into a block

```
m1 Miner(i, iter) = if (iter < blkSize[i]) {LockUp(i, iter)}
m2                  else { BlkUpdate(i, 0)};
```

Figure 8 The miner process

The mining process *Miner*(*i, iter*) in Figure 8 starts with a checking on whether the block index *iter* 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 (*m*1). If not, that is the functions have been fully executed, the block will be updated in process *BlkUpdate*(*i*, 0) (*m*2).

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 ($l4 - l7$ in Figure 9). The remaining amount for the user with address

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

```
l1 LockUp(i, iter) =
l2  weiLockUp{
l3    var addr = blkTxAddr[i][iter];
l4    var val = blkTxVal[i][iter];
l5    var gas = blkTxGas[i][iter];
l6    var gasPrice = blkTxGasPrice[i][iter];
l7    var lockedTotal = val + (gas * gasPrice);
l8    userWallet[i][addr] = userWallet[i][addr] - lockedTotal;
l9    lockedWallet[i] = lockedTotal;} ->
l10  TxExec(i, iter);
```

Figure 9 The lock up process

```
e1 TxExec(i, iter) =
e2  case {
e3    blkTxFunction[i][iter] == SwitchFunction:
e4      if (contractOwner == blkTxAddr[i][iter]) {
e5        GasConsume(i, iter, UPDATE) || Execution(i, iter)}
e6      else { LockedReturn(i, iter, false) }
e7    blkTxFunction[i][iter] == ProposeFunction:
e8      if (contractSwitch[i] == on) {
e9        GasConsume(i, iter, UPDATE) || Execution(i, iter)}
e10      else {LockedReturn(i, iter, false)}
e11    blkTxFunction[i][iter] == SettleFunction:
e12      if (contractOwner == blkTxAddr[i][iter]) {
e13        GasConsume(i, iter, UPDATE + TRANSFER) || Execution(i, iter)}
e14      else {LockedReturn(i, iter, false)}
e15    blkTxFunction[i][iter] == FetchFunction:
e16      if (contractOwner == blkTxAddr[i][iter]) {
e17        GasConsume(i, iter, FETCH) || Execution(i, iter)}
e18      else {LockedReturn(i, iter, false)}};
```

Figure 10 The overall process of transaction execution

addr (*l*3) is also calculated by deducting the locked amount (*l*8). Once the amount for a transaction *iter* is locked (*l*9), the miner executes the transaction by calling *TxExec*(*i*, *iter*) (*l*10).

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

- When *SwitchFunction* is called (*e*3), the miner checks whether the user who calls the function is the contract owner (*e*4). If so, the miner executes the transaction and updates the gas with price of *UPDATE* (*e*5); Otherwise returns false to indicate that the function is not successfully executed (*e*6).
- When *ProposeFucntion* is called (*e*7), the miner checks whether the contract is switched on (*e*8). If so, the miner updates the gas with price of *UPDATE* and executes the transaction (*e*9); Otherwise returns false (*e*10).
- When *SettleFunction* is called (*e*11), the miner checks whether the user who calls the function is the contract owner (*e*12). If so, the miner executes the transaction and update the gas with price of *UPDATE* + *TRANSFER* (*e*13); Otherwise returns false (*e*14). Differing from the previous two functions, this function costs more, including the updating fee and the transferring fee.
- When *FetchFunction* is called (*e*15), the miner checks whether the user who calls the function is the contract owner (*e*16). If so, the miner executes the transaction and updates the gas with the price of *FETCH* (*e*17); Otherwise returns false (*e*18).

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

```
g1 GasConsume(i, iter, opcode) =
g2  if (estimatedGas[i][iter] + opcode > blkTxGas[i][iter]) {
g3              ExecFail(i, iter)}
g4  else {
g5    consuming{
g6      var gasPrice = blkTxGasPrice[i][iter];
g7      var price = opcode * gasPrice;
g8      minerCoinbase[i] = minerCoinbase[i] + price;
g9      lockedWallet[i] = lockedWallet[i] - price;
g10      estimatedGas[i][iter] = estimatedGas[i][iter] + opcode;} ->
g11      consumed ->
g12      Skip};

g13 ExecFail(i, iter) =
g14  lockedReset{ lockedWallet[i] = 0;} ->
g15  BlkDetect(i, iter, false);
```

Figure 11 The gas update process

In any function call, before executing the functions in the transaction, the miner first checks whether the transaction has enough gas ($e5$, $e9$, $e13$, $e17$). This is modelled in Figure 11 ($g2$). If not, the execution fails, as modelled in the process *ExecFail*(*i, iter*) ($g3$, $g13 - g15$). 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 ($g5 - g10$). Then the miner initiates the execution, modelled by the synchronized event *consumed* ($g11$).

Once the event *consumed* is synced ($c1$), the miner executes the function ($c2 - c10$). We model the execution of a function *Execution*(i, iter) by giving the transaction a unique ID ($c3$) and storing the transaction owner ($c4$), contract address ($c5$), transaction function ($c6$) and transaction fee in a set of arrays ($c9$) 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*) ($c11$). Then the miner returns the remaining locked gas to the user ($r1 - r6$). Before the miner executes the next transaction, she checks whether there is any block from another miner ($r7$). If so, the miner chooses to execute the first block she receives, modelled in process *BlkDetect*(*i, iter, success*) ($d1 - d3$).

Recall that the miner process starts with a check on whether the block index is smaller than the block size of the miner i ($m2$ in Figure 8). If not, that is the transaction pool for the miner is full, the miner updates the block ($a2 - a10$), rewards the miner ($a11 - a12$) and broadcasts the block ($a13$), which is modelled in the process *BlkUpdate*(i, *iter*) in Figure 13. To broadcast a block, the miner sends the block number and block ID to other miners ($s3 - s5$ Figure 14. Note that s2 avoids that a node sends the block to himself/herself). Once this has been done, the miner increases the block ID by 1 ($s7$) and appends the block to her chain ($s8$ process *BlkAppend*(i, i, *newBlockNum, blockid*)). In the process *BlkAppend*(*i, i, newBlockNum, blockid*) defined in Figure 15, the minder appends the latest block ($k2 - k4$) and updates the state in the blockchain ($k5$).

The process of updating the blockchain actually updates the results of the functions as shown in Figure 16. When the operation is *FetchFuction* ($h4$), we add a fetch event to denote it ($h5$); When the operation is *SwitchFunction* ($h7$), the process switches the smart contract from the current state *on*/*off* to the alternative state *off*/*on* ($h8 - h10$); When the operation is *ProposeFunction* ($h12$), the process executes a *propose event* ($h13 - h19$), where the proposal initiator ($h17$) and the proposal state ($h18$) are recorded; When the operation is *SettleFunction* ($h21$), the process executes a *settle* event ($h22 - h26$), which records the one who settle the deal ($h25$) and the corresponding transaction state ($h26$). Once an operation is finished, the process goes back to itself for the operation in the next transaction ($h6$, $h11$, $h20$,

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

```
c1 Execution(i, iter) = consumed ->
c2                      execution{
c3                        pendUid[i][iter] = txUid;
c4                        pendFromAddr[i][iter] = blkTxAddr[i][iter];
c5                        pendToAddr[i][iter] = contractAddr;
c6                        pendOp[i][iter] = blkTxFunction[i][iter];
c7                        pendField[i][iter] = na;
c8                        pendState[i][iter] = 1;
c9                        pendValue[i][iter] = blkTxVal[i][iter];
c10                        txUid++;} ->
c11                      LockedReturn(i, iter, true);

r1 LockedReturn(i, iter, success) =
r2  returnGas{
r3    var addr = blkTxAddr[i][iter];
r4    var return = (blkTxGas[i][iter] - estimatedGas[i][iter]) * blkTxGasPrice[i][iter];
r5    userWallet[i][addr] = userWallet[i][addr] + return;
r6    lockedWallet[i] = 0; } ->
r7  BlkDetect(i, iter, success);

d1 BlkDetect(i, iter, success) =
d2  if (call(cempty, bnet[i])) {Miner(i, iter+1)}
d3  else {bnet[i]?j.newBlockNum.blockid -> BlkAppend(i, j, newBlockNum, blockid)};
```

Figure 12 The actual execution process

```
a1 BlkUpdate(i, iter) =
a2  if (iter < blkSize[i]) {
a3    update{
a4      block[blockUid][iter] = pendUid[i][iter];
a5      txFromAddr[pendUid[i][iter]] = pendFromAddr[i][iter];
a6      txToAddr[pendUid[i][iter]] = pendToAddr[i][iter];
a7      txOp[pendUid[i][iter]] = pendOp[i][iter];
a8      txField[pendUid[i][iter]] = pendField[i][iter];
a9      txState[pendUid[i][iter]] = pendState[i][iter];} ->
a10    BlkUpdate(i, iter+1)}
a11  else {reward{minerCoinbase[i] = minerCoinbase[i] +
a12         succAppendPrice; rewardCount++;} ->
a13  BlkBrdcst(i, 0, blockNum[i], blockUid)};
```

Figure 13 The block update process

```
s1 BlkBrdcst(i, iter, newBlockNum, blockid) =
s2  if (i == iter && iter < M) {BlkBrdcst(i, iter+1, newBlockNum, blockid)}
s3  else if (iter < M) {
s4    bnet[iter]!i.newBlockNum.blockid ->
s5    BlkBrdcst(i, iter+1, newBlockNum, blockid)}
s6  else {
s7    updateBlockUid{blockUid++;} ->
s8    BlkAppend(i, i, newBlockNum, blockid)};
```

Figure 14 The block broadcasting process

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

```
k1 BlkAppend(i, j, newBlockNum, blockid) =
k2  append{chain[i][newBlockNum] = blockid;
k3         blockNum[i]++;
k4         blkSize[i] = blkSize[j];} ->
k5  ChainUpdate(i, 0, blockid);
```

Figure 15 The block appending process

```
h1 ChainUpdate(i, iter, blockid) =
h2  if (iter < blkSize[i]) {
h3    case {
h4      txOp[block[blockid][iter]] == FetchFunction:
h5        fetch ->
h6        ChainUpdate(i, iter+1, blockid)
h7      txOp[block[blockid][iter]] == SwitchFunction:
h8        switch{var id = block[blockid][iter];
h9                if (contractSwitch[i] == off) {contractSwitch[i] = on;}
h10                else {contractSwitch[i] = off;}} ->
h11        ChainUpdate(i, iter+1, blockid)
h12      txOp[block[blockid][iter]] == ProposeFunction:
h13        propose{var id = block[blockid][iter];
h14                var addr = txFromAddr[id];
h15                var data = txState[id];
h16                var ptr = proposalPtr[i];
h17                proposalFrom[i][ptr] = addr;
h18                proposal[i][ptr] = data;
h19                proposalPtr[i]++; } ->
h20        ChainUpdate(i, iter+1, blockid)
h21      txOp[block[blockid][iter]] == SettleFunction:
h22        settle{var id = block[blockid][iter];
h23               var addr = txFromAddr[id];
h24               var data = txState[id];
h25               settledWith[i] = data;
h26               userWallet[i][data] = userWallet[i][data] + txValue[id]; } ->
h27        ChainUpdate(i, iter+1, blockid)}}
h28  else {Reset(i, 0) };
```

Figure 16 The chain updating process

*h*27). 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) (*h*28).

In the reset process, the miner clears each transaction container by setting the values to 0 shown as follows in Figure 17 ($f2 - f16$). Once all the transactions in a block is cleared, the miner clears the block by setting the gas and size to 0 ($f17 - f19$) and going back to process *TxBlock* for the next round.

To summarize, the flow of the processes can be abstractly represented as in Figure 18. In the figure, the arrow $A \rightarrow B$ indicates that process A calls process B; and the dashed arrow $A \rightarrow B$ means that process A sends a message to process B.

The user initiates a transaction (process *User*) and broadcasts to miners (process *TxBrdcst*). The transaction is received by the process *TxPool*. If there is enough gas, the transaction is added to the pool in process *TxInsert*. Once the pool is full or there is no further transactions, the process *TxInsert* triggers the block forming by sending a command to the process *TxBlock*. The process forms a block and calls the

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

```
f1 Reset(i, iter) =
f2  if (iter < blkSize[i]) {clearTx{
f3                                  pendUid[i][iter] = 0;
f4                                  pendFromAddr[i][iter] = 0;
f5                                  pendToAddr[i][iter] = 0;
f6                                  pendOp[i][iter] = 0;
f7                                  pendField[i][iter] = 0;
f8                                  pendState[i][iter] = 0;
f9                                  pendValue[i][iter] = 0;
f10                                 blkTxAddr[i][iter] = 0;
f11                                 blkTxVal[i][iter] = 0;
f12                                 blkTxGas[i][iter] = 0;
f13                                 blkTxGasPrice[i][iter] = 0;
f14                                 blkTxFunction[i][iter] = 0;
f15                                 estimatedGas[i][iter] = 0;} ->
f16                           Reset(i, iter+1)}
f17  else {clearBlk{
f18                 blkTotalGas[i] = 0;
f19                 blkSize[i] = 0;} ->
f20        TxBlock(i)};
```

Figure 17 The reset process

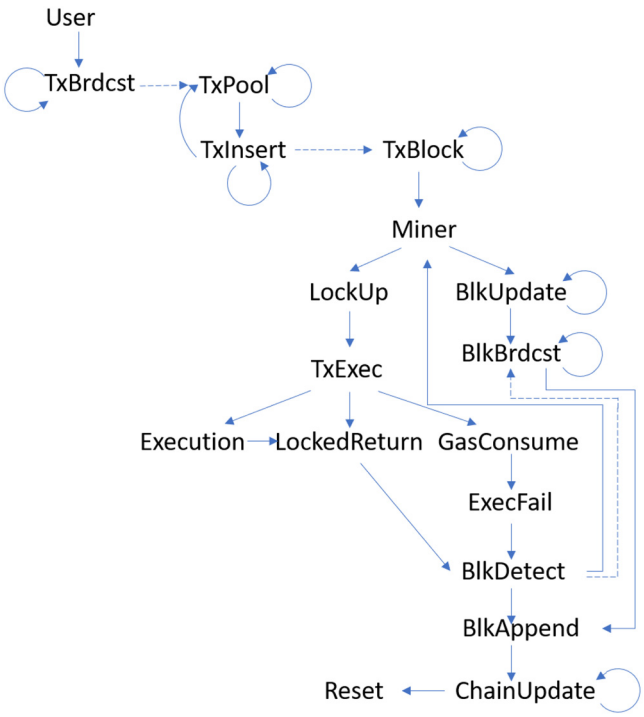

Figure 18 An overall relation between the processes

miner (process *Miner*). The miner locks up the gas (calling process *LockUp*) for each transaction; until no more transaction left, the miner updates to form a new block (process *BlkUpdate*).

- If the gas is locked, a transaction is passed to the process *TxExec* to further execute. Depending on different functions in the transaction, the process *TxExec* deducts different gas (process *GasConsume*) and executes different functions (process *Execution*). If anything wrong happens, the process *TxExec* goes to *LockedReturn* to release the locked gas. Similarly, if error happens during execution in process

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

Table 1 Formalization of the property

Property	Formalization in PAT
Deadlockfree	*deadlockfree*
GasRunOut	$estimatedGas[0][0] > gasLimit$
SameBlockNumEventually	$\&\&i : \{0..M\} @ blockNum[i] == 1$
ReceiveSettlement	$\&\&i : \{0..M\} @ settledWith[i] == 1$
ProposalReceived	$proposalPtr[0] >= 1$

```
q1 ProposerExecution = User(1, 0, gasLimit, 5, ProposeFunction) |||
q2                    (|||i:{0..M-1} @ (TxPool(i) ||| TxBlock(i)));
```

Figure 19 An overall system model with only one user

Execution, the process *LockedReturn* is called. In the process *GasConsume*, if there is no enough gas, *ExecFail* will be triggered. *ExecFail* resets the locked wallet. At this point, a new block from other miners may be sent, so the process calls the process *BlkDetect*. If a new block is received, the process appends the block to the chain *BlkAppedn* and sends a message to trigger the process *BlkBrdcst* in which new blocks will be broadcasted.

- In the case where the block is updated (process *BlkUpdate*), the block is broadcasted to others in the process *BlkBrdcst*, and the block is also appended to the chain in process *BlkAppend*.

Once the chain is appended, the process *ChainUpdate* is called by the process *BlkAppend* to update the actual chain. If it does not succeed, the whole process will be *Reset*.

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.

5 System verification

In order to show that the system design (formally modelled in CSP#) satisfies a set of desired requirements, we perform formal verification to the above formal model. 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 five properties that the system needs to satisfy, which 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 formalized as an assertion in PAT for automatic formal verification as shown in Table 1. Note that the symbol && is logical 'and'. Thus, the formula $\&\&i : \{0..M\} @ blockNum[i] == 1$

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

Table 2 Formalization of executions

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	$User(contractOwner, 200, gasLimit, 5, SettleFunction)\|\|\|$ $(\|\|\|i : \{0..M-1\} @ (TxPool(i)\|\|\|TxBlock(i)))$;

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

Figure 20 The process Listener

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 $\forall i \in \{0..M\} : settledWith[i] == 1$.

For each property, we match one execution. And the five 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.
- UnavailableExecution: 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 formalization of the executions is defined in Table 2. Note that the 'ProposeExecution' differs from the 'MultipleUsersExecution' in the number of users; the executions 'ProposeExecution', 'UnavialableExecution', 'NotOwnerExecution' and 'SettlementExecution' differ in the function parameters (*ProposeFunction, SwithchFunction, FetchFunction* and *SettelFunction* respectively). The 'ListenerExecution' adds one more process to the 'ProposeExecution'. The additional process *Listener*(*contractOwner*) is shown in Figure 20, which defines a user interface that listens to the state change.

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

- Assertion 1: *ProposerExecution Deadlockfree*
- Assertion 2: $ProposerExecution| = []!GasRunOut$
- Assertion 3: *ProposerExecution reaches SameBlockNumEventually*
- Assertion 4: $ListenerExecution| = ProposalReceived \rightarrow listenerReceiving$
- Assertion 5: $UnavailableExecution| = []!ProposalReceived$
- Assertion 6: $NotOwnerExecution| = []!Fetch$
- Assertion 7: *MultipleUsersExecution reaches SameBlockNumEventually*
- Assertion 8: *SettlementExecution reaches ReceiveSettlement*

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

```
#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 21 The experiment settings

The Assertion 1 states that the execution *ProposerExecution* is deadlock free. 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 Verification settings

For the simplicity of verification, we assume the following settings: there are five hoteliers or travellers and two miners; and the length of blockchain is five with the maximally 20 transactions in a block. The maximum gas limit for a block is set to be a large number 20 000, 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 asynchronized 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 21.

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 five users and two miners. Note that in the table, we also show the checked states, state transactions

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

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

Assertion	States	State transitions	Time(s)	Result
Assertion 1	8328	18 823	1.76	Valid
Assertion 2	13 413	30 563	2.68	Valid
Assertion 3	285	336	0.04	Valid
Assertion 4	13 413	30 563	2.76	Valid
Assertion 5	13 413	30 563	2.73	Valid
Assertion 6	4537	11 940	0.78	Valid
Assertion 7	504	521	0.05	Valid
Assertion 8	362	605	0.06	Valid

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

Assertion	States	State Transitions	Time(s)	Result
Assertion 1	6 774 622	68 280 939	3647.43	Incomplete
Assertion 2	169 626	2 165 266	112.52	Incomplete
Assertion 3	297 717	415 836	21.45	Valid
Assertion 4	230 685	2 897 974	415.428	Incomplete
Assertion 5	169 929	2 168 902	111.65	Incomplete
Assertion 6	317 081	1 511 802	189.67	Incomplete
Assertion 7	10 312	10 425	0.98	Valid
Assertion 8	3 313 604	3 233 2261	1607.98	Incomplete

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

Assertion	States	State Transitions	Time(s)	Result
Assertion 1	512 033	523 985	201.99	Incomplete
Assertion 2	12 140	1 240 978	194.12	Incomplete
Assertion 3	1 005 567	1 034 176	659.31	Incomplete
Assertion 4	15 906	1 500 982	124.36	Incomplete
Assertion 5	15 972	1 507 186	130.03	Incomplete
Assertion 6	181 892	907 271	510.04	Incomplete
Assertion 7	239 832	240 425	102.25	Valid
Assertion 8	650 388	667 096	224.03	Incomplete

and the time used for verification using PAT. Generally speaking, model checking in PAT verifies all the possible reaching states (defined by the state variables) and detects whether a bad state that some of the assertions are violated. States are connected by transitions, which defines the possible execution traces of the system behaviour. However, when C and M are set larger, the state space is too large to be verified. The states and transactions increase exponentially as shown in Tables 4 and 5. The increasing numbers of miners lead to more state variables and thus increase the states exponentially. Similarly, since the miners work in parallel with interleaving actions, the number of possible actions (i.e. the transactions) grows steeply; especially as the model includes peer-to-peer broadcasting, the increasing numbers of miners lead to exponential growth of the broadcasting messages.

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

6 System implementation

We have implemented a prototype system (https://github.com/naipengdong/Knowledge_engineering_Review), which contains three 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: 0*xfdd*7*aa*0*b*1*bcbb*77*df*50*b*9*ef*2*a*5559808d07396*d*7 and the customer has an account: 0*x*1*c*5*f*708395*f*2c13*cb*3471*ea*3*e*4332*b*1b3*a*6408*ca*.

The implementation of the Hotel Contract DApp demonstrates the application where

1. A traveller 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.

To test the implementation, we set up an Ethereum environment, including a local Ethereum network using GETH as EVM node, Truffle for compilation and deployment of contracts, XAMPP to host a HTTP webserver and Metamask plugin + Chrome to be the web3 Javascript enabled browser to talk to the EVM. In the EVM, only one miner was used. Transactions are fired every 5 seconds, for both payment and checkout actions. The number of transactions allowed per minute/second is handled by the Ethereum platform, and in the test, the rate at which the transactions were fired is 12 transactions per minute (one every 5 seconds). The mining difficulty was set as low as possible at 1. In a test run of the code, 1000 users were auto-generated with a script which simulated the users booking and purchasing packages, subsequently another script was ran to simulate checkouts in 8 hours. This generated 2002 transactions, of which 2000 transactions were from user payment and subsequent checkout, while 2 transactions were from creating the packages. Note that this test only demonstrates that the system works. It takes 8 hours to host 2000 transactions because there is the only miner deal with 1 transaction per time. By enlarging the number of miners and the block size, for example by allowing 2000 transactions per block, the system can host much more transactions. And this depends on the blockchain setting, for example the number of miners, the block size, the difficulty of mining, etc. which is beyond the application level that this system is focusing.

7 Related work

There has been a growing number and range of applications being built using blockchain as a platform/service (a.k.a. blockchain 2.0), following the well-known applications in cryptocurrencies (a.k.a. blockchain 1.0).

Blockchain-based applications. Smart contract supported by blockchain (e.g. Ethereum) enables complex processes and interactions between participants without a centralized authority. The capabilities and applications of smart contracts have been explored by both academic and industry. For instance, depending on their application domains, blockchain has been used in integrity verification, such as intellectual property management (e.g. De La Rosa *et al*., 2017), insurance (e.g. Vo *et al*., 2017) and provenance (e.g. Kim and laskowski,2016), Governance, such as citizenship (e.g. Lee, 2018), smart cities (e.g. Biswas and Muthu, 2016) and voting (Hsiao *et al*., 2018), Internet of Things (e.g. Nova, 2018), healthcare management (e.g. Patel, 2018), privacy and security (Liang *et al*., 2018), business and

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

industry, such as supply chain (e.g. IBM, 2020) and energy sector (e.g. Kyriakarakos and Papadakis, 2018), education (e.g. Bore *et al*., 2017), data management (e.g. Yang *et al*., 2018) etc. (Casino *et al*., 2019).

Blockchain in travel industry. One of the popular industry domain of blockchain-based applications is travelling, where some of the travel agency functionalities can be decentralized using blockchain, to increase transparency, reduce monopolies and prevent cheating, which are currently challenging in the travel industry. The applications include decentralized booking marketplaces, loyalty schemes, identity services, baggage tracking and travel insurance. Among them, decentralized booking has been adopted by companies like Singapore Airlines, Travelport, Webjet, etc. (Rijmenam, 2019). As ticketing is a type of contract, blockchain-based booking, including airplane booking and hotel booking, is a natural application. There have been a few blockchain-based hotel booking systems that allow clients to book without any commission fee and reduce cost, for example Locktrip (2020), Winding Tree (2020), GOeureka (2020), Atlas Atlas (2020), BTU-Hotel (French, 2019), etc. (Krietemeyer, 2020), as well as patents like the bidding system hold by Mastercard (Patent, 2020).

Evaluation on blockchain and smart contract. Most of the blockchain-based hotel booking applications, implemented as new blockchains or as smart contracts based on Ethereum, are following the try-and-error style in order to quickly enter the market; no verification or proof of security has been found on the design and implementation before deployment. This may potentially lead to various types of attacks (e.g. Chen *et al*., 2019). To evaluate the blockchain-based applications, existing software-based approaches like testing (Wang *et al*., 2018) and debugging (Lin *et al*., 2017, 2018) can be applied. For instance, static analysis tools working on the smart contract code has been proposed to detect vulnerabilities, for example OYENTE (Luu *et al*., 2016) and dynamic analysis methods, like fuzzing has been used on smart contracts (Nguyen *et al*., to appear). These methods have the advantage of efficiency but are limited in precision. The approach used in this work—formal verification of design—has the advantage of reducing the potential design flaws. For instance, formal verification has been used to analyze blockchain consensus, for example in Thin *et al*. (2018). And this work demonstrates that designs of blockchain-based applications can be first formally verified before implementation and deployment to make sure the design goals are achieved in a precise manner.

8 Conclusions and future directions

In this work, we designed a decentralized booking system that links the hoteliers and the travellers, based on the blockchain techniques. The blockchain network replaces the traditional centralized 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 demonstrated its feasibility.

Limitations and future directions. Regarding formal verification, this work is limited to certain nodes due to state space explosion. More efficient verification algorithms working on complex blockchain-based applications are needed. In addition, the attacker behaviour in the current model is limited. Advanced attacker models can be defined for not only security but also privacy properties, following the style of (Dong and Muller, 2018), and the automatic verification algorithms need to be developed with respect to the attacks, for example using logic reasoning of attacks such as (Li *et al*., 2017). In addition, the formal verification in this work only focuses on the application design; the implementation may introduce vulnerabilities; therefore, formal verification on the implementation is needed. There have been notably a few works on the source code level, for example Bhargavan *et al*. (2016), Yang and Lei (2019). However, they require the analysts to learn another language to reinterpret the applications; and the learning curve of these languages is steep. It is still a challenge to directly verify the source code for smart contracts. Regarding the application, an interesting direction is to analyze security of the deployed systems, for example using dynamic monitoring, to ensure the reliability of the application even in an untrusted environment with potentially vulnerable components.

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press

References

Atlas. *Atlas - A Universal Blockchain Platform for The Travel Industry*. https://atlas.world/viewer/whitepaper.html, visited at 9 March 2020.

Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A. Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T. & Swamy, N. 2016. Formal verification of smart contracts: Short paper. In *ACM Workshop on Programming Languages and Analysis for Security*, 91–96.

Biswas, K. & Muthukkumarasamy, V. 2016. Securing smart cities using blockchain technology. In *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 1392–1393.

Bore, N., Karumba, S., Mutahi, J., Darnell, S.S., Wayua, C. & Weldemariam, K. 2017. Towards blockchain-enabled school information Hub. In *Ninth International Conference on Information and Communication Technologies and Development*, 19.

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

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

Casino, F., Dasaklis, T. K. & Patsakis, C. 2019. A systematic literature review of blockchain-based applications: Current status, classification and open issues. *Telematics and Informatics* **36**, 55–81.

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

Chen, H., Pendleton, M., Njilla, L. & Xu, S. 2019. *A Survey on Ethereum Systems Security: Vulnerabilities, Attacks and Defenses*. https://arxiv.org/abs/1908.04507.

Chinchilla, C. 2019. *A Next-Generation Smart Contract and Decentralized Application Platform (Ethereum White Paper)*. https://github.com/ethereum/wiki/wiki/White-Paper, 2019, visited at 14 March 2020.

De La Rosa, J. L., El-Fakdi, A., Torres, V. & Amengual, X. 2017. Logo recognition by consensus for enabling blockchain implementations. *Frontiers in Artificial Intelligence and Applications* **300**, 257–262.

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

Dong, N. & Muller, T. 2018. The foul adversary: formal models. In *International Conference on Formal Engineering Methods (ICFEM)*, 37–53.

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

French, J. 2019. *BTU Protocol Launches BTU Hotel at CES, Drives Commissions to 0%*. https://blocktelegraph.io/btu-protcol-hotel-ces/, January 6, 2019, visited at 9 March 2020.

GOeureka. *GOeureka: Next-Gen Solution Shaping the Future of Online Hotel Booking*. https://goeureka.io/, visited at 9 March 2020.

Hsiao, J. H., Tso, R., Chen, C. M. & Wu, M. E. 2018. Decentralized E-voting systems based on the blockchain technology. In *Advances in Computer Science and Ubiquitous Computing*, Lecture Notes in Electrical Engineering, **474**, 305–309.

IBM Corporation. *IBM Sterling Supply Chain*. https://www.ibm.com/au-en/supply-chain, visited at 6 March 2020.

Kim, H. M. & Laskowski, M. 2016. *Towards an Ontology-Driven Blockchain Design for Supply Chain Provenance*. http://arxiv.org/abs/1610.02922, submitted 2016, visited at 6 March 2020.

Krietemeyer, M.-L. 2020. *Blockchain Technologies Influence on Hotel Bookings*. https://pdfs.semanticscholar.org/aa84/35b68db2a6f2081e877925cbf0ef3aeb7598.pdf, visited at 9 March 2020.

Kyriakarakos, G. & Papadakis, G. 2018. Microgrids for productive uses of energy in the developing world and blockchain: a promising future. *Applied Sciences (Switzerland)* **8**(4), 580.

Lee, J.-H. 2018. BIDaaS: blockchain based ID as a service. *IEEE Access* **6**, 2274–2278.

Li, L., Dong, N., Pang, N., Sun, J., Bai, G., Liu, Y. & Dong, J.S. 2017. A verification framework for stateful security protocols. In *International Conference on Formal Engineering Methods (ICFEM)*, 262–280.

Liang, G., Weller, S. R., Luo, F., Zhao, J. & Dong, Z. Y. 2018. Distributed blockchain-based data protection framework for modern power systems against cyber attacks. *IEEE Transactions on Smart Grid*. **10**, 3162–3173.

Lin, Y., Sun, J., Tran, L., Bai, G., Wang, H. & Dong, J.S. 2018. Break the dead end of dynamic slicing: localizing data and control omission bug. In *33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 509–519.

Lin, Y., Sun, J., Xue, Y., Liu, Y. & Dong, J.S. 2017. Feedback-based debugging. In *39th ACM SIGSOFT International Conference on Software Engineering (ICSE)*, 393–403.

Locktrip. *BlockChain Hotels & Rentals Travel Marketplace with 0% Commissions*. https://locktrip.com/, visited at 9 March 2020.

Luu, L. Chu, D.-H., Olickel, H., Saxena, P. & Hobor, A. 2016. Making smart contracts smarter. In *ACM SIGSAC Conference on Computer and Communications Security*, 254–269.

Marr, B. 2019. *How Blockchain Will Transform The Supply Chain And Logistics Industry*. 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. 2019. *Bitcoin is not just Digital Currency*. It's Napster for finance. http://fortune.com/2014/01/21/bitcoin-is-not-just-digital-currency-its-napster-for-finance/, visited at 15 June 2019.

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

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

Narayanan, A., Bonneau, J., Felten, E., Miller, A. & Goldfeder, S. 2016. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*, Princeton University Press.

Nguyen, D. T., Pham, L. H., Sun, J., Lin, Y. & Tran M. Q. to appear. sFuzz: an efficient adaptive fuzzer for solidity smart contracts. In *42nd International Conference on Software Engineering (ICSE)*.

Novo, O. 2018. Blockchain meets IoT: an architecture for scalable access management in IoT. *IEEE Internet of Things Journal* **5**(2), 1184–1195.

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

Patel, V. 2018. A framework for secure and decentralized sharing of medical imaging data via blockchain consensus. *Health Informatics Journal* **25**(4), 1398–1411.

Patent Application (Mastercard). *Method and System for Travel Itinerary Bidding via Blockchain (Patent US20180157999)*. https://patents.justia.com/patent/20180157999, visited at 9 March 2020.

Rijmenam, M. V. 2019. *5 Ways How Blockchain Will Change the Travel Industry*. https://vanrijmenam.nl/how-blockchain-changes-travel-industry/, August 21, 2019, visited at 9 March 2020.

Sarasola. M. R. 2019. *So Maybe You Figured Out What Blockchain is But What Can You Do With It?* 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. 2014. *The Ripple Protocol Consensus Algorithm*. Ripple Labs Inc, White Paper 5.

Sun, J., Liu, Y., Dong, J. S. & Chen, C. Q. 2009. Integrating Specification and Programs for System Modeling and Verification. In *the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 127–135.

Sun, J., Liu, Y., Dong, J. S. & Pang, J. 2019. Pat: towards flexible verification under fairness. In *International Conference on Computer Aided Verification*, 709–714. Springer.

Swan, M. 2015. *Blockchain: Blueprint for a New Economy*. OReilly Media, Inc.

Thin, W. Y. M. M., Dong, N., Bai, G. & Dong, J. S. 2018. Formal analysis of a proof-of-stake blockchain. In *23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, 197–200.

Vo, H. T., Mehedy, L., Mohania, M. & Abebe, E. 2017. Blockchain-based data management and analytics for micro-insurance applications. In *ACM on Conference on Information and Knowledge Management*, 2539–2542.

Wang, X., Sun, J., Chen, Z., Zhang, P., Wang, J. & Lin, Y. 2018. Towards optimal concolic testing. In *40th International Conference on Software Engineering (ICSE)*, 291–302.

Windingtree.com. https://windingtree.com/, visited at 9 March 2020.

Yang, C., Chen, X. & Xiang, Y. 2018. Blockchain-based publicly verifiable data deletion scheme for cloud storage. *Journal of Network and Computer Applications* **103**, 185–193.

Yang, Z. & Lei, H. 2019. Fether: an extensible definitional interpreter for smart-contract verifications in coq. *IEEE Access* **7**, 37770–37791.

https://doi.org/10.1017/S0269888920000260 Published online by Cambridge University Press