Template-Based Smart Contract Verification: A Case Study on Maritime Transportation Domain

Xufeng Zhao[1,3], Qiuyang Wei[1,2], Xue-Yang Zhu[1,3][0000−0002−2832−5590], and Wenhui Zhang[1,3]
{zhaoxf,weiqy,zxy,zwh}@ios.ac.cn

[1] Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China
[2] Hangzhou Institute for Advanced Study, University of Chinese Academy of Sciences, Hangzhou, China
[3] University of the Chinese Academy of Sciences, Beijing, China

Abstract. Maritime transportation business suffers from trust issues and burdensome paperwork. Blockchain-based smart contracts are a promising solution. Due to the nature of the blockchain, it is important to verify smart contracts before deployment, especially for its functionality and legality. In this paper, we propose a verification framework that automatically verifies the functionality and legality requirements of maritime transportation smart contracts. Smart contracts of an application, based on a set of templates, are modeled in a network of timed automata; domain-specific requirements are collected and formulated as temporal logic formulas; real-time model checking tool UPPAAL is then used to check whether these requirements are satisfied. We carry out experiments on nine real-world smart contracts to show the effectiveness and feasibility of our framework. We also compare our work with existing tools to show its effectiveness and efficiency.

Keywords: Model checking · UPPAAL · Smart contract · Solidity · Maritime transportation.

1 Introduction

Formal verification techniques are widely acknowledged for improving system reliability, but are also notorious for being difficult to use. In this paper, we propose a domain-specific verification method to ease the pain of using them.

Maritime transportation is the backbone of international trade, which accounts for over 90% of the global trade volume [37]. The maritime transportation industry is facing challenges from trustless participants, and burdensome paperwork. For example, a flower export consignment from Kenya to Netherlands requires more than 200 bilateral communications among 20 organizations, and it usually costs about 10 days to deal with the paperwork [9].

A smart contract is 'a computerized transaction protocol that executes the terms of a contract', defined by Nick Szabo [38] in 1994. Smart contracts rise with emergence of Blockchain platforms, such as Ethereum [14]. Due to the decentralized, tamper-proof and indisputable nature of the blockchain, smart contracts can perform reliable transactions among trustless participants, and thus gained attention from maritime transportation. Smart contracts are introduced to maritime transportation to improve the efficiency, traceability, and transparency [3, 18, 19, 23, 34].

Due to the tamper-proof nature of blockchain, a smart contract is hard to modify after deployment. Therefore, verification before deployment is essential. Symbolic execution tools [28, 32] are developed to detect specific vulnerabilities patterns. Development frameworks also play a part in avoiding vulnerabilities. For example, smart contracts developed from MariSmart templates [42] are free from re-entrance and transaction order dependency attacks. However, such approaches cannot be used to check various customer requirements.

Formal methods based tools [12, 13, 24, 30] are promising to verify a variety of properties. However, formal methods, i.e. model checking [17], usually require the skills of writing formal models and logic formulas, which makes it challenging to apply formal methods for engineers who lack solid mathematical background.

To ease the difficulty of using model checking techniques, we focus on the automated verification problem of smart contracts specific to maritime transportation domain. Specifically, two challenges are faced when dealing with this problem.

The first challenge is to find the common domain-specific requirements. We collect and extract functionality and legality requirements from maritime transportation related researches [9, 19, 23, 33], reports [21, 22], and laws [1, 40]. The **functionality requirements** specify the expected behaviors and results of maritime transportation, especially the payment and delivery of the shipment. Besides, maritime transportation usually take place across countries, and must obey multiple international and domestic laws. It is important to ensure that the smart contracts comply with these laws. The **legality requirements** are the legal constraints from maritime transportation laws, such as the time constraints and duties of participants. To the best of our knowledge, we are the first to apply verification on legality issues for maritime transportation smart contracts.

The second challenge is to automatically generate formal models and properties. We propose **MariSmart**, a development and verification framework for smart contracts that monitor and manage maritime transportation business. The goal of the framework is shown in Figure 1. Users can customize the MariSmart templates for a specific MariSmart application, which can then be verified with MariSmart verification framework. The verification results can be used for revising the code. The design and use of MariSmart templates are introduced in [42]. In this paper, we introduce MariSmart verification framework, a template-based verification technique.

Our main contributions in this paper are as follows.

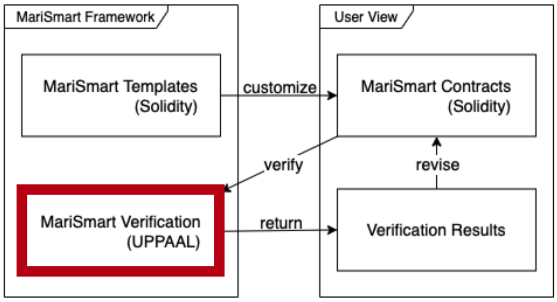


Fig. 1. The goal of the MariSmart framework. The scope of this paper is in the red bold box.

- We collect and extract common functionality and legality requirements specific to maritime transportation domain.
- We propose an automatic verification framework that transforms the MariSmart application to a network of timed automata, formalizes pre-defined requirements to timed computation tree logic formulas [5], and then verifies them with UPPAAL [12].
- We implement the verification framework, perform case studies on nine real-world smart contract systems, and compare our framework with existing works. The code of the framework and case studies are available on [2].

The reminder of this paper is organized as follows. Section 2 introduces the preliminaries. Section 3 illustrates the formal modeling method. Section 4 introduces the domain-specific requirements and formulates them. Experiments and case studies are shown in Section 5. Related work is reviewed in Section 6. Section 7 concludes and discusses the future work.

2 Preliminaries

In this section, we introduce the smart contract language Solidity, MariSmart templates and applications, and the back-end verification tool UPPAAL.

2.1 Solidity

Solidity is one of the most popular smart contract languages in Ethereum [14], which can be compiled and executed on the Ethereum Virtual Machine (EVM). A smart contract in Solidity contains declarations of State Variables, Functions, Function Modifiers, Events, Errors, Struct Types, and Enum Types.

Functions are the execution units of the smart contract. There can be a call chain of multiple functions in a transaction. For example in Figure 2, the function *depart* in (a) calls function *depart* in (b), which calls another internal function *internalTransfer*.

```
contract Carrier is Stakeholder {
  ...
  function depart(uint _UID)
      public virtual onlyOwner {
    shipments[_UID].depart();
  }
}
```

(a)

```
contract Shipment is IShipment {
  ...
  modifier pre_depart() virtual
      override {
    require(msg.sender == carrier)
        ;
    require(state == State.
        exported);
    _;
  }
  function depart() external
      virtual override pre_depart
      {
    internalTransfer(consignee,
        shipper, down_payment);
    depart_time = block.timestamp;
    state = State.departed;
    emit ShipmentDeparted(msg.
        sender, block.timestamp);
  }
}
```

(b)

```
contract NFTCarrier is Carrier {
  ...
  function depart(uint _UID)
      public override onlyOwner {
    shipments[_UID].depart();
    /* custom logic begins here */
    ContainerNFT(ContainerNFT_addr
        ).transferFrom(
      shipments[_UID].getShipper()
          ,
      address(this),
      NFTShipment(address(
          shipments[_UID])).
          getNFTID()
    );
  }
}
```

(c)

Fig. 2. Code fragments of MariSmart application. (a) *Depart* function of Carrier template; (b) corresponding function of Shipment template; (c) customized *depart* function of NFTCarrier contract.

Modifiers are used to specify the pre-conditions of the functions. Note that the whole transaction will revert if any of the *require* statement throws an exception.

Inheritance is important feature of Solidity. A Solidity contract can inherit from multiple contracts with the keyword *is*. Users can either add new functions, modifiers and variables to the contract, or override the existing ones with keyword *override*. For example, contract *NFTCarrier* in Figure 2(c) inherits contract *Carrier* in Figure 2(a), and overrides function *depart*.

Time is captured with an unsigned integer variable *block.timestamp*. Variables assigned with *block.timestamp* are called *time-related variables*. For example, *depart_ time* shown in Figure 2(b) is a time-related variable.

Other features required in this paper are introduced when they are mentioned.

2.2 MariSmart Application

An application of the maritime transportation developed with MariSmart templates [42] is called a *MariSmart application*. A MariSmart application usually includes several Stakeholder contracts and one Shipment contract.

The Stakeholder contracts implement the activities of participants of the transportation. Six stakeholders are formulated in the templates: *Shipper*, *Carrier*, *Consignee*, *Pre_ Shipment_ Inspector*, *Export_ Port_ Operator* and *Import_ Port_ Operator*. For example, the activity of *Carrier* departing the export part is implemented as function *depart* in Figure 2(a).

The Shipment contract indicates how the states of the shipment change. There are 12 states: *created*, *signed*, *inspected*, *exported*, *departed*, *lost*, *arrived*, *imported*, *rearranged*, *received*, *claimed*, and *closed*. A change occurs when its corresponding function is called and the function's pre-condition regulated by a modifier is satisfied.

The MariSmart templates can be customized for specific workflow and activities. For example, the customized contract in Figure 2(c) overrides the function *depart* with additional logic. Please refer to [42] for more details about MariSmart templates and applications.

2.3 UPPAAL

UPPAAL [12] is a real-time model checking tool. The model language of UPPAAL is based on the theory of *timed automaton* (TA). Its specification language is a variation of *timed computation tree logic* (TCTL).

Definition 1 (Syntax of UPPAAL TA). *A TA is a tuple* (L, l_0, A, C, V, E, I), *where*

- L *is a set of locations,*
- $l_0 \in L$ *is the initial location,*
- $A \subseteq Action$ *is a set of actions.* $Action := \{a?|a \in Chan\} \cup \{a!|a \in Chan\}$ *includes input actions* ($a?$) *and output actions* ($a!$). $Chan$ *is a set of channels.*
- C *is a set of clocks,*
- V *is a set of bounded integer variables,*
- $E \subseteq L \times G(C,V) \times A \times U(C,V) \times L$ *is a set of edges, where* $G(C,V)$ *is a set of linear constraints over* C *and* V *and* $U(C,V)$ *is a set of updates over* C *and* V. *Clocks can only be reset to zero.*
- $I : L \to G(C,V)$ *is a mapping from locations to constraints.*

A *network of timed automata* (NTA) includes a set of TA, sharing a set of global channels, variables and clocks.

The semantics of NTA in UPPAAL is defined as a transition system over the states of TA, where two kinds of transitions are included, namely *delay transitions* and *discrete transitions*. In delay transitions, no edge of TA is fired and the clocks increase; in discrete transitions the edge is fired and clocks are frozen. Specially, edges in different TA may fire synchronously, when there is a pair of actions $c?$ and $c!$ in the edges.

Compared to the theory of TA [6], additional features of NTA in UPPAAL mainly includes the following.

- Bounded integer variables are declared and used in guards, invariants and assignments.
- Committed locations are introduced in UPPAAL. When a NTA is at a *committed* location, it cannot delay and the next transition must involve an outgoing edge of the location.
- User functions are supported in UPPAAL with a syntax similar to C [27].

The properties involved in this work can be divided into two types, namely liveness properties and safety properties. Let β be any propositional formula. Liveness properties are usually in the form of $A\Diamond\ \beta$, meaning that on any execution path β will eventually be true, and $\beta_1 \longrightarrow \beta_2$, meaning that once β_1 true β_2 will eventually be true. Safety properties are usually in the form of $A\Box\ \beta$, meaning that on any execution path β is always true.

3 Contract Modeling

In this section, we illustrate how to model a MariSmart application as a NTA. We first define the structures to represent the MariSmart application, then illustrate how to formally model it. Finally we discuss how to deal with the time-related statements.

3.1 System Description

As is introduced in Section 2.2, a MariSmart application includes a Shipment contract and several Stakeholder contracts. Before modeling them as TA, we first abstract the Solidity function, Shipment contract, Stakeholder contracts and the MariSmart application as tuples for further interpretation.

Let Var be a set of variables and Var_t be a set of time-related variables.

Definition 2 (Solidity Function). *A public function is defined as a tuple* $FUNC := (Name, Para, Stmt, Stmt_r, Stmt_t, Var, Var_t)$*, where* $Name$ *is the function name,* $Para$ *is a set of parameters,* $Stmt$ *is a set of statements,* $Stmt_r \subseteq Stmt$ *is a set of* require *statements,* $Stmt_t \subseteq Stmt$ *is a set of time-related statements, and* $Var_t \subseteq Var$*.*

Take function *depart* in Figure 2(b) for example. Its name is *depart*, Var and $Stmt$ are taken from both *depart* and its modifier *pre_ depart*, $Stmt_r$ contains the statements in Line 4-5, $Stmt_t$ contains the statement in Line 10, and $Var_t = \{depart\_time\}$.

Definition 3 (Shipment Contract). *A Shipment contract is defined as a tuple* $SHC := (State, Var, Var_t, Func)$*, where* $State$ *is a set of shipment states,* $Var_t \subseteq Var$*, and* $Func$ *is a set of functions in the contract.*

Definition 4 (Stakeholder Contract). *A stakeholder contract is defined as a tuple* $STC := (Type, Var, Var_t, Func)$*, where* $Type$ *is the type of the stakeholder,* $Var_t \subseteq Var$*, and* $Func$ *is a set of functions in the contract.*

In SHC and STCs, we especially define Var_t and $Stmt_t$, because time-related variables and statements require special modeling in UPPAAL, which is proposed in Section 3.3. As a MariSmart application contains one Shipment contract and several Stakeholder contracts, it can be represented by a set of SHC and STCs.

Definition 5 (MariSmart Application). *A MariSmart Application with n stakeholders is defined as a tuple* $MA := (SHC, STC_1, STC_2, ..., STC_n)$.

3.2 Formal Modeling

We model a MariSmart application in a template-based way. That is, the behaviour of each element in the application, Shipment contract or Stakeholder contract, is modeled with a TA. The overview of the NTA of a MariSmart application is shown in Figure 3. The Shipment contract is modeled as *Shipment TA*, where the states of shipment are modeled as locations, and transitions of the states are modeled as edges. Each Stakeholder contract is modeled as a *Stakeholder TA*. For example, Carrier contract is modeled as Carrier TA, where the function *depart* is modeled as two edges. One from *idle* to *depart_ called* models the modifier and statements of the function. The other one from *depart_ called* back to *idle* models calling the Shipment function. It fires synchronously with the edge of Shipment TA through channel *chan_ depart*.

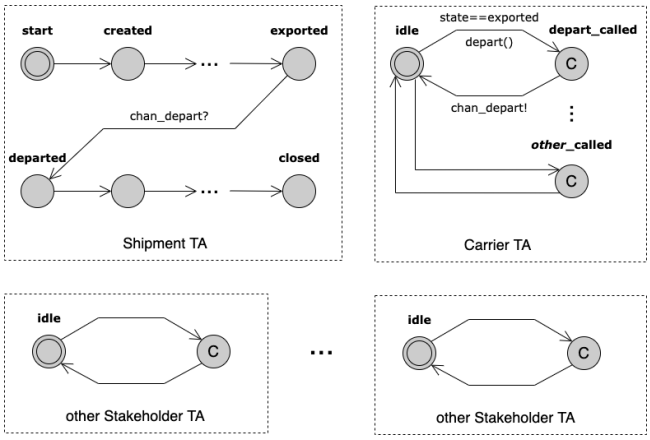


Fig. 3. Overview of NTA of a MariSmart application.

Definition 6 (Shipment TA). *The behavior of Shipment contract* $SHC = (State, Var, Var_t, Func)$ *is Shipment TA* $TA_{SHC} := (L, l_0, A, C, V, E, I)$, *where*

- $L = State \cup \{start\}$, *where start denotes the state before the shipment is created.*
- $l_0 = start$.

- $A = \{chan\_n?|n = f.Name, f \in Func\}$.
- $C = Var_t$.
- $V = Var \setminus Var_t$.
- *for each function* $f \in Func$*, there is one or more edges in* E*, s.t. the source is specified in* $f.Stmt_r$*, the destination is specified in* $f.Stmt$*, the guard condition is* $f.Stmt_r$*, and the action is* $chan\_name?$*, where* $name = f.Name$.
- I *is empty.*

A Shipment TA is demonstrated in Figure 4, where states of the shipment are modeled as locations and functions of Shipment contract is modeled as edges. The source and destination of edges are extracted from the modifier and function body. For example, function *depart* in Figure 2(b) is modeled as the edge from location *exported* to location *departed* in Figure 4.

Generally speaking, there is one edge in Shipment TA for each function of Shipment contract. However, there are exceptions. For example, function *inspect* can update the shipment state from *signed* to *inspected* when the inspection is passed, or from *signed* to *closed* alternatively. We introduce an auxiliary variable *inspect_flag* to determine which edge to fire. During the transformation, we first scan the Shipment contract to find if there are multiple assignments for the shipment state in one function. If so, an assignment for the auxiliary variable is inserted and transformed in UPPAAL User Function later.

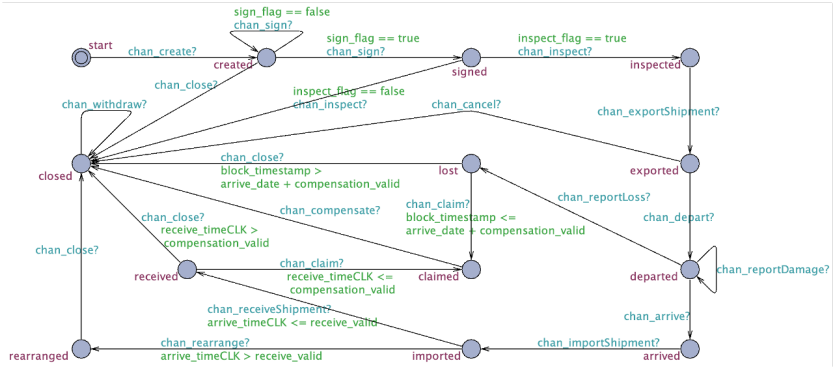


Fig. 4. Overview of TA_{SHC}.

A Stakeholder contract is modeled as a Stakeholder TA, where the function f is modeled as edges between *idle* and *f_called*. The statements in functions are modeled as UPPAAL User Functions, which supports most statements, such as assignment, **for** statement, **if** statement. Time-related statements need to be modeled in a special way, which is illustrated in Section 3.3. We also introduce an auxiliary clock *waitCLK*. It is reset when Stakeholder TA enter location *idle*, and should not exceed a constant limit at *idle*. This way, Stakeholder TA cannot permanently stay at location *idle*, so that the liveness properties can be verified properly.

Definition 7 (Stakeholder TA). *The behavior of Stakeholder contract* $STC = (Type, Var, Var_t, Func)$ *is Stakeholder TA* $TA_{STC} := (L, l_0, A, C, V, E, I)$*, where*

- $L = \{n\_called | n = f.Name, f \in Func\} \cup \{idle\}$.
- $l_0 = idle$.
- $A = \{chan\_n! | n = f.Name, f \in Func\}$.
- $C = \{tCLK | t \in Var_t\} \cup \{waitCLK\}$.
- $V = Var \setminus Var_t$.
- *for each function* $f \in Func$*, there are edges from idle to* n_called*,* $n = f.name$*, s.t. the value of* $f.Para$ *is selected, the guard is* $f.Stmt_r$*, the update resets clocks and trigger the User Function modeled from* f.
- *for each function* $f \in Func$*, and each Shipment function* g *called by* f*, there are edges from* n_called *to idle, s.t. the update resets* $waitCLK$*, the action is* $chan\_m!$*,* $m = g.Name$.
- $I(idle) = \{waitCLK \leq Constant\}$.

Take the contract shown in Figure 5(a) for example. Consignee contract is modeled as Consignee TA shown in Figure 5(b), where function *receiveShipment* is modeled as edges in Consignee TA in Figure 5(c). In the edge from *idle* to *receiveShipment_ called*, the value of parameter *received* is selected (in the yellow line), the *require* statement is examined in guard (in the green line) and the User Function (see in Figure 5(c)) is triggered in update (in the blue lines). Note that *state* is a reserved word in UPPAAL, so we replace it with *status* in Stakeholder TA.

For each function of the Shipment contract, there is usually one corresponding function in the Stakeholder contracts. Consignee contract shown in Figure 5(a) has an exception, where function *receiveShipment* can either call functions *receiveShipment* or *close*. When the framework transforms this function to a UPPAAL User Function, two auxiliary variable *call_ close* and *call_ receiveShipment* are introduced and assigned before the corresponding call statements. In this way, once the marked statements in Figure 5(c) are executed, only the corresponding edge in Figure 5(b) can be fired.

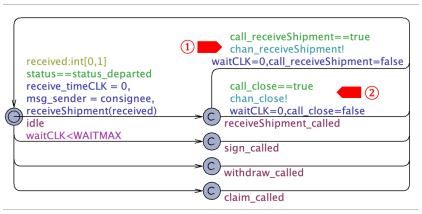
Definition 8 (MariSmart Application NTA). *The behavior of a MariSmart application* $MA = (SHC, STC_1, STC_2, ..., STC_n)$ *is modeled with an NTA* $NTA_{MA} := TA_{SHC} \parallel TA_{STC_1} \parallel TA_{STC_2} \parallel ... \parallel TA_{STC_n}$ *with global clocks.*

3.3 Dealing with Time

Modeling of time-related statements in smart contract is not as straightforward as other statements. In Solidity, current time is captured with an unsigned integer variable *block.timestamp*. We model *block.timestamp* and other variables assigned with it as clocks in UPPAAL. However, clocks cannot be used in UPPAAL User Functions. Therefore, we need to model the statements that contain time-related variables in specific ways. Three kinds of such statements need to be modeled specifically. They are demostrated in Table 1.

```
contract Consignee is Stakeholder{
  ...
  function receiveShipment(
    uint _UID,
    bool received)
    public override onlyOwner {
    if (received) {
      shipments[_UID].
          receiveShipment(false);
    } else {
      shipments[_UID].close();
  }}
  function sign(...){...}
  function withdraw(...){...}
  function claim(...){...}
}
```

(a)



(b)

```
void receiveShipment(bool received){
  if( received ){
    call_receiveShipment = true;    shipment_receiveShipment (false);
  }else{
      call_close = true;    shipment_close ();
}}
```

(c)

Fig. 5. An example of modeling Stakeholder contract as TA. (a) Consignee contract; (b) Consignee TA, some edges are simplified for brevity; (c) User Function modeled from *receiveShipment*, where the marked statements determine which edge to fire.

The first kind is assignment statements. When a variable v is assigned with *block.timestamp*, the corresponding clock v_CLK is reset to zero in the edge from location *idle* to location *activity_called*. An example is shown in the first row of Table 1, in which the time-related variable is *depart_time*.

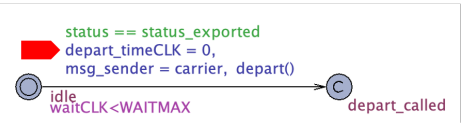
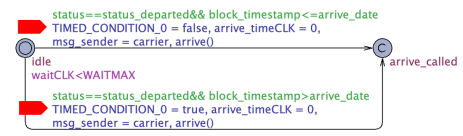
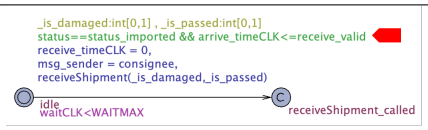
The second kind is **if** statements with time-related condition expressions. We replace such conditions with a set of auxiliary Boolean variables named by $TIMED\_CONDITION\_\#$. The rest of the statements are modeled as the UPPAAL User Function. The value of each condition expression is arbitrary, and all possible values are modeled as guard conditions of separate edges from location *idle* to location *activity_called*. Take function *arrive* in the second row of Table 1 for example, we split the edge from location *idle* to location *arrive_called* into two, denoting the two possible value of $TIMED\_CONDITION\_0$ accordingly. For functions that include multiple time-related condition expressions, all combinations of their possible values are modeled similarly.

The third kind is **require** statements. Time-related require statements are modeled as guards of the edge from location *idle* to location *activity_called*. For example in the third row of Table 1, the require statements in modifier *pre_receive* is modeled as the guard in TA. Notably, the clock *arrive_timeCLK* is equivalent to *block.timestamp-arrive_time*, so we transform the third require statement to $arrive\_timeCLK <= \text{receive\_valid}$.

4 Requirement Extraction and Formalization

In this section, we introduce how we collect and extract functionality and legality requirements and how to formalize them. The collected requirements can be checked with one-button verification in our framework.

Table 1. Modeling time-related statements.

Time-Related Statements	Transitions in TA
`function depart() external virtual override pre_depart{ internalTransfer(consignee, shipper, down_payment); depart_time = block.timestamp; state = State.departed; emit ShipmentDeparted(msg.sender, block.timestamp);}` The *block.timestamp* is assigned to *depart_time*	 Reset the corresponding clock named as *function_nameCLK* in the Stakeholder TA (as marked above).
`function arrive() external virtual override pre_arrive { internalTransfer(shipper, carrier, transportation_fee); arrive_time = block.timestamp; state = State.arrived; if (block.timestamp > arrive_date) { is_delayed = true; emit ShipmentArrivedDelayed(msg.sender, block.timestamp); } else emit ShipmentArrivedInTime(msg.sender, block.timestamp);}` The *block.timestamp* is used in a condition expression *block.timestamp* > *arrive_date*.	`// UPPAAL User Function: void arrive(){ balances[shipper] -= transportation_fee; balances[carrier] += transportation_fee; status = status_arrived; if(TIMED_CONDITION_0){ is_delayed = true;}}`  Replace the expression with a Boolean variable, and assign it with all possible values in separate edges (as marked above).
`modifier pre_receiveShipment() virtual override { require(msg.sender == consignee); require(state == State.imported); require(block.timestamp <= arrive_time + receive_valid); _;}` The third *require* statement contains both *block.timestamp* and a time-related variable *arrive_time*.	 Note that *arrive_timeCLK* is equivalent to *block.timestamp-arrive_time*. Reshape the expression and insert it into the guard of the edge (as marked above).

4.1 Functionality Requirement

Functionality requirements are mainly collected from research papers [9,19,23,33] and reports [21, 22]. They are divided into three kinds.

The first kind of functionality requirements is about workflow, listed as Requirement 1 to 6 in Table 2. They depict the order and dependence of transporta-

tion activities. For example, Requirement 1 requires that Shipment contract to be created before any other activity takes place.

Basically, properties of *something always happens* are modeled as safety properties such as $A\Box\ \beta$, while properties of *something will eventually happen* are modeled as liveness properties such as $A\Diamond\ \beta$ or $\beta_1 \longrightarrow \beta_2$. For example, Requirement 6 *'Shipment will eventually close'* are modeled as $A\Diamond$ *Shipment.closed*, where *Shipment.closed* denotes that Shipment TA is at Location *closed*.

For Requirements 1 to 3, we introduce a set of auxiliary Boolean variables like *already_ create* to trace previous activities. They are initiated as false, and are set to be true when the corresponding activity takes place. Moreover, conjunctions of locations are abstracted in Requirement 1 to 3 for conciseness. Please refer to the repository in [42] for full formulas.

The second kind of functionality requirements is about the payment system. In the MariSmart application, stakeholders first deposit Ethers (the currency of Ethereum) into the Shipment contract and withdraw them after the order is closed. During the transportation, the transferring between stakeholders is implemented as editing their balances instead of calling *send* or *transfer*. We collect the safety and correctness requirements of such balances as Requirement 7 to 10.

Formalizing these requirements is similar to the workflow requirements, except that a new auxiliary array *net* is introduced to trace the net income for the shipment contract (denoted by net[0]) and stakeholders (denoted by net[1–6]). The values of *net* are set to zero at the beginning, and are updated when the corresponding stakeholders pay or receive Ethers. With array *net*, we can check whether the balances correctly record the transferring results.

The third kind of functionality requirements is about handling accidents and compensation, including Requirement 11 to 16. Accidents include delayed arrival, shipment loss and damage. A MariSmart application should correctly record them and process the compensation.

We introduce clocks *block_ timestamp* and *arrive_ timeCLK* while formalizing the third kind of requirements. Clock *block_ timestamp* is set to zero at the beginning. Clock *arrive_ timeCLK* is set to zero when the carrier arrives.

4.2 Legality Requirement

We focus on two typical issues in maritime laws, namely compensation and rearrangement. We collect the legality requirements from Hamburg Rules [40], which was signed in 1978 and has been adopted by 35 countries, and Maritime Codes of PRC. [1]. They are listed as Requirement 17 to 26 in Table 2.

When accidents occur, compensation is usually claimed by the Shipper or Consignee, and paid by the Carrier. The amount of the compensation and time period to claim for it are specified differently in maritime laws.

Take Hamburg Rules [40] as example, we extract Requirement 17 and 19 according to Paragraph 1(a) of Article 6 in [40], which writes 'The liability of the carrier for loss resulting from loss of or damage to goods according to the provisions of article 5 is limited to an amount equivalent to 835 units of account

per package or other shipping unit or 2.5 units of account per kilogram of gross weight of the goods lost or damaged, whichever is the higher.' Note that the units in this paragraph is defined as special drawing right (SDR) in Paragraph 1 of Article 26.

Similarly, we extract Requirement 18 on compensation for loss of delay, according to Paragraph 1(b) of Article 6 in [40]. For period in which consignee should claim for compensation, we extract Requirement 20 from Paragraph 2 of Article 19.

When the consignee fails to receive the shipment within a specific period after arrival, the carrier is permitted to resell or auction the shipment, which we denote as rearrangement issues. In Maritime Code of the People's Republic of China [1], for example, it rephrases in Article 88, 'If the goods under lien in accordance with the provisions of Article 87 of this Code have not been taken delivery of within 60 days from the next day of the ship's arrival at the port of discharge, the carrier may apply to the court for an order on selling the goods by auction; where the goods are perishable or the expenses for keeping such goods would exceed their value, the carrier may apply for an earlier sale by auction.' We extract Requirement 26 according to it.

Since all of legality requirements should always be guaranteed, we model them as safety properties, as is shown in Table 2.

Notably, articles in these two maritime laws share a similar structure, which makes it convenient to extract and model them. There are only differences on specific figures between Requirement 17 to 21 from Hamburg Rules, and Requirement 22 to 26 from Maritime Codes of PRC. For example, the period for claiming for compensation is 15 days in Requirement 20, and it comes to 7 days in Requirement 25.

5 Experiments

To show the effectiveness and feasibility of our framework, we rewrite 9 real-world maritime transportation smart contracts to MariSmart application and verify them. We also compare our framework with two existing tools [30] [43].

We perform the experiments on a Linux server with 32 Cores, 2.90GHz CPU and 384G RAM. Our tool and all studied MariSmart cases can be accessed from [2]. We also present a Web application for our framework at [41].

5.1 Case studies on real-world smart contracts

We apply our verification framework on 9 real-world maritime transportation smart contracts, which are denoted as Medical [3], NFT1 [19], NFT2 [18], Recall [35], PPE [34], ShipChain [36], eth-shipment [11], LNG [26] and IoT [23]. Original contracts range from 100 to 500 lines. After modeled as NTA, the scale of the Shipment TA varies with the complexity of the workflow, and the edges of Shipment TA range from 6 to 21. The detailed information is included in first two parts of Table 3.

Table 2. Requirements and their TCTL formulas.

No.	Requirements	TCTL Formulas	Timed
Functionality Requirement			
1	Shipment is always created before any other activities.	$A\Box$ (*Any Stakeholder not at idle*) imply already_create	
2	Stakeholders always sign the shipment before taking any other activities.	$A\Box$!Shipment.closed and (*Any Stakeholder not at idle or sign_called*) imply already_sign	
3	Activities on departure side take place before departing.	$A\Box$ (*Any Stakeholder acts on departure side*) imply !already_depart	
4	After departure, the shipment either arrives or is lost.	Shipment.departed $\longrightarrow$ Shipment.lost or Shipment.arrived	
5	Activities on destination side take place after arriving.	$A\Box$ (Carrier.rearrange_called or Consignee.receiveShipment_called or ImportPortOperator.importShipment_called) imply already_arrive	
6	Shipment will eventually close.	$A\Diamond$ Shipment.closed	
7	Stakeholders' balances are always non-negative.	$A\Box$ (sum(i:int[1,6])balances[i]) $\geq$ 0	
8	Sum of shipment contract and stakeholders' net proceeds is always zero.	$A\Box$ (sum(i:int[0,6])net[i]) ==0	
9	Sum of stakeholders' balances is always equal to Ethers in shipment contract.	$A\Box$ (sum(i:int[1,6])balances[i]) == net[0]	
10	After closing the shipment, balances will eventually be withdrawn.	Shipment.closed $\longrightarrow$ (forall(i:int[1,6])balances[i]==0)	
11	Delayed arrival is recorded correctly.	$A\Box$ (is_delayed imply block_timestamp > arrive_date) and (block_timestamp > arrive_date and arrive_timeCLK==0 imply is_delayed)	✓
12	Delayed arrival leads to either claiming for compensation or closing shipment after compensate period	is_delayed $\longrightarrow$ Shipment.claimed or Shipment.closed and arrive_timeCLK>compensation_valid	✓
13	Loss of shipment is recorded correctly.	$A\Box$ (is_lost imply already_reportLoss) and (already_reportLoss imply is_lost)	
14	Shipment loss leads to either claiming for compensation or closing shipment after compensate period	is_lost $\longrightarrow$ Shipment.claimed or Shipment.closed and block_timestamp>arrive_date+compensation_valid	✓
15	Damage of shipment is recorded correctly.	$A\Box$ already_reportDamage imply is_damaged	
16	Shipment damage leads to either claiming for compensation or closing shipment after compensate period	is_damaged $\longrightarrow$ Shipment.claimed or Shipment.closed and arrive_timeCLK>compensation_valid	✓
Legality Requirement			
17	Compensation limit is always less than 835 SDR or 2.5 SDR/kg, whichever is the higher. [40]	$A\Box$ compensation_limit <835 or compensation_limit<2.5*weight	
18	Compensation actually paid for delay should be less than 2.5 times of transportation fee. [40]	$A\Box$ is_delayed and !is_damaged and !is_lost imply net[carrier]+balances[carrier]≥transportation_fee*(1-2.5)	
19	Compensation actually paid for loss or damage should be less than 835 SDR or 2.5 SDR/kg, whichever is the higher. [40]	$A\Box$ (is_damaged or is_lost) imply (net[carrier]+balances[carrier]≥transportation_fee-835 or net[carrier]+balances[carrier] ≥transportation_fee -weight*2.5)	
20	Consignee should claim for compensation within 15 days after arrival. [40]	$A\Box$ Consignee.claim_called imply receive_timeCLK≤15	✓
21	Consignee should receive shipment within 90 days after arrival, otherwise carrier can resell or auction it. [40]	$A\Box$ Carrier.rearrange_called imply arrive_timeCLK>90	✓
22	Compensation limit is always less than 666.67 SDR or 2 SDR/kg, whichever is the higher. [1]	$A\Box$ compensation_limit<666.67 or compensation_limit<2*weight	
23	Compensation actually paid for delay should be less than transportation fee. [1]	$A\Box$ is_delayed and !is_damaged and !is_lost imply net[carrier]+balances[carrier]≥transportation_fee*(1-1)	
24	Compensation actually paid for loss or damage should be less than 666.67 SDR or 2 SDR/kg, whichever is the higher. [1]	$A\Box$ (is_damaged or is_lost) imply (net[carrier]+balances[carrier]≥transportation_fee-666.67 or net[carrier]+balances[carrier]≥ transportation_fee-weight*2)	
25	Consignee should claim for compensation within 7 days after arrival. [1]	$A\Box$ Consignee.claim_called imply receive_timeCLK≤7	✓
26	Consignee should receive shipment within 60 days after arrival, otherwise carrier can resell or auction it. [1]	$A\Box$ Carrier.rearrange_called imply arrive_timeCLK>60	✓

* SDR denotes Special Drawing Right.

Table 3. Experimental results.

Case	Medical		NFT1		NFT2		Recall		PPE		ShipChain		eth-shipment		LNG		IoT	
Code Lines of Original Smart Contract																		
#	210		521		325		252		125		130		193		111		229	
Edges in Shipment TA																		
#	13		13		11		9		6		7		9		21		18	
Verification Result																		
Requirement	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**	**result**	**time**
1	✗	1	✓	8470	✓	9	✓	255	✓	600	✗	1	✓	28	✓	542	✓	779
2	✗	0	✗	1	✓	8	✗	1	✗	4	✗	0	✓	28	✓	546	✓	776
3	✓	528	✓	9243	✓	7	✓	256	✓	675	✗	1	✓	28	✓	586	✓	769
4	✗	2	✗	8	✗	3	-	-	-	-	✗	0	✓	45	✓	590	✓	871
5	✓	470	✗	0	✗	3	-	-	-	-	✓	0	✓	29	✓	554	✓	774
6	✗	1	✗	1	✓	2	✗	5	✗	3	✓	1	✓	48	✓	13	✗	203
7	✓	656	✓	7465	✓	8	✓	208	✓	542	✓	0	✓	29	✓	514	✓	785
8	✓	578	✓	7724	✓	9	✓	218	✓	628	✓	0	✓	29	✓	527	✓	787
9	✓	628	✓	6790	✓	7	✓	184	✓	532	✓	0	✓	29	✓	576	✓	782
10	✗	1	✗	2	✓	23	✗	6	✗	29	✗	1	✗	2	✓	2079	✓	1266
11	✗	1	✓	9083	✓	8	✓	234	✓	642	✗	1	✓	29	✓	580	✓	776
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	623	✓	878
13	-	-	✓	7727	✓	8	✓	202	✓	594	✓	1	✓	30	✓	732	✓	798
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	564	✓	878
15	-	-	✓	7803	✓	8	✓	222	✓	544	✓	1	✓	28	✓	530	✓	798
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	639	✓	904
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	583	✗	155
18	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✗	37	✓	784
19	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✗	76	✓	775
20	-	-	✓	8943	✓	7	✓	192	✓	604	✓	1	✓	30	✓	622	✓	780
21	-	-	✗	0	✓	7	✓	190	✓	577	✓	0	✓	30	✗	42	✓	807
22	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✓	605	✗	141
23	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✗	42	✓	819
24	-	-	-	-	-	-	-	-	-	-	-	-	-	-	✗	73	✓	790
25	-	-	✓	7958	✓	8	✓	209	✓	539	✓	1	✓	27	✓	530	✓	786
26	-	-	✗	1	✓	8	✓	208	✓	558	✓	1	✓	30	✗	46	✓	784
Avg. Passed	572		8121		8		214		586		1		31		627		824	

✓: satisfied, ✗: not satisfied, -: not checked, time(ms): verification time, Avg. Passed(ms): average verification time of satisfied requirements.

* Execution times are measured in milliseconds (ms).

The experimental results are shown in the third part of Table 3. The satisfied requirements are marked with checkmarks, unsatisfied ones are marked with crossmarks. When the relevant activity is not included in the case, corresponding requirement is not checked. For example, 7 of 9 cases do not implement compensation, and the relevant Requirement 17-19 and 22-24 are not checked.

As for functionality, 7 of 9 cases violate at least one requirements of workflow. Five cases violate requirement 6, which implies that these smart contract may be deadlocked. This violation is especially harmful since the funds may be locked in the Shipment contract. After a closer examination, we find that this requirement is violated when there are unexpected entrances to the activities, and strengthening the pre-conditions of these activities may help with this issue.

As for legality requirements, Case IoT violates Requirement 17 and 22, because it does not specify the limit of compensation. Due to the nature of liquefied natural gas, Case LNG specifically assigns the limit of compensation and the time period for reselling. Although it violates 6 relevant legality requirements, its additional terms are protected by the laws, and should be deemed as legal. As for the rest of the cases, compensation and reselling are not implemented, thus relevant requirements are not checked.

Since the unsatisfied requirements take much shorter time than the satisfied ones, we evaluate the efficiency of our framework through the average time for satisfied requirements. The verification time is mainly influenced by the range and number of parameters. Number of edges in Shipment TA and code lines of the original contract may influence the verification time. For example, Case NFT1 [19] and Case NFT2 [18] are implemented by the same team, and share a similar workflow. However, the pre-condition of activity *rearrange* of Case NFT1 is too weak, so that the Shipment TA contains more edges and the verification time is about 100 times than Case NFT2.

The complexity of the requirements may also influence the verification time, as Requirement 10 takes longer time than others. It is influenced by the complexity of the TCTL formulas, and the optimization in UPPAAL.

5.2 Comparison with VeriSolid and mvSC

We compare our framework against two verification tools, VeriSolid [30] and mvSC [43], since they both deal with time and can verify customized properties. VeriSolid transforms model-based Solidity contracts as transition systems, and verifies them in nuXmv [15]. VeriSolid applies abstraction to time variables, where time-guarded transitions are specially modeled. mvSC models smart contracts as NTA and verifies them in UPPAAL. Both works consider time in modeling, and are similar to our approach.

Both tools verify a single contract only, so we simplify the Case Medical and verify the Property 1-11 with them. The result is shown in Table 4. It can be observed that our framework can deal with a wider range of properties than VeriSolid, and runs significantly faster than mvSC.

Table 4. Comparison with VeriSolid [30] and mvSC [43]

Tool	**MariSmart**		**VeriSolid**		**mvSC**	
Requirement	**result**	**time(s)**	**result**	**time(s)**	**result**	**time(s)**
1	✗	0.001	✗	<10	✓	1503.558
2	✗	0.000	✗	<10	✗	0.001
3	✓	0.528	✓	<10	✓	1611.017
4	✗	0.002	✗	<10	✗	0.001
5	✓	0.47	✓	<10	✓	1429.493
6	✗	0.001	✗	<10	✗	0.001
7	✓	0.656	-	-	✓	1585.734
8	✓	0.578	-	-	✓	1520.306
9	✓	0.628	-	-	✓	1605.873
10	✗	0.001	-	-	✗	0.000
11	✗	0.001	-	-	✗	0.001

VeriSolid can only verify Property 1-6, since it only takes calling a function or statement as propositional formula, while specifying the value of variables is

not supported. As for verification time, VeriSolid does not record verification time originally. Since it is built as a Web application, it is unfair to calculate the entire runtime including waiting for pages to respond. Basically speaking, all verification can be finished within 10 seconds.

The tool mvSC models smart contracts as NTAs, where functions are modeled as TAs and statements are modeled as edges. Compared to our framework, the NTA generated by mvSC contains more transitions, so that the verification time significantly increases. As for verification results, only the result of Property 1 is different to the other two. This is because mvSC only supports single contract, and requires to call the constructor prior to any other functions. Due to the limitations of mvSC, the counterexample in the other two tools are excluded, such that the Stakeholder contracts trigger before the shipment creation.

6 Related Work

Formal methods used in smart contract verification [39] mainly include model checking based techniques [13] [8] [30] [4] [43] and theorem proving based techniques [24] [7].

Hirai [24] formalized the EVM semantics in theorem prover Isabelle/HOL and manually proved safety properties. Da et al. [25] proposed a tool that automatically translates Michelson contract [20] to WhyML, and verifies them in Why3. As in [7], Amani et al. extended an existing EVM formalisation in Isabelle/HOL. Approaches based on theorem proving usually take EVM bytecode or other low level code as input. Most of them are performed manually, which requires users to be experts of theorem proving. By contrast, our framework automatically models the smart contracts so that users are free from writing formal properties.

Bai et al. [8] verified template-based smart contracts with the model checker SPIN [31]. Compared to our work, the template designed in [8] is highly abstract and only defines basic interactions between two contract parties. Users of [8] have to pay more effort to develop specific smart contracts for their business. As for verification, [8] only discusses an approach to verify smart contract with SPIN, and performs a case study manually. No automated framework is involved. Alqahtani et al. [4] modeled smart contracts as a finite state machine and checked the requirements in NuSMV [16]. [4] mainly proves the feasibility of the approach yet with little automation. FsolidM [29] is a framework for designing Ethereum smart contracts. The same team further proposed VeriSolid [30] based on FSolidM. This tool first generates augmented transition system, then transforms it into BIP [10] transition system, and finally verifies it with nuXmv [15]. Zhao et al. [43] proposed an approach to verify smart contracts with UPPAAL, especially the contracts with time constraints. Comparing to our work, [43] models each statement as one edge, which results in a larger amount of locations and transitions, and also a longer time for verification. Most of the works require users to write formal properties manually, while our framework can verify important domain-specific requirements automatically.

7 Conclusion

In this paper, we have proposed an automatic verification framework for template-based smart contracts and requirements specific to maritime transportation domain. The collected requirements can be checked with one-button verification in our framework. The effectiveness and the feasibility of our verification framework are shown by the experimental results on nine real-world smart contracts. Compared to existing works, our approach can verify a wider range of domain-specific requirements within a reasonable time.

Due to the limitation of UPPAAL, the syntax of input smart contracts is limited. For example, the random generator function *keccak* is not supported by the framework. In the future, we will expand the range of which the framework can process, and extend the framework to a broader domain.

Acknowledgments. This work is partially supported by the National Natural Science Foundation of China (No. 62072443).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Maritime code of PRC. (1993), `https://www.gov.cn/guoqing/2020-12/24/content_5572935.htm`
2. Marismart framework. (2023), `https://github.com/MariSmartSourceCode/MariSmart`
3. Ahmad, R.W., Salah, K., Jayaraman, R., Yaqoob, I., Omar, M., Ellahham, S.: Blockchain-based forward supply chain and waste management for covid-19 medical equipment and supplies. Ieee Access **9**, 44905–44927 (2021)
4. Alqahtani, S., He, X., Gamble, R., Mauricio, P.: Formal verification of functional requirements for smart contract compositions in supply chain management systems (2020)
5. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: [1990] Proceedings. Fifth Annual IEEE Symposium on Logic in Computer Science. pp. 414–425. IEEE (1990)
6. Alur, R., Dill, D.: Automata for modeling real-time systems. In: Automata, Languages and Programming: 17th International Colloquium Warwick University, England, July 16–20, 1990 Proceedings 17. pp. 322–335. Springer (1990)
7. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying ethereum smart contract bytecode in Isabelle/HOL. In: Proceedings of the 7th ACM SIGPLAN international conference on certified programs and proofs. pp. 66–77 (2018)
8. Bai, X., Cheng, Z., Duan, Z., Hu, K.: Formal modeling and verification of smart contracts. In: Proceedings of the 2018 7th international conference on software and computer applications. pp. 322–326 (2018)
9. Balci, G., Surucu-Balci, E.: Blockchain adoption in the maritime supply chain: Examining barriers and salient stakeholders in containerized international trade. Transportation Research Part E: Logistics and Transportation Review **156**, 102539 (2021)

10. Basu, A., Bensalem, B., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE software **28**(3), 41–48 (2011)
11. Bavosa, A.: Smart contracts. (2018), https://github.com/ajb413/eth-shipment-tracking/tree/master
12. Behrmann, G., David, A., Larsen, K.G.: A tutorial on Uppaal 4.0. Department of computer science, Aalborg university (2006)
13. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., et al.: Formal verification of smart contracts: Short paper. In: Proceedings of the 2016 ACM workshop on programming languages and analysis for security. pp. 91–96 (2016)
14. Buterin, V., et al.: A next-generation smart contract and decentralized application platform. white paper **3**(37), 2–1 (2014)
15. Cavada, R., Cimatti, A., Dorigatti, M., Griggio, A., Mariotti, A., Micheli, A., Mover, S., Roveri, M., Tonetta, S.: The nuXmv symbolic model checker. In: CAV. pp. 334–342 (2014)
16. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. In: Computer Aided Verification: 11th International Conference, CAV'99 Trento, Italy, July 6–10, 1999 Proceedings 11. pp. 495–499. Springer (1999)
17. Clarke Jr, E.M., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model Checking. MIT Press (2018)
18. Elmay, F.K., Madine, M., Salah, K., Jayaraman, R.: Nfts for trusted traceability and management of digital twins for shipping containers. In: 2023 IEEE International Conference on Pervasive Computing and Communications Workshops and other Affiliated Events (PerCom Workshops). pp. 433–438. IEEE (2023)
19. Elmay, F.K., Salah, K., Jayaraman, R., Omar, I.A.: Using nfts and blockchain for traceability and auctioning of shipping containers and cargo in maritime industry. IEEE Access **10**, 124507–124522 (2022)
20. Foundation, T.: Michelson: the language of smart contracts in tezos., https://tezos.gitlab.io/active/michelson.html#language-semantics
21. Ganne, E.: Can blockchain revolutionize international trade?[online] (2018), https://www.wto.org/english/res_e/booksp_e/blockchainrev18_e.pdf
22. Group, B.C.: Digital innovation in trade finance: Have we reached a tipping point? (2017), https://www.swift.com/news-events/news/digital-innovation-trade-finance-have-we-reached-tipping-point
23. Hasan, H., AlHadhrami, E., AlDhaheri, A., Salah, K., Jayaraman, R.: Smart contract-based approach for efficient shipment management. Computers & industrial engineering **136**, 149–159 (2019)
24. Hirai, Y.: Defining the ethereum virtual machine for interactive theorem provers. In: Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21. pp. 520–535. Springer (2017)
25. da Horta, L.P.A., Reis, J.S., de Sousa, S.M., Pereira, M.: A tool for proving michelson smart contracts in WHY3. In: 2020 IEEE International Conference on Blockchain (Blockchain). pp. 409–414. IEEE (2020)
26. Keith, M., Edward, S.: Master ex-ship LNG sales agreement between Cheniere Marketing, Inc. and Gaz De France International Trading S.A.S. https://www.sec.gov/Archives/edgar/data/3570/000119312507106384/dex102.html (2007)
27. Kernighan, B.W., Ritchie, D.M.: The C programming language (2002)

28. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. pp. 254–269 (2016)
29. Mavridou, A., Laszka, A.: Designing secure ethereum smart contracts: A finite state machine based approach. In: Financial Cryptography and Data Security: 22nd International Conference, FC 2018, Nieuwpoort, Curaçao, February 26–March 2, 2018, Revised Selected Papers 22. pp. 523–540. Springer (2018)
30. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for Ethereum. In: Financial Cryptography and Data Security: 23rd International Conference, FC 2019, Frigate Bay, St. Kitts and Nevis, February 18–22, 2019, Revised Selected Papers 23. pp. 446–465. Springer (2019)
31. Mikk, E., Lakhnech, Y., Siegel, M., Holzmann, G.J.: Implementing statecharts in PROMELA/SPIN. In: Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques. pp. 90–101. IEEE (1998)
32. Mueller, B.: A framework for bug hunting on the ethereum blockchain. ConsenSys/mythril (2017)
33. Nguyen, S., Chen, P.S.L., Du, Y.: Risk assessment of maritime container shipping blockchain-integrated systems: An analysis of multi-event scenarios. Transportation Research Part E: Logistics and Transportation Review **163**, 102764 (2022)
34. Omar, I.A., Debe, M., Jayaraman, R., Salah, K., Omar, M., Arshad, J.: Blockchain-based supply chain traceability for COVID-19 personal protective equipment. Computers & Industrial Engineering **167**, 107995 (2022)
35. Patro, P.K., Ahmad, R.W., Yaqoob, I., Salah, K., Jayaraman, R.: Blockchain-based solution for product recall management in the automotive supply chain. IEEE Access **9**, 167756–167775 (2021)
36. ShipChain: Shipchain smart contracts. (2020), `https://github.com/ShipChain/smart-contracts/tree/master`
37. Song, D.: A literature review, container shipping supply chain: Planning problems and research opportunities. Logistics **5**(2), 41 (2021)
38. Szabo, N.: Smart contracts. (1994), `http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html`
39. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Computing Surveys (CSUR) **54**(7), 1–38 (2021)
40. uncitral: Hamburg rules. (1987), `https://uncitral.un.org/zh/texts/transportgoods/conventions/hamburg_rules`
41. Zhao, X., Lu, Y.: Marismart verifier webpage. (2023), `http://124.16.137.30:50002/#/dashboard-en`
42. Zhao, X., Wei, Q., Zhu, X.Y., Zhang, W.: A smart contract development framework for maritime transportation systems. In: 2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C). pp. 310–319 (2023). https://doi.org/10.1109/QRS-C60940.2023.00091
43. Zhao, Y., Zhu, X., Li, G., Bao, Y.: Time constraint patterns of smart contracts and their formal verification. Journal of Software **33**(8), 2875–2895 (2022)