Artificial Intelligence and Law manuscript No.
(will be inserted by the editor)

Contract Automata

An Operational View of Contracts Between Interactive Parties

Shaun Azzopardi · Gordon J. Pace · Fernando Schapachnik · Gerardo Schneider

Received: date / Accepted: date

Abstract Deontic logic as a way of formally reasoning about norms, an important area in AI and law, has traditionally concerned itself about formalising provisions of general statutes. Despite the long history of deontic logic, given the wide scope of the logic, it is difficult, if not impossible, to formalise all these notions in a single formalism, and there are still ongoing debates on appropriate semantics for deontic modalities in different contexts. In this paper, we restrict our attention to contracts between interactive parties, which are both general enough to be an interesting object of study but specific enough so as to narrow down the debates regarding the meaning of modalities, and present a formalism for reasoning about them.

Keywords Deontic Logic · Contract Automata · $\mathscr{CL}$

Published in the Journal of *Artificial Intelligence and Law*, 24(3):203-243, September 2016. DOI: 10.1007/s10506-016-9185-2

Shaun Azzopardi
Department of Computer Science
University of Malta
E-mail: shaun.azzopardi@um.edu.mt

Gordon J. Pace
Department of Computer Science
University of Malta
E-mail: gordon.pace@um.edu.mt

Fernando Schapachnik
Departamento de Computación, FCEyN,
Universidad de Buenos Aires, Argentina
E-mail: fschapachnik@dc.uba.ar

Gerardo Schneider
Department of Computer Science and Engineering
University of Gothenburg
E-mail: gerardo@cse.gu.se

1 Introduction

Deontic logic as a way of formally reasoning about norms, an important area in AI and law, has traditionally concerned itself about formalising provisions of general statutes. Despite the long history of deontic logic, given the wide scope of the logic (from the philosophical perspective), it is difficult, if not impossible, to formalise all these notions in a single formalism, and there are still ongoing debates on appropriate semantics for deontic modalities in different contexts, as witnessed by the extensive research from both the philosophical and the logical point of view [42, 26]. In this paper, we restrict our attention to contracts, agreements between different parties regulating their behaviour, which are both general enough to be an interesting object of study but specific enough so as to narrow down the debates regarding the meaning of modalities. In particular, we focus on contracts over interactive parties.

To represent contracts in a formal manner, we present *contract automata* — a *kernel formalism*, i.e., not a way for people to write their contracts, but rather a formalism into which contracts could be compiled, much like Kripke structures [18] as used in model checking, or the simplified forms of Java and other languages used in program analysis [38]. In this formalism, contracts are represented using automata tagged with deontic clauses in each state. With the behaviour of the participating parties similarly modelled using synchronous automata, one can formally articulate the semantics of the contract.

This article synthesises much of our prior work about this formalism [28, 30, 32, 3] and extends it with the following new contributions:

- We present a cleaner and more complete formalisation of contract automata;
- We give an overview of tool support we have built, supporting reasoning about conflicts in contracts;
- We extend our previous discussion of alternative ways of dealing with reparations in contract automata, and present a new approach — that of Hierarchical Contract Automata;
- We present a semantic preserving bidirectional translation from contract automata to a subset of the formal language for contracts $\mathscr{CL}$ [34, 36].

To enable formal reasoning about contracts, one requires a model in which the two parties interact — where interaction is taken to mean that they have to agree as to which actions are to be performed. Here we present such a model structured as follows:

- The behaviour of each interacting party is modelled using a multi-action automaton (Definition 1). Parties interact through synchronous composition [1] and multi-action labels on transitions[1].
- Contracts that regulate the behaviour of the parties are also modelled using multi-action automata (*contract automata*, see Definition 2), tagged with deontic clauses (Definitions 5, 6 and 7).

[1] Multi-actions are necessary, since it would otherwise be impossible not to violate a contract which enforces two different obligations at the same time.

- The synchronous composition of the parties and the contract is called a *regulated two-party system* (Definition 3). It provides the ability to predicate over deontic clauses enforced at each possible step of interaction between the two parties.
- A notion of a well-behaved party is presented in Definition 10, as a party that is guaranteed not to violate the contract.
- The notion of relative *contract strictness* is then presented (Section 2.2), allowing results about deontic operators, e.g., the desired property of deontic logics that whatever is obligation is stricter than permission from the perspective of both parties (see Section 2.3).
- Finally, with the introduction of *mutually exclusive actions*, we can see more interesting relationships among deontic modalities (Section 2.4).

Once the basics are established, Section 3 discusses conflicting clauses defined over a simple axiomatic framework. Section 4 enhances the formalism to enable reasoning about reparations, discussing different alternatives including Hierarchical Contract Automata that, besides being useful for writing contract automata more compactly, serve the purpose of allowing both compositional reasoning and reparations.

The formalism has been successfully employed to provide a new insight on the discussion of whether or not the Hohfeldian modalities [16] are primitive in the context of interacting systems. That is, whether they can be deduced from standard deontic operators — which we show that in the context of contracts between interacting parties, they can. Section 5 gives a brief account of this and other applications.

Contract automata provide an explicit state view of contracts, which aids automated analysis but does not always correlate with the structure of a contract in natural language as much as logic-based formalisms typically do. In Section 6 we present a weak-correctness preserving bidirectional translation into a restricted version of $\mathscr{CL}$, another deontic formalism that allows to predicate over traces of behaviours expressed in a flavour of dynamic logic.

Finally Section 7 and Section 8 close the article with comparisons to related work and conclusions.

2 Contract Automata

In this section we present the concept of contract automata, and their semantics. Since our work is in the context of interacting systems, we start by defining the notions of multi-action automata and their interaction through synchronous composition.

Definition 1 – Multi-Action Automaton
A *multi-action automaton* S is a 4-tuple with components $\langle \Sigma, Q, q0, \rightarrow \rangle$, where Σ is the alphabet of actions, Q is the set of states, $q0 \in Q$ is the initial state and $\rightarrow \subseteq Q \times 2^{\Sigma} \times Q$ is the transition relation. We will write $acts(q)$ to be the set of all action sets on the outgoing transitions from q (defined to be $\{A \mid \exists q' \cdot q \xrightarrow{A} q'\}$).We will write $\overset{w}{\Rightarrow}$ for the transitive closure of $\rightarrow$ (with $w \in (2^{\Sigma})^*$), and for deterministic automata, we will write $q \overset{w}{\Rightarrow}$ to indicate the unique state q' such that $q \overset{w}{\Rightarrow} q'$.

The *synchronous composition* of two automata S_1 and S_2 (with $S_i = \langle \Sigma, Q_i, q0_i, \rightarrow_i \rangle$), and synchronising over alphabet G, written $S_1 \|_G S_2$, is defined to be $\langle \Sigma, Q_1 \times Q_2, (q0_1, q0_2), \rightarrow \rangle$, where $\rightarrow$ is the classical synchronous composition relation (e.g., [1]), as defined below.

$$\frac{q_1 \xrightarrow{A}_1 q_1'}{(q_1,q_2) \xrightarrow{A} (q_1',q_2)}\ A \cap G = \emptyset \qquad\qquad \frac{q_2 \xrightarrow{A}_2 q_2'}{(q_1,q_2) \xrightarrow{A} (q_1,q_2')}\ A \cap G = \emptyset$$

$$\frac{q_1 \xrightarrow{A}_1 q_1',\ q_2 \xrightarrow{B}_2 q_2'}{(q_1,q_2) \xrightarrow{A \cup B} (q_1',q_2')}\ A \cap G = B \cap G \neq \emptyset$$

We can now define contracts to be automata with each state tagged with the clauses which will be in force at that point. The contracts will be able to refer to both presence and absence of an action. Given an alphabet of actions Σ, we write $!\Sigma$ to refer to the alphabet extended with actions preceded with an exclamation mark ! to denote their absence: $!\Sigma \overset{df}{=} \Sigma \cup \{!a \mid a \in \Sigma\}$.

Each clause in a contract automata refers to one of two parties, with the set of parties being Party $= \{1,2\}$. We will use variables p, p_1 and p_2 to range over this type, and write $\overline{p}$ to refer to the party other than p (i.e., $\overline{1} = 2$ and $\overline{2} = 1$).

Contract clauses are either (i) obligation clauses of the form $\mathscr{O}_p(a)$ or $\mathscr{O}_p(!a)$, which say that party p is obliged to perform or not perform action a respectively; or (ii) permission clauses which can be either of the form of $\mathscr{P}_p(a)$ or $\mathscr{P}_p(!a)$ (party p is permitted to perform, or not perform action a respectively). We will use variables x, y and z to stand for either presence or absence of an action. E.g. $\mathscr{P}_p(x)$ would match both $\mathscr{P}_p(a)$ and $\mathscr{P}_p(!a)$. If x is already an inverted action $x =!a$, then expression $!x$ is interpreted to be a.

Definition 2 – Contract Automaton
A *contract clause* over alphabet Σ is structured as follows (where action $x \in !\Sigma$, party $p \in \{1,2\}$):

$$\text{Clause} ::= \mathscr{O}_p(x) \mid \mathscr{P}_p(x)$$

A *contract automaton* S is a total and deterministic multi-action automaton with $S = \langle \Sigma, Q, q0, \rightarrow \rangle$, together with a total function *contract* $\in Q \rightarrow 2^{\text{Clause}}$ assigning a set of clauses to each state.

Example 1
Consider, for instance, the contract between a music service provider p and a user u. The contract states that the service provider is obliged to make a one-time offer to the user, which, if accepted, permits her to listen to a piece of music up to three times.

If we use *accept* for the action with which the one-time offer is accepted, and *listen* for the listening of the piece of music, we can express the contract as a contract automaton with 5 states, the first one σ in which the user has not yet accepted the offer, and the remaining 4, named σ_0 to σ_3, representing the cases when the user still has permission to listen to the music from 0 to 3 times respectively. The automaton is depicted in Fig. 1, in which each state is tagged with its name and the set of contract clauses that are active in that state.

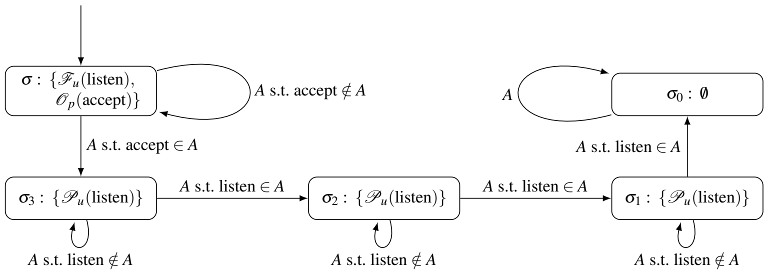


Fig. 1 Example of a music provider contract.

Note that, for the sake of conciseness in the example, rather than draw every transition, we are (visually) representing multiple transitions with a simplified notation where we write e.g. A s.t. $\mathit{listen} \in A$ as a shorthand for drawing all the transitions with a set of actions that include *listen*.

Given a two-party system (S_1, S_2), and a contract automaton $\mathscr{A}$, we can now put them together as a regulated two-party system.

Definition 3 – Regulated Two-Party System
A *regulated two-party system* synchronising over the set of actions G is a tuple $R = \langle S_1, S_2 \rangle_G^{\mathscr{A}}$, where $S_i = (\Sigma_i, Q_i, q0_i, \rightarrow_i)$ is a multi-action automaton specifying the behaviour of party i, and $\mathscr{A}$ is a contract automaton over alphabet $\Sigma_1 \cup \Sigma_2$.

The *behaviour* of a regulated two-party system R, written $[\![R]\!]$, is defined to be the automaton $(S_1 \|_G S_2) \|_\Sigma \mathscr{A}$. To make states in such systems more readable, we will write $((q_1, q_2), q_{\mathscr{A}})$ as $(q_1, q_2)_{q_{\mathscr{A}}}$.

A regulated two-party system is said to be *well-formed* if $S_1 \|_G S_2$ never deadlocks: $\forall (q_1, q_2) \cdot \mathit{acts}((q_1, q_2)) \neq \emptyset$.

In the rest of the paper we will assume that all systems are well-formed. One way of guaranteeing this may be by having all system states provide a transition with the empty action set.

Also note that the totality of the contract automaton guarantees that the system behaviour is not constrained, but simply acts to tag the states with the relevant contracts at each point in time.

We can now define whether or not either party is violating the contract when a particular state is reached or a transition is taken.

Definition 4 – Viable Action Sets
Action set A is said to be *viable with respect to a set O of obliged actions and a set F or forbidden actions*, written $\mathit{viable}(A, O, F)$, if (i) all the obliged actions are included in A but; (ii) no action which the party is forbidden (obliged not) to perform is included in A:

$$\mathit{viable}(A, O, F) \stackrel{df}{=} O \subseteq A \wedge F \cap A = \emptyset.$$

Functions $O_p(q_\mathscr{A})$ and $F_p(q_\mathscr{A})$ give the set of actions respectively obliged to be performed and forbidden (obliged not) to be performed by party p. They are defined in terms of the contract clauses in the state.

$$\begin{aligned} O_p(q_\mathscr{A}) &\overset{df}{=} \{a \mid \mathscr{O}_p(a) \in contract(q_\mathscr{A})\} \\ F_p(q_\mathscr{A}) &\overset{df}{=} \{a \mid \mathscr{O}_p(!a) \in contract(q_\mathscr{A})\}. \end{aligned}$$

We thus can define the viable actions for a party p in a state $q_\mathscr{A}$: action set A is said to be viable for party p in a contract automaton state $q_\mathscr{A}$, written $viable_p(q_\mathscr{A}, A)$, if (i) all her obliged actions are included in A but; (ii) no actions which the party is obliged not to perform are included in A:

$$viable_p(q_\mathscr{A}, A) \overset{df}{=} viable(A, O_p(q_\mathscr{A}), F_p(q_\mathscr{A})).$$

Since we would like to be able to place blame in the case of a violation, we parametrise contract satisfaction by party. It is also worth noting that while obligation to perform an action, for instance, is violated in a transition which does not include the action, permission is violated by a state in which the opportunity to perform the permitted action is not present. The satisfaction operator $\vdash_p$ will thus be overloaded to be applicable to both states and transitions. The operator $X \vdash_p C$ will denote that reaching state X, or traversing transition X, does not constitute a violation of clause C for party p. X ranges over states and transitions in the composed system.

We start by defining the satisfaction of the different deontic operators. Note that in these definitions we use q_{p_1} to refer to the action of party p_1, i.e., q_{p_1} can refer to either q_1 or q_2.

Definition 5 – Permission
If party p is *permitted* to perform shared action a, then the other party $\overline{p}$ must provide p with at least one viable outgoing transition which contains a but does not include any forbidden actions. Permission to perform unsychronised actions cannot be violated.

Satisfaction of a single permission is defined as:

$$(q_1, q_2)_{q_\mathscr{A}} \vdash_{p_1} \mathscr{P}_{p_2}(a) \overset{df}{=} \begin{cases} true & \text{if } p_1 = p_2 \vee a \notin G \\ \\ \exists A \in acts(q_{p_1}),\ A' \subseteq (\Sigma - G) \cdot & \\ \qquad a \in A \ \wedge\ viable_{p_2}(q_\mathscr{A}, A \cup A') & \text{if } p_1 \neq p_2 \wedge a \in G \end{cases}$$

Similarly, if party p is permitted to not perform action a, then the other party $\overline{p}$ must provide p with at least one viable outgoing transition which does not include a nor any forbidden action. Permission to perform local actions can never be violated. In the case of a single permission, this can be expressed as follows:

$$(q_1, q_2)_{q_\mathscr{A}} \vdash_{p_1} \mathscr{P}_{p_2}(!a) \overset{df}{=} \begin{cases} true & \text{if } p_1 = p_2 \vee a \notin G \\ \\ \exists A \in acts(q_{p_1}),\ A' \subseteq (\Sigma - G) \cdot & \\ \qquad a \notin A \ \wedge\ viable_{p_2}(q_\mathscr{A}, A \cup A') & \text{if } p_1 \neq p_2 \wedge a \in G \end{cases}$$

While actual obligation violations occur when an action is not performed, violations of a permission occur when no appropriate action is possible. In this paper we give a semantics that tags as a violation a state in which one party is permitted to perform an action, while the other provides no way of actually doing so.

Definition 6 – Obligation
In an interacting system, *obligations* put constraints on both parties. Given that party p is obliged to perform action a in a state means that (i) party p must include the action in any outgoing transition in the composed system in which it participates; and (ii) the other party $\overline{p}$ must provide a viable synchronisation action set which, together with other asynchronous actions performed by p, allows p to perform *all* its obligations, positive and negative. Obligation to not perform action a ($\mathscr{O}_p(!a)$) can be similarly expressed. We combine all positive and negative obligations in the following definitions.

Note that we treat differently the obligations imposed on a party and the ones that arise in order to let the other fulfil her own obligations. In the latter case we tag the violation at the state level (i.e., we tag the non-existence of any valid outgoing transition). In the former case, we want to flag individual transitions as satisfying or not the obligations. Thus, we overload the satisfaction operator to also consider transitions.

$$(q_1,q_2)_{q_\mathscr{A}} \vdash_{p_1} \mathscr{O}_{p_2}(x) \stackrel{df}{=} \begin{cases} true & \text{if } p_1 = p_2 \\ \exists A \in acts(q_{p_1}),\ A' \subseteq (\Sigma - G) \cdot \\ \qquad viable_{p_2}(q_\mathscr{A}, A \cup A') & \text{if } p_1 \neq p_2 \end{cases}$$

$$(q_1,q_2)_{q_\mathscr{A}} \xrightarrow{A} (q_1',q_2')_{q_\mathscr{A}'} \vdash_{p_1} \mathscr{O}_{p_2}(x) \stackrel{df}{=} \begin{cases} viable_{p_1}(q_\mathscr{A}, A) & \text{if } p_1 = p_2 \\ true & \text{if } p_1 \neq p_2 \end{cases}$$

Example 2
Consider a contract between a passenger p and the airline representative a represented as a contract automaton. When the passenger arrives at the flight boarding-check, the contract will be in a state with three clauses:

- The passenger is permitted to board one piece of hand luggage (encoded as action l): $\mathscr{P}_p(l)$.
- The passenger is obliged to show her id card (encoded as action s): $\mathscr{O}_p(s)$.
- The passenger is also obliged to show her boarding pass (action b):$\mathscr{O}_p(b)$.

For the sake of this example, we will assume that actions l and b are shared, but s is not. Now, if the airline system is in a state with at least an outgoing transition labelled with action set $\{l,b\}$, the permission to board with one piece of hand luggage is satisfied, even though the current state of the passenger's automaton has no matching transition. Note that action s need not to be present since it is a local action. I.e., no synchronisation takes place on that action.

However, to fulfil the obligation to show the id card and the boarding pass, the passenger's current state must have an outgoing transition including the set of actions $\{s,b\}$. Note that even though s is not shared and thus still not needed for the synchronisation, it must nevertheless be present to comply with the obligation.

We can now complete the definition of the basic deontic modalities.

Definition 7 – Prohibition and Equivalences for Absence of Actions

- Party p not being permitted to perform an action is equivalent to p being obliged not to perform the action:
$$!\mathscr{P}_p(a) \overset{df}{=} \mathscr{O}_p(!a) \qquad !\mathscr{P}_p(!a) \overset{df}{=} \mathscr{O}_p(a)$$
- Party p not being obliged to perform an action is equivalent to p being permitted not to perform the action:
$$!\mathscr{O}_p(a) \overset{df}{=} \mathscr{P}_p(!a) \qquad !\mathscr{O}_p(!a) \overset{df}{=} \mathscr{P}_p(a)$$
- Prohibition contract clauses $\mathscr{F}_p(a)$ and $\mathscr{F}_p(!a)$, prohibiting party p from performing and not performing a respectively, can be expressed in terms of permission:
$$\mathscr{F}_p(a) \overset{df}{=} !\mathscr{P}_p(a) \qquad \mathscr{F}_p(!a) \overset{df}{=} !\mathscr{P}_p(!a)$$
- Prohibition to perform an action is equivalent to obligation not to perform the action:
$$\mathscr{F}_p(x) = \mathscr{O}_p(!x)$$

It should be noted that we are equating lack of permission to do a to an obligation to perform an action set which does not include a. Although at first glance, this may appear to disallow a party from doing nothing as a way of satisfying lack of permission, this is not the case, since the empty set of actions does satisfy the constraint.

It is interesting to note that in a two party system there are alternative notions of opposites to permission and obligation. Consider party p not being permitted to perform action a. Apart from the interpretation we gave, in which the norm places the onus on party p not to perform a, an alternative view is to push the responsibility to $\overline{p}$ and interpret it as: *party $\overline{p}$ may not provide a viable action set which includes a.* This is distinct from $!\mathscr{P}_p(a)$ (and indeed from the other modalities we have).

Similarly, consider party p not being obliged to perform action a. The interpretation we adopted permits party p to not perform a, but once again, alternative definitions may be adopted. One possibility is to push the responsibility to $\overline{p}$ and interpret it as: *party $\overline{p}$ must provide a viable transition which does not include a.*

These duals, in which the outer negation of a norm also corresponds to shifting of responsibility, give an interesting alternative view of norm opposites in a two-party system. Another interesting alternative would be to interpret these negations as modalities whose only effect is the cancelling of existing clauses. We will not explore these alternative modalities any further in this paper, since the modalities we adopt provide a clean notion of conflicts, as discussed in Section 3. Should they be needed for a particular application, any of the above mentioned interpretations could be included as alternative type of negation. One of the advantages of clear formal semantics is that there is no need to dispute the meaning of a given term, since

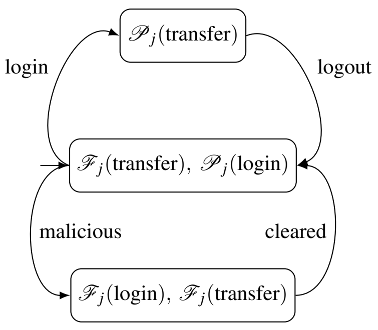

$\mathscr{P}_j$(transfer)
login
logout
$\mathscr{F}_j$(transfer), $\mathscr{P}_j$(login)
malicious
cleared
$\mathscr{F}_j$(login), $\mathscr{F}_j$(transfer)

Fig. 2 Internet banking contracts.

different ones can be defined and the appropriate one be picked to convey specific meanings.

Definition 8 – Contract Satisfaction
Contract satisfaction for a set of clauses $\mathscr{C}$ is defined as:

$$(q_1,q_2)_{q_\mathscr{A}} \vdash_p \mathscr{C} \overset{df}{=} \forall C \in \mathscr{C} \cdot (q_1,q_2)_{q_\mathscr{A}} \vdash_p C$$

General contract satisfaction for contract automata $\mathscr{A}$ is defined as follows. Note that in the case of a transition the set of clauses that the transition should satisfy is taken from the source state.

$$sat_p^\mathscr{A}((q_1,q_2)_{q_\mathscr{A}}) \overset{df}{=} (q_1,q_2)_{q_\mathscr{A}} \vdash_p contract(q_\mathscr{A})$$
$$sat_p^\mathscr{A}((q_1,q_2)_{q_\mathscr{A}} \xrightarrow{A} (q_1',q_2')_{q_\mathscr{A}'}) \overset{df}{=} (q_1,q_2)_{q_\mathscr{A}} \xrightarrow{A} (q_1',q_2')_{q_\mathscr{A}'} \vdash_p contract(q_\mathscr{A})$$

When $\mathscr{A}$ is clear for the context we will write $sat_p(X)$ instead of $sat_p^\mathscr{A}(X)$.

Example 3
If p is permitted to withdraw money from the bank, permitted not to deposit, obliged to pay the fee, and obliged not to steal ($\mathscr{P}_p(w)$, $\mathscr{P}_p(!d)$, $\mathscr{O}_p(f)$, $\mathscr{O}_p(!s)$), $\overline{p}$ should provide at least one transition that contains both a w and an f and contains neither a d nor an s.

Example 4
Consider John signing a contract with his bank. The contract says that (i) whenever he is logged into his Internet banking account, he is to be permitted to make money transfers; and (ii) if a malicious attempt to log in to his account is identified, logging in and making transfers will be prohibited until the situation is cleared. The two statements can be expressed in the contract automaton shown in Fig. 2.

2.1 Compliance

We extend the operational semantics of contract automata by tagging transitions with the compliance status of each party: $\xrightarrow[(\psi,\psi')]{A}$ where ψ and ψ' are one of $\checkmark$, $\times$ and R indicating satisfaction, violation without the possibility of reparation and violation but going to a reparation state respectively[2].

Definition 9 A state of a regulated system and action set pair (q,A) is said to be violating for party p, written $viol_p(q,A)$ if the state or any of its outgoing transitions does not satisfy the active contract clauses:

$$viol_p(q,A) \equiv \neg(sat_p(q) \wedge \forall q' \cdot sat_p(q \xrightarrow{A} q'))$$

We will write $viol(q,A)$ to denote when either party has violated the contract: $viol_p(q,A) \vee viol_{\overline{p}}(q,A)$. By defining $\delta^{\psi}_{\psi'}(p,q,A)$ to be ψ' if party p violates the contract when the system is in state q and action set A is performed ($viol_p(q,A)$) and ψ otherwise, we can define the extended operation semantics: for a transition $q \xrightarrow{A} q'$, we can deduce

$$q \xrightarrow[(\delta^{\checkmark}_{\times}(1,q,A),\,\delta^{\checkmark}_{\times}(2,q,A))]{A} q'$$

Definition 10 – Breach Incapable
A party p is said to be *incapable of breaching a contract* in a regulated two-party system $R = \langle S_1, S_2 \rangle^{\mathscr{A}}_{G}$, written $breachIncapable_p(R)$, if p cannot be in violation in any of the reachable states and transitions of R. We write $breachIncapable(R)$ if it holds for both parties.

Note that a party being breach-incapable is stronger than just being compliant for one specific run — $breachIncapable_p(R)$ means that there is no possible trace of R, in which p breaches the contract.

2.2 Contract Strength

We can now define strictness relationships over contracts.

Definition 11 – Contract Strictness
A contract automaton $\mathscr{A}'$ is said to be *stricter than* contract automaton $\mathscr{A}$ for party p (or $\mathscr{A}$ said to be *more lenient* than $\mathscr{A}'$ for party p), written $\mathscr{A} \sqsubseteq_p \mathscr{A}'$, if for any systems S_1 and S_2, p being incapable of breaching $\mathscr{A}'$ implies that p is incapable of breaching $\mathscr{A}$. We say that two contract automata $\mathscr{A}$ and $\mathscr{A}'$ are equivalent for party p, written $\mathscr{A} =_p \mathscr{A}'$, if $\mathscr{A} \sqsubseteq_p \mathscr{A}'$ and $\mathscr{A}' \sqsubseteq_p \mathscr{A}$. Similarly, we define strictly more lenient relation $\sqsubset_p$ as $\sqsubseteq_p \setminus =_p$. We define global contract strictness $\mathscr{A} \sqsubseteq \mathscr{A}'$ to hold if $\mathscr{A} \sqsubseteq_p \mathscr{A}'$ holds for all parties p, and similarly global contract equivalence $\mathscr{A} = \mathscr{A}'$.

[2] Although we have no notion of reparation yet, we introduce this possible state to be used in Section 4.

Proposition 1 *The relation over contracts $\sqsubseteq$ is a partial order.*

Structurally isomorphic contract automata provide a useful proof technique that we would use on the rest of the article many times.

Proposition 2 *Given two structurally isomorphic contract automata $\mathscr{A}$ and $\mathscr{A}'$, $\mathscr{A} \sqsubseteq \mathscr{A}'$ if and only if, for any state or transition X, $\text{sat}_p^{\mathscr{A}'}(X) \implies \text{sat}_p^{\mathscr{A}}(X)$.*

This proof principle can be proved to hold by showing that (i) the automata obtained from the synchronous composition with the two contracts are structurally identical; and (ii) using the definition of breach incapability. The principle can be used to prove that contract automata are monotonic.

Proposition 3 *Given two structurally isomorphic contract automata $\mathscr{A}$ and $\mathscr{A}'$, with contract clause functions* contract *and* contract$'$ *respectively, satisfying that $\forall q \cdot$* $\text{contract}(q) \subseteq \text{contract}'(q)$*, it follows that $\mathscr{A} \sqsubseteq \mathscr{A}'$.*

The proof follows from the observation that $sat_p(X)$ is essentially a conjunction of a proposition for each contract clause in the state. Hence, $sat_p^{\mathscr{A}'}(X)$ (which has a larger set of clauses) implies $sat_p^{\mathscr{A}}(X)$. Applying Proposition 2 to this observation completes the proof.

Although contracts are expressed as automata, we would like to be able to compare individual clauses. To do this we will need to relate contract automata which are equivalent except for a particular clause replaced by another.

Definition 12 – Clause Strictness
Given two contract clauses C and C', the relation over contract automata $[C \rightarrow C'] \subseteq \mathscr{CA} \times \mathscr{CA}$ relates two contract automata $\mathscr{A}$ and $\mathscr{A}'$ if $\mathscr{A}$ is equivalent to $\mathscr{A}'$ except possibly for a number of instances of clause C replaced by C'.

We extend the notion of strictness to contract clauses. We say that clause C' is stricter than clause C for party p, written $C \sqsubseteq_p C'$, if for any contract automata $\mathscr{A}$ and $\mathscr{A}'$ such that $(\mathscr{A}, \mathscr{A}') \in [C \rightarrow C']$, it follows that $\mathscr{A} \sqsubseteq_p \mathscr{A}'$. We similarly extend the notion of strictness for all parties $\sqsubseteq$.

The following proposition allows us to use the proof principle given in Proposition 2 for reasoning about clause strictness:

Proposition 4 *Given clauses C and C', any two contract automata related by $[C \rightarrow C']$ are structurally isomorphic.*

2.3 Strictness Theorems

The strictness relationship between clauses allows us to state the following theorems.

Theorem 1 *Obligation is stricter than permission: (i) $\mathscr{P}_p(a) \sqsubseteq \mathscr{O}_p(a)$; and (ii) $\mathscr{P}_p(!a) \sqsubseteq \mathscr{O}_p(!a)$.*

Proof *We present the proof of (i) — the proof of (ii) is very similar. We need to prove that for any contract automata $\mathscr{A}$ and $\mathscr{A}'$ such that $(\mathscr{A},\mathscr{A}') \in [\mathscr{P}_p(a) \to \mathscr{O}_p(a)]$, then it follows that $\mathscr{A} \sqsubseteq \mathscr{A}'$. Using Proposition 4, we know that $\mathscr{A}$ and $\mathscr{A}'$ are structurally isomorphic, allowing us to apply the proof principle of Proposition 2.*

We thus have to show that $\mathrm{sat}_p^{\mathscr{A}'}(X)$ implies $\mathrm{sat}_p^{\mathscr{A}}(X)$. Since the permission in $\mathscr{A}$ which is replaced by an obligation, never yields violations for party p nor for any party on transitions, it suffices to prove that this implication holds on states for party $\overline{p}$.

The satisfaction function for $\overline{p}$'s obligations in states is:

$$\exists A \in \mathrm{acts}(q_{\overline{p}}),\ A' \subseteq (\Sigma - G)\ \cdot\ \mathrm{viable}_p(q_{\mathscr{A}'}, A \cup A')$$

If $a \in G$, and since $a \in O_p(q_{\mathscr{A}'})$, we can conclude that $a \in A$:

$$a \in G \implies \exists A \in \mathrm{acts}(q_p),\ A' \subseteq (\Sigma - G)\ \cdot\ a \in A \wedge \mathrm{viable}_{\overline{p}}(q_{\mathscr{A}'}, A \cup A')$$

Furthermore, since $q_{\mathscr{A}}$ has less obligations than $q_{\mathscr{A}'}$, viability for $q_{\mathscr{A}'}$ implies viability for $q_{\mathscr{A}}$:

$$a \in G \implies \exists A \in \mathrm{acts}(q_p),\ A' \subseteq (\Sigma - G)\ \cdot\ a \in A \wedge \mathrm{viable}_{\overline{p}}(q_{\mathscr{A}}, A \cup A')$$

Hence, the satisfaction function for the permission $\mathscr{P}_p(a)$ holds and thus, by Proposition 2 we can conclude that $\mathscr{A} \sqsubseteq \mathscr{A}'$.

Theorem 2 *For synchronised actions, obligation for one party is stricter than permission for the other: (i) $\mathscr{P}_p(a) \sqsubseteq \mathscr{O}_{\overline{p}}(a)$; and (ii) $\mathscr{P}_p(!a) \sqsubseteq \mathscr{O}_{\overline{p}}(!a)$.*

Proof *As in the previous theorem, we observe that $\mathscr{P}_p(a)$ can only yield violations for states and for party $\overline{p}$.*

Observe that the obligation $\mathscr{O}_{\overline{p}}(a)$ in a state $q_{\mathscr{A}'}$ guarantees that all outgoing transitions from the state $(q_1,q_2)_{q_{\mathscr{A}'}} \xrightarrow{A} (q_1',q_2')_{q'_{\mathscr{A}'}}$ satisfy $\mathrm{viable}_{\overline{p}}(q_{\mathscr{A}'}, A)$.

Since we assume that the system does not deadlock, there is at least one such transition which party p participates in. Furthermore, if $a \in G$, it must also appear in the actions on the transition:

$$a \in G \implies \exists A \in \mathrm{acts}(q_p),\ A' \subseteq (\Sigma - G)\ \cdot\ a \in A \wedge \mathrm{viable}_{\overline{p}}(q_{\mathscr{A}'}, A \cup A').$$

This guarantees that $(q_1,q_2)_{q_{\mathscr{A}}} \vdash_p \mathscr{P}_p(a)$, and allows us to complete the proof using Proposition 2.

It is interesting to note that if we had a weaker semantics which simply identifies a violation without identifying the guilty party, we would be able to show equivalence between $\mathscr{O}_p(a)$ and $\mathscr{O}_{\overline{p}}(a)$, since a lack of *a* on a transition would cause a violation of both obligations. However, since our semantics characterise violations for the parties separately, and the partial order $\sqsubseteq_p$ is parametrised by the party, we can show that the two obligations are in fact different [29].

2.4 Mutually Exclusive Actions

Although we adopt a multi-action approach, modelling real-world scenarios means that certain actions should never occur concurrently. For instance, one would expect that the automata never perform the action *openDoor* and *closeDoor* on the same transition. This allows us to identify strictness laws which hold only for mutually exclusive actions.

Definition 13 – Mutually Exclusive Actions
Given a multi-action automaton $\langle \Sigma,\ Q,\ q0,\ \rightarrow \rangle$, two actions a and b ($\{a,b\} \subseteq \Sigma$) are said to be *mutually exclusive*, written $a \bowtie b$, if they can never appear in the same set of actions on transitions. Thus, for any automaton, it should be the case that:

$$\forall (q,A,q') \in \rightarrow \ \cdot\ a \in A \implies b \notin A.$$

In the rest of the article we will assume that mutually exclusive actions never appear in the synchronisation sets. This is done to simplify the presentation, since otherwise we would need a more complex rule for synchronous composition (not allowing synchronisation when the asynchronous actions of party are in conflict with those of the other) and a modified definition for the satisfaction of obligations (the other party must provide a viable action set which does not include any actions which may conflict with the obligations of the party to whom the obligation applies). Removing this restriction, however, does not affect the results we present.

The following theorem shows how mutually exclusive actions and action absence are related together under both obligation and permission:

Theorem 3 *If $a \bowtie b$ then (i) $\mathscr{O}_p(!a) \sqsubseteq \mathscr{O}_p(b)$; and (ii) $\mathscr{P}_p(!a) \sqsubseteq \mathscr{P}_p(b)$.*

Proof *To show (i), we need to prove that for any contract automata $\mathscr{A}$ and $\mathscr{A}'$ such that $(\mathscr{A},\ \mathscr{A}') \in [\mathscr{O}_p(!a) \rightarrow \mathscr{O}_p(b)]$, then it follows that $\mathscr{A} \sqsubseteq \mathscr{A}'$. As in the previous proofs, we can use Proposition 4 to conclude that $\mathscr{A}$ and $\mathscr{A}'$ are structurally isomorphic, allowing us to apply the proof principle of Proposition 2.*

We thus have to show that $\mathrm{sat}_p^{\mathscr{A}'}(X)$ implies $\mathrm{sat}_p^{\mathscr{A}}(X)$. We look at transitions and states separately:

Transitions: The satisfaction function for the combined obligations for a transition $(q_1,q_2)_{q_{\mathscr{A}'}} \xrightarrow{A} (q_1',q_2')_{q'_{\mathscr{A}'}}$ in automaton $\mathscr{A}'$ is that $\mathrm{viable}_p(q_{\mathscr{A}'},A)$. By definition of viability and the obligation $\mathscr{O}_p(b)$ in $q_{\mathscr{A}'}$, we can thus conclude that $b \in A$. However, since $a \bowtie b$, this means that $a \notin A$, from which we can conclude that $\mathrm{viable}_p(q_{\mathscr{A}},A)$ and hence that the satisfaction function also holds for transitions in $\mathscr{A}$.

States: The satisfaction function applied to states acts on the other party $\overline{p}$. For state $(q_1,q_2)_{q_{\mathscr{A}'}}$, it is defined to be $\exists A \in \mathrm{acts}(q_{\overline{p}}),\ A' \subseteq (\Sigma - G)\ \cdot\ \mathrm{viable}_p(q_{\mathscr{A}'},A \cup A')$. Since $a \in G$, the proof is identical to the previous case.

Hence, the satisfaction function for $\mathscr{O}_p(a)$ holds and thus, by Proposition 2 we can conclude that $\mathscr{A} \sqsubseteq \mathscr{A}'$ and hence (i) holds.

The proof of (ii) follows similarly.

A similar result can be shown, but referring to the other party in the contract:

Theorem 4 *If $a \bowtie b$ then $\mathscr{O}_{\overline{p}}(!b) \sqsubseteq \mathscr{O}_p(a)$.*

Proof *We take an approach identical to the previous theorems and prove that for any contract automata $\mathscr{A}$ and $\mathscr{A}'$ such that $(\mathscr{A},\ \mathscr{A}') \in [\mathscr{O}_{\overline{p}}(!b) \rightarrow \mathscr{O}_p(a)]$, then it follows that $\mathscr{A} \sqsubseteq \mathscr{A}'$. Propositions 4 and 2 can then be used to complete the proof. As before, we consider the satisfaction relation on states and transitions separately:*

Transitions: The satisfaction function for the combined obligations for a transition $(q_1,q_2)_{q_{\mathscr{A}'}} \xrightarrow{A} (q'_1,q'_2)_{q'_{\mathscr{A}'}}$ *in automaton* $\mathscr{A}'$ *is that* $\text{viable}_p(q_{\mathscr{A}'},A)$. *By definition of viability and the obligation* $\mathscr{O}_p(a)$ *in* $q_{\mathscr{A}'}$*, we can thus conclude that* $a \in A$. *However, since* $a \bowtie b$*, this means that* $b \notin A$*. The same transition must be viable for* $\overline{p}$ *in* $\mathscr{A}'$*, so* $\text{viable}_{\overline{p}}(q_{\mathscr{A}'},A)$ *holds. The absence of b also allows us to conclude that* $\text{viable}_{\overline{p}}(q_{\mathscr{A}},A)$*, which is the satisfaction function for* $\mathscr{O}_{\overline{p}}(!b)$ *over transitions in* $\mathscr{A}$*.*

States: For state $(q_1,q_2)_{q_{\mathscr{A}'}}$*, since we assume deadlock freedom and satisfaction of the obligation to perform a, we know of the existence of an outgoing transition with action a such that* $a \in A$*. Since party p is participating in this transition, and* $a \in G$*, we can conclude that there is a transition viable for* $\overline{p}$*, leaving from* q_p *and with an action set which includes a and hence not b. Propositions 4 and 2 can then be conclude that* $\exists A \in \text{acts}(q_p),\ A' \subseteq (\Sigma - G)\ \cdot\ \text{viable}_{\overline{p}}(q_{\mathscr{A}},A \cup A')$.

Although one may be tempted to induce that a similar result can be shown for permission (analogous to part (ii) of Theorem 3) — $\mathscr{P}_{\overline{p}}(!b) \sqsubseteq \mathscr{P}_p(a)$ does not always hold. As a simple example of a system satisfying $\mathscr{P}_p(a)$ but not $\mathscr{P}_{\overline{p}}(!b)$, consider party p being able to perform just one transition with action set $\{b\}$, and party $\overline{p}$ being able to perform one of two transitions: one with action set $\{a\}$, the other with action set $\{b\}$. Party p is permitted to perform a but party $\overline{p}$ is not permitted to perform $!b$.

3 Conflicts and Tool Support

Contract clauses are not always compatible with one another. Many definitions of conflict are possible — in this article we deal only with one particular class of conflicts which focusses on conflicting norms and mutually exclusive actions, but some interesting issues arise from party interdependence.

As expected, the obligation on a party to perform an action a and the obligation on the same party not to perform the same action can never be satisfied together. Another interesting example is that of $\mathscr{P}_p(!a)$ and $\mathscr{O}_p(a)$. Although one is tempted to intuitively think that having the possibility of doing something other than a does not conflict with the obligation of doing a, multi-action semantics in contracts are different: to satisfy the permission party $\overline{p}$ must provide a-free action sets which allow p to satisfy her obligations, but that requires that they contain a.

3.1 Definitions and Results

In this section we axiomatise the notion of conflicts in interacting two-party systems and investigate some consequences.

Definition 14 *Contract conflicts* is a relation between contract clauses $\maltese \in \text{Clause} \leftrightarrow \text{Clause}$ and is defined to be the least relation satisfying the following axioms:
Axiom 1: Opposite permissions conflict: $\vdash \mathscr{P}_p(x) \maltese\ !\mathscr{P}_p(x)$.
Axiom 2: Obligation to perform mutually exclusive actions is a conflict: $a \bowtie b \vdash \mathscr{O}_p(a) \maltese \mathscr{O}_p(b)$.

Axiom 3: Conflicts are closed under symmetry: $C \maltese C' \vdash C' \maltese C$.
Axiom 4: Conflicts are closed under increased strictness: $C \maltese C' \wedge C' \sqsubseteq C'' \vdash C \maltese C''$.

Although conflicts are only identified for opposing permissions in the axioms, they also arise in opposing obligations, and can be shown to follow from the axioms.

Proposition 5 *Opposite obligations conflict with each other: $\mathscr{O}_p(x) \maltese\, !\mathscr{O}_p(x)$.*

Proof The proof uses the definition of negated permission and obligation to derive the desired result:

$$\begin{array}{ll} & \textit{definition of conflict on opposing permissions} \\ \Longrightarrow & \mathscr{P}_p(x) \maltese\, !\mathscr{P}_p(x) \\ \Longrightarrow & \textit{for some } y,\ x = !y \\ & \mathscr{P}_p(!y) \maltese\, !\mathscr{P}_p(!y) \\ \Longrightarrow & \textit{definition of } !\mathscr{P}_p(y) \textit{ and } !\mathscr{O}_p(y) \\ & !\mathscr{O}_p(y) \maltese \mathscr{O}_p(y) \\ \Longrightarrow & \textit{symmetry of } \maltese \\ & \mathscr{O}_p(y) \maltese\, !\mathscr{O}_p(y) \end{array}$$

Various other conflicts can be derived from the axioms. The following show conflicts between permissions and obligations and arising from enforcing norms over both the presence and absence of an action.

Proposition 6 *Obligation to perform an action conflicts with both permission and obligation to not perform it: (i) $\mathscr{O}_p(x) \maltese \mathscr{P}_p(!x)$; and (ii) $\mathscr{O}_p(x) \maltese \mathscr{O}_p(!x)$. Obligation to perform an action also conflicts with lack of permission to perform the action: (iii) $\mathscr{O}_p(x) \maltese\, !\mathscr{P}_p(x)$.*

Proof By Proposition 5, we know that $\mathscr{O}_p(x) \maltese\, !\mathscr{O}_p(x)$, which, by definition of $!\mathscr{O}_p(x)$ is equivalent to $\mathscr{O}_p(x) \maltese \mathscr{P}_p(!x)$, hence completing the proof for (i).
By result (i) and $\mathscr{P}_p(!x) \sqsubseteq \mathscr{O}_p(!x)$, we can use the strictness axiom of conflicts to conclude that (ii) holds: $\mathscr{O}_p(x) \maltese \mathscr{O}_p(!x)$.
Result (iii) follows directly from the definition of $!\mathscr{P}_p(x)$ and result (ii).

Finally, we show how making two conflicting contracts stricter does not get rid of the conflict:

Proposition 7 *Given two conflicting clauses $C_1 \maltese C_2$, making the two clauses stricter does not resolve the conflict: if $C_1 \sqsubseteq C_1'$ and $C_2 \sqsubseteq C_2'$, then $C_1' \maltese C_2'$.*

Proof The proof follows by applying axiom of closure under increased strictness twice and the axiom of symmetry.

Example: For example, consider John signing a contract with his bank. The contract says that (i) whenever he is logged into his Internet banking account, he is to be permitted to make money transfers; and (ii) if a malicious attempt to log in to his account is identified, logging in and making transfers will be prohibited until the situation is cleared. The two statements can be expressed in the two contract automata shown in Fig. 3. Combining the two statements, however results in an automaton where initially, after performing action set $\{\text{login}, \text{malicious}\}$, one ends up in a state with both $\mathscr{P}_p(\text{transfer})$ and $\mathscr{F}_p(\text{transfer})$, which are in conflict by Proposition 6.

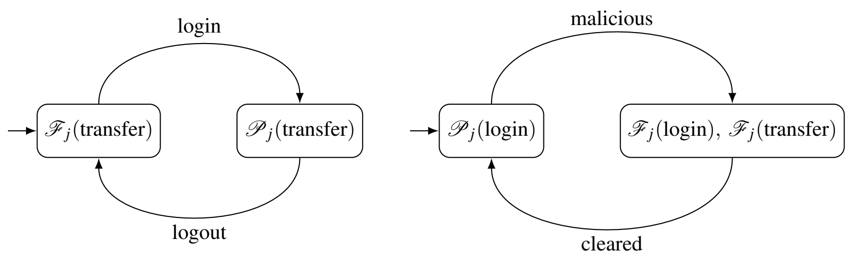

login
$\mathscr{F}_j$(transfer)
$\mathscr{P}_j$(transfer)
logout
malicious
$\mathscr{P}_j$(login)
$\mathscr{F}_j$(login), $\mathscr{F}_j$(transfer)
cleared

Fig. 3 Internet banking contracts.

t

```
Looking for conflicts in the set of clauses:
   F_1(a)
   O_1(a)

Normalised to:
   O_1(!a)
   O_1(a)

Conflict discovered between O_1(a) and O_1(!a)
PROOF:
  O_1(a) # P_1(!a)
    (by Axiom 1 of conflicts (P_p(a) # !P_p(a)) and
        Axiom 3 of conflicts (symmetry of #))
  P_1(!a) <= O_1(!a) (proved below)
     P_1(!a)
     <= { Theorem: O_p(x) => P_p(x) }
     O_1(!a)
  O_1(a) # O_1(!a)
    (by Axiom 4 - closure under increased strictness)
```

Fig. 4 Tool support for automated conflict discovery.

3.2 Automated Conflict Discovery

Conflict discovery has been implemented in a tool written in Haskell, enabling the automated analysis of sets of clauses. Fig. 4 shows the output from the command-line tool when attempting to find conflicts in the set of clauses $\{\mathscr{O}_p(a),\ \mathscr{F}_p(a)\}$. Internally, the tool attempts to discover a proof of a conflict between a pair of clauses in the given set. At an abstract level, the algorithm works as follows:

1. The clauses are normalised by replacing prohibitions with obligations not to perform an action.
2. A search for clause conflicts identifiable through the use of Axioms 1–3 is made.
3. The set of clauses is expanded using strictness closure by applying the results from Section 2.3 (trying to identify conflicts through the use of Axiom 4).
4. Equivalent clauses are removed from the set, and if any new clauses have been identified, the search is repeated from step 2.

A more concrete version of the algorithm is given as Algorithm 1.

Example 5
Let us assume we give clauses $\mathscr{F}_1(a)$ and $\mathscr{O}_1(a)$ as input to the tool (in variable `C`). The normalisation process (lines 1–6) will result in `C_new` containing $\{!\mathscr{P}_1(a),\ \mathscr{O}_1(a)\}$. The analysis loop (lines 10–27) will be triggered since we have new clauses to consider, but no conflicts will be discovered in terms of Axiom 1 which handles opposing permissions (lines 11–15) or Axiom 2 which handles mutually exclusive actions (lines 17–19). The analysis takes commutativity of Axiom 3 into consideration.

Since these clauses will not lead to a conflict, the strictness results given in this paper are used to identify stricter clauses (lines 21–22), resulting in `C_stricter` becoming $\{\mathscr{P}_1(a),\ \mathscr{P}_2(a)\}$. These are both new clauses, and the loop will be iterated.

This round of the loop, permission $\mathscr{P}_1(a) \in$ `C_new` will be identified in line 12 to clash with $!\mathscr{P}_1(a) \in$ `C_discovered`, resulting in line 13 returning a `CONFLICT`.

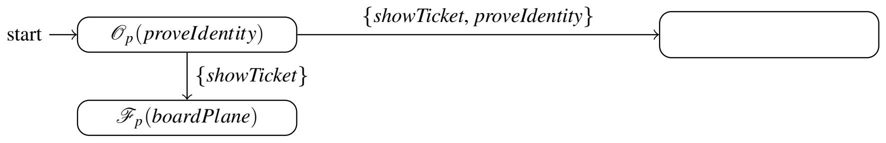
If a conflict is identified, the tool returns a proof, as shown in Fig. 4. Furthermore, a module for state-based (as opposed to action-based) deontic clauses is also available, by encoding the state into two mutually exclusive actions — when it becomes true, and when it becomes false. This tool has been used in [31] to automate the search for all the possible “types of rights”, i.e., combinations of Hohfeldian modalities (see Section 5).

Algorithm 1

```
 1  // Normalise contract
 2  C_new = $\emptyset$
 3  foreach c $\in$ C
 4    case c of
 5      F(p,x)    -> C_new = C_new $\cup$ { !P(p,x) }
 6      otherwise -> C_new = C_new $\cup$ { c }
 7
 8  // Search for conflicts until no new clauses remain to be analysed
 9  C_discovered = C_new
10  while (C_new $\neq$ $\emptyset$) {
11    // Apply axiom 1 (including effect of axiom 3)
12    foreach P(p,x) $\in$ C_new
13      if ( !P(p,x) $\in$ C_discovered ) then return CONFLICT
14    foreach !P(p,x) $\in$ C_new
15      if ( P(p,x) $\in$ C_discovered ) then return CONFLICT
16
17    // Apply axiom 2 (includes effect of axiom 3)
18    foreach O(p,x) $\in$ C_new, O(p,x') $\in$ C_discovered
19      if ( x $\bowtie$ x' ) then return CONFLICT
20
21    // Add stricter clauses
22    C_stricter = immediateStricter(C_new)
23
24    // Check what still needs to be analysed
25    C_discovered = C_discovered $\cup$ C_new
26    C_new = C_stricter $\setminus$ C_discovered
27  }
28
29  return NO-CONFLICT
```

4 Reparations

Since the semantics of contract automata continue enforcing a contract even after violation, they can simulate reparations in a limited way — consider the contrary-to-duty clause: *The passenger is obliged to show a means of identification when presenting the ticket, and would otherwise be prohibited from boarding*. This can be partially emulated using the contract automaton shown below.


start
$\mathscr{O}_p(proveIdentity)$
$\{showTicket, proveIdentity\}$
$\{showTicket\}$
$\mathscr{F}_p(boardPlane)$

This approach, however, has a number of drawbacks. Firstly, we have no implicit notion of which transitions are violating from the automaton. Secondly, although obligation and prohibition reparation can be emulated in this manner, there is no way we can express a reparation in the case of a permission. Finally, the approach is only partial, in that it does not distinguish between whether the passenger chose not to present a means of identification (and hence the airline can apply the reparation) or whether the airline never even gave the option to the passenger to show the means of identification by not providing a synchronising action (and hence the passenger can apply the reparation).

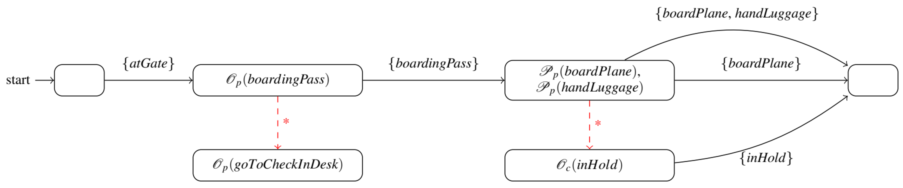
These limitations indicate the need for reparation to be provided as a first class notion in contract automata. In the rest of this section, we present three different forms of reparation extensions to these automata.

4.1 Reparation Automata

Reparations are transitions conditionally taken upon contract violation which can only be detected upon combining the system's behaviours. The first extension to contract automata hinges on this distinction, providing means of specifying two types of transitions — reparation ones which are taken if a violation has taken place and is to be reparated, and normal ones which are otherwise taken.

Consider a contract which states that: *(i) The passenger is obliged to present his boarding pass or would otherwise be obliged to go back to the check-in desk; after which (ii) he is permitted to board the plane with hand-luggage but if stopped from doing so, the airline company is obliged to put his hand-luggage in the hold and allow him to board.* The reparation automaton for this agreement is given in the figure below — red dashed edges are used to identify reparation transitions:[3]

[3] An asterisk $*$ on a transition is used to denote that any action set not matching any other outgoing transition from the source state would follow this transition. Formally, it would be a set of transitions, one for each uncatered for action set.



start
{atGate}
$\mathscr{O}_p(boardingPass)$
{boardingPass}
$\mathscr{P}_p(boardPlane)$, $\mathscr{P}_p(handLuggage)$
{boardPlane, handLuggage}
{boardPlane}
$\mathscr{O}_p(goToCheckInDesk)$
$\mathscr{O}_c(inHold)$
{inHold}
*
*

Definition 15 A *reparation automaton* is a contract automaton with two transition relations $\rightarrow_N$ (normal) and $\rightarrow_R$ (reparation), each a subset of $Q \times 2^{\Sigma} \times Q$. While the normal relation is to be total and deterministic as in the case of contract automata, the reparation one need not be total but must be deterministic. We will write $hasRep(q,A)$ if for some state q' there is a reparation transition $q \xrightarrow{A}_R q'$.

We can now define the tagged operational semantics of a regulated two party system using the following rules (we write q and q' to denote the combined states $(q_1,q_2)_{q_{3_\mathscr{A}}}$ and $(q'_1,q'_2)_{q'_{3_\mathscr{A}}}$ respectively):

$$\frac{(q_1,q_2) \xrightarrow{A} (q'_1,q'_2),\ q_{3_\mathscr{A}} \xrightarrow{A}_N q'_{3_\mathscr{A}}}{q \xrightarrow[(\checkmark,\checkmark)]{A} q'} \neg viol(q,A)$$

$$\frac{(q_1,q_2) \xrightarrow{A} (q'_1,q'_2),\ q_{3_\mathscr{A}} \xrightarrow{A}_R q'_{3_\mathscr{A}}}{q \xrightarrow[(\delta^{\checkmark}_{\mathrm{R}}(1,q,A),\,\delta^{\checkmark}_{\mathrm{R}}(2,q,A))]{A} q'} viol(q,A)$$

$$\frac{(q_1,q_2) \xrightarrow{A} (q'_1,q'_2),\ q_{3_\mathscr{A}} \xrightarrow{A}_N q'_{3_\mathscr{A}}}{q \xrightarrow[(\delta^{\checkmark}_{\times}(1,q,A),\,\delta^{\checkmark}_{\times}(2,q,A))]{A} q'} viol(q,A) \wedge \neg hasRep(q,A)$$

Reparation automata enable the description of reparations but are still limited in a number of ways. Unlike contract automata, they are not closed under synchronous composition, since non-determinism can result from the composition. This happens when a state has more than one active clause since it is impossible to distinguish which of these has been violated. This is one of the major shortcomings of reparation automata, which we seek to address in extended reparation automata.

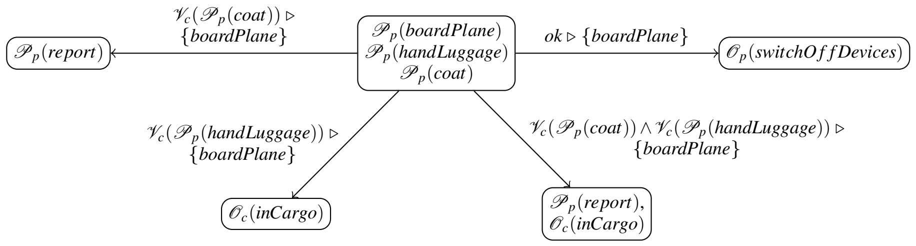
4.2 Extended Reparation Automata

Some situations require identifying which combination of clauses was violated, as each may be reparated in a different way. For instance, if *the passenger is (i) permitted to have one piece of hand luggage, but if not allowed on board, the crew is obliged to send it in as cargo; and (ii) he is also permitted to have a coat, but if not allowed on* *board, he may report the issue.* To be able to identify which clauses the reparation is related to, one can tag reparation transitions with the contract clauses the reparation addresses.

The solution we adopt is to have transitions tagged not only with an action set, but also with an expression specifying which clauses were violated (or not) by a party ($\mathscr{V}_p(c)$ and $\neg\mathscr{V}_p(c)$). We use *ok* to denote that none of the clauses in the source state are violated by either party:

$$\mathrm{VExp} ::= ok \mid \mathscr{V}_{\mathrm{Party}}(Clause) \mid \neg\mathrm{VExp} \mid \mathrm{VExp} \wedge \mathrm{VExp}$$

To illustrate the use of this approach, recall the reparation automaton given in the example discussed earlier. Reparation automata can only define one reparation transition for the action *boardPlane*, however a party could be in violation in multiple ways, i.e., either the passenger was not allowed the hand luggage, or the coat, or both. This can be modelled using extended reparation automata as shown below:[4]


$\mathscr{V}_c(\mathscr{P}_p(coat)) \rhd \{boardPlane\}$
$\mathscr{P}_p(report)$
$\mathscr{P}_p(boardPlane)$ $\mathscr{P}_p(handLuggage)$ $\mathscr{P}_p(coat)$
$ok \rhd \{boardPlane\}$
$\mathscr{O}_p(switchOffDevices)$
$\mathscr{V}_c(\mathscr{P}_p(handLuggage)) \rhd \{boardPlane\}$
$\mathscr{V}_c(\mathscr{P}_p(coat)) \wedge \mathscr{V}_c(\mathscr{P}_p(handLuggage)) \rhd \{boardPlane\}$
$\mathscr{O}_c(inCargo)$
$\mathscr{P}_p(report)$, $\mathscr{O}_c(inCargo)$

Definition 16 An *extended reparation automaton* is a contract automaton where the transition relation is augmented with a boolean expression over clause violation: $\rightarrow \subseteq \mathrm{Q} \times \mathrm{VExp} \times 2^{\Sigma} \times \mathrm{Q}$. Totality and determinism of the transition relation is still required — for any action set A, the disjunction of the violation expressions on transitions tagged by A must be a tautology and any two such expressions must be mutually exclusive.

We can now give the tagged semantics of extended reparation automata as follows (we write $(q,A) \vdash V$ to denote that violation expression $V \in \mathrm{VExp}$ is satisfied when action set A is taken from state q):

$$\frac{(q_1,q_2) \xrightarrow{A} (q_1',q_2'),\ q_{3_{\mathscr{A}}} \xrightarrow{ok \rhd A} q'_{3_{\mathscr{A}}}}{q \xrightarrow[(\checkmark,\checkmark)]{A} q'}\ (q,A) \vdash ok$$

$$\frac{(q_1,q_2) \xrightarrow{A} (q_1',q_2'),\ q_{3_{\mathscr{A}}} \xrightarrow{V \rhd A} q'_{3_{\mathscr{A}}}}{q \xrightarrow[(\delta_{\mathrm{R}}^{\checkmark}(1,q,A),\, \delta_{\mathrm{R}}^{\checkmark}(2,q,A))]{A} q'}\ V \neq ok,\ (q,A) \vdash V$$

[4] The expression *ok* is used to denote that none of the clauses in the source state are violated by either of the parties.

The totality of the transition relation means that no transitions are unmitigatable violating ones. We can address this problem by having a violation state, in which violations with no reparation are sent to. Alternatively, one could modify the semantics of extended reparation automata to allow for partial transitions and treat missing transitions or the transition (ok, A) as a catch-all when A takes place and no other transition is activated. The semantics given above are, however, more compositional and thus preferred. Given that one can now differentiate between a norm being violated or not, extended reparation automata are closed under synchronous composition.

4.3 Hierarchical Reparation Automata

Logic-based approaches provide compositional structure such as nesting to contracts which is potentially lost when using an automaton-based approach. For instance, the reparation for an airline not to allow a passenger to board might be to have to offer the passenger overnight accommodation, and (later) book them on the next available flight. By looking at an automata (of both reparation or extended reparation automata types) describing this behaviour it is not possible to know whether the reparation is the overnight accommodation offer or whether it extends to the booking on the next available flight or whether it extends even further. A standard way of introducing structure and nesting in an automaton-based formalism is to use *hierarchical automata* (e.g., see [27]) to be able to encapsulate a whole automaton in a single state, resulting in different levels of behavioural detail thus giving a means of grouping behaviour together.

We have used such a nested approach in *hierarchical reparation automata*, in which any state can be refined into two automata — the first giving the contract in force (without reparations) and the second being another contract which will be triggered if any part of the first contract is violated. This nesting can be done to any arbitrary depth to enable the description of reparations triggered in case a reparation itself is violated.

Definition 17 A *hierarchical reparation automaton $H \in \mathscr{H}$* consists of (i) a multi-action automaton $S = \langle \Sigma, \mathrm{Q}, q_0, \rightarrow \rangle$, with: (i) a state partition $Q = Q_0 \cup Q_N$ (with $Q_0 \cap Q_N = \emptyset$) where Q_0 are the normal states and Q_N are the nested states; (ii) a set of final states $F \subseteq Q$; (iii) a sink state $V \in Q_0$; (iv) a set of clauses associated to each normal state contract $\in Q_0 \rightarrow 2^{Clause}$; and (v) two functions used to access the normal contract and the reparation automata in a nested state: $\mathrm{nrm}, \mathrm{rep} \in Q_N \rightarrow \mathscr{H}$.

An example of such an automaton can be seen in Fig. 5. This automaton represents a contract that is intended to start holding in case of a flight cancellation: The airline is obliged to book a hotel room for the passenger. If it does not do this it is obliged to provide a hotel voucher that the passenger can use to book the hotel themselves, otherwise a monetary compensation is to be given. Whichever, the passenger is then obliged to give a copy of the hotel's booking receipts to the airline. Then, two hours before the flight the passenger is obliged to check in, otherwise he is forbidden to board the plane.

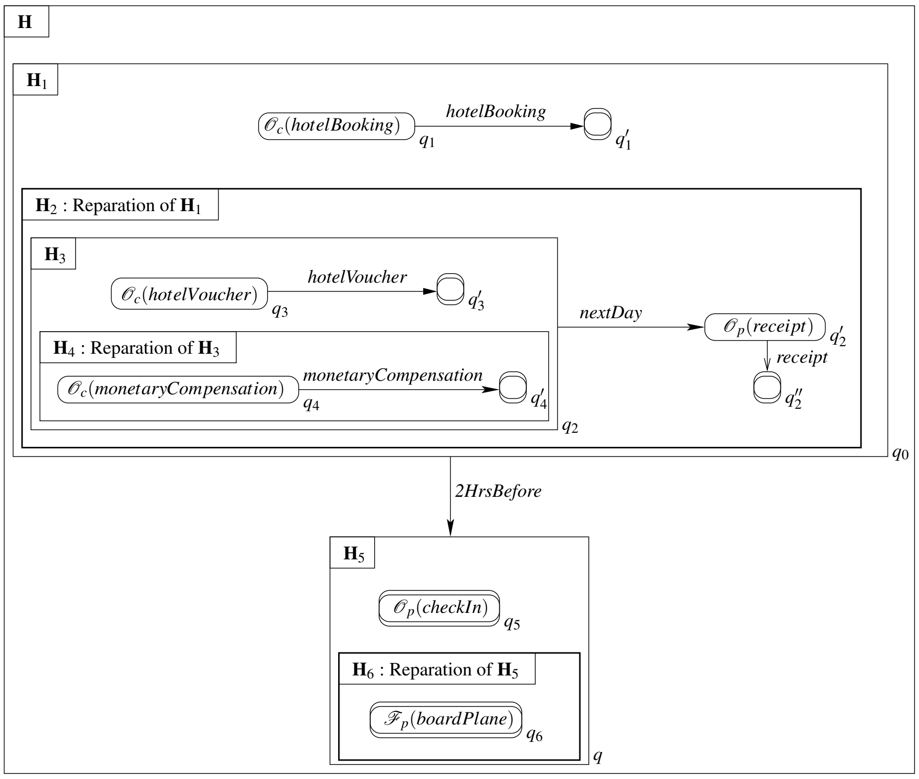


Fig. 5 Example hierarchical reparation automaton.

The configuration of a hierarchical reparation automaton consists of a stack of pairs of states and automata specifying where the contract currently lies. In an automaton H, the stack[5] just contains $(H,\ q_0(H))$ (which we write as $\mathit{conf}_0(H)$) and advances by (i) moving along the top automaton if no violation is detected, also removing the item from the stack if a final state is reached or adding an item if the state moved into is a nested one; (ii) moving to the initial state of the reparation automaton of the next item on the stack (if any) in the case of a violation; or (iii) moving to the sink state V in case of no reparation automaton. The rules specifying the behaviour of a regulated system using hierarchical reparation automata are shown in Fig. 6 (we write q to denote $(q_1,q_2)_{q_{3_{\mathscr{A}}}}$).

We can illustrate how these rules are applied when taking a specific trace of Fig. 5. The trace starts with the crew being obliged to book a hotel for the passenger ($(H,q_0(H))$ which in turn resolves to $(H_1,q_1) : (H,q)$), but the crew does not do this, leading to $\mathit{conf}_0(H_2) : (H,q)$ using REP, which then resolves to $(H_3,q_3) : (H_2,q_2) : (H,q)$. The crew is then obliged to book a hotel room for the passenger, however they do not find any empty rooms in its hotel contacts, resulting in a viola-

[5] We represent a stack in the following way: $(H,q_{\mathscr{CA}}) : hs$, where $(H,q_{\mathscr{CA}})$ is the item on top of the stack, $q_{\mathscr{CA}}$ is the current state in automaton H and hs is the rest of the stack.

If doing A does not violate the contract, and the next contract state is not a final or nested state, then transition normally:

$$\frac{(q_1,q_2) \xrightarrow{A} (q_1',q_2'),\ q_{3_{\mathscr{A}}} \xrightarrow{A} q_{3_{\mathscr{A}}}'}{(q_1,q_2)_{(H,q_{3_{\mathscr{A}}}):hs} \overset{A}{\Rightarrow} (q_1',q_2')_{(H,q_{3_{\mathscr{A}}}'):hs}} \neg viol(q,A) \wedge q_{3_{\mathscr{A}}}' \notin F(H) \cup Q_N(H) \qquad \text{(NORMAL)}$$

If doing A does not violate the contract, and the next contract state is a non-nested, final state, then exit the current automaton:

$$\frac{(q_1,q_2) \xrightarrow{A} (q_1',q_2'),\ q_{3_{\mathscr{A}}} \xrightarrow{A} q_{3_{\mathscr{A}}}'}{(q_1,q_2)_{(H,q_{3_{\mathscr{A}}}):(H',q_{4_{\mathscr{A}}}'):hs} \overset{A}{\Rightarrow} (q_1',q_2')_{(H',q_{4_{\mathscr{A}}}'):hs}} \neg viol(q,A) \wedge q_{3_{\mathscr{A}}}' \in F(H) \setminus Q_N(H) \qquad \text{(POP)}$$

If doing A does not violate the contract, and the next contract state is a nested state, then transition to the start state of the automaton associated with the nested state:

$$\frac{(q_1,q_2) \xrightarrow{A} (q_1',q_2'),\ q_{3_{\mathscr{A}}} \xrightarrow{A} q_{3_{\mathscr{A}}}'}{(q_1,q_2)_{(H,q_{3_{\mathscr{A}}}):hs} \overset{A}{\Rightarrow} (q_1',q_2')_{conf_0(H'):(H,q_{3_{\mathscr{A}}}'):hs}} \neg viol(q,A) \wedge q_{3_{\mathscr{A}}}' \in Q_N(H) \wedge \mathrm{nrm}(q_{3_{\mathscr{A}}}') = H'$$

(PUSH)

If doing A violates the contract, and the current contract state has a reparation, then transition to the start state of the automaton reparating the current state on the stack:

$$\frac{(q_1,q_2) \xrightarrow{A} (q_1',q_2'),\ q_{3_{\mathscr{A}}} \xrightarrow{A} q_{3_{\mathscr{A}}}'}{(q_1,q_2)_{(H,q_{3_{\mathscr{A}}}):(H',q_{4_{\mathscr{A}}}'):hs} \overset{A}{\Rightarrow} (q_1',q_2')_{conf_0(H''):(H',q_{4_{\mathscr{A}}}'):hs}} viol(q,A) \wedge q_{4_{\mathscr{A}}}' \in Q_N(H') \wedge \exists H'' \cdot H'' = \mathrm{rep}(q_{4_{\mathscr{A}}}')$$

(REP)

If doing A violates the contract, and the current contract state does not have a reparation, then transition to the sink state:

$$\frac{(q_1,q_2) \xrightarrow{A} (q_1',q_2'),\ q_{3_{\mathscr{A}}} \xrightarrow{A} q_{3_{\mathscr{A}}}'}{(q_1,q_2)_{(H,q_{3_{\mathscr{A}}}):hs} \overset{A}{\Rightarrow} (q_1',q_2')_V} viol(q,A) \wedge \nexists H'' \cdot H'' = \mathrm{rep}(q_{3_{\mathscr{A}}}) \qquad \text{(VIOL)}$$

Fig. 6 The formal semantics for hierarchical reparation automata.

tion that leads to $conf_0(H_4) : (H_2,q_2) : (H,q)$ using REP, which in turn resolves to $(H_4,q_4) : (H_2,q_2) : (H,q)$. In exchange, the crew gives the passenger a monetary compensation to make up for the hotel's lack of capacity, which leads to $(H_2,q_2) : (H,q)$ using NORMAL. With the passenger booking the hotel and the night passing, the reparation is satisfied and the obligation to show the receipt to the airline is activated, leading to $(H_2,q_2') : (H,q)$ using NORMAL. Then the passenger shows the receipt, which leads to (H,q) using NORMAL. Two hours before the flight the passenger is then obliged to check in to his new flight, leading to $(conf_0(H_5)) : (H,q')$ using *PUSH*.

As in reparation automata, we cannot differentiate between the norms being violated and thus hierarchical reparation automaton are not closed under synchronous composition. Although one could tag reparation sub-automata with an expression specifying upon which violation it is to be triggered (similar to extended reparation automata), this would still not suffice since the normal behaviour automaton may contain multiple states and thus a single norm may be violated from different states.

5 Applications

Contract automata can be used as a reasoning mechanism for many problems in the deontic realm. In this section we visit some applications of it by the authors of this article.

Contract Synthesis and Contract Strictness

In [32] an algorithm for synthesising *imposed* contracts is presented based on the idea that one party's behaviour may impose restrictions on how others may behave when interacting with it. These restrictions may be seen as *implicit contracts* which the other party has to conform to and may thus be considered inappropriate or excessive if they overregulate one of the parties.

As an example, consider the contract that binds a customer and a bank, which stipulates that opening new accounts is free of charge. And yet, at the moment of opening an account, the bank requires the release of personal information and allowance to send the customer promotional material. The bank is not strictly breaching the contract, but maybe it is asking "too much". Can this "too much" be quantified?

As another example consider an airline, which compensates for missed connections due to delays by providing the traveller with food and lodging. However, the airline has a policy of not providing this service unless the customers explicitly demands for it. In a way, the airline is turning its unconditional obligation of providing aid into a conditional or restricted one: given that the customer asks for help, support will be provided.

The notions of conditional permission and contract strictness come into play so a party that wants to engage in a contract with another can compare among different bidders to see who offers the less strict contract – i.e., the one that imposes less restrictions on its own behaviour.

Hohfeldian Modalities in Contracts

In [31] we used contract automata to clarify Hohfeld's *claim right*, *power*, *freedom* and *immunity* modalities [16] and contribute to the debate of whether they embed new building blocks of the deontic though or can be reconstructed from well-known, more traditional, deontic operators.

Based on Kanger's prior analysis of the modalities [17], we give formal semantics to the operators in the context of action-based, interacting two-party systems and we prove that, at least in this context, neither claim, nor power, nor freedom nor immunity are foundational modalities, as they can be defined in terms of positive and negative permissions and obligations, over presence or lack of actions.

For instance, if for a given state of affairs S we define $S^{\uparrow}$ as the action that brings about the state of affairs S and $S^{\downarrow}$ as the action that makes S stop holding, in the context of contracts p's claim right over $\overline{p}$ for state of affairs S becomes $\mathscr{O}_{\overline{p}}(S^{\uparrow})$. Similarly, power, freedom and immunity can be encoded as $\mathscr{P}_p(S^{\uparrow})$, $\mathscr{P}_p(S^{\uparrow})$ and $\mathscr{O}_{\overline{p}}(!S^{\downarrow})$ respectively.

Runtime Monitoring and Enforcement

Contract automata provides a way of encoding contracts at a low-level of abstraction. Although it is not meant as a formalism which humans will use to write contracts directly in, it provides an excellent level of abstraction for automated reasoning technologies such as runtime monitoring and enforcement.

Using runtime verification [20] techniques, the structure of contract automata can be automatically used by various tools, with system behaviour being checked at runtime against the contract clauses in the states to enable identification of violations. Such an approach can be used, for instance, to monitor web services for compliance by ensuring that all communication with the service goes through a central monitor which is traversing the contract represented as a contract automaton. Using such an approach, one can go further by *enforcing* a contract by similarly monitoring progress of the system using a contract automaton, but enacting the active contract clauses whenever the state changes. In [33] we have been looking at a slight variant of contract automata to enable the runtime verification of privacy policies.

Conflict Analysis

In [2] we have looked at means of analysing natural language contracts to identify conflicts in an automated manner. Although a deontic logic is used to provide a semantics to the contracts themselves, the conflict analysis procedure is an extension of the one presented in this paper combined with several off-the-shelf NLP techniques to provide a more intelligent contract editing tool to lawyers. As a result, a tool that works as a plugin for a text editor has been developed. The tool was tested by lawyers and notaries, getting overall positive feedback with suggestions for further work.

6 Comparison to Other Formalisms

Contract automata have been designed specifically to act as a specification language for "contracts" between synchronising systems, while most of the existing formal language for contracts have a different kind of system as target. This, and other specific details about contract automata as explained below, make it difficult to perform a fair comparison. For instance, we note that the notions of obligation, permission and prohibition may vary across logics: in the case of contract automata the semantics deal with interacting systems, thus making an obligation on one party to perform action a also oblige the other party to allow action a to happen. Also, most logics do not include party identification and thus, we somehow need to encode this information which is needed in contract automata.

$$\begin{aligned}
C &:= C_O \mid C_P \mid C_F \mid C \wedge C \mid [\beta]C \mid \top \mid \bot \\
C_O &:= O_C(\alpha) \mid C_O \oplus C_O \\
C_P &:= P(\alpha) \mid C_P \oplus C_P \\
C_F &:= F_C(\alpha) \\
\alpha &:= 0 \mid 1 \mid a \mid \overline{a} \mid \alpha \& \alpha \mid \alpha.\alpha \mid \alpha + \alpha \\
\beta &:= 0 \mid 1 \mid a \mid \overline{a} \mid \beta \& \beta \mid \beta.\beta \mid \beta + \beta \mid \beta^*
\end{aligned}$$

Fig. 7 $\mathscr{CL}$ syntax.

Despite the above constrains, we believe it is instructive to compare our formalism with previously defined languages in the literature. In order to make this comparison we need to consider formalisms that are close in spirit (it should allow at least for the representation of permissions, prohibitions and obligations), and make some explicit assumptions or consider a suitable subset of the language meaningful to be compared with.

In the rest of this section we compare contract automata with the formal language for contracts $\mathscr{CL}$ [34, 35, 36], which is briefly introduced in Section 6.1. In Section 6.2 we informally describe the relation between $\mathscr{CL}$ and $\mathscr{CA}$, whereas in Section 6.3 we introduce $\mathscr{CL}_{rest}$, a suitable subset of $\mathscr{CL}$ comparable to $\mathscr{CA}$. In Section 6.4 we give some preliminaries needed for the comparison. We then proceed to show the translation from $\mathscr{CL}_{rest}$ to $\mathscr{CA}$ in Section 6.5, and from $\mathscr{CA}$ to $\mathscr{CL}_{rest}$ in Section 6.6.

6.1 The contract language $\mathscr{CL}$

In what follows we present the syntax of $\mathscr{CL}$, and give a brief intuitive explanation of its notation and terminology, following [35]. A contract in $\mathscr{CL}$ may be obtained by using the syntax grammar rules shown in Fig. 7.

A $\mathscr{CL}$ contract consists of a conjunction of clauses representing normative expressions, where each clause may, by definition, by itself be considered a contract. The essential concepts are obligations (C_O), permissions (C_P), and prohibitions (C_F), which are then combined by conjunction. They can be "guarded" or conditioned, represented by using the dynamic logic square brackets. $\top$ and $\bot$ are the trivially satisfied and violating contracts respectively. O, P and F are deontic modalities; the obligation to perform an action α is written as $O_C(\alpha)$, showing the primary obligation to perform α, and the reparation contract C if α is not performed. This represents what is usually called in the deontic community a *Contrary-to-Duty* (*CTD*), as it specifies what is to be done if the primary obligation is not fulfilled. The prohibition to perform α is represented by the formula $F_C(\alpha)$, which not only specifies what is forbidden but also what is to be done in case the prohibition is violated (the contract C); this is called *Contrary-to-Prohibition* (*CTP*). Both CTDs and CTPs are useful to represent normal (expected) behaviour, as well as alternative (exceptional) behaviour. $P(\alpha)$ represents the permission of performing a given (complex) action α. Note that in $\mathscr{CL}$ it is assumed that permissions cannot be violated, and thus they do not have reparations.

In the description of the syntax, we have also represented what are the allowed actions (α and β in Fig. 7). It should be noted that the usage of the Kleene star (*) –which is used to model repetition of actions– is not allowed inside the above described deontic modalities, though they can be used in dynamic logic-style conditions. Indeed, actions β may be used inside the dynamic logic modality (the bracket $[\cdot]$) representing a condition in which the contract C must be executed if action β is performed. The binary constructors (&, ., and $+$) represent (true) concurrency, sequence and choice over basic actions respectively. Compound actions are formed from basic ones by using these operators. Conjunction of clauses can be expressed using the $\wedge$ operator; the exclusive choice operator ($\oplus$) can only be used in a restricted manner. 0 and 1 are two special actions that represent the impossible action and the skip action (matching any action) respectively.

The concurrency (or *synchrony*) action operator & should only be applied to actions that can happen simultaneously. $\mathscr{CL}$ offers the possibility to explicitly specify such actions by defining the following relation between actions: *a#b* if and only if it is not the case that *a&b*. We call such actions *mutually exclusive* (or *contradictory*).

$\mathscr{CL}$ has been designed to avoid deontic paradoxes, as this is a common problem when defining a language formalising normative concepts (*cf.* [26]). Besides, $\mathscr{CL}$ enjoys additional properties concerning the relation between the different normative notions, as for instance that obligations implies permissions, and that prohibition may be defined as the negation of permission. It has also been proven that some undesirable properties do not hold, such as that the permission of performing a simple action does not imply the permission of performing concurrent actions containing that simple action (similarly for prohibitions). See [34, 36] for a more detailed presentation of $\mathscr{CL}$, including a proof of how deontic paradoxes are avoided as well as the properties of the language.

6.2 Informal comparison between $\mathscr{CL}$ and $\mathscr{CA}$

$\mathscr{CL}$ and $\mathscr{CA}$ address different types of systems — whereas the former takes a logic-based approach to describe systems in which the parties behave autonomously, the latter takes an operational approach for systems in which the parties interact and actions may be synchronised. The different formalisms thus have different strengths, even if the differences in application domains can be addressed by encoding party synchronisation within $\mathscr{CL}$, or by having no interaction in $\mathscr{CA}$.

Due to different domains addressed, $\mathscr{CL}$ and $\mathscr{CA}$ are not strictly comparable. For instance, $\mathscr{CA}$ has an inherent notion of parties which is not in $\mathscr{CL}$, while $\mathscr{CL}$ can handle exclusive disjunction over contracts, which $\mathscr{CA}$ cannot deal with. Another important point of difference is that $\mathscr{CA}$ assumes interactive (and synchronous) systems, which results in inherently different semantics of the basic deontic notions. E.g., obligation on a party to perform an action in $\mathscr{CA}$ automatically induces a weak obligation on the other party to provide the handshake action, thus allowing the origi-

$$\begin{aligned} C &:= O_{\perp}(a) \mid P(a) \mid F_{\perp}(a) \mid C \wedge C \mid [\beta]C \mid \top \mid \bot \\ \beta &:= 0 \mid 1 \mid a \mid \overline{a} \mid \beta \& \beta \ \mid \beta.\beta \mid \beta^* \end{aligned}$$

Fig. 8 $\mathscr{CL}_{rest}$ syntax.

nal party to satisfy their obligation. Such weak obligations cannot be encoded in $\mathscr{CL}$, and conversely the one-sided obligation of $\mathscr{CL}$ cannot be encoded in $\mathscr{CA}$.

Due to these differences, we limit our comparison of $\mathscr{CA}$ to a syntactic subset of $\mathscr{CL}$ with the following constraints:

– Modalities (O, P and F) are only applied to simple actions;
– No $\oplus$ (exclusive or) between deontic operators;
– No action complementation;
– No reparations.

The last constraint is not necessary if we extend our view to hierarchical contract automata, but we will limit our discussion to contract automata. In the rest of this section we will compare $\mathscr{CA}$ with this restricted version of $\mathscr{CL}$, which we will refer to as $\mathscr{CL}_{rest}$.

6.3 $\mathscr{CL}_{rest}$: A subset of the contract language $\mathscr{CL}$

As explained above, we will be considering not the full language $\mathscr{CL}$ but a subset, which we will cal $\mathscr{CL}_{rest}$. The syntax of this new language is presented in Fig. 8. We provide here a trace semantics for $\mathscr{CL}_{rest}$ (cf. Fig. 9), which has been obtained by constraining the deontic trace semantics of $\mathscr{CL}$ [9].

For a contract with action alphabet Σ, we will introduce its deontic alphabet Σ_d which consists of O_a, P_a and F_a for each action $a \in \Sigma$, that will be used to represent which normative behaviour is enacted at a particular moment.

Given a $\mathscr{CL}_{rest}$ contract C with action alphabet Σ, the semantics will be expressed in the form $\sigma, \sigma_d \vDash C$, where σ is a finite trace of sets of concurrent actions in Σ and σ_d is a finite trace consisting on sets of deontic information in Σ_d. The statement $\sigma, \sigma_d \vDash C$ is said to be well-formed if $length(\sigma) = length(\sigma_d)$. In the rest of the paper we will consider only well-formed semantic statements.

A well-formed statement $\sigma, \sigma_d \vDash C$ will correspond to the statement that action sequence σ is possible under (will not break) contract C, with σ_d being the deontic statements enforced from the contract.

Example 6
As an example, let us consider the $\mathscr{CL}_{rest}$ contract $C = [a]O(b) \wedge [b]F(b)$, and the trace $\sigma = \langle \{a\}, \{b\} \rangle$, then $\sigma_d = \langle \emptyset, \{O_b\} \rangle$, and we have that $\sigma, \sigma_d \vDash C$. The contract $C' = F(c) \wedge [1](O(a) \wedge F(b))$, for example, stipulates that it is forbidden to perform action c and that after the execution of any action, there is an obligation to perform an a (while prohibiting the execution of b), so we can write $\sigma_d = \langle \{F_c\}, \{O_a, F_b\} \rangle$. The contract allows, for instance, the execution of actions a and b concurrently, and then a concurrently with c ($\sigma = \langle \{a,b\}, \{a,c\} \rangle$), and we have that $\sigma, \sigma_d \vDash C'$. On the other

$$\begin{array}{rlll}
\sigma,\sigma_d \vDash C & \text{if} & \mathit{length}(\sigma) = \mathit{length}(\sigma_d) = 0 & (1)\\
\sigma,\sigma_d \vDash \top & \text{if} & \sigma_d(0) = \emptyset \text{ and } \sigma(1..),\sigma_d(1..) \vDash \top & (2)\\
\sigma,\sigma_d \vDash C_1 \wedge C_2 & \text{if} & \sigma,\sigma'_d \vDash C_1 \text{ and } \sigma,\sigma''_d \vDash C_2 \text{ and } \sigma_d = \sigma'_d \cup \sigma''_d & (3)\\
\sigma,\sigma_d \vDash [\varepsilon]C & \text{if} & \sigma,\sigma_d \vDash C & (4)\\
\sigma,\sigma_d \vDash [\alpha_\&]C & \text{if} & (\alpha_\& \not\subseteq \sigma(0) \Rightarrow \sigma,\sigma_d \vDash \top) \text{ and} & (5)\\
 & & (\alpha_\& \subseteq \sigma(0) \Rightarrow (\sigma_d(0) = \emptyset \text{ and } \sigma(1..),\sigma_d(1..) \vDash C)) & (6)\\
\sigma,\sigma_d \vDash [\overline{a}]C & \text{if} & (a \in \sigma(0) \Rightarrow \sigma,\sigma_d \vDash \top) \text{ and} & (7)\\
 & & (a \notin \sigma(0) \Rightarrow (\sigma_d(0) = \emptyset \text{ and } \sigma(1..),\sigma_d(1..) \vDash C)) & (8)\\
\sigma,\sigma_d \vDash [\beta;\beta']C & \text{if} & \sigma,\sigma_d \vDash [\beta][\beta']C & (9)\\
\sigma,\sigma_d \vDash [\beta+\beta']C & \text{if} & \sigma,\sigma_d \vDash [\beta]C \wedge [\beta']C & (10)\\
\sigma,\sigma_d \vDash [\beta^*]C & \text{if} & \sigma,\sigma_d \vDash C \wedge [\beta][\beta^*]C & (11)\\
\sigma,\sigma_d \vDash O_C(a) & \text{if} & \sigma_d(0) = O_a \text{ and} & (12)\\
 & & (a \in \sigma(0) \Rightarrow \sigma(1..),\sigma_d(1..) \vDash \top) \text{ and} & (13)\\
 & & (a \notin \sigma(0) \Rightarrow \sigma(1..),\sigma_d(1..) \vDash C) & (14)\\
\sigma,\sigma_d \vDash F_C(a) & \text{if} & \sigma_d(0) = F_a \text{ and} & (15)\\
 & & (a \in \sigma(0) \Rightarrow \sigma(1..),\sigma_d(1..) \vDash C) \text{ and} & (16)\\
 & & (a \notin \sigma(0) \Rightarrow \sigma(1..),\sigma_d(1..) \vDash \top) & (17)\\
\sigma,\sigma_d \vDash P(a) & \text{if} & \sigma_d(0) = P_a \text{ and } \sigma(1..),\sigma_d(1..) \vDash \top & (18)
\end{array}$$

Fig. 9 The deontic trace semantics of $\mathcal{CL}_{rest}$.

hand, any trace starting with c will be rejected. Similarly, any trace starting with any action other than c but followed by a b or anything but an a, will also be rejected.

Given two traces σ_1 and σ_2, we will use $\sigma_1;\sigma_2$ to denote their *concatenation*, and $\sigma_1 \cup \sigma_2$ (provided the length of σ_1 is equal to that of σ_2) to denote the *point-wise union* of the traces: $\langle \sigma_1(0) \cup \sigma_2(0),\ \sigma_1(1) \cup \sigma_2(1),\ \ldots \sigma_1(n) \cup \sigma_2(n) \rangle$. Besides, we will use the notation $w' = w +\!\!+ \langle a \rangle$ to denote that the last action of trace w' is a. Similarly, we use the same notation for the deontic trace $w_d = w_d +\!\!+ \langle c \rangle$, where c is a (possible empty) set containing O_a, P_a, or F_a for any a.

In what follows we explain the trace semantics of $\mathcal{CL}_{rest}$ (cf. Fig. 9), which is based on that presented for the conflict detection algorithm for $\mathcal{CL}$ [9].[6]

Basic conditions: Empty traces satisfy any contract, as shown in Fig. 9-(1).

Done, Break: The simplest definitions are those of the trivially satisfiable contract $\top$, and the unsatisfiable contract $\bot$. In the case of $\bot$, only an empty sequence will not have yet broken the contract, while in the case of $\top$, any sequence of actions satisfies the contract (whenever no obligation, prohibition, or permission is present on the trace). See Fig. 9 line (2).

Conjunctions: For the conjunction of two contracts, the action trace must satisfy both contracts, and the deontic traces are combined point-wise. See Fig. 9 line (3).

[6] We do not present the trivial cases of actions 0 and 1; they are omitted in the rest of the paper.

Conditions: Conditions are handled structurally. Note that using the normal form defined in [19], one can push concurrent actions to the bottom level. See Fig. 9 lines (4)–(11).

Obligations: Obligations, like conditions, are defined structurally on action expressions. The base case of the action simply consisting of a conjunction of actions that can be dealt with by ensuring that if the actions are present in the action trace, then the contract is satisfied, otherwise the reparation is enacted. The case for the sequential composition of two action sequences is handled simply by rewriting into a pair of obligations. The case of choice (+) is the most complex case, in which we have to consider the possibility of having either obligation satisfied or neither satisfied, hence triggering the reparation. Recall that the star operator cannot appear within obligations. See Fig. 9 lines (12)–(14).

Prohibitions: Dealing with prohibitions is similar to obligations, with the main difference being that prohibition of choice is more straightforward to express. See Fig. 9 lines (15)–(17).

Permissions: See Fig. 9 line (18) for the semantics.

6.4 Preliminaries for the comparison between $\mathscr{CA}$ and $\mathscr{CL}_{rest}$

As discussed at the beginning of this section the semantics of contract automata deal with interacting systems, where an obligation on one party to perform action a implicitly includes an obligation on the other party to allow action a to take place. This is a different semantics of obligation when compared to obligations in $\mathscr{CL}_{rest}$— the additional weak obligation on the other party in contract automata (the obligation to provide a viable action set which includes action a) cannot be modelled directly in $\mathscr{CL}_{rest}$ and thus strong equivalence between the two formalisms cannot be achieved. However, we will show that if we limit ourselves to a weaker form of equivalence which considers only which deontic clauses (obligations, permissions and prohibitions) are actively enforced after a particular trace, we can indeed show that contract automata and $\mathscr{CL}_{rest}$ are, in this sense, *weakly equivalent*.

In order to compare $\mathscr{CA}$ and $\mathscr{CL}_{rest}$ we need some preliminaries. Without lost of generality we will define a *system execution trace* (or simply a *system trace*) to be a sequence of events. We assume that each event is in fact a pair containing the event itself and who is the actor (subject, or performer of the event), and that the set *Names* contains all the possible actors included in a contract (in bilateral contracts as we are having here, there usually be only two subjects, namely the two parties involved in the contract). We also assume two projection functions giving the action itself and the subject, namely $actor : Events \rightarrow Names$, and $action : Events \rightarrow Actions$. For example, if the event $e = p : a$ is "the customer sends the document by email to the provider", then $p = actor(e) = \textit{the customer}$ and $a = action(e) = \textit{sends the document by email to the provider}$.

We assume then that traces in $\mathscr{CL}_{rest}$ and in $\mathscr{CA}$ may in fact be always syntactically translated into a system trace and that the alphabet in both formalisms is the same (meaning that we have the same set of actors, actions, and events). So, w.l.o.g. we will use w (or other letter from the end of the alphabet or a primed version of such letters) to characterize system traces in both formalisms, and we simply call them *traces* without distinguishing whether we are in a $\mathscr{CL}_{rest}$ or in a $\mathscr{CA}$ setting.

Definition 18 – $\mathscr{CA}$ - $\mathscr{CL}_{rest}$ Clause Equivalence
We define the *clauses in force of a $\mathscr{CA}$ $\mathscr{A} = \langle \Sigma, Q, q0, \rightarrow \rangle$ after following w* to be $ClauseInForce_{CA}(w, \mathscr{A}) = \{\mathbb{D}_{p:a} \mid \mathbb{D}_p(a) \in contract(q_0 \overset{w}{\Rightarrow})\}$. The set of *formulae in force of a $\mathscr{CL}_{rest}$ formulae ψ after following w* is defined to be the set of deontic formulae c' such that $w,\ w_d +\!\!+ \langle c' \rangle \vdash \psi$.

Finally, we define the function $ca2cl$: Clause $\rightarrow \mathscr{CL}_{rest}$ as follows: $ca2cl(O_p(a)) = O_\perp(p:a)$, $ca2cl(F_p(a)) = F_\perp(p:a)$, $ca2cl(P_p(a)) = P(p:a)$.

Definition 19 – $\mathscr{CA}$ - $\mathscr{CL}_{rest}$ Weak Equivalence
We say that a contract automaton $\mathscr{A}$ is *weakly equivalent* to a $\mathscr{CL}_{rest}$ formula ψ, written $\mathscr{A} \xleftrightarrow{CA\text{-}CLr} \psi$, if and only if for any prefix of a trace w, it is always the case that the clauses in force are the same. That is, iff $ca2cl(ClauseInForce_{CA}(w, \mathscr{A})) = ClauseInForce_{\mathscr{CL}_{rest}}(w, \psi)$.

We say that the formalisms $\mathscr{CA}$ and $\mathscr{CL}_{rest}$ are weakly equivalent iff for any $\mathscr{CA}$ automaton $\mathscr{A}$ we can always find a corresponding formula ψ in $\mathscr{CL}_{rest}$ such that $\mathscr{A} \xleftrightarrow{CA\text{-}CLr} \psi$, and vice-versa.

6.5 Translation of $\mathscr{CL}_{rest}$ into $\mathscr{CA}$

We provide in what follows a translation of $\mathscr{CL}_{rest}$ into $\mathscr{CA}$. The translation algorithm is heavily inspired by the one for $\mathscr{CL}$ conflict detection [9], implemented in the tool CLAN [10]. In particular we are interested only in the automata generation part of the algorithm (for the sublanguage given by the $\mathscr{CL}_{rest}$ syntax).

In what follows $\alpha_{\&}$ will denote basic actions or complex actions constructed from basic actions only using the concurrent operator & (for example $a, a\&b$). It can be shown that every action expression can be transformed into an equivalent representation where & appears only at the innermost level (see [36]). This representation is referred to as the *canonical form*; we assume all such terms are in canonical form in $\mathscr{CL}_{rest}$ formulae.

Given a contractual formula $\psi \in \mathscr{CL}_{rest}$, over an action alphabet Σ and corresponding deontic alphabet Σ_d, we can construct a contract automaton $\mathscr{A}(\psi) = \langle \Sigma, Q, q0, \rightarrow \rangle$ with clauses $contract \in Q \rightarrow 2^{\text{Clause}}$. Before giving the precise construction (cf. Algorithm 2) we need some preliminaries. The construction of the automaton uses the residual contract function *residual* which, given a $\mathscr{CL}_{rest}$ formula ψ and a set of actions $A \in 2^{\Sigma}$, will return the subformula that needs to hold in a following step after actions in A are executed; note that A may in particular be a single action. This function is defined in Fig. 10.

As an example, let us consider the $\mathscr{CL}_{rest}$ formula $\psi \overset{df}{=} [a]O(b) \wedge [b]F(a)$. Applying the residual function to all possible actions we get: $residual(\psi, \{a\}) = O(b)$, $residual(\psi, \{b\}) = F(a)$, $residual(\psi, \{a,b\}) = O(b) \wedge F(a)$.

$$\begin{array}{rcl}
\mathit{residual} & : & \mathscr{CL}_{rest} \times 2^{\Sigma} \to \mathscr{CL}_{rest} \\
\mathit{residual}(\top,\varphi) & = & \top \\
\mathit{residual}(\bot,\varphi) & = & \bot \\
\mathit{residual}(C_1 \wedge C_2,\varphi) & = & \mathit{residual}(C_1,\varphi) \wedge \mathit{residual}(C_2,\varphi) \\
\mathit{residual}([\alpha_{\&}]C,\varphi) & = & \begin{cases} C & \text{if } \alpha_{\&} \subseteq \varphi \\ \top & \text{otherwise} \end{cases} \\
\mathit{residual}([\overline{a}]C,\varphi) & = & C \quad \text{if } a \notin \varphi \\
\mathit{residual}([\beta;\beta']C,\varphi) & = & \mathit{residual}([\beta][\beta']C,\varphi) \\
\mathit{residual}([\beta+\beta']C,\varphi) & = & \mathit{residual}([\beta]C \wedge [\beta']C,\varphi) \\
\mathit{residual}([\beta^*]C,\varphi) & = & \mathit{residual}(C \wedge [\beta][\beta^*]C,\varphi) \\
\mathit{residual}(O_C(a),\varphi) & = & \begin{cases} \top & \text{if } a \in \varphi \\ C & \text{otherwise} \end{cases} \\
\mathit{residual}(F_C(a),\varphi) & = & \begin{cases} C & \text{if } a \in \varphi \\ \top & \text{otherwise} \end{cases} \\
\mathit{residual}(P(a),\varphi) & = & \top
\end{array}$$

Fig. 10 The *residual* function.

$$\begin{array}{ll}
\mathit{deontic}(C_1 \wedge C_2) & = \mathit{deontic}(C_1) \cup \mathit{deontic}(C_2) \\
\mathit{deontic}(O(a)) & = \{O_a\} \\
\mathit{deontic}(F(a)) & = \{F_a\} \\
\mathit{deontic}(P(a)) & = \{P_a\} \\
\mathit{deontic}(\text{ otherwise }) & = \emptyset
\end{array}$$

Fig. 11 The labelling function *deontic*.

Our translation algorithm needs to consider the sequence of subformulae *reachable* from a given formula, that is, all the subformulae that are to be enforced when a given action is performed starting with the original formula.

Definition 20 Given a $\mathscr{CL}_{rest}$ formula ψ, we define the set of formulae *transitively reachable from* ψ, written $\mathscr{R}_{\psi}$, to be all formulae ψ' such that there exist (i) a sequence of formulae $\psi_0, \psi_1 \ldots \psi_n$ with $\psi_0 = \psi$ and $\psi_n = \psi'$); and (ii) a sequence of action sets $A_0, A_2 \ldots A_{n-1}$, such that for all i, $\mathit{residual}(\psi_i, A_i) = \psi_{i+1}$.

It can be proved that $\mathscr{R}_{\psi}$ is finite for any $\psi \in \mathscr{CL}_{rest}$. We also need to define a labelling function *deontic* taking a $\mathscr{CL}_{rest}$ formula and giving the corresponding deontic statements corresponding to the deontic modality to be enforced at a given moment. The definition of this function is shown in Fig. 11. We can now present the translation algorithm taking a $\mathscr{CL}_{rest}$ formula and giving a contract automaton (cf. Algorithm 2).

Algorithm 2 *Given a $\mathscr{CL}_{rest}$ formula ψ we define the contract automaton $\mathscr{A}(\psi) = \langle \Sigma, Q, q0, \rightarrow \rangle$ with clauses* contract $\in Q \rightarrow 2^{Clause}$, *as follows:*

- *The alphabet Σ is the same as for $\mathscr{CL}_{rest}$;*
- *The set of states Q is given by all the formulae $\mathscr{R}_{\psi}$ reachable from ψ;*
- *q_0 is defined to be ψ;*
- *The transition relationship is defined as follows: $(q, A, q') \in \rightarrow$ if residual$(q, A) = q'$;*
- *The clauses given by the function* contract *on each state are given by the labelling function deontic.*

The automaton thus constructed is an instance of the one provided for conflict analysis for $\mathscr{CL}$ [9]. One can easily show that such an automaton is total and deterministic, and is thus a correct contract automaton. Furthermore, $\mathscr{A}(\psi)$ is weakly equivalent to ψ.

Lemma 1 *A $\mathscr{CL}_{rest}$ formula ψ is weakly equivalent to the contract automaton constructed from ψ following Algorithm 2: $\mathscr{A}(\psi) \xleftrightarrow{CA\text{-}CL} \psi$.*

The proof is an instance of the proof given for $\mathscr{CL}$ for the conflict detection algorithm given in [9]. From Algorithm 2 and the above lemma we get the following theorem.

Theorem 5 *Given a $\mathscr{CL}_{rest}$ contract ψ, there exists a contract automaton $\mathscr{A}$ such that $\mathscr{A} \xleftrightarrow{CA\text{-}CL} \psi$.*

By recursively applying the residual function (that is by taking the closure of the continuation relation starting with the original formula) we will get all the subformulae given the states of the automaton. The transition relationship $\mathscr{R}_\psi$ will "connect" all the above produced subformulae thus giving the automaton as expected. Again, the proof of the theorem is an instance of the corresponding proof given in [9].

6.6 Transforming $\mathscr{CA}$ into an equivalent $\mathscr{CL}_{rest}$ specification

We will approach the problem of transforming a contract automaton into a $\mathscr{CL}$ formula by noting that (i) it is possible to obtain a regular expression representing all finite strings which lead to a particular state in a contract automaton; (ii) a contract automaton with a single clause in a single state is equivalent to the $\mathscr{CL}_{rest}$ formula $[e]c$ (where e is the regular expression over events which lead to the only non-empty state which contains the clause c); (iii) any contract automaton can be decomposed into the synchronous composition of a number of contract automata each containing a single clause; and (iv) synchronous composition over contract automata corresponds to conjunction over formulae in $\mathscr{CL}_{rest}$.

We will define these terms and formally show the consistency of the argument.

Definition 21 A contract automaton $\langle \Sigma, Q, q0, \rightarrow \rangle$ with clauses $contract \in Q \rightarrow 2^{\text{Clause}}$ is said to be *singularly claused* if it contains exactly one clause in exactly one state (the rest of the states have no clauses): $\sum_{q \in Q} \#(contract(q)) = 1$. We call the class of such automata $\mathscr{CA}_1$.

We start by showing how a contract automaton can be decomposed into the synchronous composition of a number of singularly claused contract automata.

Lemma 2 *Given a contract automaton $\mathscr{A} = \langle S, \text{contract} \rangle$ (with $S = \langle \Sigma, Q, q0, \rightarrow \rangle$ and clauses* $\text{contract} \in Q \rightarrow 2^{Clause}$*), there exists a set of singularly claused contract automata $\{\mathscr{A}_1, \ldots, \mathscr{A}_n\} \in 2^{\mathscr{CA}_1}$ such that $\mathscr{A} = \|_{i \in \{1,\ldots,n\}} \mathscr{A}_i$ synchronising over the whole alphabet.*

Proof The proof is by induction over the total number of clauses in $\mathscr{A}$. We can show that given a clause c_1 in state q_1, we can construct the two automata $\mathscr{A}_k = \langle S, contract_k\rangle$ and $C_1 = \langle S, contract_1\rangle$ where:

$$contract_k(q) \overset{df}{=} \begin{cases} contract(q) \setminus \{c_1\} & \text{if } q = q_1 \\ contract(q) & \text{otherwise} \end{cases}$$
$$contract_1(q) \overset{df}{=} \begin{cases} \{c_1\} & \text{if } q = q_1 \\ \emptyset & \text{otherwise} \end{cases}$$

Note that:

1. $\mathscr{A}_1$ is singularly claused: $\mathscr{A}_1 \in \mathscr{C}\mathscr{A}_1$.
2. $\mathscr{A}_k$ has one less clause than the original contract automaton $\mathscr{A}$.
3. It can be shown that the reachable state space of $\mathscr{A}_k \|_\Sigma \mathscr{A}_1$ is identical to $\mathscr{A}$.

By induction we can thus reduce $\mathscr{A}$ into the synchronous composition of singularly claused contract automata. □

Example 7
Let us consider the contract automaton $\mathscr{A} = \langle S, contract\rangle$ (where $S = \langle \Sigma, Q,\ q0,\ \rightarrow\rangle$, with clauses $contract \in Q \rightarrow 2^{\text{Clause}}$), where: $\Sigma = \{a,b,c\}$, $Q = \{q,q'\}$, $q0 = q$, $\rightarrow = \{(q,\{a\},q'),(q',\{c\},q)\}$, and $contract = \{(q,\{P_p(a)\}),(q',\{O_p(b)\})\}$. The above construction will give the following two singularly claused automata:
$\mathscr{A}_1 = \langle S_1, contract_1\rangle$, where $\Sigma = \{a,b,c\}$, $Q_1 = \{q,q'\}$, $q0_1 = q$, $\rightarrow = \{(q,\{a\},q'),(q',\{c\},q)\}$, and $contract = \{(q,\{P_p(a)\}),(q',\emptyset)\}$; and
$\mathscr{A}_2 = \langle S_2, contract_2\rangle$, where $\Sigma = \{a,b,c\}$, $Q_2 = \{q,q'\}$, $q0_2 = q$, $\rightarrow = \{(q,\{a\},q'),(q',\{c\},q)\}$, and $contract = \{(q,\emptyset),(q',\{O_p(b)\})\}$.

It is easy to see that the synchronous composition of $\mathscr{A}_1$ and $\mathscr{A}_2$ is equivalent to the original contract $\mathscr{A}$.

We now show how to obtain a $\mathscr{C}\mathscr{L}_{rest}$ formula from a singularly claused $\mathscr{C}\mathscr{A}$.

Lemma 3 *For any singularly claused contract automaton $\mathscr{A}$ one can find a $\mathscr{C}\mathscr{L}$ formula ψ_C such that the two are equivalent: $\mathscr{A} \xleftrightarrow{CA\text{-}CL} \psi$.*

Proof Let c be the only clause appearing in $\mathscr{A}$, to be found in state q. Using standard automata results, we can identify a regular expression e over sets of actions which represents the set of traces taking us from the initial state of $\mathscr{A}$ to state q. Now, consider the $\mathscr{C}\mathscr{L}_{rest}$ formula $[e]c$. It is easy to show that: (i) after any trace matching e, the clause in force is the single clause c for both the $\mathscr{C}\mathscr{L}_{rest}$ formula and $\mathscr{A}$; and (ii) after any other trace, neither contract $\mathscr{A}$ nor the $\mathscr{C}\mathscr{L}_{rest}$ formula $[e]c$ will enforce any clause. □

Example 8
Let us consider again the above example with contract automaton $\mathscr{A} = \langle S, contract\rangle$ (where $S = \langle \Sigma, Q,\ q0,\ \rightarrow\rangle$, with clauses $contract \in Q \rightarrow 2^{\text{Clause}}$), where: $\Sigma = \{a,b,c\}$, $Q = \{q,q'\}$, $q0 = q$, $\rightarrow = \{(q,\{a\},q'),(q',\{c\},q)\}$, and $contract = \{(q,\{P_p(a)\}),(q',\{O_p(b)\})\}$, and with singularly claused automata $\mathscr{A}_1$ and $\mathscr{A}_2$ as above. The $\mathscr{C}\mathscr{L}$ formulae obtained by following the procedure described in the proof of Lemma 3 are $[(a.c)^*]P(a)$ (obtained from $\mathscr{A}_1$) and $[a.(c.a)^*]O(b)$ (obtained from $\mathscr{A}_2$).

In order to get our final result, we need the following auxiliary lemma relating conjunction in $\mathscr{CL}_{rest}$ and synchronous composition in contract automata.

Lemma 4 *Conjunction in $\mathscr{CL}_{rest}$ is semantically equivalent to synchronous composition over contract automata: if $\mathscr{A}_1 \xleftrightarrow{CA\text{-}CL} \psi_1$ and $\mathscr{A}_2 \xleftrightarrow{CA\text{-}CL} \psi_2$, then $\mathscr{A}_1 \|_G \mathscr{A}_2 \xleftrightarrow{CA\text{-}CL} \psi_1 \wedge \psi_2$.*

Proof The proof follows from the fact that the active clauses deriving from formula $\psi_1 \wedge \psi_2$ after a trace w are equivalent to the union of the clauses deriving from ψ_1 and ψ_2 separately, with trace w. Similarly, based on the definition of synchronous composition it can be shown that $contract((q_{0_1}, q_{0_2}) \overset{w}{\Rightarrow}) = contract(q_{0_1}) \cup contract(q_{0_2})$. □

Based on these last two lemma, the following theorem follows immediately.

Theorem 6 *Given contract automaton $\mathscr{A}$, there always exists an equivalent $\mathscr{CL}_{rest}$ formula ψ: $\mathscr{A} \xleftrightarrow{CA\text{-}CL} \psi$.*

7 Related Work

Despite the fact that contracts are by definition an agreement between two or more parties, most formal studies regulate the parties independently and do not analyse how permissions, obligations or prohibitions for one party affect the other, or do so in limited ways. Here we summarise the most closely related work.

Xu [43] presents a multi-party contract model where a contract is represented by a set of actions, graph describing possible interaction among the parties, and set of *commitments*, which is the name the article gives to obligations. In contrast with our approach, the model does not allow to describe permissions nor prohibitions.

Marjanovic *et al.* [23] aims at formalisations of contracts for e-commerce but focuses only on analysing temporal consistency. Governatori *et al.* [13] deals with obligation violations in contracts using the domain specific BCL language [12], introducing contrary-to-duty clauses and directed obligations. Neither of them analyses the reciprocity of deontic clauses in a contract. That is, how the clauses imposed on one of the parties also put restrictions on the behaviour of the other, an aspect that is central to our approach.

A related line of research was started by Herrestad *et al.* [15], later followed upon by various others ([41, 8], etc.) — although not explicitly about contracts, they look at a flavour of axiomatic deontic logic with obligations being directed from one individual towards another, termed *directed obligations*. Their main concern is the relation of general operators with directed ones (i.e., the question of whether a general obligation is equivalent to a directed obligation imposed on every agent), and similar issues. Directed permissions have also been studied, but were termed to be conflicting because of lack of a clear counterparty, following both the *claimant theory* (were the stress is placed on who can claim in a Court of Law when a violation occurs) or the *benefit theory* (that places the emphasized on who a given norm is intended to benefit). A salient different with our approach is that the idea of synchronicity of actions and parties having to explicitly collaborate so, e.g., one of them can fulfill her obligations, is not present. Also, once one considers actions that are only realisable by the two parties in synchrony, as our approach does, the concept of permission appears more clearly.

Ryu [37] also analyses directed obligations and permissions but in a defeasible axiomatic model where the responsibilities are not mutual. The motivation for his approach can be seen in the following quote:

A contractual obligation is an obligation that is relativized to two persons, an obligor and a benefactor, i.e., 'the obligor α_1 has an obligation to the benefactor α_2 that φ, given that ψ'. For α_1, some situation in which both φ and ψ are true is legally better than any situation in which ψ is true but φ is not, because in a situation in which ψ is true but φ is not, the obligor α_1 is legally responsible and the law prescribes sanctions against α_1. However, the benefactor α_2 does not have any legal responsibility for situations in which ψ is true but φ is not. Instead, if α_1 fails to bring about it that φ or see to it that φ when ψ, α_2 has recourse to certain legal actions against α_1, which are ratified by the law.

As can be seen Ryu's approach also does not consider that some actions might require collaboration and thus is explicit about placing the onus of compliance solely on one of the parties.

Kanger [17] also introduces the notion of directionality of modalities, stating that, e.g., party p has versus party p' a *claim right* that $S(p, p')$. However, as noted by many (e.g., [22, 15]), the notion of the counterparty is weekly present in Kanger's theory, and there is not a notion of the 'weak' obligations that the permission on one party impose on the other. Indeed, that difference is very clear when contract automata is used to formalise Kanger's types of rights: some deontic combinations that are valid in Kanger's approach become conflicting because of the onuses imposed into the other party. See [31] for a full analysis.

Although our model deals with interaction, it does not provide explicitly for the notion of *interference* that has been analysed by many, notably Hohfeld [16] and Lindahl [21], It is important to understand, however, that the difference between *vested* and *naked* liberties (i.e., warranty of immunity from interference) relates to a real concern in the context of general law but blurs in the context of a contract where one party allowing the other to perform a shared action, but reserving itself the right to interfere, does not have practical sense. More specifically, in our formal model $\mathscr{P}_p(a)$ means not only that p may attempt to perform a — it means that p would succeed in doing a should she try. If the notion of *attempting* to do an action a that can be interfered by others needs to be modeled, then another action *attempt_a* should be added and the permission placed onto the latter. Another alternative is to introduce modalities for trying, as in Santos *et al.* [39].

Surden [40] describes the advantages of computable contracts so that their rules can be monitored. More recently Flood *et al.* [11] proposed representing financial agreements as automata. Their informal article describes advantages of using directed finite automata as a graphical representation for contracts where *happy* and *unhappy* paths can be described.

Lindahl [21] studies *liberty spaces* to present the concept of *less free than*, a relationship between maximally consistent sets of deontic positions. The general idea is somewhat similar to our definition of strictness (see Section 2.2); however, as Lindahl notes, most of the maximally consistent sets are incomparable using this relationship, whereas our notion of strictness provides interesting theorems.

Many of the above mentioned authors, and also others, deal with some definition of conflicts but they usually leave out the inconsistencies that arise because of the onuses imposed to the other party (see our example of $\mathscr{P}_p(!a)$ conflicting with $\mathscr{O}_p(a)$ in Section 3).

Another automata approach to normative systems is explored by Martínez *et al.* [25, 6, 24, 7] in *C-O Diagrams*. Unlike our approach, such diagrams do not consider synchronous actions as we are doing here. On the other hand, C-O Diagrams allow for the representation of real-time constraints, something not covered by our formalism.

With a coincidentally similar name, Basile *et al.* [4] presents a formalism also called contract automata that splits actions in requests and matching offers. Unlike our approach, parties are not separately represented but implicit in the requesting and offering of the actions in the contract graph. Also, the contracts are not tagged with deontic operators. Leaned towards service-oriented computing, the underlying idea is that the contract either accept or reject a stream of actions (a word in automata parlance) that comes from the interaction of the parties.

Recently, Bench-Capon [5] has studied how transitions systems have been used to model and analyse norms, in the context of multi-agent systems. He identifies three approaches to dealing with undesirable actions: enforcement, which simply means removing the offending transitions; sanctions, which may result in additional agents, states and transitions to be added to the transition system; and by creating desirable and undesirable transitions, with the agents hopefully coordinating on the most desirable transition (done by labelling a transition with values such as *Compliance* and *Safety*). Our approach only limits itself to contracts, rather than the wide of view of norms explored by Bench-Capon. However, our approach extending contract automata with reparations is similar to the sanctions approach detailed in his article.

8 Conclusions

In this paper we have presented contract automata as a low-level formalism to enable the analysis of contracts. We show how such a formalism can be effectively used for analysis in a variety of settings — from the algebraic approach to contract conflict analysis to the compositional analysis of contract synthesis and combinational analysis and its relation to other deontic formalisms. Apart from the previously published results, we have presented more complete results about dealing with reparations in contract automata, and presented a new means of reparation handling through the use of hierarchical contract automata. Furthermore, we formally compare $\mathscr{CL}$ and contract automata in terms of expressivity and translation between the formalisms. Finally, we have presented a tool for the automated analysis of conflicts.

One of the things shown in this paper is how the formalism is both tractable for pencil-and-paper proofs, although still being low-level to be targeted effectively by automated static and dynamic analysis tools. Although in this paper we only present one such automated analysis technique for conflict analysis, we are currently looking into the building of a toolset for the generation and analysis of contract automata.

There are various other directions we are currently exploring — on one hand, we are looking at the use of contract automata to characterise different deontic concepts such as *conditional permissions*, and analysis of contracts with unknown sub-parts which is essential for natural language text contract analysis. The building of effective tools for the analysis of contract automata is crucial for the practical impact of this work, but at the other end, we also techniques for the generation of contract automata from more abstract formalisms. Building upon the translation from $\mathscr{CL}$ in this paper, we are currently looking at other deontic notations, including natural language texts.

Acknowledgements This research has been partially supported by: UBACyT 20020130200032BA, the European ICT COST Action IC1402 (*ARVI: Runtime Verification beyond Monitoring*), and the Swedish Research Council under grants Nr. 2012-5746 (*ReMU: Reliable Multilingual Digital Communication: Methods and Applications*) and Nr. 2015-04154 (*PolUser: Rich User-Controlled Privacy Policies*).

References

1. Arnold, A.: Nivat's processes and their synchronization. Theor. Comput. Sci. **281**, 31–36 (2002)
2. Azzopardi, S.: Extending contract automata with reparation, hypothetical and conditional clauses. Tech. rep., University of Malta (2014)
3. Azzopardi, S., Pace, G.J., Schapachnik, F.: Contract automata with reparations. In: Legal Knowledge and Information Systems - JURIX 2014: The Twenty-Seventh Annual Conference, Jagiellonian University, Krakow, Poland, 10-12 December 2014, *Frontiers in Artificial Intelligence and Applications*, vol. 271, pp. 49–54. IOS Press (2014)
4. Basile, D., Degano, P., Ferrari, G.L.: Automata for analysing service contracts. In: Trustworthy Global Computing - 9th International Symposium, TGC 2014, Rome, Italy, September 5-6, 2014. Revised Selected Papers, *Lecture Notes in Computer Science*, vol. 8902, pp. 34–50. Springer (2014)
5. Bench-Capon, T.J.M.: Analysing norms with transition systems. In: Legal Knowledge and Information Systems - JURIX 2014: The Twenty-Seventh Annual Conference, Jagiellonian University, Krakow, Poland, 10-12 December 2014, *Frontiers in Artificial Intelligence and Applications*, vol. 271, pp. 29–38. IOS Press (2014)
6. Díaz, G., Cambronero, M.E., Martínez, E., Schneider, G.: Timed Automata Semantics for Visual e-Contracts. In: 5th International Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'11), *Electronic Proceedings in Theoretical Computer Science*, vol. 68, pp. 7–21. Málaga, Spain (2011)

7. Díaz, G., Cambronero, M.E., Martínez, E., Schneider, G.: Specification and verification of normative texts using c-o diagrams. IEEE Transactions on Software Engineering **99**, 1 (2013)
8. Fasli, M.: On commitments, roles, and obligations. In: Revised Papers from the Second International Workshop of Central and Eastern Europe on Multi-Agent Systems: From Theory to Practice in Multi-Agent Systems, CEEMAS '01, pp. 93–102. Springer-Verlag (2002)
9. Fenech, S., Pace, G.J., Schneider, G.: Automatic Conflict Detection on Contracts. In: ICTAC'09, *LNCS*, vol. 5684, pp. 200–214. Springer (2009)
10. Fenech, S., Pace, G.J., Schneider, G.: CLAN: A tool for contract analysis and conflict discovery. In: ATVA'09, *LNCS*, vol. 5799, pp. 90–96. Springer (2009)
11. Flood, M.D., Goodenough, O.R.: Contract as automaton: The computational representation of financial agreements. Available at SSRN 2538224 (2014)
12. Governatori, G.: Representing business contracts in *RuleML*. Int. J. Cooperative Inf. Syst. **14**(2-3), 181–216 (2005)
13. Governatori, G., Milosevic, Z.: Dealing with contract violations: formalism and domain specific language. In: EDOC Enterprise Computing Conference, 2005 Ninth IEEE International, pp. 46–57. IEEE (2005)
14. Hage, J.: Contrary to Duty Obligations — A Study in Legal Ontology. In: Legal Knowledge and Information Systems (JURIX 2001) (2001)
15. Herrestad, H., Krogh, C.: Deontic logic relativised to bearers and counterparties. Anniversary Anthology in Computers and Law pp. 453–522 (1995)
16. Hohfeld, W.: Some fundamental legal conceptions as applied in judicial reasoning. Yale Lj **23**, 16 (1913)
17. Kanger, S., Kanger, H.: Rights and parliamentarism. Theoria **32**(2), 85–115 (1966)
18. Kripke, S.: Semantical considerations on modal logic. Acta Philosophica Fennica **16**, 83–94 (1963)
19. Kyas, M., Prisacariu, C., Schneider, G.: Run-time monitoring of electronic contracts. In: 6th International Symposium on Automated Technology for Verification and Analysis (ATVA'08), *LNCS*, vol. 5311, pp. 397–407. Springer-Verlag, Seoul, South Korea (2008)
20. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5), 293–303 (2009)
21. Lindahl, L.: Position and change: A study in law and logic, vol. 112. Springer (1977)
22. Makinson, D.: On the formal representation of rights relations. Journal of philosophical Logic **15**(4), 403–425 (1986)
23. Marjanovic, O., Milosevic, Z.: Towards formal modeling of e-contracts. In: Proceedings of the 5th IEEE International Conference on Enterprise Distributed Object Computing, EDOC '01, pp. 59–. IEEE Computer Society, Washington, DC, USA (2001)
24. Martínez, E., Díaz, G., Cambronero, M.: Contractually compliant service compositions. In: Proceedings of the 9th international conference on Service-Oriented Computing (ICSOC'11), *Lecture Notes in Computer Science*, vol. 7084, pp. 636–644. Springer-Verlag Berlin, Paphos, Cyprus (2011)

25. Martínez, E., Díaz, G., Cambronero, M.E., Schneider, G.: A model for visual specification of e-contracts. In: The 7th IEEE International Conference on Services Computing (IEEE SCC'10), pp. 1–8. IEEE Computer Society, Miami, USA (2010)
26. McNamara, P.: Deontic Logic. In: Gabbay, D.M., Woods, J., eds.: Handbook of the History of Logic, vol. 7, pp. 197–289. North-Holland Publishing (2006)
27. Mikk, E., Lakhnech, Y., Siegel, M.: Hierarchical automata as model for statecharts. In: Third Asian Computing Science Conference. Advances in Computing Science – ASIAN'97, *Lecture Notes in Computer Science*, vol. 1345. Springer Verlag (1997)
28. Pace, G.J., Schapachnik, F.: Permissions in Contracts, a Logical Insight. In: The 24th International Conference on Legal Knowledge and Information Systems (JURIX'11), *Frontiers in Artificial Intelligence and Applications*, vol. 235. IOS Press, University of Vienna, Austria (2011)
29. Pace, G.J., Schapachnik, F.: Permissions in contracts, a logical insight. In: JURIX, pp. 140–144 (2011)
30. Pace, G.J., Schapachnik, F.: Contracts for interacting two-party systems. In: FLACOS'12, *ENTCS*, vol. 94 (2012)
31. Pace, G.J., Schapachnik, F.: Types of rights in two-party systems: A formal analysis. In: Legal Knowledge and Information Systems - JURIX 2012: The Twenty-Fifth Annual Conference, University of Amsterdam, The Netherlands, 17-19 December 2012, *Frontiers in Artificial Intelligence and Applications*, vol. 250, pp. 105–114. IOS Press (2012)
32. Pace, G.J., Schapachnik, F.: Synthesising implicit contracts. In: ICAIL'13, pp. 217–221. ACM, New York, NY, USA (2013)
33. Pardo, R., Colombo, C., Pace, G., Schneider, G.: An automata-based approach to evolving privacy policies for social networks. In: 16th International Conference on Runtime Verification (RV) (Madrid, Spain. 2016), LNCS (2016)
34. Prisacariu, C., Schneider, G.: A Formal Language for Electronic Contracts. In: FMOODS, *LNCS*, vol. 4468, pp. 174–189. Springer (2007)
35. Prisacariu, C., Schneider, G.: CL: An Action-based Logic for Reasoning about Contracts. In: WOLLIC'09, *LNCS*, vol. 5514, pp. 335–349. Springer (2009)
36. Prisacariu, C., Schneider, G.: A dynamic deontic logic for complex contracts. Journal of Logic and Algebraic Programming **81**(4), 458–490 (2012)
37. Ryu, Y.: Specification of contractual obligations in formal business communication. Data & knowledge engineering **26**(3), 309–326 (1998)
38. Salcianu, A., Rinard, M.: Pointer and escape analysis for multithreaded programs. In: ACM SIGPLAN Notices, vol. 36, pp. 12–23. ACM (2001)
39. Santos, F., Jones, A., Carmo, J.: Action concepts for describing organised interaction. In: System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on, vol. 5, pp. 373–382. IEEE (1997)
40. Surdan, H.: Computable contracts. UCDL Rev. **46**, 629 (2012)
41. Tan, Y.H., Thoen, W.: A logical model of directed obligations and permissions to support electronic contracting. Int. J. Electron. Commerce **3**, 87–104 (1998)
42. Von Wright, G.: Deontic logic: A personal view. Ratio Juris **12**, 26–38 (1999)
43. Xu, L.: A multi-party contract model. SIGecom Exchanges **5**(1), 13–23 (2004)