Runtime Verification of Contracts with Themulus

Alberto Aranda García[1], María-Emilia Cambronero[1], and Christian Colombo[3] and Luis Llana[24] and Gordon J. Pace[3]

[1] University of Castilla-La Mancha, Albacete, Spain
[2] University Complutense of Madrid, Madrid, Spain
[3] University of Malta, Malta.

Abstract. Contracts regulating the behaviour of multiple interacting parties go beyond the notion of pure properties, but allow one to document and analyse the ideal behaviour. In this paper we build upon a real-time deontic logic allowing the description of such contracts and present a runtime verification tool for monitoring of such contracts. We present a verification algorithm used to monitor contracts written in this logic and an airport agreement is used as a case study to illustrate how such agreements and contracts can be monitored using our tool with reasonable processing costs.

Keywords: Deontic Logic, Formal Methods, Runtime verification, LARVA

1 Introduction

With the rise of multi-party systems and services, the formal notion of contracts or regulated interaction between participating parties has increased in importance. For many such situations, viewing a contract as a logical property which is not to be violated is sufficient — whether it is for the verification, monitoring or enforcement of the communication between parties. However, another view is that the notion of contracts should be first-class objects which speak about ideal behaviour but also cover the possibility that the actual behaviour does not match the ideal one and its consequences. Deontic logics [10] address precisely this aspect, typically using ideal-behaviour modalities such as obligations, permissions and prohibitions. Consider a multi-party system in which, whenever a file is downloaded, then the user should pay. From a deontic perspective, we would see an obligation to perform a *pay* action whenever we see a *download* action. Within the deontic logic, one can also express reparation clauses e.g. an obligation to pay a fine is added if the obligation to pay is not satisfied. Various approaches have been proposed in the literature for the formal reasoning about

⋆ Research partially supported by the Spanish MINECO/FEDER projects DArDOS (TIN2015-65845-C3-1-R) and FAME (RTI2018-093608-B-C31) and the Comunidad de Madrid project FORTE-CM (S2018/TCS-4314) co-funded by EIE Funds of the European Union.

(O1)	$\mathcal{O}_k(a)[d] \xrightarrow{a,k} \top$
(O2)	$\mathcal{O}_k(a)[d] \xrightarrow{\overline{(a,k)}} \top$
(O3)	$\mathcal{O}_k(a)[d] \xrightarrow{b,l} \mathcal{O}_k(a)[d]\,,$ $(a,k) \neq (b,l)$
(O4)	$\mathcal{O}_k(a)[d] \xrightarrow{\overline{(b,l)}} \mathcal{O}_k(a)[d]\,,$ $(a,k) \neq (b,l)$
(O5)	$\mathcal{O}_k(a)[d] \overset{d'}{\rightsquigarrow} \mathcal{O}_k(a)[d-d']\,,$ $0 < d' \leqslant d$

(F1)	$\mathcal{F}_k(a)[d] \xrightarrow{a,k} \bot$
(F2)	$\mathcal{F}_k(a)[d] \xrightarrow{\overline{(a,k)}} \bot$
(F3)	$\mathcal{F}_k(a)[d] \xrightarrow{b,l} \mathcal{F}_k(a)[d]\,,$ $(b,l) \neq (a,k)$
(F4)	$\mathcal{F}_k(a)[d] \xrightarrow{\overline{(b,l)}} \mathcal{F}_k(a)[d]\,,$ $(b,l) \neq (a,k)$
(F5)	$\mathcal{F}_k(a)[d] \overset{d'}{\rightsquigarrow} \mathcal{F}_k(a)[d-d']\,,$ $0 < d' \leqslant d$

(P1)	$\mathcal{P}_k(a)[d] \xrightarrow{a,k} \top$
(P2)	$\mathcal{P}_k(a)[d] \xrightarrow{b,l} \mathcal{P}_k(a)[d],$ $(a,k) \neq (b,l)$
(P3)	$\mathcal{P}_k(a)[d] \xrightarrow{\overline{(a,k)}} \bot$
(P4)	$\mathcal{P}_k(a)[d] \xrightarrow{\overline{(b,l)}} \mathcal{P}_k(a)[d],$ $(a,k) \neq (b,l)$
(P5)	$\mathcal{P}_k(a)[d] \overset{d'}{\rightsquigarrow} \mathcal{P}_k(a)[d-d']\,,$ $0 < d' \leqslant d$

(C1)	$\mathrm{cond}_k(a)[d](\varphi,\psi) \xrightarrow{a,k} \varphi$
(C2)	$\mathrm{cond}_k(a)[d](\varphi,\psi) \xrightarrow{\overline{(a,k)}} \varphi$
(C3)	$\mathrm{cond}_k(a)[d](\varphi,\psi) \xrightarrow{b,l} \psi,$ $(b,l) \neq (a,k)$
(C4)	$\mathrm{cond}_k(a)[d](\varphi,\psi) \xrightarrow{\overline{(b,l)}} \psi,$ $(b,l) \neq (a,k)$
(C5)	$\mathrm{cond}_k(a)[d](\varphi,\psi) \overset{d'}{\rightsquigarrow}$ $\mathrm{cond}_k(a)[d-d'](\varphi,\psi)\,,$ $0 < d' \leqslant d$

Fig. 1. Operational Semantics transition rules 1/2

such deontic concepts e.g. defeasible logic in [11], event calculus-based in [12], dynamic logic style in $\mathcal{CL}$ [15] and automata-based in [13]. The common theme across these and other related approaches is looking at contracts as first-class entities which can be analysed and transformed independently of the systems they regulate enabling analysis such as conflict analysis or contract bias evaluation.

One recurring challenge across these approaches, beyond the deontic one, is that of addressing temporal issues [14], however, there has been limited work on deontic logic allowing reasoning about continuous time contracts, and no tools we are aware of. Recently, we have proposed Themulus [2], a real-time deontic calculus to reason about contracts over continuous time. From a practical perspective, the use of the calculus for actual verification of system behaviour with respect to a contract raises challenges, particularly due to the real-time nature of the calculus. In this paper, we present some results proving the soundness of a runtime verification algorithm of contracts written in Themulus. We have implemented this algorithm on top of the runtime verification tool Larva [6] in order to enable real-world agreements to be monitored.

2 Contracts and agents

In this section, we present some previous definitions published in [2]. We include them here to make the paper self-contained. We will assume a time domain $\mathbb{T}$ ranging over the non-negative reals[4]. In order to deal with the recursion operator, we assume a set of variables fvars over which recursion will be defined.

[4] It is worth noting that the logic we present works equally well if the natural numbers are used for a discrete time domain. However, we allow for real time values to cater for any temporal constraints.

(AO1)	$\dfrac{\varphi \xrightarrow{\alpha} \varphi',\ \psi \xrightarrow{\alpha} \psi'}{\varphi \mathsf{op} \psi \xrightarrow{\alpha} \varphi' \mathsf{op} \psi'}$
(AO2)	$\dfrac{\varphi \overset{d}{\rightsquigarrow} \varphi',\ \psi \overset{d}{\rightsquigarrow} \psi'}{\varphi \mathsf{op} \psi \overset{d}{\rightsquigarrow} \varphi' \mathsf{op} \psi'}$
(AO3)	$\dfrac{\varphi \overset{d}{\rightsquigarrow} \top,\ \psi \overset{d'}{\rightsquigarrow} \psi'}{\varphi \wedge \psi \overset{d'}{\rightsquigarrow} \psi'},\ d' \geqslant d$
(AO4)	$\dfrac{\varphi \overset{d}{\rightsquigarrow} \varphi',\ \psi \overset{d'}{\rightsquigarrow} \top}{\varphi \wedge \psi \overset{d}{\rightsquigarrow} \varphi'},\ d \geqslant d'$
(AO5)	$\dfrac{\varphi \overset{d}{\rightsquigarrow} \bot,\ \psi \overset{d'}{\rightsquigarrow} \psi'}{\varphi \vee \psi \overset{d'}{\rightsquigarrow} \psi'},\ d' \geqslant d$
(AO6)	$\dfrac{\varphi \overset{d}{\rightsquigarrow} \varphi',\ \psi \overset{d'}{\rightsquigarrow} \bot}{\varphi \vee \psi \overset{d}{\rightsquigarrow} \varphi'},\ d \geqslant d'$

(wait1)	$\mathsf{wait}(d) \overset{d'}{\rightsquigarrow} \mathsf{wait}(d-d')\ ,\ 0 < d' \leqslant d$
(wait2)	$\mathsf{wait}(d) \xrightarrow{\alpha} \mathsf{wait}(d)$

(V1)	$\dfrac{\varphi \xrightarrow{\alpha} \varphi'}{\varphi \triangleright \psi \xrightarrow{\alpha} \varphi' \triangleright \psi}$
(V2)	$\dfrac{\varphi \overset{d}{\rightsquigarrow} \varphi'}{\varphi \triangleright \psi \overset{d}{\rightsquigarrow} \varphi' \triangleright \psi}$
(V3)	$\dfrac{\varphi \overset{d}{\rightsquigarrow} \bot, \psi \overset{d'}{\rightsquigarrow} \psi'}{\varphi \triangleright \psi \overset{d+d'}{\rightsquigarrow} \psi'}$
(S1)	$\dfrac{\varphi \xrightarrow{\alpha} \varphi'}{\varphi; \psi \xrightarrow{\alpha} \varphi'; \psi}$
(S2)	$\dfrac{\varphi \overset{d}{\rightsquigarrow} \varphi'}{\varphi; \psi \overset{d}{\rightsquigarrow} \varphi'; \psi}$
(S3)	$\dfrac{\varphi \overset{d}{\rightsquigarrow} \top,\ \psi \overset{d'}{\rightsquigarrow} \psi'}{\varphi; \psi \overset{d+d'}{\rightsquigarrow} \psi'}$

(rec1)	$\dfrac{\varphi \xrightarrow{\alpha} \varphi'}{\mathsf{rec}\ x.\varphi \xrightarrow{\alpha} \varphi'[x/\mathsf{rec}\ x.\varphi]}$
(rec2)	$\dfrac{\varphi \overset{d}{\rightsquigarrow} \varphi'}{\mathsf{rec}\ x.\varphi \overset{d}{\rightsquigarrow} \varphi'[x/\mathsf{rec}\ x.\varphi]}$

Fig. 2. Operational Semantics transition rules 2/2

Contracts regulate the behaviour of a number of agents, or parties running in parallel. In this section we present the notation we will use to describe these agents and their behaviour in order to be able to formalize contracts in the following sections.

Structurally, the underlying system consists of a number of indexed agents running in parallel, using variables A, A' to represent the individual agents. The system as a whole will consist of the parallel composition of all agents indexed by a finite set $\mathcal{I}$ i.e. the system will be of the form $||_{i \in \mathcal{I}} A_i$. We will use variables $\mathcal{A}$, $\mathcal{A}'$ to denote the state of the system as a whole. Agents semantics are thus assumed to be represented as *timed labelled transition systems*:

- $A \xrightarrow{a} A'$, for $a \in \mathsf{Act}$, indicates that agent A changes to A' upon performing action a. As it is usual in process algebrae [17], the execution of actions do not consume time. The transition $A \xrightarrow{a}\!\!\!\!\!\!\rightarrow$ indicates that agent A cannot perform action a: $A \xrightarrow{a}\!\!\!\!\!\!\rightarrow \stackrel{\text{df}}{=} \neg \exists A' \cdot A \xrightarrow{a} A'$.
- $A \overset{d}{\rightsquigarrow} A'$, for $d > 0 \in \mathbb{T}$, indicates that agent A evolves to A' after d time units pass.

2.1 Contract Syntax

Definition 1. *The set of contract formulae denoted by C (with variable $\varphi \in \mathcal{C}$ to range over the contracts) is syntactically defined as follows:*

$$\varphi ::= \top \mid \bot \mid \mathcal{P}_k(a)[d] \mid \mathcal{O}_k(a)[d] \mid \mathcal{F}_k(a)[d] \mid \mathsf{wait}(d) \mid \mathit{cond}_k(a)[d](\varphi_1, \varphi_2) \\ \mid \varphi_1; \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \triangleright \varphi_2 \mid \mathsf{rec}\ x.\varphi \mid x$$

where $a \in \mathsf{Act}$, $x \in \mathsf{fvars}$, $k \in \mathcal{I}$ and $d \in \mathbb{T} \cup \{\infty\}$.

The basic formulae $\top$ and $\bot$ indicate, respectively, the contracts that are trivially satisfied and violated. Then we have the operators that represent the modalities from the deontic logic: obligations $\mathcal{O}_k(a)[d]$, prohibitions $\mathcal{F}_k(a)[d]$ and permissions $\mathcal{P}_k(a)[d]$. In all three cases the operator indicates the agent k, the action a, and the time constraint d (the modality is in force within d units of time). We can find the contract disjunction $\varphi_1 \vee \varphi_2$, and contract conjunction $\varphi_1 \wedge \varphi_2$, sequential composition $\varphi_1; \varphi_2$, delay $\mathsf{wait}(d)$, the conditional contract $\mathrm{cond}_k(a)[d](\varphi_1, \varphi_2)$, and the reparation operator $\varphi_1 \blacktriangleright \varphi_2$. Finally, there is the possibility of repetition introduced by the variable $x \in \mathsf{fvars}$ and the recursion operator $\mathsf{rec}\ x.\varphi$. Using these basic contract combinators, we can define more complex ones, for example a prohibition which persists until a particular action is performed — a prohibition on agent k from performing action a until party l performs action b, written $\mathcal{F}([a,k]\,\mathcal{U}\,[b,l])$, and defined as follows:

$$\mathcal{F}([a,k]\,\mathcal{U}\,[b,l]) \stackrel{\text{df}}{=} \mathsf{rec}\ x.\big(\mathrm{cond}_k(a)[\infty](\bot,\top) \wedge \mathrm{cond}_l(b)[\infty](\top, x)\big)$$

In order to simplify the semantics of the language, we define a congruence in the language.

Definition 2. *We define the relation $\equiv \subseteq \mathcal{C} \times \mathcal{C}$ as the least congruence relation that includes:*

1. $\varphi \wedge \top \equiv \varphi$
2. $\top \wedge \varphi \equiv \varphi$
3. $\bot \wedge \varphi \equiv \bot$
4. $\varphi \wedge \bot \equiv \bot$
5. $\varphi \vee \top \equiv \top$
6. $\top \vee \varphi \equiv \top$
7. $\varphi \vee \bot \equiv \varphi$
8. $\bot \vee \varphi \equiv \varphi$
9. $\top; \varphi \equiv \varphi$
10. $\bot; \varphi \equiv \bot$
11. $\top \blacktriangleright \varphi \equiv \top$
12. $\bot \blacktriangleright \varphi \equiv \varphi$
13. $\mathcal{O}_k(a)[0] \equiv \bot$
14. $\mathcal{F}_k(a)[0] \equiv \top$
15. $\mathcal{P}_k(a)[0] \equiv \top$
16. $\mathsf{wait}(0) \equiv \top$
17. $cond_k(a)[0](\varphi, \psi) \equiv \psi$

The operational semantics of the language is defined by the rules appearing in Figures 1 and 2. The operational semantics of the contracts has three kind of transitions: (i) $\varphi \xrightarrow{a,k} \varphi'$ to denote that contract φ can evolve (in one step) to φ' when action a is performed, which involves party k (and possibly other parties); or (ii) $\varphi \xrightarrow{\overline{(a,k)}} \varphi'$ indicating that the contract φ can evolve to φ' when the action a is not offered by any party other than k; or (iii) $\varphi \overset{d}{\rightsquigarrow} \varphi'$ to represent that contract φ can evolve to contract φ' when d time units pass. We will use variable α to stand for a label of either form: (a,k) or $\overline{(a,k)}$. The rules of the operational semantics are always applied to irreducible terms.

Next, define the predicate $\mathsf{vio}(\varphi)$ that indicates if a contract is *currently violated*.

Definition 3. *We say that an irreducible contract φ is in a* violated *state, written $\mathsf{vio}(\varphi)$, if and only if the contract has already been violated:*

$$\begin{aligned}
\mathsf{vio}(\top) &\stackrel{df}{=} \mathit{ff} & \mathsf{vio}(\bot) &\stackrel{df}{=} \mathit{tt}\\
\mathsf{vio}(\mathcal{P}_k(a)[d]) &\stackrel{df}{=} \overline{(a,k)} & \mathsf{vio}(\mathcal{O}_k(a)[d]) &\stackrel{df}{=} \mathit{ff}\\
\mathsf{vio}(\mathcal{F}_k(a)[d]) &\stackrel{df}{=} (a,k) & \mathsf{vio}(\mathsf{wait}(d)) &\stackrel{df}{=} \mathit{ff}\\
\mathsf{vio}(\varphi \wedge \varphi') &\stackrel{df}{=} \mathsf{vio}(\varphi) \vee \mathsf{vio}(\varphi') & \mathsf{vio}(\varphi \vee \varphi') &\stackrel{df}{=} \mathsf{vio}(\varphi) \wedge \mathsf{vio}(\varphi')\\
\mathsf{vio}(\varphi \blacktriangleright \varphi') &\stackrel{df}{=} \mathsf{vio}(\varphi) \wedge \mathsf{vio}(\varphi') & \mathsf{vio}(cond_k(a)[d](\varphi,\varphi')) &\stackrel{df}{=} \mathit{ff}\\
\mathsf{vio}(\varphi; \varphi') &\stackrel{df}{=} \mathsf{vio}(\varphi) & \mathsf{vio}(\mathsf{rec}\ x.\varphi) &\stackrel{df}{=} \mathsf{vio}(\varphi)
\end{aligned}$$

We can now define how contracts evolve alongside a system, and what it means for a system to satisfy a contract.

Definition 4. *Given a contract $\varphi \in \mathcal{C}$ with alphabet Act' and a system $\mathcal{A}$, we define the semantics of $\varphi \| \mathcal{A}$ — the combination of the system with the contract — with alphabet Act with $\mathsf{Act}' \subseteq \mathsf{Act}$ through the following rules:*

(M1) $\dfrac{\varphi \xrightarrow{a,k} \varphi',\ \mathcal{A} \xrightarrow{a,s} \mathcal{A}'}{\varphi \parallel \mathcal{A} \Rightarrow \varphi' \parallel \mathcal{A}'} k \in s$	***(M2)*** $\dfrac{\varphi \xrightarrow{\overline{(a,k)}} \varphi',\ \mathcal{A} \vDash \langle a, \overline{k} \rangle}{\varphi \parallel \mathcal{A} \Rightarrow \varphi' \parallel \mathcal{A}}$
(M3) $\dfrac{\mathcal{A} \xrightarrow{a,s} \mathcal{A}'}{\varphi \parallel \mathcal{A} \Rightarrow \varphi \parallel \mathcal{A}'} a \notin \mathsf{Act}'$	***(M4)*** $\dfrac{\begin{array}{c}\mathcal{A} \overset{d}{\rightsquigarrow} \mathcal{A}',\ \varphi \overset{d}{\rightsquigarrow} \varphi', \\ \forall d' < d \cdot \text{ if } \mathcal{A} \overset{d'}{\rightsquigarrow} \mathcal{A}'' \text{ and} \\ \varphi \overset{d'}{\rightsquigarrow} \varphi'' \text{ then } \mathcal{A}'' \nvDash \mathsf{vio}(\varphi'')\end{array}}{\varphi \parallel \mathcal{A} \Rightarrow \varphi' \parallel \mathcal{A}'}$

Rule **M1** and **M2** handles synchronization between the contract and the system. If an action a performed by the system is of interest to the contract, the contract evolves alongside the system (**M1**), if the contract allows an agent to perform an action but only agent k (and no other agent) is willing to engage in the action, then only the contract evolves (**M2**). Rule **M3** handles actions on the system which the contract is not interested in. Finally, rule **M4** ensures that time cannot skip over a violation.

Definition 5. *Let $\mathcal{A}$ be a system and $\varphi \in \mathcal{C}$ be a contract.*

- *System $\mathcal{A}$ can* break *φ, written $\mathsf{break}(\mathcal{A}, \varphi)$, if there exists a computation that leads to a violation of the contract: for some $n \geqslant 0$ and contracts φ_0 till φ_n such that:*

$$\varphi \parallel \mathcal{A} = \varphi_0 \parallel \mathcal{A}_0 \Rightarrow \varphi_1 \parallel \mathcal{A}_1 \Rightarrow \ldots \varphi_{n-1} \parallel \mathcal{A}_{n-1} \Rightarrow \varphi_n \parallel \mathcal{A}_n,$$

  and $\mathcal{A}_n \vDash \mathsf{vio}(\varphi_n)$.
- *System $\mathcal{A}$ may* fulfil *φ, written $\mathsf{fulfill}(\mathcal{A}, \varphi)$, if there exists a computation of the system that fulfils the contract: for some $n \geqslant 0$ and contracts φ_0 till φ_n:*

$$\varphi \parallel \mathcal{A} = \varphi_0 \parallel \mathcal{A}_0 \Rightarrow \varphi_1 \parallel \mathcal{A}_1 \Rightarrow \ldots \varphi_{n-1} \parallel \mathcal{A}_{n-1} \Rightarrow \varphi_n \parallel \mathcal{A}_n \ ,$$

  and $\mathcal{A} \nvDash \mathsf{vio}(\varphi_k)$ for $0 \leqslant k < n$, and $\varphi_n \equiv \top$.

Note that there are contracts which may never be fulfilled. An example of such a contract is $\varphi = \mathsf{rec}\ x.[a, k, \infty](\bot, \infty)$, which may never be fulfilled since there are no transitions from this contract leading to $\top$. Nevertheless, if agent k never performs action a, then neither is the contract broken.

3 Case Study

The case study presented in this section (Figure 3) is based on the Madrid Barajas airport regulations [1]. The contract describing our case study can be

1. The passenger is permitted to check in her luggage according to the stipulations of the class of ticket they purchased from their respective airline company. It is necessary that the passenger arrives at the airport, at least two hours before her flight, in order to check-in her luggage and pass the security controls. Then, the passenger is permitted to use the check-in desk within two hours before the plane takes off (t_0). φ_0
2. At the check-in desk, the passenger is obliged to present her boarding pass within 5 minutes. φ_1
3. After presenting the boarding pass, the passenger must show her passport. She has 5 minutes for this purpose. φ_2
4. The passenger is permitted to carry two pieces (of hand luggage): one personal article and one carry-on luggage. If the passenger has carry-on luggage, she is obliged to fit it into the device for hand luggage allowance, situated next to the check-in desks. φ_3
5. After presenting her passport, the passenger is permitted to board within 90 minutes and to present the hand-luggage to the airport staff within 10 minutes. φ_4 (the part before the reparation)
6. The airline company is obliged to allow the passenger to board within 90 minutes. φ_4 (the reparation part)
7. The passenger is obliged to pass the filters or security checkpoints, before they access the restricted safety areas of the airport, as boarding gates and passenger-only zones, in accordance with the safety regulations, within 60 minutes. φ_5
8. These security checkpoints consist of metal-detector arches for the passengers and X-ray detectors for their luggage. The airport security staff are permitted to carry both systems manually. φ_6
9. If the hand luggage is a personal computer or another electronic device, the airport security staff is permitted to ask the passenger to take it out of its protective case in order to be examined. φ_7
10. The passenger is obliged to take out the personal computer protection if the airport staff need to examine it. φ_8
11. The passenger is forbidden from taking articles to the security restricted area, or to the cabin of the aircraft, which constitute a risk for the health of other passengers, the crew and the safety of the aircraft and the cargo. φ_9
12. The passenger is obliged to included risk articles at check-in as baggage and/or apply to them the relevant procedure to be accepted on board. Otherwise, the security staff can requisition the articles. φ_{10}
13. Security staff is permitted to deny access to the boarding area and the airplane cabin to any passenger in possession of an object which, even if not considered forbidden, arouses their suspicions φ_7. If the passenger is stopped from carrying luggage, the airline company is obliged to put the passenger's hand luggage in the hold within 20 minutes φ_4 (reparation part).
14. The passenger is obliged to transport the liquids in individual containers with a capacity of fewer than 100 ml. These containers must be carried in a resealable, transparent plastic bag (for its easy inspection), with a capacity of not more than 1 liter. Maximum one bag per passenger. φ_{12}
15. The passenger is obliged to accompany her medication with a corresponding receipt, a medical prescription or a specified statement about the passenger's health condition, in case the security staff requires it. φ_{13}
16. Even if the regulations for liquids do not apply in the medication case, the passenger is obliged to demonstrate all liquid medication to the security staff, apart from the transparent plastic bag, used for the transport of other liquids. φ_{14}
17. The passenger is obliged to check in all firearms, which may not be transported. φ_{15}
18. The passenger has an obligation to know her rights if she wants to file a complaint. In this case, she may ask for a document at any airport in Spain, in which his rights are described, including some advice about how to act. Over and above, the passenger may also contact the Spanish National Aviation Agency (Agencia Estatal de Seguridad Aérea — AESA). φ_{16}
19. The passenger is entitled to present a claim, in case of any violation of those rights, which can result in a financial compensation or any other kind of compensation. If a passenger is unhappy with the service during a flight but is not entitled to present a claim, he still has the option to lodge a complaint or a suggestion. φ_{17}

Fig. 3. Adaptation of the Madrid Barajas airport regulations

formalized using our contract calculus as follows:

$$\begin{aligned} PBS ::= & ((\varphi_0 \wedge \varphi_{10} \wedge \varphi_{15}); \varphi_1; \varphi_2; \varphi_3; \\ & (\varphi_5 \wedge \varphi_6 \wedge \varphi_7 \wedge \varphi_8 \wedge \varphi_9 \wedge \varphi_{12} \wedge \varphi_{13} \wedge \varphi_{14}); \\ & (\varphi_4 \wedge \varphi_{11})) \wedge \varphi_{16} \wedge \varphi_{17}) \end{aligned}$$

Where the formulas $\varphi_1, \ldots, \varphi_{17}$ are defined in Figure 4 such that p, S, c refer to the passenger, the security airport staff, and the airline company, respectively; while t_0 is departure estimated time. Note that the clauses φ_0 to φ_{17} are used to express the different parts of the contract, and combined in the top-level contract expression *PBS*. The translation is quite straightforward although it is not automated. We have attach a formula to each point of the contract indicating where it has been formalized.

$\varphi_0 ::= \mathcal{P}_p(\texttt{checkin})[t_0 - 120]$
$\varphi_1 ::= \mathcal{O}_p(\texttt{PBP})[5]$
$\varphi_2 ::= \mathcal{O}_p(\texttt{ShP})[5]$
$\varphi_3 ::= \mathcal{P}_p(\texttt{CToHL})[120]$
$\varphi_4 ::= (\mathcal{P}_p(\texttt{board})[90]; \mathcal{P}_p(\texttt{hl})[10]) \triangleright (\mathcal{O}_c(\texttt{board})[90]; \mathcal{O}_c(\texttt{hlhold})[20])$
$\varphi_5 ::= \mathcal{O}_p(\texttt{PSC})[60]$
$\varphi_6 ::= \mathcal{P}_S(\texttt{CD})[120]$
$\varphi_7 ::= \mathcal{P}_S(\texttt{EPP})[120]$
$\varphi_8 ::= \mathcal{O}_p(\texttt{TOP})[120]$
$\varphi_9 ::= (\mathcal{F}([\texttt{TRA}, p]\,\mathcal{U}\,[\texttt{landing}, p]))$
$\varphi_{10} ::= \mathcal{O}_p(\texttt{CRA})[t_0 - 120] \triangleright \mathcal{P}_S(\texttt{RRA})[10]$
$\varphi_{11} ::= \mathcal{P}_S(\texttt{DRA})[10]$
$\varphi_{12} ::= \mathcal{O}_p(\texttt{liquids})[120]$
$\varphi_{13} ::= \mathcal{O}_p(\texttt{medication})[120]$
$\varphi_{14} ::= \mathcal{O}_p(\texttt{liq})[120]$
$\varphi_{15} ::= \mathcal{O}_p(\texttt{firearms})[t_0 - 120]$
$\varphi_{16} ::= \mathcal{O}_p(\texttt{rights})[120]$
$\varphi_{17} ::= \mathcal{P}_p(\texttt{complaint})[120]$

`checkin`	Go to the checkin desk
`PBP`	Present boarding pass
`ShP`	Show her passport
`CToHL`	Carry two hand luggage
`board`	Board
`hl`	Board with hand luggage
`PSC`	Pass the security checkpoints
`CD`	Carry detectors
`EPP`	Examine passanger PC
`TOP`	Take out PC
`TRA`	Taking risk articles
`CRA`	Check in risk articles
`RRA`	Requisition of risk articles
`DRA`	Access to boarding area
`hlhold`	Put her hand luggage in the hold
`board`	Board
`liquids`	Liquids of 100ml
`medication`	Medication with receipt
`liq`	Demonstrate liquid medication
`firearms`	Check in the firearms
`rights`	Know her rights
`complaint`	File a complaint

Fig. 4. Madrid Barajas airport regulations formulae.

Then, this approach allows us to check and determine if any of the agents involved in the plane boarding system breaks the contract. In this case, our formalism allows us to determine, for instance, if it was the passenger who violated the contract and if so why, e.g. because she did not present her boarding pass within the specified time at the check-in desk, or because she was taking articles to the restricted security area.

4 Runtime Verification

The operational semantics we give to contracts provides us with a framework for contract monitoring: to monitor contract $\psi \in \mathcal{C}$, we start the monitor in state ψ and update the state whenever the system performs an action according to the operational semantics. A violation is reached once the violation predicate is satisfied by the system. In the rest of this section, we concretely show how our logic can be automatically monitored using a derivative-based algorithm [4].

The idea behind derivative-based or term rewriting-based monitoring is that the formula still to be monitored is used as the state of the monitoring system. Whenever an event e is received with the system being in state ψ, the state is updated to ψ' such that any trace of events es matches ψ', if and only if $e : es$ (the trace starting with e, followed by es) matches ψ. This is repeated and a violation is reported when (and if) the monitoring state is reduced to a formula which matches the empty trace. In our contract logic, the operational

semantics provide precisely this information, with ψ' being chosen to be the (unique) formula such that[5] $\psi \xrightarrow{a,k};\longmapsto \psi'$ (where the monitoring systems observes the action a performed by party k), and $\mathsf{vio}(\psi)$ indicates whether ψ matches the empty string (immediately violates the contract).

In timed logics, this approach has to be augmented with timeout events which, in the absence of a system event, still change the formula. For example, the contract $\mathsf{wait}(d);\varphi$ would evolve to φ upon d time units elapsing. Similarly, if d time units elapse (with no system events received), we evolve the contract $\mathcal{O}_k(a)[d];\varphi$ to $\bot;\varphi$, which is equivalent to $\bot$, thus enabling us to flag the violation as soon as it happens. If we were to wait for a system event, the violation might end up being identified too late. In our case, we use the timeout function to enable the setting of a timer to trigger the monitoring state update, evolving φ to the unique formula φ' according to the timed operational semantics $\varphi \overset{\mathrm{timeout}(\varphi)}{\rightsquigarrow};\longmapsto \varphi'$. Also, system events carry a timestamp, through which the contract can be moved ahead in time upon receiving the event.

In order to formalise these ideas, we need a notion of structural equivalence. Intuitively, two contracts are structurally equivalent if they only differ in the time constraints and so they can perform the same actions.

Definition 6. *Consider the relation $\mathcal{R} \subseteq \mathcal{C} \times \mathcal{C}$ defined as follows:*

$$
\begin{aligned}
\mathcal{R} \overset{df}{=} & \{(\top,\top),(\bot,\bot)\} \\
& \cup \{(\mathcal{F}_k(a)[d], \mathcal{F}_k(a)[d']) \mid d, d' > 0\} \\
& \cup \{(\mathcal{P}_k(a)[d], \mathcal{P}_k(a)[d']) \mid d, d' > 0\} \\
& \cup \{(\mathcal{O}_a(k)[d], \mathcal{O}_a(k)[d']) \mid d, d' > 0\} \\
& \cup \{(\mathsf{wait}(d), \mathsf{wait}(d')) \mid d, d' > 0\} \\
& \cup \{(\mathit{cond}_k(a)[d](\varphi_1,\varphi_2), \mathit{cond}_k(a)[d'](\varphi_1,\varphi_2)) \mid d, d' > 0, \\
& \qquad \varphi_1, \varphi_2 \in \mathcal{C}\}
\end{aligned}
$$

The structural equivalence congruence*, denoted by $\equiv_s$ is the smallest congruence containing $\mathcal{R}$.*

Proposition 1. *Let $\varphi, \psi \in \mathcal{C}$ be two structurally equivalent contracts, $\varphi \equiv_s \psi$. Then $\mathsf{vio}(\varphi) = \mathsf{vio}(\psi)$ and $\varphi \xrightarrow{\alpha} \chi$ iff $\psi \xrightarrow{\alpha} \chi$.*

Proof. The proof follows using structural induction.

Definition 7. *The* timeout *of a contract φ, written timeout(φ), is inductively defined as follows:*

[5] We write $r;s$ to indicate the forward composition of the two relations r and s, and use $\longmapsto$ to denote the reflexive transitive closure of the timed labelled transition systems.

$$
\begin{aligned}
timeout(\top) &\stackrel{df}{=} \infty \\
timeout(\mathcal{P}_k(a)[d]) &\stackrel{df}{=} d \\
timeout(\mathcal{F}_k(a)[d]) &\stackrel{df}{=} d \\
timeout(\mathsf{wait}(d)) &\stackrel{df}{=} d \\
timeout(\varphi_1 \triangleright \varphi_2) &\stackrel{df}{=} timeout(\varphi_1) \\
timeout(\varphi_1 \wedge \varphi_2) &\stackrel{df}{=} \min\{timeout(\varphi_1), timeout(\varphi_2)\} \\
timeout(\varphi_1 \vee \varphi_2) &\stackrel{df}{=} \min\{timeout(\varphi_1), timeout(\varphi_2)\}
\end{aligned}
\qquad
\begin{aligned}
timeout(\bot) &\stackrel{df}{=} \infty \\
timeout(\mathcal{O}_k(a)[d]) &\stackrel{df}{=} d \\
timeout(cond_k(a)[d](\varphi_1, \varphi_2)) &\stackrel{df}{=} d \\
timeout(\varphi_1; \varphi_2) &\stackrel{df}{=} timeout(\varphi_1) \\
timeout(\mathsf{rec}\ x.\varphi \mid x) &\stackrel{df}{=} timeout(\varphi)
\end{aligned}
$$

Finally, we obtain the result that we need: any timed transition taking less than the timeout of a contract preserves the structure of a contract.

Proposition 2. *Given contract φ and time $t < timeout(\varphi)$, advancing φ by t time units preserves the structure of the contract: if $\varphi \overset{t}{\rightsquigarrow} \varphi'$, then: $\varphi \equiv_s \varphi'$.*

Proof. The proof is simple by structural induction.

We can now define the closure of a contract φ as all formulae reachable from φ through action transitions and timeout time transitions.

Definition 8. *We define the* closure *of a contract formula φ, written $closure(\varphi)$, to be the set of all contract formulae reachable through a combination of visible action transitions and timeout transitions. Formally, $closure(\varphi)$ is the smallest set such that: (i) $\varphi \in closure(\varphi)$; (ii) if $\varphi_1 \in closure(\varphi)$, and $\varphi_1 \xrightarrow{a,k} \varphi_2$, then $\varphi_2 \in closure(\varphi)$; and (iii) if $\varphi_1 \in closure(\varphi)$, and $\varphi_1 \overset{timeout(\varphi_1)}{\rightsquigarrow} \varphi_2$, then $\varphi_2 \in closure(\varphi)$.*

It is easy to prove, that for a contract φ whose time constraints are non-zero constants, the closure of φ does not exhibit Zeno-like behaviour[6]. It also follows that the relations of timeout time steps and visible event steps are sufficient to characterise the operational semantics progress of a contract to a violation or otherwise.

4.1 A Monitoring Algorithm

The monitoring algorithm for our contract logic is shown in Algorithm 1. The state of the monitor is stored in variable *contract* while variable *systime* keeps track of the last timestamp processed by the system. Initially, these variables are set to ψ and 0 (line 1). The monitoring algorithm is effectively a loop (lines 2–17) which checks whether there was a violation upon every iteration. Upon entering the loop, any pending timer triggers are replaced (lines 3–5), enacting a process which creates a special *timeout* event (line 4) to be launched (asynchronously) after the current contract times out. In the meantime, execution is blocked until

[6] By Zeno-like behaviour, we mean an infinite number of arbitrarily smaller time steps whose sum converges, thus blocking time from progressing.

an event is received (line 6). If the event received is the *timeout* event, the monitored formula is updated accordingly using the *timestep* function which returns the unique formula satisfying $\psi \overset{t}{\rightsquigarrow} \mathit{timestep}(\varphi, t)$ with the time advanced by $\mathit{timeout}(\psi)$ time units (lines 7–10). If, however, the event received is a system event e with timestamp t, the monitoring state is updated by first advancing time by $(t - \mathit{systime})$ time units, and then stepping forward using the *step* function which returns the unique formula such that $\psi \xrightarrow{e} \mathit{step}(\psi, e)$ (lines 11–14). Finally, the *systime* variable is updated accordingly (line 16).

It is worth noting that the algorithm replicates the two types of transitions required to advance a contract: (i) maximally advancing time until the structure of the contract changes (the case of a timeout event); and (ii) processing a system event. This ensures that the state of the contract monitor advances steadily in correct steps (assuming that *timestep* and *step* correctly implement the rules from the semantics). Furthermore, progress is ensured since stepping along maximal time steps never results in Zeno-like behaviour.

```
 1 contract = φ; systime = 0;
 2 while ¬vio(contract) do
 3 |   reset timer to timeout(contract))
 4 |   |   createEvent(Timeout);
 5 |   end
 6 |   switch getEvent() do
 7 |   |   case Timeout do
 8 |   |   |   Δt = timeout(contract);
 9 |   |   |   contract = timestep(contract, Δt);
10 |   |   end
11 |   |   case Event e with Timestamp t do
12 |   |   |   Δt = t − systime;
13 |   |   |   contract = step(timestep(contract, Δt), e);
14 |   |   end
15 |   end
16 |   systime = systime + Δt;
17 end
18 report(Violation);
```

Algorithm 1: Algorithm to monitor timed contracts

5 Runtime Verification Using Larva

Rather than programming the runtime verification algorithm from scratch, we have built it on top of an existing runtime verification tool. We used Larva [6], which uses Dynamic Automata with Timers and Events (DATEs) as a specification language. DATEs are symbolic timed automata enriched in many aspects.

There are three main elements in DATE transitions: (i) *Events* refer to observable actions which a DATE may react to. (ii) *Conditions* are boolean expressions taking into consideration both the DATE's symbolic state and the system state, which decides whether a transition is taken or not. (iii) *Actions* are modifications that are done to the DATE or system state upon observing an event and satisfying the condition. What follows formally defines these concepts.

Events which a DATE will be able to react to are either (i) system events over an alphabet SystemEvent, corresponding to control- or data-flow points of interest during the execution of the system; or (ii) timer events which are triggered upon a timer t reaching a threshold limit L written $t@L$. Timer limits can either take the form of a time constant $T \in \mathbb{T}$ (where $\mathbb{T}$ refers to the continuous time domain), or deadline variables $D \in \text{Deadline}$. Unlike constant limits, deadline variables can be dynamically modified during the traversal of the DATE.

$$\text{Event} ::= \text{SystemEvent} \mid \text{Timer}@(\mathbb{T} \cup \text{Deadline})$$

DATE conditions and actions may also refer to the state of the system which is being monitored e.g. to react to a *login* event only if the system is in alert mode. The state of the system σ will be assumed to range over the type State_s. Besides, monitors may keep their own state, e.g. the monitor may keep track of how many users are logged in, in order to react to a *login* only when more than 100 concurrent users are using the system. The symbolic DATE state μ will be assumed to range over the type State_m.

In addition, a DATE configuration will also keep track of the timer values $\tau \in \text{State}_T$, assigning a time value to each time such that $\text{State}_T = \text{Timer} \to \mathbb{T}$. Similarly, it keeps track of the current value of the timer variable deadlines $\delta \in \text{State}_D$ where $\text{State}_D = \text{Deadline} \to \mathbb{T}$. We will abuse notation and write $\delta(L)$ to extend the function to work also on constant deadlines (in which case that constant deadline is returned) and $\tau + \Delta$ (where $\Delta \in \mathbb{T}$) to denote the timer state in which all timers are advanced by Δ time units. The symbolic state of a DATE is thus defined to be a combination of all these parts: $\text{State}^+_M = \text{State}_m \times \text{State}_T \times \text{State}_D$.

Conditions $c \in \text{Condition}$ are predicates over the system and full monitoring state:

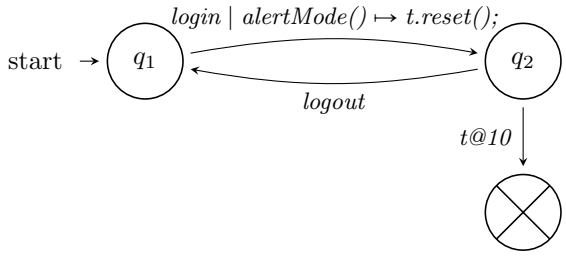
$$\text{Condition} = (\text{State}_s \times \text{State}^+_M) \to \mathbb{B}$$

Similarly, actions $\alpha \in \text{Action}$ are functions which, based on the system state, may update any part of the full monitoring state:

$$\text{Action} = (\text{State}_s \times \text{State}^+_M) \to \text{State}^+_M$$

Formally Defining DATEs A DATE is quadruple $\langle Q, q_0, K, B \rangle$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $K \subseteq (Q \times \text{Event} \times \text{Condition} \times \text{Action} \times Q)$ is a set of event-condition-action transitions, and $B \subseteq Q$ is a set of bad states. We will write $q \xrightarrow{e|c \mapsto \alpha} q'$ to denote a transition $(q, e, c, \alpha, q') \in K$.

The following figure shows an example of a DATE whereupon the detection of a login event on alert mode, a timer of ten minutes is started. If the ten minutes elapse before a logout, the DATE reaches a bad state.



The *event triggers* from a DATE state q, written $\mathit{triggers}(q)$, are defined to be all events which appear on transitions outgoing from q: $\mathit{triggers}(q) \stackrel{\text{df}}{=} \{e \mid \exists q', c, \alpha \cdot q \xrightarrow{e|c\mapsto\alpha} q'\}$.

The semantics of a DATE with dynamic timer deadlines can now be defined using this notation. The configuration of the monitor consists of (i) the state $q \in Q$ of the DATE; and (ii) the symbolic monitoring state $\sigma \in \textsc{State}_M^+$ of the monitor. Given a monitoring configuration, the *earliest timer trigger* is the least time which will trigger an outgoing timer event transition if no other event is received:

$$\begin{aligned}\mathit{earliest}(q,(\mu,\tau,\delta)) \stackrel{\text{df}}{=} \min\{\delta(L)-\tau(t) \mid t@L \in \mathit{triggers}(q)\\ \wedge\ \delta(L) \geqslant \tau(t)\}\end{aligned}$$

The semantics of DATEs will specify how the configuration of the DATE changes upon event triggering or time passing. We will have two forms of operational semantics relations: (i) $C \xrightarrow{e,\Delta,\sigma} C'$ to denote that the monitor moves from configuration C to C' upon the system receiving event e after Δ time units (from the last transition) and with the system state snapshot at that time being σ; (ii) $C \overset{\Delta,\sigma}{\rightsquigarrow} C'$ to denote that the monitor goes from configuration C to C' after Δ time units of inactivity at the end of which the system state is σ.

The first relation is defined with the implicit condition that an event e has triggered and another two preconditions: the existence of a transition triggering on e and the satisfaction of the condition c. The side-condition ensures that no timer-triggered transitions should have modified the configuration before.

$$\frac{q \xrightarrow{e|c\mapsto\alpha} q' \qquad c(\sigma,(\mu,\tau,\delta))}{(q,(\mu,\tau,\delta)) \xrightarrow{e,\Delta,\sigma} (q',\alpha(\mu,\tau+\Delta,\delta))}\ \mathit{earliest}(q,(\mu,\tau,\delta)) > 0$$

The second relation is similar to the previous but, while not requiring the occurrence of a system event, requires that the timer has reached its deadline.

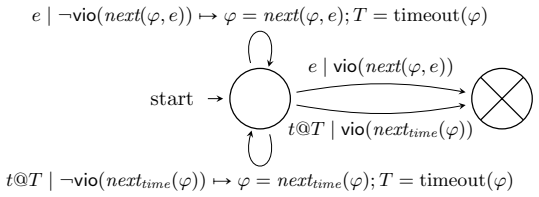
$$\frac{q \xrightarrow{t@L|c\mapsto\alpha} q' \qquad \tau(t) + \Delta = \delta(L) \qquad c(\sigma,(\mu,\tau,\delta))}{(q,(\mu,\tau,\delta)) \overset{\Delta,\sigma}{\rightsquigarrow} (q',\alpha(\mu,\tau+\Delta,\delta))}\ \mathit{earliest}(q,(\mu,\tau,\delta)) = 0$$

Building on the earlier example, consider the case where the automaton is in the second state with the timer just reset: $(q_2, (\varnothing, t \mapsto 0, t \mapsto 10))$. If a logout event occurs after six minutes, then $\Delta = 6$, while $\mathit{earliest}(q_2, (\varnothing, t \mapsto 6, t \mapsto 10)) = 10-6 = 4$. Therefore the first relation would apply, updating the configuration to $(q_1, (\varnothing, t \mapsto 6, t \mapsto 10))$. On the other hand, if no logout event occurs within ten minutes, then $\mathit{earliest}(q_2, (\varnothing, t \mapsto 0, t \mapsto 10)) = 10 - 0 = 10$ causing the second relation to be applied resulting in the configuration $(q_\times, (\varnothing, t \mapsto 10, t \mapsto 10))$.

6 Implementation

The operational semantics given to the contracts, along with the timers and events, provide the framework to transform contracts in our calculus into DATEs which can be used to runtime verify the performance of parties involved. More details about the implementation can be found in [3] and in the Appendix.

We have adapted the derivative-based [4] runtime verification algorithm in order to obtain a DATE which reacts to the events appropriately. The resulting two-state DATE keeps track of the current contract in a variable φ and updates triggers on either an event or the timeout triggers. Initially, φ is set to the contract to monitor, while T to timeout(φ):



The function $\mathit{next}(\varphi, e)$ corresponds to $\mathit{step}(\mathit{timestep}(\varphi, \Delta t), e)$ while $\mathit{next}_{time}(\varphi)$ corresponds to $\mathit{timestep}(\varphi, \Delta t)$ — in both cases Δt is the time elapsed since the last processed event. Using this construction, a trace leads to the bad state of the DATE if and only if it violates the initial contract. It is worth noting that since any computable function can be embedded as the action of a DATE transition, the translation is made possible by the computability of derivatives over time and event steps.

Practical Evaluation To test our approach, we implemented the case study in Java with each action represented as a method call. When evaluated empirically, runtime verification tools, typically (e.g. [5]) get evaluated by comparing the time needed to run the system with and without monitoring. In this case, this is not practical since the system is simply a sequence of dummy method executions.

Instead, the purpose of this quantitative evaluation is (i) to verify the correct behaviour of the monitor, i.e., that a violation is indeed reported when it actually occurs and vice versa; and (ii) to verify our intuition that the monitor will scale linearly with the size of the execution of the underlying system.

A number of test cases were generated, each representing the interactions of a single user involving a varying number of actions. In each case, the monitor verdict was as expected. Subsequently, different levels of traffic were generated by launching several users in parallel ranging from 100 to 100,000 users. The experiment[7] was run on a laptop with an Intel i7-855U processor, 16GB RAM.

Thousands of users	**0.1**	**0.5**	**1**	**5**	**10**	**50**	**100**
No monitoring (s)	0.0556	0.231	0.261	0.736	0.811	7.00	12.1
With monitoring (s)	0.104	0.433	0.595	2.16	4.05	18.1	38.4
Difference (s)	0.0480	0.202	0.334	1.43	3.24	11.1	26.4
Difference per user (ms)	0.480	0.404	0.334	0.285	0.324	0.223	0.264

The results in the table above show that as the number of users increases, the CPU time per user stabilises at around 0.3 milliseconds. This confirms our reasoning that since the individual user monitors do not interact, the monitoring effort scales linearly to the number of users. One would only expect this trend to stop when reaching a large number of users such that the performance of an underlying framework starts deteriorating, e.g., the thread pool grows larger than what is efficiently manageable.

Regarding memory, since contract monitors only need to keep track of a state of bounded size per user per contract, this was not considered to be an issue.

7 Conclusions and Future Work

In this paper, we have presented a runtime verification algorithm for a real-time contract calculus, proved to be correct. Also, we presented an implementation of the algorithm as part of an established runtime verification tool Larva.

It is worth noting that despite the fact that there is much work on real-time deontic logics (see [2] for a summary and comparisons of such works), and limited work on monitoring of deontic logics (e.g. see [7,16,8]), the overlap between the two has been largely neglected.

There are various research directions this research opens. Regarding the runtime verification aspect, an interesting challenge is how we can use our techniques for runtime enforcement: starting from a specification, how we can synthesise algorithmic machinery to ensure that the system under scrutiny does not violate the specification, e.g. by delaying or injecting events. In particular, there is a body of work on runtime enforcement of timed properties, e.g. [9] which could offer insight on how our work can be extended to build contract enforcement engines, a notion that has not been widely explored in the deontic logic world.

[7] Code is available at: `https://github.com/aarandag/larva-timedcontracts`

References

1. Madrid-Barajas Airport. Airport regulations. `https://www.aeropuertomadrid-barajas.com/eng/air-passenger-rights.htm`, `https://www.aeropuertomadrid-barajas.com/eng/regulations-hand-luggage.htm` and `https://www.aeropuertomadrid-barajas.com/eng/checkin-madrid-airport.htm`, last access 2020/05/25, 2020.
2. Alberto Arana, María-Emilia Cambronero, Christian Colombo, Luis Llana, and Gordon J. Pace. Themulus: A timed contract-calculus. In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development*, pages 193–204, 2020.
3. Alberto Aranda García, María-Emilia Cambronero, Christian Colombo, Luis Llana, and Gordon J. Pace. Themulus: A timed contract-calculus. Technical Report TR-01-20, Universidad Complutense de Madrid, 2020.
4. Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, October 1964.
5. Feng Chen and Grigore Rosu. Mop: an efficient and generic runtime verification framework. In *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2007, October 21-25, 2007, Montreal, Quebec, Canada*, pages 569–588, 2007.
6. Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems (FMICS)*, volume 5596 of *Lecture Notes in Computer Science*, pages 135–149, L'Aquila, Italy, 2008.
7. Stephen Cranefield. A rule language for modelling and monitoring social expectations in multi-agent systems. In Olivier Boissier, Julian A. Padget, Virginia Dignum, Gabriela Lindemann, Eric T. Matson, Sascha Ossowski, Jaime Simão Sichman, and Javier Vázquez-Salceda, editors, *Coordination, Organizations, Institutions, and Norms in Multi-Agent Systems, AAMAS 2005 International Workshops on Agents, Norms and Institutions for Regulated Multi-Agent Systems, ANIREM 2005, and Organizations in Multi-Agent Systems, OOOP 2005, Utrecht, The Netherlands, July 25-26, 2005, Revised Selected Papers*, volume 3913 of *Lecture Notes in Computer Science*, pages 246–258. Springer, 2005.
8. Mehdi Dastani, Paolo Torroni, and Neil Yorke-Smith. Monitoring norms: a multi-disciplinary perspective. *Knowledge Eng. Review*, 33:e25, 2018.
9. Yliès Falcone, Thierry Jéron, Hervé Marchand, and Srinivas Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Sci. Comput. Program.*, 123:2–41, 2016.
10. Georg Henrik Von Wright. Deontic Logic. *Mind*, 60(237):1–15, January 1951.
11. Guido Governatori, Antonino Rotolo, and Giovanni Sartor. Temporalised normative positions in defeasible logic. In *The Tenth International Conference on Artificial Intelligence and Law, Proceedings of the Conference, June 6-11, 2005, Bologna, Italy*, pages 25–34, 2005.
12. Mustafa Hashmi, Guido Governatori, and Moe Thandar Wynn. Modeling obligations with event-calculus. In *Rules on the Web. From Theory to Applications — 8th International Symposium, RuleML 2014, Co-located with the 21st European Conference on Artificial Intelligence, ECAI 2014, Prague, Czech Republic, August 18-20, 2014. Proceedings*, pages 296–310, 2014.
13. Gordon J. Pace and Fernando Schapachnik. Contracts for Interacting Two-Party Systems. In *FLACOS'12*, volume 94 of *ENTCS*, pages 21–30, 2012.

14. Gordon J. Pace and Gerardo Schneider. Challenges in the specification of full contracts. In *Integrated Formal Methods, 7th International Conference, IFM 2009, Düsseldorf, Germany, February 16-19, 2009. Proceedings*, pages 292–306, 2009.
15. Cristian Prisacariu and Gerardo Schneider. A dynamic deontic logic for complex contracts. *The Journal of Logic and Algebraic Programming*, 81(4):458 – 490, 2012. Special Issue: NWPT 2009.
16. Bas Testerink, Mehdi Dastani, and John-Jules Ch. Meyer. Norm monitoring through observation sharing. In *Proceedings of the European Conference on Social Intelligence (ECSI-2014), Barcelona, Spain, November 3-5, 2014*, pages 291–304, 2014.
17. Wang Yi. CCS + time = an interleaving model for real time systems. In *Automata, Languages and Programming, 18th International Colloquium, ICALP91, Madrid, Spain, July 8-12, 1991, Proceedings*, pages 217–228, 1991.