An Event-B based approach for formal modelling and verification of smart contracts

Asma Lahbib, Abderrahim Ait Wakrime, Anis Laouiti, Khalifa Toumi, Steven Martin

An Event-B based Approach for Formal Modelling and Verification of Smart Contracts

Asma LAHBIB*, Abderrahim Ait Wakrime**, Anis Laouiti*, Khalifa Toumi***, and Steven Martin****

*SAMOVAR, Télécom SudParis, CNRS, Université Paris-Saclay, 9 rue Charles Fourier 91011 Evry, France.
Email: {asma.lahbib, anis.laouiti}@telecom-sudparis.eu
**Computer Science Department, Faculty of Sciences, Mohammed V University, Rabat, Morocco.
Email: abderrahim.aitwakrime@gmail.com
***IRT SystemX, 8 Avenue de la Vauve, 91127 Palaiseau, France.
Email: {khalifa.toumi}@irt-systemx.fr
****LRI, Université Paris-Sud, 15 Rue Georges Clemenceau, 91400 Orsay, France.
Email: {steven.martin}@lri.fr

***Abstract*— While smart contracts are becoming widely recognized as the most successful application of the blockchain technology that could be applied into various industries and for different purposes such as e-commerce, energy tradings, assets management, and healthcare services, their implementation has posed several challenges insofar that they could handle large amount of money and digital assets in addition to their ability to manipulate critical data and transactions related information which makes them attractive targets of security threats and attacks that could lead to significant problems like money losses, privacy leakage and data breach. To better deal with such issues, reasoning about the correctness, the safety and the functional accuracy of smart contracts before their deployment on the blockchain network is critical and no important than ever. In this context model checking tools are well adopted for the formal verification of smart contracts in order to assure their execution as parties' willingness as well as their reliable and secure interaction with users. In this direction, this paper uses Event-B formal verification method to formally model solidity written smart contracts in order to verify and validate their safety, correctness and functional accuracy in addition to their compliance with their specification for given behaviors. The verification is conducted using a model checking tool along which expected safety properties are formalized, validated and judged to be satisfied or unsatisfied. To illustrate the proposed approach, its application to a realistic industrial use case is described.**

***Index Terms*— Formal verification, Event-B, Smart contracts, Blockchain, Solidity**

I. INTRODUCTION

Blockchain, as a decentralized and distributed public ledger technology in a peer to peer network, has attracted recently a lot of interest in both the research and the industrial communities. Originally invented as the underlying infrastructure of Bitcoin, this technology can be applied into diverse applications far beyond cryptocurrencies and financial services. The range of potential use cases of blockchain is seemingly endless, for instance this last could be applied into various environments and for different purposes such as supply chains management, legal and notary services, big data applications, Internet of Things (IoT) scenarios, energy tradings, healthcare services and so on.

How blockchain has attracted researchers and industrials' attention this way is mainly due to the significant number of business benefits if offers including transparency, traceability, security, availability and efficiency. Moreover, one of the most important features that has given rise to a stronger interest in blockchain is the auto processing property of established transactions satisfied through the use of smart contracts.

Smart contracts could be defined as an executable code deployed and residing at a specific address on the blockchain network. They permit trusted transactions and agreements to be established among parties without relying on a centralized authority or giving participants the ability to go back on, thus ensuring transactions transparent, and irreversible execution. However and with the rapid development of blockchain, smart contracts have been exposed to an increasing number of security threats and attacks [1] that have led to significant malicious scenarios resulting in terrible losses such as the DAO attack, that has caused more than 3 million ETH separated from the DAO resources pool which is worth around $60 million.

In order to better deal with such issues, reasoning about the correctness, the safety and the functional accuracy of smart contracts before their deployment on the blockchain network is critical and no important than ever. In this context, many formal verification methods are proposed and several tools such as Why3, F*, Oyente are developed to verify the correctness of the program regarding existing programming errors. However, these ones rarely test the behavior of the smart contract while interacting with users or under specific scenarios.

For this purpose model checking is well adopted in behavior based verification [2] in order to verify whether the smart contract can interact with the user in a reliable and secure way or not.

In this direction, this study will focus on behavior based formal verification of smart contracts in order to verify their compliance with the specification for given behaviors. Such contracts are written in Solidity (which is a high level programming language designed for implementing smart contracts running on the Ethereum blockchain). In order to model their design in a formal way, we adopt Event-B formal method (which is a state-based formal method for modeling and analyzing systems, based on classical logic and set theory), as in our previous work [20], [21]. Therefore a verification of the formal model is conducted using a model checking tool along which expected safety properties are formalized, validated and judged to be satisfied or unsatisfied. The described approach of this proposal is applied to a use case application that has been the subject of a previous work, called Distributed Resource Management Framework for Industry 4.0 environments (DRMF) [3].

The rest of this paper is organized as follows. Section 2 presents the considered case study in which this approach is applied. Section 3 recalls the basic concepts of blockchain and smart contracts along with a discussion of their security issues and vulnerabilities followed by a basic definition of Event-B formal method. Thereafter a presentation of related proposals carried out in the area of

smart contracts formal verification is given in Section 4. Section 5 describes the proposed approach from a theoretical and formal point of view and details the proposed Event-B formalization followed by the verification process of smart contract's behavior in Section 6. Finally in Section 7, the paper ends up with some conclusions and an outlook of our future work to study in this area.

II. Use case study

Throughout this paper, we will project our approach into a real world use case study conducted in the scope of some application domains of Industry 4.0.

As a case study, we consider a set of automaker factories collaborating all together along the production processes and alongside with automotive suppliers buying and shipping goods and products through transportation partners as it is shown in Fig.1. These parties

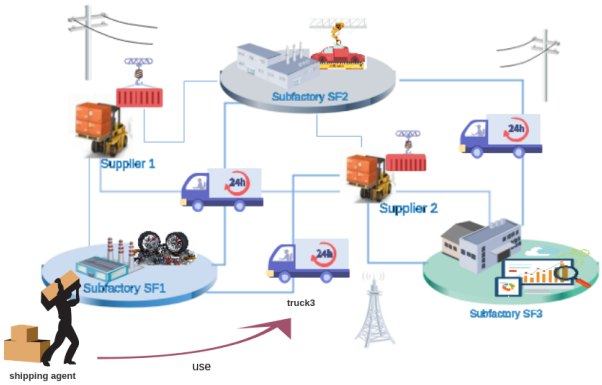

Fig. 1. use case study

while looking to ensure sophisticated shipping operations, agreed to invest in both shipping equipments and logistic technologies to manage their day-to-day trucking operations instead of involving specific services and third parties to ensure them.

We remind here that sharing such resources among several parties collaborating and working all together and especially that could not always have a strong confidence established in advance could raise several problems mainly related to the management and the access made over common resources that should be in most time fully distributed, secure, traceable and notarized. In an attempt to resolve such challenges, we have used in our previous proposal [3], blockchain based smart contracts technology in order to keep a living document trace about the flow of resources being shared among collaborating parties while integrating the OrBAC access control model to implement distributed, dynamic and secure resource access authorization.

To do so, a smart contract named Access Contract (AC) was developed where the OrBAC access control model was integrated within a distributed ledger to express access control policies and to manage access authorization made over shared resources. This last carries out transactions related to access requests, the evaluation of access control policies for each tuple (resource, requester, action to be done, the context in question) as well as the access decision to be granted.

Returning to the presented use case, let us assume that a shipping agent within the subfactory SF1 wants to have access over the shared resource Truck 3 to perform a shipping operation. To do so, an evaluation of the agent's access request is needed first to allow or not the demanded access. Here a verification of the well conduct and functioning of both the requesting entity and the requested resource is essential, hence an authorization over the access control smart contract is required to decide whether the access is permitted or prohibited. The access authorization and according to the OrBAC access control model depends on a set of contextual conditions whose activation will activate the corresponding rule. In this example, to have the requested access accepted, the human agent should have a trust score above the defined threshold for the corresponding role, he should belong to the shipping department and request to use the Truck during his working hours,and he should not make too frequent access requests during a certain minimum period of time.

III. Background and problem statement

In this section, we will introduce first the main preliminaries used in our proposal specifically blockchain, smart contracts and formal verification, we will point out then potential security issues that smart contracts may have and we will review research proposals related to their security verification in recent years.

A. Blockchain technology

Originally designed for keeping a financial ledger and meeting the purpose of cryptocurrency applications, the blockchain paradigm can be extended to provide a generalized framework for managing any movements of data related to goods, devices, information records, etc. This last could be defined as a distributed ledger of transactions whereby records of all established interactions are registered providing thereof a proof of existence, of ownership and modification of this data during interaction [4], [5]. The established transactions are hold within blocks chained together and containing within their headers the hash of the previous block in order to ensure immutability since blocks once chained, data contained within will be available and couldn't be easily changed or altered. To ensure that all entities have the same copy of the ledger, a consensus is required to maintain the blockchain architecture and to ensure its functioning. This last makes sure that an agreement is reached between a set of predefined entities to support a decision making. After reaching consensus, valid blocks are added to the blockchain. Moreover, each node in this distributed peer-to-peer network holds the same copy of transaction records, which provides robustness against single point of failure attacks.

B. Smart contracts

A smart contract is an executable code deployed and residing at a specific address on the blockchain network. The main aim of a smart contract is to automatically execute the terms of an agreement once the specified conditions are met. It include a set of data which are the state variables and code corresponding to the executable functions. These last are executed when transactions are made, broadcast to the network and addressed to its address. Called smart contract then runs independently and automatically in a prescribed manner on every node in the network, according to the data that was included as input in the related transaction, as a result an eventual return value is shown to the outside.

Smart contracts can be developed and deployed in different blockchain platforms where each one of them offers distinctive features for development supported by different high-level programming languages. In Ethereum blockchain platform, advanced and customized smart contracts are supported with the help of Turing complete programming language. The code of an Ethereum contract is in a low-level, stack-based bytecode language referred to as

Proposal	Target	Category	Focus	Tool
[12]	Solidity SC and EVM bytecode	Program based formal verification	Translated SC into F* to check the correctness.	F*
[13]	EVM bytecode	Program based formal verification	Analyzed EVM bytecode of contracts statically.	Isabelle/HOL
[14]	Solidity SC	Behavior based formal verification	Considered the interaction between users, programs and the environment.	Statistical Model Checking
[15]	DSCP BITHALO	Behavior based formal verification	Used probabilistic formal models for verification.	Game theory Markov Decision Process Prism tool
[16]	Solidity SC	Behavior based formal verification	Proposed a generic modeling method then applied the model checking.	NUSMV Model checking
[17]	LLVM bitcode supports different high level languages	Program based formal verification	Proposed a source code translator to convert the smart contract embedded with policy assertions to LLVM bitcode and a verifier to determine assertion violations.	Zeus (the prototype implemented)

TABLE I
SMART CONTRACTS BASED FORMAL VERIFICATION APPROACHES

Ethereum virtual machine (EVM) code. Users define contracts using high-level programming languages compiled into EVM code. The most widespread language is Solidity [6] which is a JavaScript style contract-oriented, statically-typed, high-level programming language designed for implementing smart contracts.

C. Potential security issues and assurance of smart contracts

Besides their correct execution, it is also crucial that the design and the implementation of smart contracts are secure against vulnerabilities aiming at tampering and stealing assets they handle. Indeed, several security vulnerabilities in Ethereum smart contracts have been discovered. A recent analysis reveals that among 19336 smart contracts deployed on the public Ethereum blockchain, 8333 contracts suffer from at least one security issue [7]. An example of attack was in June 2016, the DAO (the worlds largest crowdfunding project deployed on the Ethereum) was attacked by hackers, causing more than 3 million ETH separated from the DAO resources pool which is worth around $60 million.

A survey of possible attacks on Ethereum contracts was presented in [1] where security vulnerabilities of smart contracts were grouped into three classes according to the level in which they are introduced namely Solidity, EVM bytecode, and blockchain.

Another issue that smart contracts may have is their dependency on external calls especially when they execute external contracts' codes within their functions, call other contracts' function and wait for its returned value, or even call another contract that may change its global state. If there is an exception raised (e.g., not enough gas, exceeding call stack limit) in the called contract, the calling contract terminates, reverts its state and returns false. However, depending on how the call is made, the exception in the calling contract may or may not get propagated. In these cases, the contracts control flow should not be influenced by an adversary contract.

To deal with such issues, reasoning about the correctness of smart contracts before their deployment on the blockchain network is critical and no important than ever. How to write reliable smart contracts was presented in [2], where two aspects were considered for the correctness verification and security insurance of smart contracts including programming correctness and formal verification.

Formal verification provides a powerful technology for the correctness verification of the established specification of smart contracts. In the following subsection, we will review research proposals carried out in this field.

D. Event-B formal method

The Event-B method is an evolution of the B method [8], [9]. This method is based on the notions of pre-conditions and post-conditions of Hoare, the weakest pre-condition of Dijkstra and the substitution calculus. Event-B method is based on mathematical foundations namely first order logic and set theory. A formal model in Event-B uses two types of entities to describe a system: Machines and Contexts. A Machine represents the dynamic part of a model i.e. states and transitions. A Context contains the static part of the model. A context may extend an existing context and a machine may see a context. Generally, a model is defined by a name, sets, constants and their properties, variables and their invariants and events. An event takes the form:

$$\textbf{evt} \triangleq \textbf{any}\ \text{x}\ \textbf{where}\ \text{G}\ \textbf{then}\ \text{Act}\ \textbf{end}.$$

Where x is the list of event parameters, G represents predicates which define the guard of the event and Act is an action that modifies some state variables. When the guard is satisfied, the event can be fired.

Event-B is supported by the Atelier-B platform [11] and can be used in conjunction with the Pro-B animator/model-checker [10] in order to animate and validate a formal development. Due to lack of space, we outline in this paper the short overview of Event-B and to have more detail about Event-B method, these references can be useful [8], [9].

IV. RELATED WORKS

Multiple efforts have been carried out in the current literature for the correctness verification of the established specification of smart contracts through formal verification approaches.

In [12], authors proposed a verification method based on programming language. They translated Solidity written smart contracts into an F* language (functional programming language aimed at program verification) in order to check the safety, and the functional accuracy of implemented contracts. The translation is made both at the source level (functional correction specifications) and bytecode level (low-level properties).

In [13], the Isabelle proof assistant was used to verify the binary Ethereum code whose corresponding sequences were organized into linear code blocks. A logic program was then created where each block is processed as a set of instructions. Each part of the verification is validated in a single trusted logical framework from the perspective of bytecode.

In [14], a new verification method is proposed to verify a smart contract's behavior in its execution environment. The proposed approach considered the interaction between both users and programs, plus programs and the environment. The Behavior Interaction Priorities (BIP) framework was used for components modeling, therefore the Statistical Model Checking (SMC) tool was used for verification.
In [15], game theory approach was combined with formal methods to address the challenging aspect of smart contracts, the proposed approach focused on DSCP (a Decentralized Smart Contract Protocol inspired by BITHALO), therefore a probabilistic formal model was proposed to verify smart contracts based on PRISM tool.
In [16], authors proposed a generic modeling method of smart contracts based Ethereum applications, the model checking approach was then considered to verify the implementation's compliance with the specification. The proposed model was written in NuSMV language with properties formalized into temporal logic CTL to be subsequently applied to a case study based on smart contracts and coming from the energy market place.
In [17], a framework called ZEUS was proposed to automatically verify the correctness of implemented smart contracts using both abstract interpretation and symbolic model checking and to validate their fairness. The proposed framework consists of three components including a policy builder against which the smart contract must be verified, a source code translator for the conversion of this last to LLVM bitcode, and third a verifier in charge of determining and reporting policy violations.
Table I summarizes the already presented works regarding the target they address, the category to which they belong, the focus they are interested at and the tool they use.

V. Proposed Approach

A. Overview

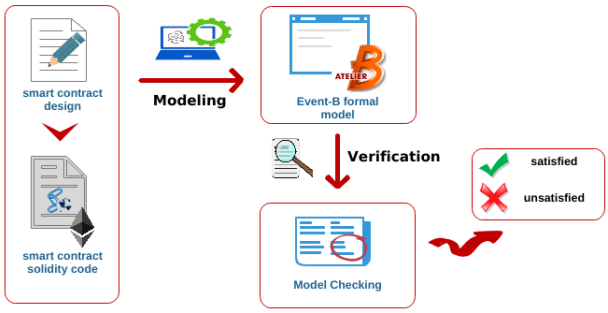

Fig. 2. Overview

The main objective of this work is to formally model an Ethereum blockchain application based on smart contracts formal verification methods. We rely alongside this work on Event-B which is a state based model-oriented formal method intended for system development. Its strength lies in a well-defined modelling and analysis process, which allows one to specify a system from an abstract specification to a concrete one.
In order to verify and validate their safety, correctness and functional accuracy, smart contracts, as illustrated in Fig. 2, are translated from solidity (which is a high level programming language designed for implementing smart contracts running on the Ethereum blockchain), into Event-B formal language. Therefore, and in order to check whether implemented smart contracts behave as they are supposed to do, a verification of the formal model is conducted using a model checking tool along which expected safety properties are formalized, validated and judged to be satisfied or unsatisfied.

B. DRMF framework, Access contract

The described approach of this proposal is applied to a use case application that has been the subject of a previous work, called Distributed Resource Management Framework for Industry 4.0 (DRMF) [**?**]. In what follows, we recall the basic concepts of DRMF to focus on the specification and implementation details of one of the considered smart contracts which is the Access contract (AC).
In brief, the DRMF framework utilizes blockchain technology to support shared resources management and usage between collaborating parties while dynamically manage access authorization over considered resources. The corresponding implementation is composed of three smart contracts coded in solidity programming language.
In this work we will mainly focus on the Access Contract (AC) which is designed to achieve distributed, interoperable, contextual, trustworthy and secure access control for multi organization systems. It is based on the integration of the OrBAC access control model within an Ethereum distributed ledger to express access control policies. Its operation is based on a set of data variables and methods that will be formally modeled into Event-B formal method as it will be presented in the following paragraphs.

C. AC Event-B formal model

In Listing 1, the set of enumerated types, the state variables, the constants and the properties of defined attributes constituting the machine ACCESS CONTRACT are introduced. For instance, AccessType_AS, Agent_AS, Activity_AS are specified as sets containing predefined constants an AccessType (respectively an Agent, an Activity) variable could have. Table II illustrates a brief description of each variable specified within the ACCESS CONTRACT machine.
Reminding here that a role structure is characterized by its description, the Organization by which it is issued (e.g. the subfactory SF1, SF2, etc.), the department to which it does belongs (e.g. logistics), and the trust threshold it has. On the other hand, the context structure is identified by its description (that could be for example shipping, renting, etc.), minInterval (which is the minimum allowable time interval between two successive access requests), nb_req_threshold (that corresponds to the maximum number of access requests in the minimum interval), working hours (corresponds to the working hours interval).

MACHINE *ACCESS_CONTRACT*
SETS
$AccessType\_AS = \{permission, prohibition\}$;
$Agent\_AS = \{agent1, agent2, agent3\}$;
$Activity\_AS = \{use, check, add, update\}$;
$Resource\_AS = \{truck1, truck2, truck3, truck4, record, trustfile\}$;
$Context\_AS = \{ctx1, ctx2, ctx3\}$;
$ContextDesc\_AS = \{ctxdesc1, ctxdesc2, ctxdesc3\}$;
$SecurityRule\_AS = \{SR1, SR2, SR3\}$;
$Role\_AS = \{role1, role2, role3\}$;
$RoleDesc\_AS = \{shippingworker, productionworker, supervisor\}$;
$Organisation\_AS = \{SF1, SF2, SF3\}$;
$Departement\_AS = \{logistics, manufacturing, administration\}$;
$State = \{free, occupied\}$

ABSTRACT_VARIABLES
$AccessType, Agent, Activity,$
$Resource, Context, SecurityRule,$
$SecurityRuleList, Role, RoleDesc,$
$Organisation, Departement, TrustThres,$
$RoleList, WorkingHours, ContextList,$

$ContextDesc, minInterval, nbRequests,$
$nbReqThreshold, resultAccessControl,$
$Requests, TimeLastAccess, AccessTime,$
$LastRequestId, currentTime, TrustValue, ResourceState$

INVARIANT
\\Typing invariants
$AccessType \subseteq AccessType\_AS \ \wedge$
$Agent \subseteq Agent\_AS \ \wedge$
$Activity \subseteq Activity\_AS \ \wedge$
$Resource \subseteq Resource\_AS \ \wedge$
$Context \subseteq Context\_AS \ \wedge$
$SecurityRule \subseteq SecurityRule\_AS \ \wedge$
$Role \subseteq Role\_AS \ \wedge$
$RoleDesc \subseteq RoleDesc\_AS \ \wedge$
$Organisation \subseteq Organisation\_AS \ \wedge$
$Departement \subseteq Departement\_AS \ \wedge$
$ContextDesc \subseteq ContextDesc\_AS \ \wedge$
$AccessTime \in \mathbb{N} \ \wedge$
$LastRequestId \in \mathbb{N} \ \wedge$
$currentTime \in \mathbb{N} \ \wedge$
$TrustValue \in 0..10$
$nbReqThreshold = 2$
END

Listing 1. The description of SETS, ABSTRACT VARIABLES and INVARIANT clauses.

VARIABLE	Description
AccessType	the access type to be attributed after each access demand
Agent	the entity asking access authorization
Activity	the action to be performed once the access is authorized
Resource	the resource over which the access is requested
Role	the role demanding the access
Context	the situation that must be respected so that the access is granted
SecurityRule	the defined access control policy
SecurityRuleList	the list of security rules
RoleList	the list of roles
ContextList	the list of contexts

TABLE II
SECURITY RULES LIST EXAMPLE

The definition of sets and abstract variables are succeeded by the introduction of typing invariant properties in order to complete the model construction. Listing 1. includes the typing invariants of added variables For example, we define the variable Resource to store shared resources over which the access is demanded. Obviously *Resource* is a subset of *Resource_AS* (Inv4, Listing 1.). Listing 2. defines typing invariants related to role, context, resource, request and security rules structures. As an example, the *ResourceState* invariant (invariant 8) is used to constrain changes in the state of a resource between free and occupied states.

MACHINE *ACCESS_CONTRACT*
INVARIANT
\\Invariant 1
$RoleList \in Role \leftrightarrow (RoleDesc * Organisation * Departement) \ \wedge$
\\Invariant 2
$TrustThres \in Role \rightarrow 0..10 \ \wedge$
\\Invariant 3
$minInterval \in Context \rightarrow (0..10) \ \wedge$
\\Invariant 4
$nbRequests \in Context \rightarrow (0..2) \ \wedge$
\\Invariant 5
$ContextList \in Context \leftrightarrow (ContextDesc * Resource) \ \wedge$
\\Invariant 6
$WorkingHours \in Context \rightarrow \mathbb{N} * \mathbb{N} \ \wedge$
\\Invariant 7
$TimeLastAccess \in ContextDesc \leftrightarrow \mathbb{N} \ \wedge$
\\Invariant 8
$ResourceState \in Resource \rightarrow State \ \wedge$
\\Invariant 9
$Requests \in \mathbb{N} \leftrightarrow (Context * \mathbb{N} * Role) \ \wedge$
\\Invariant 10
$resultAccessControl \in Requests \leftrightarrow BOOL$
\\Invariant 11
$SecurityRuleList \in SecurityRule \leftrightarrow$
$(AccessType * Activity * Resource * Role * Context)$
END

Listing 2. Role, Context, Resource, Request and Security Rule typing invariants.

As a next step, this model is enriched by safety invariant properties plus the definition of events. The guard and action of these last must be specified in such a way that it establishes invariants preservation.

Safety invariants:
In order to ensure consistent, correct and safe functioning process of the considered framework, we define a set of constraints as it is illustrated in Listing 3. These last, and that must be preserved by events specification, are formalized as follow:

1) Property 1: Once the smart contract is called upon a request is made, an access result shall always be returned.
2) Property 2: Ensure that there is only one instance for each access rule defined and shared within the Access smart contract.
3) Property 3: Once a shared resource is used by a certain entity, the access over it must be blocked and could never be attributed to another entity.

MACHINE *ACCESS_CONTRACT*
INVARIANT
\\Invariant 12
$card(SecurityRuleList) \leq card(SecurityRule\_AS) \ \wedge$
\\Invariant 13
$\forall(sr1, sr2, accessT1, act1, rsr1, role1, context1, accessT2, act2, rsr2, role2, context2).$
$(accessT1 \in AccessType \ \wedge \ act1 \in Activity \ \wedge \ rsr1 \in Resource \ \wedge$
$context1 \in Context \ \wedge \ role1 \in Role \ \wedge \ accessT2 \in AccessType \ \wedge$
$act2 \in Activity \ \wedge \ rsr2 \in Resource \ \wedge \ context2 \in Context \ \wedge \ role2 \in Role \ \wedge$
$sr1 \in SecurityRule \ \wedge \ sr2 \in SecurityRule \ \wedge$
$(sr1 \mapsto (accessT1 \mapsto act1 \mapsto rsr1 \mapsto role1 \mapsto context1)) \neq$
$(sr2 \mapsto (accessT1 \mapsto act1 \mapsto rsr1 \mapsto role1 \mapsto context1)) \Rightarrow sr1 \neq sr2)$
\\Invariant 14
$\forall(rsr, req).$
$(rsr \in Resource \ \wedge \ req \in Requests \ \wedge \ ResourceState(rsr) = occupied$
$\Rightarrow resultAccessControl(req) = FALSE)$
END

Listing 3. Safety invariant properties.

Events Specification In the following Listings, we use events to describe the behavior of our *ACCESS_CONTRACT* Event-B machine. In Listing 4, we define the first event called addPolicy permitting to define and to add a new access control policy, based on the role demanding the access "*role*", the context defining the situation "*context*", the resource to be used "*rsr*", the action to be performed "*act*" and the access type to be attributed "*accessT*". This event requires the execution of addRole and addContext events before to be completed. As pre-condition, input parameters must be already defined and the security rule must not exist before within the "*SecurityRuleList*". The triggering of this event allows a new policy item to be added to the policies list whose size will be incremented.

MACHINE *ACCESS_CONTRACT*
EVENTS
$addPolicy = ANY \ \ role, context, rsr, act, accessT, sr$
WHERE $role \in Role \ \wedge$
$context \in Context \ \wedge$
$rsr \in Resource \ \wedge$
$act \in Activity \ \wedge$
$accessT \in AccessType \ \wedge$
$sr \in SecurityRule \ \wedge$
$sr \notin dom(SecurityRuleList) \ \wedge$
$(accessT \mapsto act \mapsto rsr \mapsto role \mapsto context) \notin ran(SecurityRuleList)$
THEN $SecurityRuleList := SecurityRuleList \cup$
$\{(sr \mapsto (accessT \mapsto act \mapsto rsr \mapsto role \mapsto context))\}$

```
    END;
END
```

Listing 4. *add Policy* event.

Listing 5. and Listing 6. are responsible for defining and adding a new role, respectively a new context to their corresponding list, Rolelist and ContextList. For instance, addRole event uses in input the role name "*rl*", the corresponding description "*rldesc*", the organization to which the role belongs "*org*", the corresponding department "*dept*", and finally the trust threshold above which access demanders should have their associated trust scores "*trust*".

```
MACHINE ACCESS_CONTRACT
EVENTS
        addRole = ANY rl, rldesc, org, dept, trust
WHERE   rl ∈ Role ∧
        rldesc ∈ RoleDesc ∧
        org ∈ Organisation ∧
        dept ∈ Departement ∧
        trust ∈ TrustThres ∧
        rl ∉ dom(RoleList) ∧
        (rldesc ↦ org ↦ dept) ∉ ran(RoleList) ∧
        ∀(rr, oo, dd).(rr ∈ RoleDesc ∧ oo ∈ Organisation ∧
        dd ∈ Departement ∧ (rr ↦ oo ↦ dd) ∈ ran(RoleList)
            ⇒ rr ≠ rldesc ∧ (oo ≠ org ∨ dd ≠ dept))
THEN    RoleList := RoleList ∪ {(rl ↦ (rldesc ↦ org ↦ dept))}
    END;
END
```

Listing 5. *add Role* event.

```
MACHINE ACCESS_CONTRACT
EVENTS
        addContext = ANY ctx, ctxdesc, minIn, nbReq, rsr
WHERE   ctx ∈ Context ∧
        ctxdesc ∈ ContextDesc ∧
        minIn ∈ 0..3 ∧
        nbReq ∈ 0..3 ∧
        rsr ∈ Resource ∧
        ctx ∉ dom(ContextList) ∧
        (ctxdesc ↦ rsr) ∉ ran(ContextList) ∧
        ∀(cc, rr).(cc ∈ ContextDesc ∧ rr : Resource ∧
            (cc ↦ rr) ∈ ran(ContextList) ⇒ cc ≠ ctxdesc ∧ rr ≠ rsr)
THEN    ContextList := ContextList ∪ {(ctx ↦ (ctxdesc ↦ rsr))} ||
        minInterval := minInterval ⩤ {ctx ↦ minIn}
    END;
END
```

Listing 6. *add Context* event.

Listing 7. introduces the AccessControlAccept event. Through this last, an access request made by the role "*rl*", upon a certain resource "*rsr*" to perform the activity "*act*" within the specific context "*ctx*" is authorized. To do so, a set of guards are generated, these last represent the set of conditions that should be respected, they include the typing guards related to input parameters, the belonging guards over which the existence of role "*rl*" and context "*ctx*" within the Rolelist and respectively the Contextlist will be checked, finally the comparison guards defined to check the condition to be respected as well as the legitimate behavior of the access requesting agent. This last is evaluated through (i) the corresponding trust score "*TrustValue*" that should be above the trust threshold "*TrustThres*" of the associated role, (ii) the access requests number to detect a potential doubtful access demand, this last should not exceed the "*nbReqThreshold*", (iii) the last access request "*TimeLastAccess*" that should be launched within the minimum allowable time interval "*minInterval*", and (iv) finally the current access request that shall be launched during the working hours interval. As a result of this event, a list of variables will be modified through the event action clause specifically the "*Requests*" set, the "*LastRequestId*" and the "*TimeLastAccess*". Therefore the state of the resource will be changed to occupied and the access will be authorized.

```
MACHINE ACCESS_CONTRACT
EVENTS
        accessControlAccept = ANY rsr, rl, act, ctx
WHERE   \\Typing guards
        rsr ∈ Resource ∧
        rl ∈ Role ∧
        act ∈ Activity ∧
        ctx ∈ ContextDesc ∧
        \\Belonging guards
        rl ∈ dom(RoleList) ∧
        (ctx ↦ rsr) ∈ ran(ContextList) ∧
        \\Comparison guards
        ResourceState(rsr) = free ∧
        TrustValue ≥ TrustThres(rl) ∧
        nbRequests(ContextList (ctx ↦ rsr)) < nbReqThreshold ∧
        currentTime − TimeLastAccess(ctx) >
            minInterval(ContextList (ctx ↦ rsr)) ∧
        (∀(xx, yy).(xx ∈ ℕ ∧ yy ∈ ℕ ∧
            (xx ↦ yy) = WorkingHours(ContextList⁻¹(ctx ↦ rsr)) ⇒
            currentTime > xx ∧ currentTime < yy))
THEN    Requests := Requests ∪ {LastRequestId + 1
            ↦ (ContextList⁻¹(ctx ↦ rsr) ↦ currentTime ↦ rl)} ||
        LastRequestId := LastRequestId + 1 ||
        TimeLastAccess(ctx) := currentTime ||
        resultAccessControl(LastRequestId + 1 ↦
            (ContextList∼(ctx ↦ rsr) ↦ currentTime ↦ rl)) := TRUE ||
        ResourceState(rsr) := occupied
    END;
END
```

Listing 7. *Access Control Accept* event.

```
MACHINE ACCESS_CONTRACT
EVENTS
        accessControlRelease = ANY rsr, rl, act, ctx
WHERE   \\Typing guards
        rsr ∈ Resource ∧
        rl ∈ Role ∧
        act ∈ Activity ∧
        ctx ∈ ContextDesc ∧
        \\Belonging guards
        rl ∈ dom(RoleList) ∧
        (ctx ↦ rsr) ∈ ran(ContextList) ∧
        \\Comparison guards
        ResourceState(rsr) = occupied ∧
        (currentTime − TimeLastAccess(ctx) ≤
            minInterval(ContextList (ctx ↦ rsr)) ∨
        nbRequests(ContextList (ctx ↦ rsr)) ≥ nbReqThreshold ∨
        TrustValue < TrustThres(rl) ∨
        ¬((∀(xx, yy).(xx ∈ ℕ ∧ yy ∈ ℕ ∧
            (xx ↦ yy) = WorkingHours(ContextList⁻¹(ctx ↦ rsr)) ⇒
            currentTime > xx ∧ currentTime < yy)))
THEN    Requests := Requests ∪ {LastRequestId + 1
            ↦ (ContextList⁻¹(ctx ↦ rsr) ↦ currentTime ↦ rl)} ||
        LastRequestId := LastRequestId + 1 ||
        TimeLastAccess(ctx) := currentTime ||
        resultAccessControl(LastRequestId + 1 ↦
            (ContextList∼(ctx ↦ rsr) ↦ currentTime ↦ rl)) := FALSE
    END
END
```

Listing 8. *Access Control Reject* event.

VI. Verification of the smart contract behavior

In this section, our main objective is to demonstrate our approach by (i) validating the formal model introduced in Section. V-C and then (ii) checking that both the presented implementation and design of the smart contract verifies well some required typing and safety properties. More details about the proposed framework are provided in our previous work [3]. As a first attempt to validate the Event-B models, we have applied these last to a simplified real example derived from the use case study illustrated in Section. II.

Indeed in this scenario, we added two new roles called shipping worker and 'production worker' belonging respectively to the 'logistics' and 'manufacturing' departments within the subFactory SF1 and having as trust threshold the value of '0.5', correspondingly

a new context is added to the contexts list, named 'shipping', it concerns the resource 'truck3' during the working interval '8to18' and where 2 access requests are permitted during the minimum time interval set up to '5 minutes'.

Afterwards, a corresponding security rule is defined and added to the security rules list after being specified as follow: SR1(access type = "permission", activity= "use", resource= "truck3", role= "shipping worker", context= "shipping"), in other words, this security rule authorizes the shipping worker to use the truck3 resource according to the context shipping, that is, his trust score is above the trust threshold, the current time is within the working interval and at most 2 access requests are launched during the last 5 minutes.

Subsequently, we defined 3 access demands corresponding respectively to 2 legitimate behaviors and a malicious one. For the legitimate behaviors, two shipping workers from the logistics department with a trust score equal respectively to 0.7 and 0.65 demand for the first time to use the truck3 resource during their working hours interval. For the malicious behavior, we assume that a shipping agent launches an on-off attack resulting on a low trust score while demanding access over the truck3 resource.

In order to demonstrate that the formal specification of access contract models is correct, we aim to validate Event-B models using ProB model-checker and animator [18]. This validation allows to verify that the invariants are preserved by all events. Since these models are deterministic and have finite state spaces, the model-checking and animation are sufficient to validate our model for a given initial state than theorem proving that requires considerable efforts.

a) Verification using model-checking: Model-checking [19] is an automated approach for verifying that a system model conforms to its specifications. The system behavior is formally modeled and the specifications, expressing the expected properties of the system, are also formally expressed, in our case via formulas of the first-order logic. All experiments were conducted on a 64-bit PC, Ubuntu 16.04 operating system, an Intel Core i5, 2.3 GHz Processor with 4 cores and 8 GB RAM. Using the ProB model-checker and based on mixed breadth and depth search strategy, we have explored all states: 100% of checked states with 641 distinct states and 1613 transitions during 1987 milliseconds. No invariant violation was found, and all the operations were covered. This verification ensures that invariants are preserved by each event. Otherwise, a counter-example would be generated.

b) Verification using animation: ProB can be used as a complementary of model-checker as an animator. Verification using animation is very important and can detect a series of problems, such as unexpected behavior of a model. ProB animator allows to visualize the dynamic behavior of a Event-B machine and we can systematically explore all the accessible states of a Event-B machine to check the studied properties. We have successfully applied the animation of ProB on the operational scenario that described at the beginning of this section. The animation of this scenario demonstrates the behavior of the our specification which implies that we have a verified and validated specification. However, if this is not the case, then we have to go back to the initial specification to correct the unacceptable behavior and re-apply the animation until the specification conforms to requirements.

VII. Conclusion

In this paper, we have applied the formal verification concept to smart contracts developed and deployed within an Ethereum based blockchain application. The aim is mainly to verify the compliance of these last with their specification while validating their safety and functional accuracy under specific properties and for given behaviors. To do so, we adopted the Event-B formal method for translating solidity written smart contracts. Model checking technique has been exercised therefore on the resulted formal model to judge and validate some properties of interest. The presented method was applied to a simplified real example derived from a use case study describing shared resources management among collaborating organizations under Industry 4.0 environments.

As a future work, we will focus on enriching our model with other properties when considering malicious parties introducing specific scenarios of attacks.

References

[1] Atzei, Nicola, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). International Conference on Principles of Security and Trust. Springer, Berlin, Heidelberg, 2017.

[2] Liu, Jing, and Zhentian Liu. A Survey on Security Verification of Blockchain Smart Contracts. IEEE Access (2019).

[3] Lahbib, Asma, et al. Distributed Resource Management Framework for Industry 4.0 environments (DRMF). 2019 IEEE 18th International Symposium on Network Computing and Applications (NCA). IEEE, 2019. (to appear)

[4] Zheng, Zibin, et al. An overview of blockchain technology: Architecture, consensus, and future trends. 2017 IEEE International Congress on Big Data (BigData Congress). IEEE, 2017.

[5] Abeyratne, Saveen A., and Radmehr P. Monfared. Blockchain ready manufacturing supply chain using distributed ledger. (2016).

[6] Solidity. Programming language for smart contracts. Available online: https://solidity.readthedocs.io/en/v0.5.3/

[7] Luu, Loi, et al. Making smart contracts smarter. Proceedings of the 2016 ACM SIGSAC conference on computer and communications security. ACM, 2016.

[8] Abrial, Jean-Raymond, and Jean-Raymond Abrial. The B-book: assigning programs to meanings. Cambridge University Press, 2005.

[9] Abrial, Jean-Raymond. Modeling in Event-B: system and software engineering. Cambridge University Press, 2010.

[10] Leuschel, Michael, and Michael Butler. "ProB: A model checker for B." International Symposium of Formal Methods Europe. Springer, Berlin, Heidelberg, 2003.

[11] ClearSy: Atelier B: B System (2019). http://clearsy.com/

[12] Bhargavan, Karthikeyan, et al. Formal verification of smart contracts: Short paper. Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security. ACM, 2016.

[13] Amani, Sidney, et al. Towards verifying ethereum smart contract bytecode in Isabelle/HOL. Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. ACM, 2018.

[14] Abdellatif, Tesnim, and Kei-Lo Brousmiche. Formal verification of smart contracts based on users and blockchain behaviors models. 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS). IEEE, 2018.

[15] Bigi, Giancarlo, et al. Validation of decentralised smart contracts through game theory and formal methods. Programming Languages with Applications to Biology and Security. Springer, Cham, 2015. 142-161.

[16] Nehai, Zeinab, Pierre-Yves Piriou, and Frederic Daumas. Model-checking of smart contracts. 2018 IEEE 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). IEEE, 2018.

[17] Kalra, Sukrit, et al. ZEUS: Analyzing Safety of Smart Contracts. NDSS. 2018.

[18] Leuschel, Michael, and Michael Butler. "ProB: an automated analysis toolset for the B method." International Journal on Software Tools for Technology Transfer 10.2 (2008): 185-203.

[19] Clarke Jr, Edmund M., et al. Model checking. MIT press, 2018.

[20] Ait Wakrime, Abderrahim, J. Paul Gibson, and Jean-Luc Raffy. "Formalising the Requirements of an E-Voting Software Product Line Using Event-B." 2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE). IEEE, 2018.

[21] Ait Wakrime, Abderrahim, et al. "Formalizing railway signaling system ERTMS/ETCS using UML/Event-B." International Conference on Model and Data Engineering. Springer, Cham, 2018.