Legal Knowledge and Information Systems
M. Palmirani (Ed.)

doi:10.3233/978-1-61499-935-5-21

On Observing Contracts: Deontic Contracts Meet Smart Contracts

Shaun AZZOPARDI [a,1] and Gordon J. PACE [a,b] Fernando SCHAPACHNIK [c,2]
[a] *Department of Computer Science, University of Malta, Msida, Malta*
[b] *Centre for Distributed Ledger Technologies, University of Malta, Msida, Malta*
[c] *Universidad de Buenos Aires, Facultad de Ciencias Exactas y Naturales, Departamento de Computación and ICC UBA-CONICET, Buenos Aires, Argentina*

Abstract. Smart contracts have been proposed as executable implementations enforcing real-life contracts. Unfortunately, the semantic gap between these allows for the smart contract to diverge from its intended deontic behaviour. In this paper we show how a deontic contract can be used for real-time monitoring of smart contracts specifically and request-based interactive systems in general, allowing for the identification of any violations. The deontic logic of actions we present takes into account the possibility of action failure (which we can observe in smart contracts), allowing us to consider novel monitorable semantics for deontic norms. For example, taking a rights-based view of permissions allows us to detect the violation of a permission when a permitted action is not allowed to succeed. A case study is presented showing this approach in action for Ethereum smart contracts.

Keywords. blockchain, smart contracts, contracts, deontic logic, monitoring

1. Introduction

Regulating the behaviour of interactive systems[3] has long been studied in a formal manner — from the formal semantics of contracts describing the expected modalities of behaviour to analysis techniques to enable automated verification of compliance of a system with such contracts. Deontic logics have been shown to be particularly effective in capturing the concepts behind such contracts, enabling them to be described as first-class logical objects which one can analyse, manipulate and transform. The deontic modalities of obligation and prohibition lend themselves easily to verification techniques, since through the observation of actions or the state of the system regulated by such a contract, one can easily ascertain whether or not an obligation or a prohibition has been violated. On the other hand, the notion of permission, if we take a rights-based view of it, is more problematic when it comes to verification, because an action that is permitted but (illegitimately) denied might be mistaken for one than never occurred. If a user is allowed to close an account but the bank does not allow it, the account remains open. From a (oversimplistic but common) point of view, there is no trace of the violation.

[1] Corresponding Author: Shaun Azzopardi, Department of Computer Science, University of Malta, Msida, Malta; E-mail: shaun.azzopardi@um.edu.mt.
[2] Partially supported by PICT-201-0112 and UBACyT 20020170100172BA.
[3] By interactive systems we refer to systems made up of different components interacting in such a manner that parts of their behaviour can be enabled or blocked by the other components.

1. *This contract is between ⟨buyer-name⟩, henceforth referred to as 'the buyer' and ⟨seller-name⟩, henceforth referred to as 'the seller'. The contract will hold until either party requests its termination.*
2. *The buyer is obliged to order at least ⟨minimum-items⟩, but no more than ⟨maximum-items⟩ items for a fixed price ⟨price⟩ before the termination of this contract.*
3. *Notwithstanding clause 1, no request for termination will be accepted before ⟨contract-end-date⟩. Furthermore, the seller may not terminate the contract as long as there are pending orders.*
4. *Upon enactment of this contract, the buyer is obliged to place the cost of the minimum number of items to be ordered in escrow.*
5. *Upon accepting this contract, the seller is obliged to place the amount of ⟨performance-guarantee⟩ in escrow, otherwise, if only a partial amount is placed, the seller is obliged to place the rest by a time period at the buyer's discretion.*
6. *While the contract has not been terminated, the buyer has the right to place an order for an amount of items and a specified time-frame as long as (i) the running number of items ordered does not exceed the maximum stipulated in clause 2; and (ii) the time-frame must be of at least 24 hours, but may not extend beyond the contract end date specified in clause 2.*
7. *Upon placing an order, the buyer is obliged to ensure that there is enough money in escrow to cover payment of all pending orders.*
8. *Before termination of the contract, upon delivery the seller must receive payment of the order.*
9. *Upon termination of the contract, if either any orders were undelivered or more than 25% of the orders were delivered late, the buyer has the right to receive the performance guarantee placed in escrow according to clause 5.*

Figure 1. A legal contract regulating a procurement process.

In open systems (i.e. systems whose internal structure is not fully known), such as ones where one of the intervening parties is a human, verification is problematic and the best one can hope for is to observe or monitor the interaction between the parties and deduce whether or not it is compliant with the contract. Even through simple monitoring, obligations and prohibitions are easily analysed for compliance or violation. Observed behaviour, however, does not always give sufficient information to enable a decision of whether or not there was a violation of a permission.

In this paper we limit ourselves to request-based interactive systems — systems in which a number of parties may initiate interaction (hence *request-based*), but for each interaction, one of the parties has the power to allow or reject such a request (hence *interactive*). In such systems, we observe that as long as the *intention* to initiate an action, and its success or otherwise can be observed, we can also monitor for violation of permissions i.e. if the user has permission to press a button and attempts to do so, but the backend rejects the request, then we can flag that the backend has violated the user's permission.

In particular, we focus on smart contracts as an instance of such a request-based interactive system, and show how a deontic logic with permission can be fully monitored on smart contracts on a platform such as Ethereum [19]. Since smart contracts have been proposed as executable implementations enforcing real-life contracts, the ability to ensure that an implementation on a distributed ledger technology (or blockchain), matches the description of the contract is a highly desirable feature. In order to illustrate our approach, we consider a smart contract regulated by a procurement contract, shown in Figure 1, as adapted from [1]. The contract regulates the behaviour of a buyer and a seller, with the contract setting minimum and maximum order counts, a contract end date, and ensures that there is enough money in escrow to ensure at least the minimum amount of orders is financially covered, along with a monetary performance guarantee from the seller to the buyer.

The paper starts off by describing similar and related approaches in the literature in Section 2. We then go on to define the semantics of an action-based deontic logic for request-based interactive systems in Section 3 and show how smart contracts written for the Ethereum platform in Solidity can implement sound and complete monitors of such contracts in Section 4. The approach is illustrated on a procurement smart contract in Section 5. We discuss the viability of our approach and conclude in Section 6.

2. Related Work

The interaction of deontic logic and smart contracts is an area of recent interest. Idelberger et al. [9] analyse pros and cons of logic-based smart contracts (as opposed to the more common procedural-code based ones). They discuss the ability to monitor logic-based contracts as an advantage but do not explore possible implementations, which is the topic of this article. With the blooming of smart contracts questions have begun to arise around the issue of how to be sure that the smart contract is correct (eg, [11,1]).

Indeed deontic-based contract monitoring is a well known topic, with proposals for using deontic languages to monitor business processes [15], building monitors out of logic contract specifications [4], or for using certain monitoring architectures [12], even using blockchain as base technology [18][4], amongst others.

However, these proposals takes a lightweight approach to handling permissions. For instance Modgil et al. [12] consider the monitoring of permissions, but only flag when they are exercised and not when denied by another party, as we do (see Section 3).

In [2] we presented the notion of a contract between interacting parties where permissions on one party could be denied by the other by not cooperating. That work, however, flagged violations when, at a given state, one party did not provide the needed synchronisation for a shared action, independently of whether the permission-bearer party attempted the action or not — if A is permitted to borrow a book from B, and B's specification does not include the book-lending action, a violation is still flagged. However, one may argue that perhaps A never intended to exercise his or her right to borrow a book at the time — should that still count as a violation of the permission? Moreover that work requires *a priori* a full specification of the possible party actions, which is not available in the kind of interactive systems we consider here.

What was missing there, and this work addresses is the notion of *attempts*. It has been long established that a logic that only predicates over actions may be insufficient for certain deontic applications that refer to other concepts such as *intention* (e.g. [8]), much in the same sense as BDI agents. Later on, Lorini and Herzig [10] took a step further by proposing a logic that includes (and differentiates among) *intention* and *attempt*. While an *intention* is a mental state that has no direct observable effect, an *attempt* is an observable action, which may or may not succeed. Attempts permit to separate the state that the action is intended to bring about from whether the agent took some transition with a visible output. As an example, Bob might try to sell his car, yet failed to do so. The end state is Bob keeping the car and not having extra money, yet he engaged in the action of trying to sell it.

This work was built on prior philosophical traditions on what it means for an agent to intend to do an action. For space reasons we concentrate only on the two that are more

[4]Note that [18] discusses the use of blockchain as a decentralized way to monitor business processes but not the monitoring of the smart contract itself as we do.

relevant to our work: Sellars [17] poses that *"A tries to do α"* if and only if *A exercises actions to bring about the state α without being sure whether they will succeed or not.* Schroeder [16] however, reverses the word *"trying"* to the observation of the action being done by another agent: *B* perceives *A* trying an action *a* if *B* is not certain that action *a* will succeed. The labelling of something as an attempt is done by an observer, not by the exerciser of the action. Others, such as [3], have presented an alternative notion of attempt having to do with the probability p of bringing about a certain state α by certain action a. Interestingly, as the chance of $\neg\alpha$ is $1-p$, then a becomes both an attempt at α and an attempt at $\neg\alpha$ at the same time.

Our work follows the tradition of Lorini and Herzig (and others) of having actions that might succeed or not, but in our case the failure is an observation of an action attempted and denied by the system (which in turn, might be a permission violation).

3. An Operational Semantics for Deontic Contracts

In this section we present a simple action-based deontic logic for request-based interactive systems based on approaches from the literature such as [7,2], which will be used to synthesise smart contract violation monitors. By *request-based interactive systems* we mean systems in which only one party initiates interaction (hence *request-based*), but the other party may allow or reject any such request (hence *interactive*). Although we could have adopted other existing deontic logics which deal with interaction, we choose to define our own in order to obtain a clean translation process, which can then be adapted to other existing logics by adding the features as required.

Well-formed formulae in the logic will be used to denote a contract specifying the expected behaviour of the user interacting with a service. The interaction with the system will take the form of actions which are initiated by the user and accepted or rejected by the service. Formally, the logic is defined on an alphabet of event names Σ — we will use variables a, b to range over Σ. Given event name a, we will write a^Y to denote a successful attempt to perform event a, and a^N to denote that event a was attempted but was rejected. We will write $\mathbb{\Sigma}$ to denote the set of all possible observations over event names Σ: $\mathbb{\Sigma} \stackrel{df}{=} \{a^Y, a^N \mid a \in \Sigma\}$. We use variables x, y to range over $\mathbb{\Sigma}$.

For example, in the procurement contract, actions may include *login*, *requestExtension*, etc. Consider a contract where the user is allowed to request an extension only once. Then the first time the user would call *requestExtension*, she or he would expect it to succeed, but it may fail thereafter (i.e. any further requests are rejected).

The logic itself will allow the expression of obligations, prohibitions and permissions over these actions from the point of view of the user. It is worth noting that in a request-based interactive system, the obligations and prohibitions of invoking actions are the responsibility of the user (the caller), whereas satisfying the permissions which the user has is the responsibility of the underlying service. The syntax of the logic is the following:

$$\mathscr{C} ::= \top \mid \bot \mid \mathscr{O}(a) \mid \mathscr{F}(a) \mid \mathscr{P}(a) \mid [\psi]\mathscr{C} \mid \mathscr{C} \,\&\, \mathscr{C} \mid \mathscr{C};\mathscr{C} \mid \mathscr{C} \triangleright \mathscr{C} \mid \text{rec } X.\mathscr{C} \mid X$$

The logic contains the trivially satisfied ($\top$) and violated ($\bot$) contracts and ways of expressing obligation to perform an action ($\mathscr{O}(a)$), prohibition ($\mathscr{F}(a)$) and permission ($\mathscr{P}(a)$). Contracts can also be guarded by a condition on the actions which are performed ($[\psi]C$) — with the contract trivially holding if the condition does not hold. Contracts can be combined via conjunction ($C_1 \,\&\, C_2$), sequential composition ($C_1;C_2$) and reparation

$(C_1 \triangleright C_2)$, with the last operator indicating that if the first contract is violated, the second comes into force. Finally, one can use recursion (rec $X.C$) to express repeating contracts. We will use variables C, C_1, C_2 to range over contracts. In the rest of the paper, we will assume that all recursion is well-formed in that it is guarded by a condition or a deontic modality e.g. rec $X.\mathscr{O}(a);X \,\&\, [hasNotPaid]X$ is fine, but not rec $X.(X \,\&\, \mathscr{P}(a));\mathscr{F}(b)$.

Consider a contract clause in the procurement example, which states that the user is permitted to terminate the contract once a delivery has been made. This can be expressed as: rec $X.[noDelivery]X \,\&\, [madeDelivery]\mathscr{P}(endContract)$.

We define a syntactic equivalence relation over contract formulae:

$$C \,\&\, \top \equiv C \qquad \bot \,\&\, C \equiv \bot \qquad \top \,;\, C \equiv C \qquad \text{rec } X.C \equiv C[X \backslash \text{rec } X.C]$$
$$\top \,\&\, C \equiv C \qquad C \,\&\, \bot \equiv \bot \qquad \bot \triangleright C \equiv C$$

As typically done in operational semantics, this relation will be used to simplify the presentation of the semantics, and will be applied in between the steps defined by the operational semantics. The only equivalence worth commenting on is the one for recursion, which states that free instances of the recursion variable may be replaced by the recursive formula itself. If unguarded recursion were allowed, this rule can be applied infinitely often. However, with guarded recursion, we note that if the equivalence relation is applied from left to right, we can prove that the process of applying the relation until no further reductions are available acts as a total and deterministic function. If, starting from C, the result obtained is C', we will write $C \hookrightarrow C'$. We say that a contract C is normalised if $C \hookrightarrow C$.

Note that the logic allows for multiple obligations to be in place at the same time, in order to enable conformance to such concurrent obligations, we give a semantics in which a set of concurrent actions (an *action set*) is observed at once. This approach follows other similar work in the literature (e.g. [14,2]). We will use variables A, B to range over action sets (i.e. in 2^{Σ})[5].

We can now define the semantics of the logic as a relation such that $C \xrightarrow{A} C'$ if contract $C \in \mathscr{C}$ evolves to contract $C' \in \mathscr{C}$ if an action set $A \subseteq \Sigma$ is observed. The full semantics is illustrated in Figure 2.

It is worth noting a number of features of the semantics of this logic: (i) obligations can be satisfied upon an attempt by the user to perform the action, irrespective whether or not it was approved, and dually, prohibition is violated by an attempt by the user to perform the action even if it is rejected; (ii) permission[6] is violated if an attempt to perform the action occurs but is rejected by the system; (iii) conditions are assumed to be predicates over the set of actions which the user has performed. However, one may adopt alternative choices (e.g. one may choose that an attempt to perform a forbidden action which is rejected not to be a violation) without changing the results in the rest of the paper. Finally, note that although at first sight some interactions may appear missing (e.g. semantics of recursion and the triggering of a reparation) they are handled by the equivalence relation presented earlier.

Given a trace of action-sets $T \in (2^{\Sigma})^*$, we write $C \xRightarrow{T} C'$ to indicate that contract C evolves to contract C' following trace T, and applying the syntactic equivalences after

[5] In this article we use standard notation of 2^X to denote the power set of X, and X^* for finite lists over set X.

[6] We use the term permission to define a modality that other authors in the Hohfeldian tradition call a *right*, in the sense that what is permitted to a party posses a burden of compliance on the other party (or the system in our case).

$$\frac{}{\top \xrightarrow{A} \top} \qquad \frac{}{\bot \xrightarrow{A} \bot}$$

$$\frac{}{\mathscr{O}(a) \xrightarrow{A} \top}\ \{a^Y, a^N\} \cap A \neq \emptyset \qquad \frac{}{\mathscr{O}(a) \xrightarrow{A} \bot}\ \{a^Y, a^N\} \cap A = \emptyset$$

$$\frac{}{\mathscr{F}(a) \xrightarrow{A} \top}\ \{a^Y, a^N\} \cap A = \emptyset \qquad \frac{}{\mathscr{F}(a) \xrightarrow{A} \bot}\ \{a^Y, a^N\} \cap A \neq \emptyset$$

$$\frac{}{\mathscr{P}(a) \xrightarrow{A} \top}\ a^N \notin A \qquad \frac{}{\mathscr{P}(a) \xrightarrow{A} \bot}\ a^N \in A$$

$$\frac{C_1 \xrightarrow{A} C_1' \qquad C_2 \xrightarrow{A} C_2'}{C_1 \,\&\, C_2 \xrightarrow{A} C_1' \,\&\, C_2'} \qquad \frac{C_1 \xrightarrow{A} C_1'}{C_1 ; C_2 \xrightarrow{A} C_1' ; C_2} \qquad \frac{C_1 \xrightarrow{A} C_1'}{C_1 \triangleright C_2 \xrightarrow{A} C_1' \triangleright C_2}$$

$$\frac{C \xrightarrow{A} C'}{[\psi]C \xrightarrow{A} C'}\ \psi(A) \qquad \frac{}{[\psi]C \xrightarrow{A} \top}\ \neg\psi(A)$$

Figure 2. Semantics of deontic contract logic.

each step, defined as the smallest relation such that[7]: (i) if $C \hookrightarrow C'$, then $C \overset{\langle\rangle}{\Longrightarrow} C'$; and (ii) if $C \hookrightarrow C' \xrightarrow{A} C'' \overset{T}{\Longrightarrow} C'''$, then $C \overset{A:T}{\Longrightarrow} C'''$.

Although the contract semantics allows transitions over action sets, most services are accessed as a sequence of individual events. We choose to resolve concurrent norms by a special event that marks a time unit, using it to group single events into action sets. This may correspond to a temporal period (e.g. a day, with an event marking the occurrence of midnight) or otherwise (e.g. the time unit may be a login session, and the logging out event marks the end of the time unit). We will now show how such a small-step semantics can be defined using the action set semantics in order to be able to deal with systems such as smart contracts in which events happen individually.

The small-step semantics will process one action at a time, with the special event *tick* (not in Σ) to denote the end of the current time unit. We will write $\mathbb{\Sigma}_{tick}$ to denote the observable events $\mathbb{\Sigma}$ augmented by *tick*: $\mathbb{\Sigma}_{tick} \overset{df}{=} \{tick\} \cup \mathbb{\Sigma}$, and we will use variables α, β to range over this set. The small step semantics will be of the form $C_A \overset{\alpha}{\rightsquigarrow} C'_{A'}$ to denote that upon receiving event α, if $A \subseteq \mathbb{\Sigma}$ were the events observed since the last tick event, then contract $C \in \mathscr{C}$ will evolve to contract $C' \in \mathscr{C}$ with the accumulated observed events now being $A' \subseteq \mathbb{\Sigma}$:

$$\frac{}{C_A \overset{x}{\rightsquigarrow} C_{\{x\} \cup A}} \qquad \frac{C \xrightarrow{A} C' \qquad C' \hookrightarrow C''}{C_A \overset{tick}{\rightsquigarrow} C''_{\emptyset}}$$

As we did before with action set semantics, for a trace $t \in \mathbb{\Sigma}^*_{tick}$ we define $C_A \overset{t}{\rightsquigarrow} C'_{A'}$ to be the transitive closure of the small step semantics.

Another reasonable choice here is to immediately flag a violation. Note also how a tick event is essential, since at some point the actions performed need to be evaluated — an obligation to do an action without an explicit or implicit time limit is not an obligation.

[7] In the rest of the paper we use standard notation for traces: $\langle\rangle$ denotes the empty list, $x : xs$ denotes the list consisting of item x followed by list xs, $xs +\!\!+ ys$ denotes the concatenation of lists xs and ys and items(xs) denotes the set of elements appearing in list xs.

We also define a relation $\flat \in (2^{\Sigma})^* \leftrightarrow \Sigma^*_{tick}$, such that action-set trace $T \in (2^{\Sigma})^*$ is related with singleton-action trace $t \in \Sigma^*_{tick}$ if the actions in t split by tick actions are the same as the actions sets appearing in T. This relation is defined as the least relation satisfying the following:

$$\langle\rangle \flat \langle\rangle \quad \stackrel{df}{=} true$$
$$(A : T) \flat (t_1 +\!\!+ \langle tick \rangle +\!\!+ t_2) \stackrel{df}{=} A = \text{items}(t_1) \wedge tick \notin A \wedge T \flat t_2$$

Using this relation, we can formulate and prove the correctness of the small step semantics with respect to the action set semantics.

Theorem 1. Given a normalised contract $C \in \mathscr{C}$, an action-set sequence leads to a violation if and only if equivalent singleton-action traces also lead to violations:

$$\forall T \in (2^{\Sigma})^*, t \in \Sigma^*_{tick} \cdot T \flat t \implies (C \stackrel{T}{\Rightarrow} \bot) \Leftrightarrow (C_{\emptyset} \stackrel{t}{\Rightarrow\!\!\!\!\!\!\sim} \bot_{\emptyset})$$

4. Monitoring Deontic Contracts in a Blockchain

Smart contracts are concrete instances of request-based interactive systems, with the intended purpose of serving as executable implementations of real-life contracts. By being executed on blockchains there is a certain degree of transparency and dependability. Although their out-of-the box immutable transaction record can be used for analysing past behaviour, such an approach does not allow for online monitoring with real-time and on-chain reactions to violations.

In smart contracts, actions are requested by the user (typically by calling an entry-point or function of the smart contract's interface), which triggers specific business logic. In this manner, since a smart contract is intended to serve as a regulated environment within which the parties interact, the smart contract ensures that the action is allowed (according to the counterpart real-life contract) and if so carries out the expected behaviour. For example, if a buyer attempts to place an order when there is not enough money in escrow to cover it, then the smart contract should not let the order to be placed successfully (see Clause 7 of the procurement contract).

A smart contract thus enforces a deontic contract if it only allows compliant sequences of actions. This is not always possible — for instance when certain blockchain actions should signal an off-blockchain event which the smart contract should faithfully record (e.g. that a delivery was made). However, within the domain of what is observable on the blockchain, the challenge in monitoring that the smart contract behaviour matches that of the deontic contract lies in the semantic's rules which deal with attempts, particularly the rules for permission (where an attempt which is not allowed to go through results in a violation) and prohibition (where an attempt to perform a forbidden action is sufficient to trigger a violation). In other words, being able to monitor not just successful actions (i.e. a^Y) but also failed attempts (i.e. a^N). In this section we will look at how monitoring for attempts (as opposed to actual performance of the action) can be done on the Ethereum blockchain, and we discuss how we can instrument smart contracts written using the Solidity language [6] to monitor deontic contracts.

In Solidity smart contracts, the code defines an interface which parties may invoke in order to trigger particular behaviour specified in the code itself. The code itself may trigger a failure which results in the whole invocation to be dismissed (including any effects performed before the failure). For example, the code below shows a function used to terminate a contract which uses the `require` command to fail and abort execution if invoked by anyone other than the buyer or the seller:

```
function terminateContract() public {
  require(msg.sender == seller || msg.sender == buyer); // Can only be done by the seller or buyer
  ...
}
```

The fact that such failure triggers a revert of the smart contract state, as though the invocation never happened, is problematic for monitoring, since this means that if one monitors for a failure in an online manner, failure will also obliterate the observation itself. However, Ethereum provides different modalities for function calls, allowing for encapsulating function calls within a `call`, capturing such failures and signalling the outcome using a boolean value indicating success or failure of the invocation (instead of the failure). One can thus pre-process a smart contract to package its functions in order to detect both success and failure of invocations, and which are used to keep a record of events between `tick` events:

```
function terminateContractWithFailure() public {
    if (this.call(bytes4(keccak256("terminateContract()")))) {
        addSuccessfulAction(Action.TerminateContract);
    } else {
        addFailedAction(Action.TerminateContract);
    }
}
```

This approach allows us to listen to smart contract events, and record the ones that are relevant to the deontic contract to be used according to the small-step semantics of the contract language. The gathering of events corresponds to the first rule of the small-step semantics, while by encoding the deontic contract's semantics e.g. using a finite-state automaton or a symbolic automaton which we can easily instrument a smart contract with [5], we can then trigger a transition as per the second rule of the small-step semantics. That is, when `tick` is called we attempt to transition from the current state with the events that happened since the last tick:

```
function tick() public {
    ...
    transition();
    if (isViolating(currentState))
        doSomethingUponViolation();
    resetActions();
}
```

When a *tick* is invoked, the big step semantics are thus invoked and if a mismatch between the actions and the deontic contract is identified, appropriate action takes place[8]. How triggering of the `tick` function takes place depends on the contract itself, as discussed before. Instead of using the *tick* event to mark explicit time periods, we can also let it be triggered in other ways, for instance by agreement between parties (e.g. acknowledging that both will not perform any more actions in the current round), or by just one of the parties after a certain amount of time has elapsed since the last tick (e.g. as a signal that the party is ready to move to the next stage).

5. Case Study

Let us reconsider the procurement contract from Figure 1. In Figure 3, we show how the contract can be formalised in our logic. Here events do not just reflect actions of a party,

[8]How to deal with malfunctioning smart contracts is a challenge in itself, and is largely still an open problem, although an initial attempt at addressing this can be found in [13].

C1. $\mathscr{P}(TerminateContractUnlessOtherwiseForbidden)$
C2. $\mathscr{F}(TeminateContractWithItemsNotBetweenMinAndMaxItems)$
C3. $\mathscr{F}(TerminateContractBeforeEndTimestamp) \,\&\, \mathscr{F}(TerminateContractBySellerAndWithPendingOrders)$
C4. $\mathscr{F}(EnactmentWithLessThanCostOfMinimumItems)$
C5. $\mathscr{F}(AcceptContractWithLessThanGuarantee) \triangleright \mathscr{O}(SendRestOfGuarantee)$
C6. $[ContractNotTerminated]\mathscr{P}(OrderWithLessThanMaxItemsAndDeliveryTimeLessThan24HrsAndBeforeEnd)$
C7. $\mathscr{F}(OrderWithLessThanEnoughMoneyForPendingOrders)$
C8. $[ContractNotTerminated]\mathscr{F}(DeliveryWithPaymentLessThanCost)$
C9. $[TerminateContractWithPendingOrdersOr25PercentLateOrders]\mathscr{O}(SendGuaranteeToBuyer)\,\&$
$[TerminateContractWithoutPendingOrdersAnd25PercentLateOrders]\mathscr{O}(SendGuaranteeToSeller)$

$$ProcurementContract \stackrel{df}{=} \text{rec } X.[\neg\psi]((C4\,\&\,C5);X) \;\&\; [\psi](\text{rec } Y.(C1\,\&\,C2\,\&\,C3\,\&\,C6\,\&\,C7\,\&\,C8\,\&\,C9);Y)$$
$$\text{where } \psi \stackrel{df}{=} EnactmentAndSellerAcceptanceWithEnoughInEscrow$$

Figure 3. Formalisation of each of the clauses in Figure 1.

but rather actions happening in some smart contract state. It is worth noting here that the choice of meaning of the tick action is crucial, and one has to ensure that no party is able to unilaterally trigger this action to gain advantage, e.g. quickly invoking a tick so that the other party violates a pending obligation. For the sake of our case study, we implement a third party activated tick action, but there are other solutions which could be implemented, such as a unilateral time-bounded tick, in which either party can activate a tick action if at least a certain amount of time has passed since the last tick.

Both the contract and its monitoring based on the techniques outlined in Section 4 have been implemented on Ethereum in Solidity[9], thus achieving (i) a smart contract which should implement the logic of the legal contract; and (ii) a monitor which observes whether the behaviour of the parties and the logic of the smart contract actually complies with the legal contract as written in the formal deontic logic.

It should be noted that monitoring for compliance does not come for free. Execution of code in smart contracts on platforms like Ethereum require payment for so-called *gas*. The monitoring logic we add to a contract adds to the gas required when calling the original functions in such a contract. This increase (as a percentage of the original cost) is highly dependant on (i) the original smart contract; and (ii) the number of actions received between successive ticks. However, to get an idea of the magnitude of the extra cost we have done some previous studies at evaluating overheads in instrumenting monitors on Solidity smart contracts in [1]. It is worth mentioning that (i) given the critical nature of many smart contracts, the additional cost (in many cases being in the order of 10–20% of the original cost) is typically worth paying for; and (ii) given prices of ether at the time of writing the gas added is negligible in actual cost.

6. Conclusions

In computer science circles, one frequently finds conflation of contracts with specifications, mainly because obligations and prohibitions readily translate into standard specification languages. In contrast, permission has no direct correspondence in many specification languages, particularly linear time logics. In this paper, we have argued that in request-based interactive systems, respecting permission (or at least one form of permission) corresponds to respecting attempts to perform the permitted action or achieve the

[9] See `https://github.com/shaunazzopardi/deontic-monitoring-solidity/`

permitted state. Furthermore, in certain systems, attempts and their success (or otherwise) are observable events, which enables the definition of a trace semantics encompassing permission, thus allowing runtime monitoring of deontic contracts. This monitoring approach is applied to smart contracts (in our case study, to a procurement contract) in order to enable flagging when the prescribed behaviour and the behaviour of parties diverge.

One issue we have not addressed in our approach is directed modalities e.g. which party is obliged to perform an action, which would allow us to assign blame for violations. The current semantics can easily be extended to tag the modalities with parties and handle violations in a manner similar to the one we used earlier in [2].

References

[1] Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Monitoring smart contracts: CONTRACTLARVA and open challenges beyond. In *the 18th International Conference on Runtime Verification*, 2018.

[2] Shaun Azzopardi, Gordon J. Pace, Fernando Schapachnik, and Gerardo Schneider. Contract automata - an operational view of contracts between interactive parties. *Artif. Intell. Law*, 24(3):203–243, 2016.

[3] Jan M. Broersen. Modeling attempt and action failure in probabilistic STIT logic. In Toby Walsh, editor, *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Spain, 2011*, pages 792–797. IJCAI/AAAI, 2011.

[4] María-Emilia Cambronero, Luis Llana, and Gordon J. Pace. A calculus supporting contract reasoning and monitoring. *IEEE Access*, 5:6735–6745, 2017.

[5] Joshua Ellul and Gordon J. Pace. Runtime verification of ethereum smart contracts. In *Proceedings of the First International Workshop on Blockchain Dependability*, 2018.

[6] Ethereum. Solidity v0.4.24 Language Documentation. `https://solidity.readthedocs.io/en/v0.4.24/`, 2018. [Online; accessed 09-September-2008].

[7] Stephen Fenech, Gordon J. Pace, and Gerardo Schneider. Clan: A tool for contract analysis and conflict discovery. In Zhiming Liu and Anders P. Ravn, editors, *ATVA*, volume 5799 of *Lecture Notes in Computer Science*, pages 90–96. Springer, 2009.

[8] Guido Governatori, Vineet Padmanabhan, Antonino Rotolo, and Abdul Sattar. A defeasible logic for modelling policy-based intentions and motivational attitudes. *Logic Journal of the IGPL*, 17(3), 2009.

[9] Florian Idelberger, Guido Governatori, Régis Riveret, and Giovanni Sartor. Evaluation of logic-based smart contracts for blockchain systems. In *Rule Technologies. Research, Tools, and Applications - 10th International Symposium, RuleML 2016, NY, USA, 2016. Proceedings*, pages 167–183, 2016.

[10] Emiliano Lorini and Andreas Herzig. A logic of intention and attempt. *Synthese*, 163(1):45–77, 2008.

[11] Daniele Magazzeni, Peter McBurney, and William Nash. Validation and verification of smart contracts: A research agenda. *IEEE Computer*, 50(9):50–57, 2017.

[12] Sanjay Modgil, Nir Oren, Noura Faci, Felipe Meneguzzi, Simon Miles, and Michael Luck. Monitoring compliance with e-contracts and norms. *Artif. Intell. Law*, 23(2):161–196, 2015.

[13] Gordon J. Pace, Joshua Ellul, and Christian Colombo. Contracts over smart contracts: Recovering from violations dynamically. In *8th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2018)*, 2018.

[14] Cristian Prisacariu and Gerardo Schneider. A formal language for electronic contracts. In *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 174–189. Springer, 2007.

[15] Shazia Wasim Sadiq, Guido Governatori, and Kioumars Namiri. Modeling control objectives for business process compliance. In *Business Process Management, 5th International Conference, BPM 2007, Brisbane, Australia, September 24-28, 2007, Proceedings*, pages 149–164, 2007.

[16] Severin Schroeder. The concept of trying. *Philosophical Investigations*, 24(3):213–227, 2001.

[17] Wilfrid Sellars. Science and metaphysics: Variations on kantian themes. 1968.

[18] Ingo Weber, Xiwei Xu, Régis Riveret, Guido Governatori, Alexander Ponomarev, and Jan Mendling. Untrusted business process monitoring and execution using blockchain. In *Business Process Management - 14th International Conference, (BPM) 2016, Rio de Janeiro, Brazil, September 18-22, 2016. Proceedings*, pages 329–347, 2016.

[19] Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.