Formal Verification of Functional Requirements for Smart Contract Compositions in Supply Chain Management Systems

Sarra Alqahtani, Xinchi He, Rose Gamble, Mauricio Papa
Tandy School of Computer Science
University of Tulsa
Tulsa, OK USA
{sarra-alqahtani, xinchi-he, gamble, mauricio-papa}@utulsa.edu

Abstract

The smart contract technology has increasingly attracted the attention of different industries. However, a significant number of smart contracts deployed in practice suffer from several bugs, which enable malicious users to cause damage. The research community has shifted their focus to verifying the correctness of smart contracts using model checkers and formal verification methods. The majority of the research investigates the correctness of systems built on one smart contract. This paper proposes a verification approach for systems composed of interacting smart contracts developed and controlled by different entities. We use the NuSMV model checker and the Behavioral Interaction Priority tool to model the behaviors of smart contracts and their interactions with the aim of verifying their compliance with the systems' functional requirements. These requirements are formalized by Linear Temporal Logic propositions. The applicability of our approach is illustrated using a case study from The American Petroleum Institute and implemented using Hyperledger Fabric.

1. Introduction

Blockchain first started with the cryptocurrency Bitcoin, where it serves as a decentralized, distributed, digital ledger to record all Bitcoin transactions [1]. In the Bitcoin blockchain, there is a single layer of distributed ledgers responsible for accomplishing all computations related to transferring Bitcoins between users. In order for the blockchain to be effectively used in applications other than the cryptocurrency, two layers are needed: (1) a layer for the distributed ledgers to store immutable records of data and (2) a layer to implement and run the business logic of the application, typically called the smart contract. The distributed ledgers can be kept at each stakeholder's local server or cloud so they can verify transactions without a third party.

The mission of supply chain management (SCM) is to guarantee the provenance of items being traded. Typical supply chain systems include interactions through messages to ensure the secure exchange of information and synchronization among stakeholders. Inconsistent system logic among stakeholders can lead to inconsistent data and a lack of transparency, potentially increasing the fraud rate within the supply chain. Using the smart contract to implement the system logic can presumably ensure a transparent, secure, and immutable provenance, which could benefit all parties in the supply chain by reducing fraudulent and/or malicious behavior as it relates to the supply chain goods. The definition of the smart contract proposed in [2] refers to a digital protocol that executes promises or terms that are predefined and agreed upon by several parties. When the predefined conditions occur, the corresponding contract terms are automatically triggered. This process allows the blockchain to evaluate contract clauses without third party supervision. In their book, Tapscott and Tapscott [3] refer to using blockchain technology for the end-to-end SCM as a "Blockchain Revolution." They explain that smart contracts could enable companies to contract for price, quality, and delivery dates with just a few clicks of a mouse. However, much of the discussion of the impact of blockchain on SCM remains at a relatively conceptual level. It does not explore in detail how blockchain could technically be exploited to provide the expected functional outcomes.

One of the major challenges for using blockchain in systems like SCM is related to the smart contract design and implementation. In blockchain-based SCM systems, smart contracts are controlled and implemented by different entities using low-level languages, exacerbating the system's exposure to different types of security and safety threats. Recently, various attacks have succeeded to expose security issues, such as the DAO attack [4] that resulted in the loss of almost 60 million USD worth of Ether (the digital currency in Ethereum). Another attack is the Parity wallet bug that resulted in 169 million USD worth of Ether to be locked forever [5].

URI: https://hdl.handle.net/10125/64392
978-0-9981331-3-3

To counter the aforementioned attacks and avoid breaches, several approaches have been explored, such as documenting vulnerabilities [6, 7] or model checking using formal verification techniques and game theory [8-11]. All of these approaches focus on investigating the correctness of systems built on a single smart contract. Little attention has been paid to verifying the correctness of systems built on interacting contracts. Such a research area is especially vital for SCM systems, where different stakeholders are expected to develop their own smart contracts to interact with other contracts to accomplish different business objectives. The same challenge can be faced by ERP systems but with less risk, since the entities involved in ERP systems are usually from the same domain (i.e. internal departments of the same company).

In this paper, we design a novel approach to verify the functional requirements for blockchain-based SCM systems that are built with interacting smart contracts. The proposed approach models the detailed implementations of the interacting smart contracts into an abstract design that can be utilized to verify the compliance of the overall functional requirements of the system. This high-level verification method can help the SCM experts decide if the initial design satisfied their requirements without the need to test the low-level code of each smart contract. We show our approach using a case study based on the oil and gas supply chain [12]. Although the approach is generic and can be applied to various blockchain frameworks, we use the Hyperledger Fabric framework to demonstrate the implementation of the composed smart contracts for our case study.

This paper is organized as follows. The next section further discusses the background and related work on verification methods for blockchain smart contracts. Section 3 explains the proposed approach and discusses the investigated case study, Section 4 describes the implementation of the case study and shows the verification results. Section 5 discusses the conclusion and the future work.

2. Related Work

In this section, we present a brief overview of related work. Magazzeni et al. [13] claim that it is essential to validate and verify the smart contract before deploying it to the blockchain due to security and integrity concerns. Five questions have been pinpointed regarding the assurance of smart contract execution correctness: (1) whether the written contract correctly and fully represents the understanding and intentions of all the parties, (2) whether the smart contract correctly encodes the written contract, (3) whether the smart contract does its job, (4) whether the smart contract does anything that it was not intended to do, and (5) whether the platform holds its integrity when multiple smart contract are used. The authors suggest adopting formal verification methods with model checking to address such questions. Much of the current research focuses on answering the first four questions. Our research focuses on answering question 5, which is the correctness verification of smart contract interactions within one system.

The proposed approach in [10] models the smart contract, blockchain execution environment, and the users' behaviors based on a formal model checking language. The generated model is then used to analyze the design vulnerabilities of the smart contracts using an off-the-shelf statistical model checking tool. They prove the applicability of their approach through a simple case study of name registration smart contracts.

Nehai et al. [11] propose a verification method for an Ethereum application based on smart contracts. The proposed method has three layers: (1) kernel layer to represent the behaviors of smart contracts, (2) application layer to model the smart contract logic, and (3) environment layer to model the execution framework of Ethereum. A set of modelling rules are used to translate Solidity code for the smart contract into a model checking language. Then, the system requirements and properties written in English are formalized in a temporal logic language called Computation Tree Logic (CTL) and applied to the resulting checked model. The researchers illustrate the applicability of their approach using a case study from the energy market field.

Mavridou et al. [14] developed the VeriSolid framework as a model-based, correct-by-design approach for Ethereum Solidity smart contracts. VeriSolid can be used as a tool by smart contract developers to manually specify their requirements in the abstract form of finite state machines (FSMs).VeriSolid extracts the behavioral model of the contract from the given FSMs using the Behavior Interaction Priority (BIP) tool [15]. The system requirements are specified using CTL. The model checking tool (NuSMV) is used to verify the correctness of the BIP behavioral model and the specified CTL properties. After verifying the system design, VeriSolid generates the Solidity code for the system that can be deployed directly in the Ethereum platform.

Bhargavan et al. [9] introduce a novel framework for detecting flaws in Ethereum smart contract code by translating Solidity and Ethereum Virtual Machine (EVM) bytecode contracts into the functional programming language F*. The aforementioned approaches [9-11,13,14] are solely focused on a single smart contract. In contrast, our paper investigates

verifying the functional compliance of interacting smart contracts deployed in the Hyperledger Fabric platform and written in the GO language.

3. Formal Verification of Interacting Smart Contract

The proposed approach devised for smart contract verification is depicted in Figure 1. Our smart contracts are written in Go, which is one of the smart contract coding languages for Hyperledger Fabric besides Java and JavaScript. In step 2, a set of modeling rules are adopted to separately translate each smart contract into a finite state machine (FSM).

Then, the Behavior Interaction Priority (BIP) framework [15] is used for its strong correct-by-design feature to model the interactions between FSMs. BIP has been used for constructing several correct-by-design systems, such as robotic systems and satellite onboard software [16-17]. In order to check the behavioral compliance of the smart contract interactions with the overall system functional requirements, we use the BIP-to-NuSMV tool [18] to translate our BIP models into NuSMV language, which is the language typically used for model checking. The expected behavior (i.e. functional requirements) of the system is formalized into linear temporal logic (LTL) properties and applied to the generated NuSMV model. If any property does not hold then the result is an error message with a counter example to show the cause and source of the flaw. Otherwise, the compliance of the interacting smart contracts with their system requirements is formally proved.

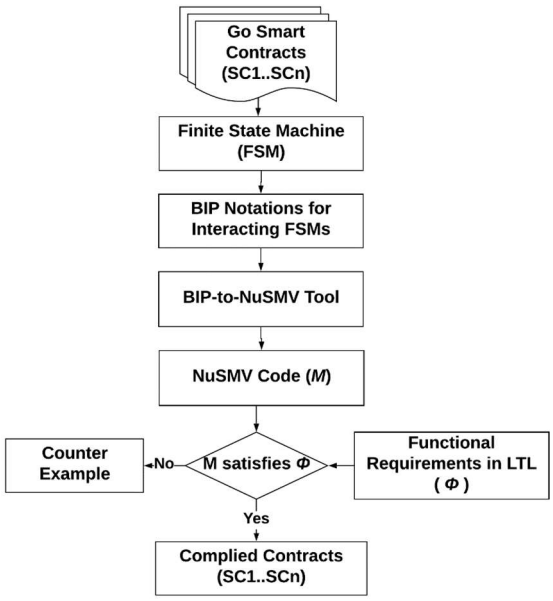

Figure 1: General approach

To illustrate our approach, we use a case study from the oil and gas supply chain [12] implemented using the Hyperledger Fabric blockchain framework. However, the proposed approach is applicable for any business system architecture built by using several smart contracts that are owned and developed by different entities in one blockchain-based system. We focus in this paper on SCM in oil and gas industry since this area falls in our research interests.

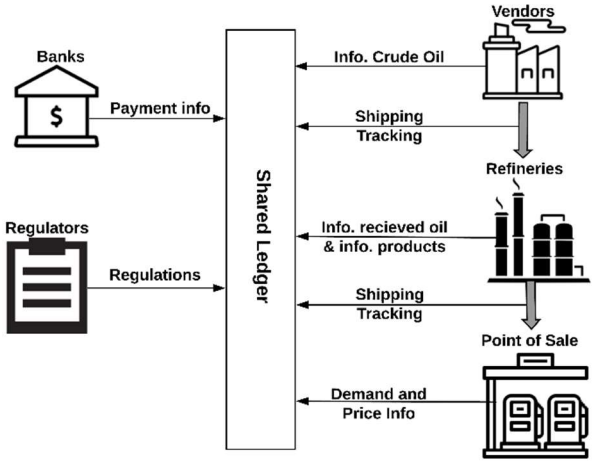

Figure 2: Blockchain Architecture of Oil and Gas Supply Chain

3.1. Fuel Supply Chain Using Composed Smart Contracts

In this section, we explain the involved smart contracts in the fuel supply chain and how their composition can be used to implement the supply chain transactions (Figure 2). The original case study is extracted from [12] then implemented on Hyperledger for the purpose of this research. We show pseudocode representations of smart contracts for three stakeholders; the supply chain, vendor, and point-of sale. We also reduce the supply chain into the above stakeholders in the rest of the paper due to the space limit. However, the same approach can be easily applied to all stakeholders mentioned in [12].

1) **Supply chain main smart contract (SCSC):** The SCSC has the core functions for the supply chain, including stakeholders' registration, fraud detection, price and production limit maintenance, and automatic payments between stakeholders once the payment conditions have been met. The pseudo code of the SCSC is shown in Algorithm 1.
2) **Vendor smart contract (VSC):** The VSC logs information about the produced oil batch in the blockchain before transferring it to the refineries,

tracks the oil batch across the supply chain, and receives the production daily limit from the SCSC. The pseudo code of VSC is shown in Algorithm 2.

3) **Transportation smart contract (TSC):** Crude oil can be transported using trains, ships, trucks, and pipelines. Assuming they all have embedded sensors to measure the quantity of shipped oil, this information can be kept in blockchain using the TSC. This logged information can be used by the SCSC to detect frauds related to stealing oil during shipping (called pipeline tapping fraud).

Algorithm 1: SCSC

```
Local variable: total_production, total_demand,
                fixed_price, production_limit,
                overcapacity_threshold
func register() (stakeholder_info)
create a record for the stakeholder
log this record in the ledger
return ID

func check_limit() (vendor_id)
v_production = aggregate_from_ledger(vendor_id)
if v_production > production_limit then
    alert(vendor_id, "Violation of production daily
      limit")
end

func check_price() (sold_batch)
if sold_batch.price != fixed_price then
    alert(sold_batch_PSO, "Violation of the gallon
      price")
end

func check_overcapacity()
total_production =
  accumulate_production_from_ledger()
total_demand = accumulate_demand_from_ledger()
if total_production − total_demand >
  overcapacity_threshold then
    update_production_limit(all_vendors)
    update_fixed_price(all_POS)
end

func check_fruad() (oil_batch)
oil_batch_array = retrieve all logged records in the
  ledger with oil_batch.id
if all amount in oil_batch_array isn't consistent then
    alert(oil_batch.vendor, "Inconsistent batches")
end

func pay() (sold_batch)
make payment from sold_batch.PSO to oil_batch_vendor
```

Algorithm 2: VSC

```
Local variable: vendor_id, production_limit
func create_batch() (info)
production = aggregate the vendor's total production
  from the ledger
if info.amount + production <= production_limit then
    oil_batch b = new oil_batch(info)
    log(b)
end

func log() (oil_batch b)
log b in the ledger
SCSC.check_limit(oil_batch)

func receive_limit() (prod_limit)
production_limit = prod_limit

func track_batch() (b.id)
retrieve batch info from the ledger
```

Algorithm 3: PoSSC

```
Local variable: pos_id, price
func create_batch() (info)
sold_batch b = new sold_batch(info)
log b in the ledger
SCSC.pay(b)
SCSC.check_price(p)

func receive_price() (p)
price = p
```

4) **Point of sale smart contract PoSSC:** The PoSSC logs information in the blockchain about the received fuel from the refineries, as well as more details about the sold fuel including the gallon price and the demand level. The pseudo code of PoSSC is shown in Algorithm 3.

   Refinery smart contract (RSC): Refineries transform crude oil into its various consumable products, such as fuel oil, diesel oil, jet fuel, and multiple essential manufacturing feedstocks. Refineries can use their smart contracts to log information about the received oil from vendors and the produced refined fuel and track their refined fuel to its point-of-sale destination.

5) **Other smart contracts:** Other smart contracts developed and owned by regulators and banks can be part of the supply chain.

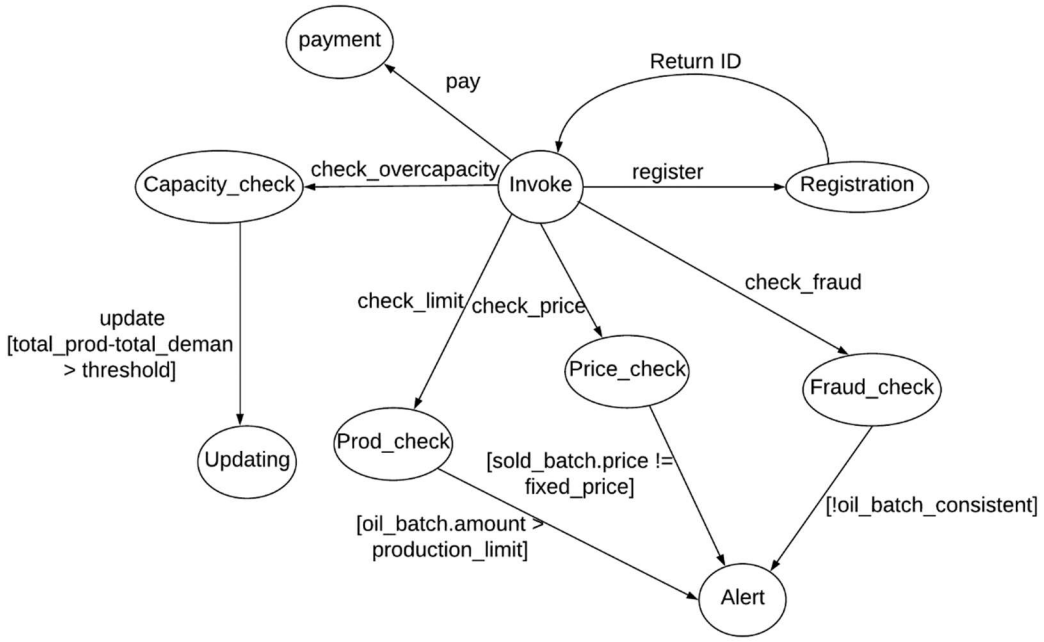

Figure 3: FSM for SCSC in Algorithm 1

3.2. Smart Contract Modeling

We represent each smart contract as a finite state machine (FSM), which comprises a set of states and a set of transitions between those states. To model a Hyperledger Fabric smart contract as an FSM, we apply a minimum set of rules [14] to simplify the overall process and reduce the errors. The rules include the following.

1- Functions should be modeled as transitions, such that invoking a transition by a user or another contract forces the contract to execute the action of the transition.
2- An action is a computation triggered by the execution of the associated transition (i.e. the function's body).
3- A guard is a predicate on variables that must be true to allow the execution of the associated transition.

Modeling the smart contract as an FSM provides an adequate level of abstraction for reasoning about the contract behavior and interaction with other contracts. Inspired by [14], we define the smart contract as an FSM in Definition 1.

Definition 1. A smart contract is a tuple (S, s_0, S_f, V, T), where

- S is a finite set of states;
- s_0 is the initial state;
- S_f is a set of final states;
- V is a set of contract variables (i.e., variable names and types);
- T is a set of transition relations, where each transition t ∈ T includes:
  - Transition name t^{name};
  - Source state $t^{from} \in S$;
  - Destination state $t^{to} \in S$;
  - Arguments $t^{input} \subset V$;
  - Transition guard t^{guard};
  - Return variables $t^{output} \subset V$.

Using Definition 1, SCSC in Algorithm 1 is modeled as a FSM in Figure 4. The transition names are identical to the function names in Algorithm 1, while guards (i.e. conditions) are placed inside square brackets. Each transition corresponds to an action that the SCSC can perform as part of the supply chain management. Please note that for better readability, not all actions are shown in Figure 3.

The SCSC has 9 states:

1- *Invoke*: The supply chain stakeholders submit their function calls to the Invoke state, which in turn directs the input towards the appropriate state based on the required transition, i.e. the called function.
2- *Registration*: The stakeholders must first register to the supply chain blockchain by giving its information.

3- *Prod_check*: When a vendor logged information about one batch of its oil production in the ledger, the SCSC automatically checks whether that vendor has already exceeded its daily limit.
4- *Price_check*: The SCSC checks if the point-of-sale has adhered to the fixed price rule set by the regulators when the POS contract logged information about the sold batch of fuel.
5- *Capacity_check*: To maintain the relation between the supply and demand of oil, SCSC checks periodically if the total production by all vendors has exceeded the demand by a specific threshold defined by the regulators.
6- *Update*: If there is an overcapacity issue, the SCSC updates the production limit and fixed price until the market becomes stable again.
7- *Fraud_check*: Whenever a new record is logged regarding an oil batch, the SCSC checks the logged amount of oil to confirm the consistency along the chain.
8- *Alert*: The SCSC triggers an alert once fraud, a violation of the production limit or the fixed price, has been detected. The alert is sent to the relevant stakeholders with an appropriate message.
9- *Payment*: This state plays the escrow role of the SCSC to transfer money between stakeholders according to their initial agreements.

3.3. Interaction Modeling Using BIP

The BIP framework offers a strong software architecture-based modeling formalism that can be used to abstract complex systems for verification purposes [18]. In this paper, we primarily adopt two concepts from BIP framework: atomic components and connectors.

An atomic component is used to model each smart contract's behavior. Each atomic component is a FSM, extended with local variables and ports. Ports represent the function names, which may be associated with variables as arguments, to be used for interacting with other components. A transition is a step, labeled by a port, from one control location to another. It may have an associated Boolean condition, called a guard, and a set of one or more actions that are computationally defined on local variables.

Connectors represent sets of ports that must be synchronously executed. For every interaction, the connector provides the guard and the data transfer to exchange data across the ports involved in the interaction between different contracts. Composite contracts are defined by associating sub-components (atomic or composite) using connectors. For more details about BIP, the reader is referred to [17, 18].

After modeling the smart contracts as FSMs, we translate each FSM into BIP atomic components. Each component transition is labeled by a port (i.e., function's name) specifying the transition's unique name. The component interface is composed of ports, which are used for interacting with other components (i.e., FSMs). The BIP notation for VSC is detailed in Figure 4 and its FSM is shown in Figure 5.

In the VSC implementation, the values of local variables in lines 3-6 are directly retrieved from the shared ledger. However, when we model smart contracts using BIP, we use local variables to store and communicate data with other contracts for the verification purposes. Ports in lines 8-11 represent the function names in the smart contract. The *Place* keyword in line 13 is used to list the available states, while *initial* in line 16 is used to identify the initial state. To create the transitions between states, keywords *on*, *from*, and *to* are used. The condition preceded by *provided* represents the guard while statements after *do* represent the actions or the function's body statement. The interactions between contracts are modelled by assembling their FSMs using connectors.

```
1  atomic type VSC
2
3    data int VendorId
4    data int limit
5    data int production
6    data int batch_id
7
8    export port ePort create_batch
9    export port ePort track_batch
10   export port onePort recieve_limit(limit)
11   export port onePort log(batch_id)
12
13   place Invoke,Batch_creating,Tracking,
14     Logging, Updating_limit
15
16   initial to Invoke
17   on create_batch from Invoke to Batch_creating
18     provided amount+production<=production_limit
19     do {production+=amount;}
20   on log from Invoke to Logging
21   on recieve_limit from Invoke to Updating_limit
22   on track_batch from Invoke to Tracking
23 end
```

Figure 4: BIP notation for VSC

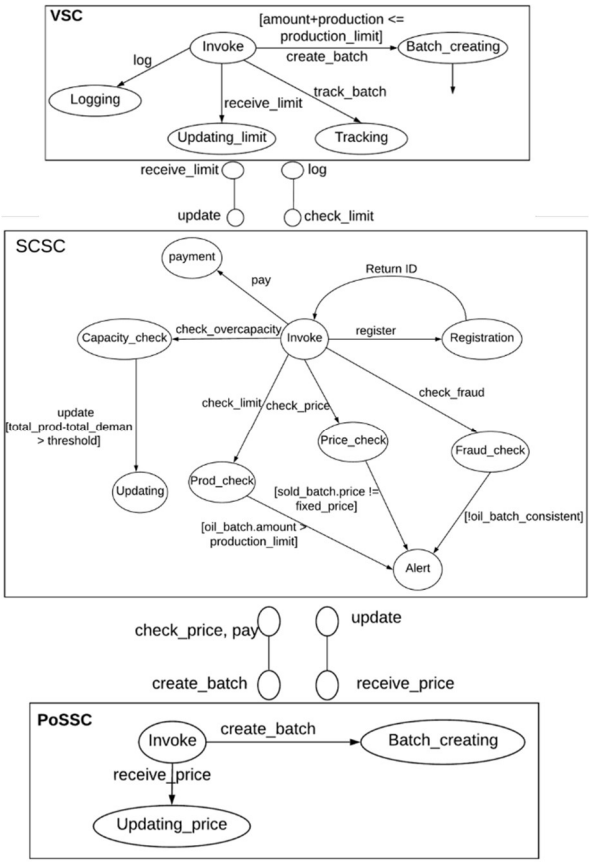

Figure 5: Interaction modeling between SCSC, VSC, PoSSC using connectors

```
compound type Fuel_supplyChain
  component SCSC scsc
  component VSC vsc
  component PoSSC possc

 // synchronization functions
connector Synch1 supply_vendor_limit(
  scsc.update, vsc.recieve_limit)
connector Synch2 supply_vendor_log(
  scsc.check_limit, vsc.log)
connector Synch2 supply_possc_price(
  scsc.update, possc.recieve_price)
connector Synch1 supply_possc_check_price(
  scsc.check_price, possc.create_batch)
connector Synch1 supply_possc_payment(
  scsc.pay, possc.create_batch)
end

/* instantiation of the root component */
component Fuel_supplyChain test

end
```

Figure 6: BIP notation for PoSSC

Connectors link ports from different contracts to represent interaction patterns between them. For every interaction, the connector provides the guard and the data transfer to exchange data across the ports involved in the interaction. We show the interaction between the supply chain main contract SCSC and the stakeholders' contracts VSC and PoSSC using connectors (circles) in Figure 5. A snippet of our BIP code to synchronize the connectors between SCSC, VSC, and PoSSC is shown Figure 6.

3.4. Model Checking Using NuSMV

We use BIP-to-NuSMV tool to generate the NuSMV code for the composed contracts in Figure 6. The functional requirements for the whole composition are tested against the NuSMV model for verification. The functional requirements are domain specific and they are usually written in English. In this paper, we use a set of simple requirements for fuel supply chain and translate them into LTL. which is the typical language used to verify the behavior of NuSMV models. Our verification approach checks whether the behavior of the composed contracts modeled by NuSMV satisfies the required LTL properties. In LTL, $\Box$ represents the safety property that must be always maintained. $\Diamond$ represents the liveness property that indicates that the following property should eventually happen. Several requirements for the fuel supply chain case study are formalized into LTL logic as follows:

Property 1: Vendors must always maintain their production under the daily limit set by the SCSC:

$$\Box (\text{VSC.production} \leq \text{SCSC.production\_limit})$$

Property 2: Vendors must always log their productions into the ledger:

$$\Box (\text{VSC.create\_batch} \Rightarrow \Diamond (\text{VSC.log}))$$

Property 3: Point-of-sales must always adhere to the price set by the SCSC:

$$\Box (\text{PoSSC.price} = \text{SCSC.fixed\_price})$$

Property 4: Once the point-of-sale receives the oil batches from a vendor, the payment must be triggered by the SCSC:

$$\Box (\text{PoSSC.create\_batch} \Rightarrow \Diamond (\text{SCSC.pay}))$$

4. Implementation and Verification Results

4.1. SCM Implementation

We implement our smart contracts following the chaincode specifications in Hyperledger Fabric [19]. Hyperledger Fabric is an open-source permissioned blockchain framework from the Linux Foundation. We inherit the blockchain network architecture from our previous work [20, 21], which have more details about the implementation and performance of the proposed architecutre applied in other applications. Each smart contract is defined with a set of typed variables and a set of functions. Every contract code must implement its own interface whose functions are called in response to received transactions.

There are two mandatory functions in Hyperledger Fabric smart contract interface. The first function is Init, which is called when the contract receives an instantiate or upgrade transaction. The Invoke function is called in response to receiving an invoke transaction to process the application functions. The Invoke function's arguments are the name of the contract application function to invoke along with its list of arguments. A snippet of Invoke function from the SCSC is shown in Figure 7.

```
1 func (t *SupplyChainMainChaincode) Invoke(
2   stub shim.ChaincodeStubInterface) pb.Response {
3   function, args := stub.GetFunctionAndParameters()
4   if function == "register" {
5     return t.register(stub, args)
6   } else if function == "check_limit" {
7     return t.checkLimit(stub, args)
8   } else if function == "check_price" {
9     return t.checkPrice(stub, args)
10  } else if function == "check_over_capacity" {
11    return t.checkOverCapacity(stub, args)
12  } else if function == "check_production_limit" {
13    return t.checkProductionLimit(stub, args)
14  } else if function == "pay" {
15    return t.pay(stub, args)
16  } else if function == "alert" {
17    return t.alert(stub, args)
18  }
19  return shim.Error(fmt.Sprintf(
20    "[ERROR] No <%s> action defined ", function))
21 }
```

Figure 7: Invoke function in Go chaincode

As shown in Figure 8, three organizations are defined: (1) vendors that produce oil, (2) point-of-sale nodes that consume and sell oil, and (3) a supply chain consortium foundation that regulates functionality and interactions throughout the entire supply chain. Hence, three smart contracts are developed based on the pseudocode for the proposed supply chain in Section 3. They are SCSC.go, VSC.go, and PoSSC.go. Hyperledger Fabric Client, a Nodejs-based library, is used to expose the blockchain network as RESTful APIs. A shell script is built so that corresponding APIs can be used to validate the deployment of the smart contracts.

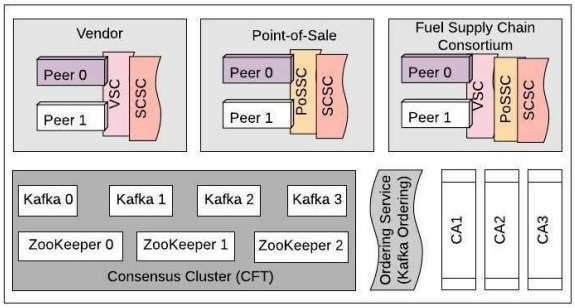

Figure 8: Hyperledger Fabric blockchain network for the oil and gas supply chain

In Hyperledger Fabric, it is a mandatory requirement to have corresponding smart contracts installed on the same peer node if there are interactions among different contracts. For example, in order for the VSC to call the SCSC, the VSC's peer node is required to have both the VSC and SCSC installed. Thus, the SCSC is installed on each peer node as it serves as the base of the overall supply chain functionality. This interaction between the contracts is required because each contract can only access its own state. Hence, the contracts involved in the supply chain need to explicitly exchange their states upon request. After verifying the design of the case study using the proposed approach, we deployed our architecture and tested the interactions between the contracts in simulated actions. Since the verification is performed offline, we did not investigate the performance of our approach. As mentioned before, the performance evaluation for the implemented architecture has been already published in our previous work [20,21] for different applications including Healthcare systems and IoT firmware updates.

4.2. Verification Results

We ran the NuSMV model checker tool on our NuSMV model and verified that the considered implementation satisfies the four stated properties. Then, we removed line 4 from the PoSSC algorithm that consequently removed the connector between PoSSC.create_batch and SCSC.pay from Figure 5. This change is expected to result in **Property 4** not being satisfied. Hence, when we re-ran the model checker using the modified PoSSC, it provided a counter example that is shown in Figure 9. We can see that in state 6 the PoSSC has created a new batch for its sold fuel but without triggering the pay function in the SCSC, which eventually falsified the functional requirement defined by **Property 4**.

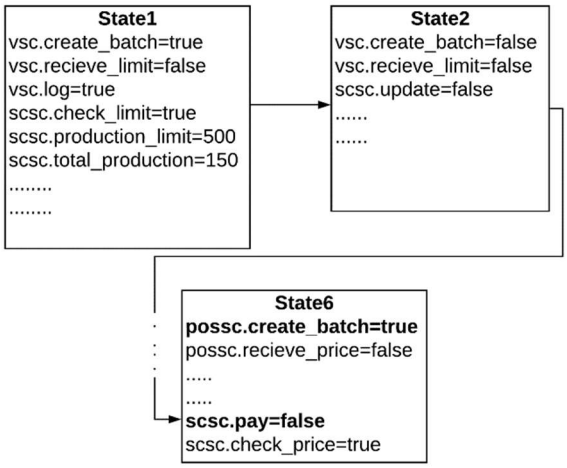

Figure 9: Counter example for Property 4

5. Conclusion and Future Work

We present a workflow approach that applies model-checking to a Hyperledger Fabric application based on interacting smart contracts. We first translated smart contracts written in Go into FSMs and then modelled their interactions using BIP. To verify behavior correctness of the smart contract composition, we used a BIP-to-NuSMV tool to translate BIP into the input language for the model checker NuSMV. The functional requirements of the behavior are specified using LTL and applied to the generated NuSMV model. We tested our approach using a supply chain system from the oil and gas industry. In future work, we plan to extend our approach to model security requirements for verification purposes. Moreover, we intend to test our approach on other blockchain platforms that have been shown to be vulnerable to security issues such as Ethereum.

6. Acknowledgement

This work was partially supported by eLynx Technologies – The University of Tulsa grant number 160115, https://www.elynxtech.com/, Tulsa, OK. We would like to thank Stephen Jackson, President and CEO of eLynx Technologies for his support.

7. References

[1] Swan, M., Blockchain: Blueprint for a New Economy, O'Reilly Media, 2015.

[2] Szabo, N., "Formalizing and Securing Relationships on Public Networks," First Monday.

[3] Tapscott, D., and A. Tapscott, Blockchain Revolution: How the Technology Behind Bitcoin Is Changing Money, Business, and the World. Portfolio; Reprint edition (May 10, 2016), p. 368.

[4] Gelvez, M. P. G., "Explaining the DAO exploit for beginners in Solidity," Available: https://medium.com/@MyPaoG/explaining-the-dao-exploit-for-beginners-in-solidity-80ee84f0d470

[5] Thomson, I., "Parity: The bug that put $169m of Ethereum on ice? Yeah, it was on the todo list for months," *The Register,* Available: https://www.theregister.co.uk/2017/11/16/parity_flaw_not_fixed/

[6] Atzei, N., M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts SoK," Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204, 2017.

[7] Delmolino, K., M. Arnett, A. Kosba, A. Miller, and E. Shi, "Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab", International Conference on Financial Cryptography and Data Security, 2016, pp. 79-94.

[8] Bigi, G., A. Bracciali, G. Meacci, and E. Tuosto, "Validation of Decentralised Smart Contracts Through Game Theory and Formal Methods," Essays Dedicated to Pierpaolo Degano on Programming Languages with Applications to Biology and Security - Volume 9465, 2015.

[9] Bhargavan , K. *et al.*, "Formal Verification of Smart Contracts: Short Paper," Proceedings of the ACM Workshop on Programming Languages and Analysis for Security, Vienna, Austria, 2016.

[10]Abdellatif, T., and K. Brousmiche, "Formal Verification of Smart Contracts Based on Users and Blockchain Behaviors Models," IFIP International Conference on New Technologies, Mobility and Security (NTMS), 2018, pp. 1-5.

[11] Nehai, Z., P.-Y. Piriou, and F. Daumas, "Model-Checking of Smart Contracts," IEEE International Conference on Blockchain, Halifax, Canada, 2018.

[12] The American Petroleum Institute (API), "Energy: Understanding Our Oil Supply Chain," Available: https://www.api.org/~/media/Files/Policy/Safety/API-Oil-Supply-Chain.pdf

[13] Magazzeni, D., P. McBurney, and W. Nash, "Validation and Verification of Smart Contracts: A Research Agenda," Computer, vol. 50, no. 9, 2017, pp. 50-57.

[14] Mavridou, A., A. Laszka, E. Stachtiari, and A. Dubey, "VeriSolid: Correct-by-Design Smart Contracts for Ethereum," arXiv.org, Available: https://arxiv.org/abs/1901.01292

[15] Bliudze, S. et al., "Formal Verification of Infinite-State BIP Models," Cham, Springer International Publishing , 2015, pp. 326-343.

[16] Basu, A. et al., "Incremental Component-Based Construction and Verification of a Robotic System," Proceedings of 18th European Conference on Artificial Intelligence, 2008.

[17] Basu, A., M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-time Components in BIP," Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods, 2006.

[18] Basu, A. *et al.*, "Rigorous Component-Based System Design Using the BIP Framework," *IEEE Software,* vol. 28, no. 3, pp. 41-48, 2011.

[19] Beckert, B., M. Herda, M. Kirsten, and J. Schiffl, " Formal Specification and Verification of Hyperledger Fabric Chaincode," 3rd Symposium on Distributed Ledger Technology (SDLT-2018) co-located with ICFEM 2018: the 20th International Conference on Formal Engineering Methods, 12 November 2018.

[20] X. He, S. Alqahtani, and R. Gamble, "Toward Privacy-Assured Health Insurance Claims," International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData), Halifax, NS, Canada, 2018, pp. 1634-1641.

[21] X. He, S. Alqahtani, R. Gamble, and M. Papa, “Securing Over-The-Air IoT Firmware Updates using Blockchain,” In Proceedings of the International Conference on Omni-Layer Intelligent Systems (COINS '19), New York, NY, USA, 2019, 164-171.