Verification of Data-Aware Commitment-Based Multiagent System

Marco Montali Diego Calvanese
KRDB Research Centre for Knowledge and Data
Free University of Bozen-Bolzano, Italy
surname@inf.unibz.it

Giuseppe De Giacomo
Dip. di Ing. Informatica, Automatica e Gestionale
Sapienza Università di Roma, Italy
degiacomo@dis.uniroma1.it

ABSTRACT

In this paper we investigate multiagent systems whose agent interaction is based on social commitments that evolve over time, in presence of (possibly incomplete) data. In particular, we are interested in modeling and verifying how data maintained by the agents impact on the dynamics of such systems, and on the evolution of their commitments. This requires to lift the commitment-related conditions studied in the literature, which are typically based on propositional logics, to a first-order setting. To this purpose, we propose a rich framework for modeling data-aware commitment-based multiagent systems. In this framework, we study verification of rich temporal properties, establishing its decidability under the condition of "state-boundedness", i.e., data items come from an infinite domain but, at every time point, each agent can store only a bounded number of them.

Categories and Subject Descriptors

I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—Multiagent Systems; D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

Data aware commitments, formal verification, temporal logics, ontologies

1. INTRODUCTION

In this paper we investigate multiagent systems (MASs) whose agent interaction is based on social commitments that evolve over time, in presence of possibly incomplete data. MASs based on social commitments have been extensively studied in the literature [8]. Intuitively, a social commitment $\textsc{cc}(\mathsf{d}, \mathsf{c}, q_p, q_d)$ models a relationship between a debtor agent d and a creditor agent c, in which d commits towards c that, whenever condition q_p holds in the system, it will bring about condition q_d in the following course of interaction. Commitments provide a semantics for the agent interaction that abstracts away from the internal agent implementation, and can be thus employed to specify business protocols and contracts. The establishment of commitments is regulated by contracts, which depend on domain-specific events and conditions. Established commitments, in turn, have a lifecycle that is regulated by a so-called *commitment machine* [11] on the basis of such contracts.

Appears in: *Proceedings of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2014), Lomuscio, Scerri, Bazzan, Huhns (eds.), May, 5–9, 2014, Paris, France.*

While in the literature, virtually all the work is based on propositional contents for such commitments [8], here we explicitly manage data described through first-order formalisms, in line with [7]. In other words, we study how data maintained by the agents impact on the dynamics of such systems, and on the evolution of their commitments. Technically, this requires to lift to first-order the notions related to contracts, commitments, and commitment machines.

As a result, we obtain a powerful framework of *data-aware commitment-based MASs* (DACMASs), which incorporates the typical notions of commitment-based MASs but in a rich, data-aware context. In our framework, the commitment machine itself becomes a special agent, called *institutional*, which is in charge of supporting the evolution of the system according to the commitments. In addition, this agent manages core information about the MAS itself, such as the list of participating agents, which changes over time as the system unfolds.

Data manipulated by agents are described in terms of a domain ontology, expressed in a lightweight description logic (DL), tailored towards ontology-based data access. This ontology provides a common ground for the agent interaction and commitments, establishing the vocabulary that is shared by all of them. In particular, we rely on *DLR-Lite* [5], the n-ary version of the DL at the base of the OWL 2 QL profile of the OWL 2 W3C standard. Note that the system dynamics and commitment-machines representation are orthogonal to how data are represented, and could be recast into a plain relational setting, at the price of losing the semantic richness of DL. Each agent has its own data about the domain and the contracts it is involved in, expressed in terms of a *DLR-Lite* ontology. Such data are manipulated through actions, in response to events and according to the commitments in place. At each point in time, only a finite number of data is present in the system. However, such data change over time: old data are removed by the agents, and new data (coming from a countably infinite domain Δ) are inserted.

The main result of this paper is that, when a DACMAS is *state-bounded*, i.e., the number of data that are simultaneously present at each moment in time is bounded, verification of rich temporal properties becomes decidable. More specifically, we are able to check DACMASs against properties expressed in a sophisticated first-order variant of μ-calculus with a controlled form of quantification across states. We do this by exploiting recent results in [2, 6], and reducing verification of state-bounded DACMASs to finite-state model checking through a faithful form of abstraction, essentially obtained by replacing real data items with a finite number of symbolic values, while correctly preserving the relationships among the real data items themselves.

2. PRELIMINARIES

Description Logics (DLs) [1] are logics that represent the domain of interest in terms of *objects*, *concepts*, denoting sets of objects, and *relations* between objects.

We consider here the DL *DLR-Lite* [5], which is a DL that belongs to the *DL-Lite* family of lightweight DLs and that is equipped with relations of arbitrary arity. In *DLR-Lite*, concepts C and relations R are built from atomic concepts N and atomic relations P (of arity ≥ 2) according to:

$$C \longrightarrow N \mid P[i] \mid C \sqcap C \qquad\qquad R \longrightarrow P \mid P[i_1, \ldots, i_h]$$

where $h \geq 2$ and for $i_1, \ldots, i_h$, which denote pairwise distinct components of relation P, we have that $\{i_1, \ldots, i_h\} \subseteq \{1, \ldots, n\}$, where n is the arity of P. Similarly, $i \in \{1, \ldots, n\}$. Intuitively, $\sqcap$ denotes concept conjunction, while $P[i_1, \ldots, i_m]$ denotes the projection of relation P on its components $i_1, \ldots, i_m$. This results in a concept if $m = 1$ and in a relation otherwise.

Formally, the semantics of DLs is given in terms of first-order *interpretations* $\mathcal{I} = (\Delta^\mathcal{I}, \cdot^\mathcal{I})$, where $\Delta^\mathcal{I}$ is a nonempty *interpretation domain*, and $\cdot^\mathcal{I}$ is an *interpretation function*, assigning to each concept C a subset $C^\mathcal{I}$ of $\Delta^\mathcal{I}$, and to each n-ary relation R an n-ary relation $R^\mathcal{I}$ over $\Delta^\mathcal{I}$ such that

$$\begin{array}{rcl}(C_1 \sqcap C_2)^\mathcal{I} & = & C_1^\mathcal{I} \cap C_2^\mathcal{I} \\ (P[i_1, \ldots, i_m])^\mathcal{I} & = & \{(o'_1, \ldots, o'_m) \mid \text{there is } \vec{o} \in P^\mathcal{I} \text{ s.t.} \\ & & \quad \vec{o}[i_j] = o'_j, \text{ for } j \in \{1, \ldots, m\}\}\end{array}$$

(Here, $\vec{o}[i]$ denotes the i-th component of tuple $\vec{o}$.) Also, $\cdot^\mathcal{I}$ assigns to each constant a an object $a^\mathcal{I}$ of $\Delta^\mathcal{I}$. We adopt the unique name assumption, i.e., $a_1 \neq a_2$ implies $a_1^\mathcal{I} \neq a_2^\mathcal{I}$.

In DLs, knowledge about the domain of interest is encoded in an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$, which is formed by a *TBox* $\mathcal{T}$, encoding intensional knowledge, and an *ABox* $\mathcal{A}$, encoding extensional knowledge about individuals objects.

A *DLR-Lite* TBox is a finite set of assertions of the form:

$$\begin{array}{rl} E_1 \sqsubseteq E_2 & \text{(concept/relation } \textit{inclusion assertion}\text{)}, \\ E_1 \sqsubseteq \neg E_2 & \text{(concept/relation } \textit{disjointness assertion}\text{)}, \\ (\mathsf{key}\ i_1, \ldots, i_\ell \colon R) & (\textit{key assertion}), \end{array}$$

where R has arity n, and $1 \leq i_1 < i_2 < \cdots < i_\ell \leq n$. To ensure decidability of inference, and good computational properties, we require that no relation P can appear both in a key assertion and in the right hand side of a relation inclusion assertion [10, 5].

A *DLR-Lite* ABox is a finite set of assertions of the form:

$$\begin{array}{rl} N(a_1) & (\textit{concept membership assertion}), \\ P(a_1, \ldots, a_n) & (\textit{relation membership assertion}), \end{array}$$

where P has arity n, and $a_1, \ldots, a_n$ denote constants.

The semantics of an ontology is given by stating when an interpretation $\mathcal{I}$ *satisfies an assertion*, where $\mathcal{I}$ satisfies: $E_1 \sqsubseteq E_2$, if $E_1^\mathcal{I} \subseteq E_2^\mathcal{I}$; $E_1 \sqsubseteq \neg E_2$, if $E_1^\mathcal{I} \cap E_2^\mathcal{I} = \emptyset$; $(\mathsf{key}\ i_1, \ldots, i_\ell \colon R)$, if there are no two distinct tuples in $R^\mathcal{I}$ that agree on all their components $i_1, \ldots, i_\ell$; $N(a_1)$, if $a_1^\mathcal{I} \in N^\mathcal{I}$; and $P(a_1, \ldots, a_n)$, if $(a_1^\mathcal{I}, \ldots, a_n^\mathcal{I}) \in P^\mathcal{I}$. A *model* of an ontology $\mathcal{O} = \langle \mathcal{T}, \mathcal{A} \rangle$ is an interpretation that satisfies all assertions in $\mathcal{T}$ and $\mathcal{A}$. An ontology $\mathcal{O}$ is *satisfiable* if it has at least one model, and it *logically implies* an assertion α, written $\mathcal{O} \models \alpha$, if all models of $\mathcal{O}$ satisfy α.

Next we introduce queries, whose answers, as usual in ontologies, are formed by constants denoting individuals explicitly mentioned in the ABox. In the following, when convenient, we identify a vector $\vec{x}$ with the set of its elements. A *union of conjunctive queries* (UCQ) q over an ontology $\langle \mathcal{T}, \mathcal{A} \rangle$ is a FOL formula of the form $\bigvee_{i=1}^{n} \exists \vec{y_i}.conj_i(\vec{x}, \vec{y_i})$ with free variables $\vec{x}$ and existentially quantified variables $\vec{y}_1, \ldots, \vec{y}_n$. Each $conj_i(\vec{x}, \vec{y_i})$ in q is a conjunction of atoms of the form $N(z)$, $P(\vec{z})$, where N and P respectively denote a concept and a role name occurring in $\mathcal{T}$, and z, $\vec{z}$ are constants of $\mathcal{A}$ or variables in $\vec{x}, \vec{y}_1, \ldots, \vec{y}_n$. The *(certain) answers* to q over $\langle \mathcal{T}, \mathcal{A} \rangle$ is the set $\text{ANS}(q, \mathcal{T}, \mathcal{A})$ of substitutions θ of the free variables of q with constants in $\mathcal{A}$ such that $q\theta$ evaluates to true in every model of $\langle \mathcal{T}, \mathcal{A} \rangle$, denoted $\langle \mathcal{T}, \mathcal{A} \rangle \models q\theta$. *DLR-Lite* enjoys nice computational properties, in particular w.r.t. query evaluation: computing the certain answers to a UCQ can be done in polynomial time in the size of $\langle \mathcal{T}, \mathcal{A} \rangle$, and in AC^0 in the size of $\mathcal{A}$ alone (i.e., in data complexity) [5]. Such result is based on the *FOL rewritability* property of *DLR-Lite* [5], which states that for every UCQ q and TBox $\mathcal{T}$, we can *rewrite* q into a new UCQ $rew_\mathcal{T}(q)$ such that $\text{ANS}(q, \mathcal{T}, \mathcal{A}) = \text{ANS}(rew_\mathcal{T}(q), \emptyset, \mathcal{A})$, for every ABox $\mathcal{A}$. In other words, the TBox can be "compiled away".

We also consider ECQs, which are FOL queries whose atoms are UCQs evaluated according to the certain answer semantics above [4]. An *ECQ* over $\mathcal{T}$ and $\mathcal{A}$ is a possibly open formula of the form (where q is a UCQ):

$$Q \longrightarrow [q] \mid \neg Q \mid Q_1 \wedge Q_2 \mid \exists x.Q$$

The *(certain) answers* to Q over $\langle \mathcal{T}, \mathcal{A} \rangle$, is the set of substitutions θ of the free variables of Q with constants in $\mathcal{A}$ defined by composing the certain answers of the UCQs q in Q through first-order constructs, and interpreting existential variables as ranging over the constants in $\mathcal{A}$ only. Hence, the first-order constructs in ECQs are interpreted under a (weaker) epistemic semantics. ECQs over *DLR-Lite* ontologies enjoy the same computational properties as UCQs, in particular FOL rewritability of query answering [4].

3. FRAMEWORK

We introduce now our framework for modeling DACMASs. Formally, a DACMAS is a tuple $\langle \mathcal{T}, \mathcal{E}, \mathcal{X}, \mathcal{I}, \mathcal{C}, \mathcal{B} \rangle$, where: *(i)* $\mathcal{T}$ is a global *DLR-Lite TBox*; *(ii)* $\mathcal{E}$ is a set of predicates denoting *events* (where the predicate name is the event type, and the arity determines the content/payload of the event); *(iii)* $\mathcal{X}$ is a finite set of *agent specifications*; *(iv)* $\mathcal{I}$ is a (partial) *specification for the institutional agent*; *(v)* $\mathcal{C}$ is a *contractual specification*; *(vi)* and $\mathcal{B}$ is a *Commitment Box* (CBox). We discuss each component in detail.

3.1 The Global TBox

The global TBox is used to represent the key concepts, relations and (semantic) constraints characterizing the domain in which the agents operate, so as to provide a *common ground* for the agent interaction. Part of this TBox is fixed for every agent system, and is used to model core notions related to the system itself. The extension of such core notions is maintained by a single, special *institutional* agent, which is also responsible for the manipulation of commitments (cf. Section 3.6). The data maintained by such an agent are publicly available and can be queried by the other agents, but only modified by the institutional agent itself. Specifically, the institutional agent maintains data about the following relations:

- Agent denotes the set of (names of) agents that currently participate to the system. Since the institutional agent is always part of the system, we fix its name as inst, and enforce that inst always belongs to the extension of Agent.
- Spec, whose extension is immutable, denotes the set of agent specification names mentioned in $\mathcal{X}$ (cf. Section 3.2).
- hasSpec connects agents with their current specification(s): $\mathsf{hasSpec}[1] \sqsubseteq \mathsf{Agent}$, $\mathsf{hasSpec}[2] \sqsubseteq \mathsf{Spec}$.

Each agent, including the institutional agent, maintains a proprietary *DLR-Lite* ABox, in which it stores its own data. Such data can be queried only by the agent itself and by the institutional agent, which exploits the results of such queries to keep track of the evolution of commitments. Furthermore, each agent progresses its own ABox during the execution in such a way that it is always consistent with the global TBox $\mathcal{T}$. Notice that the overall collection of ABoxes is not assumed to be consistent with the TBox, i.e., the TBox assertions are only required to be satisfied by each agent individually.

Since, in general, queries may involve the ABoxes of several agents, to disambiguate to which ABox a query atom refers, we augment the vocabulary of the global TBox with a *location argument* that points to an agent. We use $\mathsf{R@a}(\vec{x})$ to denote an atomic query returning the extension of R in the ABox of agent a. If a does not point to (the name of) an agent currently in the system, then $\mathsf{R@a}(\vec{x})$ evaluates to empty. Beside the special constant inst, we also use self to implicitly refer to the agent that is posing the query (similarly to "this" in object-oriented programming). When clear from the context, we omit @self and just use relations without the location argument. We denote with UCQ_ℓ (resp., ECQ_ℓ) the language obtained from UCQ (resp., ECQ) by extending atoms with a location argument.

3.2 Agent Specifications

In a DACMAS, agents interact by exchanging messages. A message is sent by a sender agent to a receiver agent, and is about the occurrence of an *event* with a *payload*, containing data to be communicated. All agents but the institutional one are only aware of the events they send and receive. As for data, the institutional agent has instead full visibility of all exchanged messages, so as to properly handle the evolution of commitments.

Agents determine the events they may send, and also how they react to events, through proactive and reactive rules. Such rules are grouped into behavioural profiles called *agent specifications*, and model: *(i)* the possible, proactive emission of an event, directed to another agent (*communicative rule*); *(ii)* conditional internal (re)actions, which lead to update the agent ABox when sending/receiving an event to/from another agent (*update rule*). The update could result in the insertion of new data items (from the countably infinite domain Δ), not already present in the system.

The exchange of a message represents a synchronization point among the sender, receiver and institutional agent. Hence, the reaction of the three agents is interpreted as a sort of transaction, such that each of them effectively enforces the update on its own ABox only if each of the three resulting ABoxes is consistent with $\mathcal{T}$. An inconsistency could potentially arise when reacting to an event either because the same data item is asserted to be member of two disjoint classes, or because a key assertion is violated.

Formally, an agent specification is a tuple $\langle \mathsf{sn}, \Pi \rangle$, where sn is the specification name, and Π is a set of *communicative* and *update rules*. Such rules are defined over the vocabulary of $\mathcal{T}$ and $\mathcal{B}$, and are applied over the ABoxes of the agent and of inst. This allows the agent to query the status of commitments and obtain the names of the other participants.

A *communicative rule* has the form

$$Q(r, \vec{x}) \textbf{ enables } \textsc{ev}(\vec{x}) \textbf{ to } r$$

where Q is an ECQ_ℓ, and $\textsc{ev}(\vec{x})$ is an event supported by the system, i.e., predicate $\textsc{ev}/|\vec{x}|$ belongs to $\mathcal{E}$. The semantics of a communicative rule is as follows. Whenever $Q(r, \vec{x})$ evaluates positively, the agent autonomously selects one of the answers θ returned by the query, using it to determine the event receiver and its payload. This states that the ground event $\textsc{ev}(\vec{x})\theta$ *can be* sent by the agent to $r\theta$, provided that $r\theta$ indeed points to an actual agent name in the system (including the two special names inst and self).

Example 3.1. Consider a DACMAS where customers and sellers interact to exchange goods. We model the behavioural rules for customers and sellers using two agent specifications. To buy from a seller, a customer must register to that seller. A registration request is modeled in the customer specification as:

$$\mathsf{Spec@inst}(sel, \mathsf{seller}) \textbf{ enables } \textsc{req\_reg} \textbf{ to } sel$$

Assuming that each seller maintains its customers and items respectively in relations MyCust and Item, the proposal of an item to a customer is modeled in the seller specification as:

$$\mathsf{MyCust}(m) \wedge \mathsf{Item}(i) \textbf{ enables } \textsc{propose}(i) \textbf{ to } m \qquad \blacksquare$$

Update rules are ECA-like rules of the form:

- **on** $\textsc{ev}(\vec{x})$ **to** r **if** $Q(r, \vec{x})$ **then** $\alpha(r, \vec{x})$ (on-send)
- **on** $\textsc{ev}(\vec{x})$ **from** s **if** $Q(s, \vec{x})$ **then** $\alpha(s, \vec{x})$ (on-receive)

where $\textsc{ev}/|\vec{x}|$ is an event type from $\mathcal{E}$, Q is an ECQ_ℓ, and α is an update action with parameters (described below). Each such rule triggers when an event is sent to/received from another agent, and Q holds. This results in the application of α using the actual event payload and receiver/sender. Action α queries the ABox of the agent and of inst, using the answers to add and remove facts to the ABox.

Formally, an *update action* is an expression $\alpha(\vec{p}) : \{e_1, \ldots, e_n\}$, where $\alpha(\vec{p})$ is the action signature (constituted by the name α and by a list $\vec{p}$ of parameters), and $\{e_1, \ldots, e_n\}$ are update effects, each of which has the form

$$[q^+(\vec{p}, \vec{x})] \wedge Q^-(\vec{p}, \vec{x}) \rightsquigarrow \textbf{add } A, \textbf{del } D$$

- q^+ is an UCQ_ℓ, and Q^- is an ECQ_ℓ whose free variables occur all among those of q^+; intuitively, q^+ selects a set of tuples from the agent ABox and that of inst, while Q^- filters away some of them.[1] During the execution, the effect is applied with a ground substitution $\vec{\mathsf{d}}$ for the action parameters, and for every answer θ to the query $[q^+(\vec{\mathsf{d}}, \vec{x})] \wedge Q^-(\vec{\mathsf{d}}, \vec{x})$.
- A is a set of facts (over the alphabet of $\mathcal{T}$ and $\mathcal{B}$) which include as terms: free variables $\vec{x}$ of q^+, action parameters $\vec{p}$ and/or Skolem terms $f(\vec{x}, \vec{p})$. We use $\textsc{skolem}(A)$ to denote all Skolem terms mentioned in A. At runtime, whenever a ground Skolem term is produced by applying θ to A, the agent autonomously substitutes it with a possibly new data item taken from Δ. This mechanism is exploited by the agent to inject new data into the system. The ground set of facts so obtained is *added* by the agent to its ABox.
- D is a set of facts which include as terms free variables $\vec{x}$ of q^+ and action parameters $\vec{p}$. At runtime, whenever a ground fact in D is obtained by applying θ, it is *removed* from the agent ABox.

As in STRIPS, we assume that additions have higher priority than deletions (i.e., if the same fact is asserted to be added and deleted during the same execution step, then the fact is added). The "**add** A" part (resp., the "**del** D" part) can be omitted if $A = \emptyset$ (resp., if $D = \emptyset$).

Example 3.2. Consider three possible reaction rules for the seller. The fact that the seller makes every agent that sends a request become one of its customers is modeled as:

$$\textbf{on } \textsc{ask\_reg} \textbf{ from } c \textbf{ if } true \textbf{ then } makeCust(c)$$

where $makeCust(x) : \{[true] \rightsquigarrow \textbf{add}\{\mathsf{MyCust}(x)\}\}$.

Assume now that the seller maintains the item cart for a customer, using relation $\mathsf{InCart(i, c)}$ to model that item i is in the cart of c. The seller reaction to an "empty cart" request is modeled as:

$$\textbf{on } \textsc{empty\_cart\_req} \textbf{ from } c \textbf{ if } \mathsf{MyCust}(c) \textbf{ then } doEmpty(c)$$

[1] The distinction between q^+ and Q^- is needed for technical reasons, borrowed from [2].

where $doEmtpy(c) : \{[\mathsf{InCart}(i,c)] \rightsquigarrow \textbf{del}\{\mathsf{InCart}(i,c)\}\}$. Note that the effect is applied to each i in the cart of c.

Consider now the case where the seller receives a new item i to be sold. It reacts by adding i and deciding its price. This is modeled with a Skolem term $p(i)$:

$$\textbf{on } \text{NEW\_ITEM}(i) \textbf{ from } a \textbf{ if } true \textbf{ then } addItem(i)$$

where $addItem(i) : \{[true] \rightsquigarrow \textbf{add}\{\mathsf{Item}(i), \mathsf{Price}(i, p(i))\}\}$. ■

3.3 Institutional Agent Specification

The institutional agent inst manages the core information of the DACMAS. Its behaviour is (partially) captured by the institutional agent specification $\mathcal{I}$, which differs from the other agent specifications in two respects.

First, since inst is aware of all messages exchanged by the other agents and can query their ABoxes, its specification is not only constituted by communicative rules and on-send/on-receive reactive rules, but also by *on-exchange* rules of the form:

$$\textbf{on } \text{EV}(\vec{x}) \textbf{ from } s \textbf{ to } r \textbf{ if } Q(s,r,\vec{x}) \textbf{ then } \alpha(s,r,\vec{x})$$

where Q and α can query the internal ABox of the institutional agent, and the ABoxes of s and r. To conveniently specify reactions of inst that do not depend on a specific event, but trigger whenever an event is exchanged, we use:

$$\textbf{on any event from } s \textbf{ to } r \textbf{ if } Q(s,r) \textbf{ then } \alpha(s,r)$$

Second, $\mathcal{I}$ is only a *partial* specification for inst. In fact, inst is also responsible for the manipulation of commitments, which results in a set of additional on-exchange rules that, starting from the contractual specification (cf. Section 3.4), encode the commitment machines for the commitments involved in the contract. These rules are automatically extracted from the contractual specification (cf. Section 3.6).

EXAMPLE 3.3. Consider a portion of institutional agent specification, modeling the creation of a new agent whenever inst receives a request (whose payload denotes the specification to be initially followed by that agent).

To handle this request, inst uses relation NewA to store a newly created agent together with is initial specification. The axiom $\mathsf{NewA}[1] \sqsubseteq \neg\mathsf{Agent}$ is part of $\mathcal{T}$, and enforces that a new agent has indeed a new name.

The behaviour is defined in two steps. In the first step, inst reacts to a creation request by choosing an agent name (using Skolem term $n()$). The reaction is applied only if there is no pending new agent to be processed.

$$\textbf{on } \text{AG\_REQ}(s) \textbf{ from } a \textbf{ if } \neg(\exists x \exists y.\mathsf{NewA}(x,y)) \textbf{ then } create(s)$$
$$create(s) : \{\ [true] \rightsquigarrow \textbf{add}\{\mathsf{NewA}(n(), s))\}\ \}$$

Note that axiom $\mathsf{NewA}[1] \sqsubseteq \neg\mathsf{Agent}$ ensures that the update is blocked if the chosen name is already used in the system.

In the second step, inst informs itself that a new agent has to be processed:

$$\mathsf{NewA}(a,s) \textbf{ enables } \text{INSERT\_AG}(a,s) \textbf{ to } \mathsf{self}$$

The corresponding reaction finalizes the insertion of the new agent, moving it to the set of participating agents:

$$\textbf{on } \text{INSERT\_AG}(a,s) \textbf{ from } \mathsf{self} \textbf{ if } true \textbf{ then } do\_ins(a,s)$$
$$do\_ins(a,s) : \{\ [true] \rightsquigarrow \textbf{add}\{\mathsf{Agent}(a), \mathsf{Spec}(a,s)\}, \textbf{del}\{\mathsf{NewA}(a,s)\}\ \}$$ ■

3.4 Contractual Specification

The contractual specification $\mathcal{C}$ consists of a set of *commitment rules*, which are reactive rules similar to on-exchange rules. The main difference is that, instead of actions, they describe (first-order) conditional commitments and their creation. Technically, a commitment rule has the form:

$$\textbf{on } \text{EV}(\vec{x}) \textbf{ from } s \textbf{ to } r \textbf{ if } Q_c(s,r,\vec{x}) \qquad (*)$$
$$\textbf{then } \text{CC}_{\mathsf{n}}(s,r,[q_p^+(s,r,\vec{x},\vec{y})] \wedge Q_p^-(s,r,\vec{x},\vec{y}), Q_d(s,r,\vec{x},\vec{y}))$$

where n is the commitment name, the ECQ_ℓ Q_c is the condition for the creation of the conditional commitment, $[q_p]^+ \wedge Q_p^-$ (where, as in update effects, q_p^+ is a UCQ_ℓ, and Q_p^- is an ECQ_ℓ whose free variables all occur among those of q_p^+) is the *precondition* determining the generation of a corresponding base-level commitment, and the ECQ_ℓ Q_d is the *discharge condition* for such base-level commitment. All the aforementioned queries can be posed over the ABoxes of s, r, and inst. We use GET-CC($\mathcal{C}$) to extract the set of conditional commitments contained in $\mathcal{C}$.

According to the literature, commitments are manipulated either explicitly via specific events (such as a commitment cancellation or delegation), or implicitly when the commitment precondition or discharge condition becomes true. The allowed commitment manipulations, together with the resulting commitment states, are captured by means of a *commitment machine* [11]. In this work, we consider a simple commitment machine, inspired by [11, 12], and show how to lift it to a first-order setting, taking into account that in our framework the precondition and the discharge condition are specified through queries over the data of the involved agents. More elaborated commitment machines, in terms of events and states, can be seamlessly incorporated.

Specifically, every commitment in GET-CC($\mathcal{C}$) is associated to a specific first-order commitment machine, which is activated using the corresponding commitment rule in $\mathcal{C}$ of the form above, instantiated possibly multiple times, depending on the agent data. The machine evolves as follows:

1. When an event of type EV is sent by agent a to agent b with payload $\vec{\mathsf{d}}$, if $Q_c(\mathsf{a},\mathsf{b},\vec{\mathsf{d}})$ is satisfied, an *instance of the conditional commitment* n is created. The debtor, creditor, and payload of this instance are respectively a, b, and $\vec{\mathsf{d}}$.
2. Such instance is explicitly or implicitly manipulated by the involved agents. Explicit manipulation is done via specific message exchanges; we consider in particular the case of delegation from the debtor a to a new debtor, and the case of cancellation. Implicitly, instead, the instance can generate one or more corresponding base-level commitment instances: whenever $[q_p^+(\mathsf{a},\mathsf{b},\vec{\mathsf{d}},\vec{\mathsf{v}})] \wedge Q_p^-(\mathsf{a},\mathsf{b},\vec{\mathsf{d}},\vec{\mathsf{v}})$ is satisfied with actual values $\vec{\mathsf{v}}$ for variables $\vec{y}$, the conditional commitment instance *creates a base-level commitment instance* with payload $\vec{\mathsf{d}}$ and $\vec{\mathsf{v}}$. Such base-level instance is put into the *active* state. The discharge condition for this instance is the instantiation of Q_d with the involved agents and specific payload, i.e., a is committed to bring about $Q_d(\mathsf{a},\mathsf{b},\vec{\mathsf{d}},\vec{\mathsf{v}})$.
3. Also a base-level commitment instance is explicitly and implicitly manipulated by the involved agents. Explicit manipulation of an active base-level instance resembles that of conditional commitment instances, with the difference that, when canceled, a base-level commitment instance enters into the *violated* state. Implicit manipulation determines instead the *discharge* of the instance as soon as $Q_d(\mathsf{a},\mathsf{b},\vec{\mathsf{d}},\vec{\mathsf{v}})$ holds, moving the instance from active to *satisfied.*

EXAMPLE 3.4. Consider a commitment rule establishing a conditional commitment that the seller takes whenever it accepts the registration of a customer c. The conditional commitment is about the delivery of items paid by c. Specifically, for each item sold by the seller, if c has paid that item, then the seller commits to ensure that c will hold *that* item. Note that the two conditions are *correlated* by the same item, and that a base-level commitment is created for each paid item. This cannot be expressed in propositional logic. Assuming that the seller stores a fact $\mathsf{Paid}(i, c)$ if c has paid for i, and that the customer stores a fact $\mathsf{Owned}(i)$ whenever it owns i, the commitment rule can be specified as:

on ACCEPT_REG **from** s **to** c **if** $\mathsf{MyCust}@s(c)$
then $\textsc{cc}_{\mathsf{Delivery}}(s, c, [\mathsf{Item}@s(i) \wedge \mathsf{Paid}@s(i, c)], \mathsf{Owns}@c(i))$

Note the use of location arguments, reflecting that payments are maintained by the seller, whereas the items owned by the customer are maintained by the customer itself. ■

3.5 Commitment Box

The commitment box $\mathcal{B}$ is a set of relations used by inst to maintain the concrete instances of conditional commitments, and the instances of their corresponding base-level commitments (with their states). In fact, due to the presence of data, commitments do not only require to keep track of the involved agents, but also of the payload associated to each of their instances. Such relations are extracted from the contractual specification as follows. Each commitment $\textsc{cc}_{\mathsf{n}}(s, r, [q_p^+(s, r, \vec{x}, \vec{y})] \wedge Q_p^-(s, r, \vec{x}, \vec{y}), Q_d(s, r, \vec{x}, \vec{y}))$ in GET-CC($\mathcal{C}$) induces two relations in $\mathcal{B}$, on the basis of the commitment name n and the payloads $\vec{x}$ and $\vec{y}$: *(i)* nCC/ar, where $ar = 2 + |\vec{x}|$ for debtor, creditor, and conditional commitment payload; *(ii)* nC/ar, where $ar = 3 + |\vec{x}| + |\vec{y}|$ for debtor, creditor, state, and base-level commitment payload.

EXAMPLE 3.5. The commitment in Example 3.4 induces the following relations in $\mathcal{B}$: $\mathsf{DeliveryCC}(\mathsf{debtor}, \mathsf{creditor})$ and $\mathsf{DeliveryC}(\mathsf{debtor}, \mathsf{creditor}, \mathsf{state}, \mathsf{item})$. ■

3.6 Commitment Machine Formalization

As anticipated in Section 3.3, the specification of inst must be complemented with a set of additional on-exchange rules, used to properly manipulate the evolution of commitments as the interaction unfolds. Commitment instances are stored by inst using the vocabulary of the CBox $\mathcal{B}$, and evolved through the application of these rules. Specifically, these rules ground the (first-order) commitment machine described in Section 3.4 to each specific commitment of GET-CC($\mathcal{C}$), according to the "templates" described in the remainder of this section. We denote with CC-RULES($\mathcal{C}$) all the commitment manipulation rules produced from $\mathcal{C}$.

When discussing the templates, we refer to a commitment rule $\rho \in \mathcal{C}$ of the form ($*$) in Section 3.4. Notice that, when n, $\vec{x}$ and $\vec{y}$ are mentioned in the rule templates, they are meant to be replaced with the actual commitment name and payload variables.

CC creation. For each $\rho \in \mathcal{C}$, a corresponding creation rule is obtained, depending on n and $\vec{x}$. When the rule triggers, a new instance of the conditional commitment nCC is created, with the actual agents and payload:

on EV($\vec{x}$) **from** s **to** r **if** $Q_c(s, r, \vec{x})$ **then** $create\_\mathsf{n}CC(s, r, \vec{x})$
$create\_\mathsf{n}CC(s, r, \vec{x})$: $\{\ [true] \rightsquigarrow \textbf{add}\{\mathsf{nCC}(s, r, \vec{x})\}\ \}$

CC delegation. The delegation of a conditional commitment instance for commitment n is triggered when the old debtor d_o sends to the new debtor d_n a DELEGATE_nCC event, specifying in the event payload the creditor and the payload of the instance to be delegated. If such an instance exists, the debtor is updated by inst:

on DELEGATE_nCC($c, \vec{x}$) **from** d_o **to** d_n
if $\mathsf{nCC}(d_o, c, \vec{x})$ **then** $changedeb\_\mathsf{n}CC(d_o, d_n, c, \vec{x})$

$changedeb\_\mathsf{n}CC(d_o, d_n, c, \vec{x})$: $\{\ [true] \rightsquigarrow \textbf{add}\{\mathsf{nCC}(d_n, c, \vec{x})\},$
$\textbf{del}\{\mathsf{nCC}(d_o, c, \vec{x})\}\}$

CC cancelation. The cancelation of a conditional commitment instance for commitment n is triggered when the debtor sends to the creditor a CANCEL_nCC event, providing the instance payload. If the instance exists, it is removed:

on CANCEL_CC($\vec{x}$) **from** d **to** c
if $\mathsf{nCC}(d, c, \vec{x})$ **then** $delete\_\mathsf{n}CC(d, c, \vec{x})$
$delete\_\mathsf{n}CC(d, c, \vec{x})$: $\{\ [true] \rightsquigarrow \textbf{del}\{\mathsf{nCC}(d, c, \vec{x})\}\ \}$

C creation. Every conditional commitment instance for relation nCC creates a base-level commitment instance whenever the precondition (whose variables $\vec{x}$ are grounded with the instance payload) holds with an answer substitution θ for variables $\vec{y}$. This results in the creation of a new tuple for relation nC with the actual, full payload. This does not depend on the specific exchanged event, but only on the actual configuration of the data. Hence, a single "any-event" rule can be used to manage the creation of all base-level instances at once:

on any event from d **to** c **if** $true$ **then** $createC(d, c)$

where, for each commitment $\textsc{cc}_{\mathsf{n}}(s, r, [q_p^+(s, r, \vec{x}, \vec{y})] \wedge Q_p^-(s, r, \vec{x}, \vec{y}), Q_d(s, r, \vec{x}, \vec{y}))$ in GET-CC($\mathcal{C}$), action $createC(d, c)$ contains the following detachment effect:

$[\mathsf{nCC}(d, c, \vec{x}) \wedge q_p^+(d, c, \vec{x}, \vec{y})] \wedge Q_p^-(d, c, \vec{x}, \vec{y})$
$\rightsquigarrow \textbf{add}\ \{\mathsf{nC}(d, c, \mathsf{active}, \vec{x}, \vec{y})\}$

Differently from the propositional formalization of a commitment machine, in which the conditional commitment detaches to a base-level one, in our setting the conditional commitment instance is maintained, and keeps waiting for other situations matching the precondition with different data.

C delegation. It resembles the CC delegation:

on DELEGATE_nC($c, \vec{x}, \vec{y}$) **from** d_o **to** d_n
if $\mathsf{nC}(d_o, c, \mathsf{active}, \vec{x}, \vec{y})$ **then** $changedeb\_\mathsf{n}C(d_o, d_n, c, \vec{x})$
$changedeb\_\mathsf{n}C(d_o, d_n, c, \vec{x}, \vec{y})$:
$\{[true] \rightsquigarrow \textbf{add}\{\mathsf{nC}(d_n, c, \mathsf{active}, \vec{x}, \vec{y})\}, \textbf{del}\{\mathsf{nC}(d_o, c, \mathsf{active}, \vec{x}, \vec{y})\}\}$

C cancelation. It determines a transition for the base-level commitment instance from the active to the violated state:

on CANCEL_C($\vec{x}, \vec{y}$) **from** d **to** c
if $\mathsf{nC}(d, c, \vec{x}, \vec{y})$ **then** $viol\_\mathsf{n}C(d, c, \vec{x}, \vec{y})$
$viol\_\mathsf{n}C(d, c, \vec{x}, \vec{y})$:
$\{[true] \rightsquigarrow \textbf{add}\{\mathsf{nC}(d, c, \mathsf{viol}, \vec{x}, \vec{y})\}, \textbf{del}\{\mathsf{nC}(d, c, \mathsf{active}, \vec{x}, \vec{y})\}\}$

C discharge. Similarly to the case of C creation, the discharge of base-level commitment instances is handled by a single "any-event" rule, which checks the discharge condition for each active commitment instance with the actual payload, evolving the instance to the satisfied state if it holds:

on any event from d **to** c **if** $true$ **then** $dischargeC(d, c)$

where, for each $\textsc{cc}_{\mathsf{n}}(s, r, [q_p^+(s, r, \vec{x}, \vec{y})] \wedge Q_p^-(s, r, \vec{x}, \vec{y}), Q_d(s, r, \vec{x}, \vec{y}))$ in CC($\mathcal{C}$), action $dischargeC(d, c)$ contains:

$[\mathsf{nC}(d, c, \mathsf{active}, \vec{x}, \vec{y})] \wedge Q_d(d, c, \vec{x}, \vec{y})$
$\rightsquigarrow \textbf{add}\{\mathsf{nC}(d, c, \mathsf{sat}, \vec{x}, \vec{y})\},\ \textbf{del}\{\mathsf{nC}(d, c, \mathsf{active}, \vec{x}, \vec{y})\}$

C removal. A last "any-event" reactive rule is used by inst to remove those instances of base-level commitments that already achieved a final state (sat or viol):

$$\textbf{on any event from } a \textbf{ to } b \textbf{ if } true \textbf{ then } removeFinal()$$

where, for each base-level commitment relation nC in $\mathcal{B}$, action $removeFinal()$ contains:

$$[\mathsf{nC}(d, c, s, \vec{x}, \vec{y})] \wedge (s = \mathsf{sat} \vee s = \mathsf{viol}) \rightsquigarrow \textbf{del}\{\mathsf{nC}(d, c, s, \vec{x}, \vec{y})\}$$

EXAMPLE 3.6. Assume that the only rule in $\mathcal{C}$ is that of Example 3.4. The following CC creation rule is produced

on ACCEPT_REG **from** s **to** c **if** $\mathsf{MyCust}@s(c)$
then $create\_DeliveryCC(s, c)$

$create\_DeliveryCC(s, c) : \{[true] \rightsquigarrow \textbf{add}\{\mathsf{DeliveryCC}(s, c)\}\}$

Furthermore, the following C creation and C discharge update actions are produced:

$createC(d, c) : \{[\mathsf{DeliveryCC}(d, c) \wedge \mathsf{Item}@d(i) \wedge \mathsf{Paid}@d(i, c)]$
$\rightsquigarrow \textbf{add}\{\mathsf{DeliveryC}(d, c, \mathsf{active}, i)\}\}$
$dischargeC(d, c) : \{[\mathsf{DeliveryC}(d, c, \mathsf{active}, i)] \wedge \mathsf{Owns}@c(i)$
$\rightsquigarrow \textbf{add}\{\mathsf{DeliveryC}(d, c, \mathsf{sat}, i)\}, \textbf{del}\{\mathsf{DeliveryC}(d, c, \mathsf{active}, i)\}\}$

4. EXECUTION SEMANTICS

The execution semantics of a DACMAS is defined in terms of a *transition systems* that, starting from a given initial state, accounts for all the possible system dynamics, considering in particular all the (possibly infinite) sequences of message exchanges, and all the possible substitutions that the agents choose during the application of update actions to provide concrete values for the Skolem terms.

Given a DACMAS $\mathcal{S} = \langle \mathcal{T}, \mathcal{E}, \mathcal{X}, \mathcal{I}, \mathcal{C}, \mathcal{B} \rangle$ and an initial state σ_0, the execution semantics of $\mathcal{S}$ over σ_0 is defined by a transition system $\Upsilon_{\mathcal{S}}^{\sigma_0} = \langle \Delta, \mathcal{T} \cup \mathcal{B}, \Sigma, \sigma_0, \Rightarrow \rangle$, where:

- Σ is a (possibly infinite) set of states. Each state $\sigma \in \Sigma$ is equipped with a function $abox$ that, given the name a of an agent, returns the ABox $\sigma.abox(\mathsf{a})$ of a in σ, if and only if a participates to the system in state σ. Specifically, $\sigma.abox(\mathsf{inst})$ is always defined, and for other agent names a, $\sigma.abox(\mathsf{a})$ is defined if and only if a belongs to the extension Agent in $\sigma.abox(\mathsf{inst})$.
- $\sigma_0 \in \Sigma$ is the initial state. We assume that every ABox $\mathcal{A}$ in σ_0 is such that $(\mathcal{T}, \mathcal{A})$ is satisfiable, and that $\mathsf{Spec}(sn) \in \sigma_0.abox(\mathsf{inst})$ if and only if $\langle sn, \_ \rangle \in \mathcal{X}$.
- $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation.

Instrumental to the definition of the transition system is the extension of answering ECQ_ℓ queries so as to take into account location arguments. Formally, given a TBox $\mathcal{T}$, we define that $R@b(\vec{x})$ *holds in state* σ *from the perspective of agent* a *under substitution* θ for $\vec{x}$, written $\mathcal{T}, \sigma, \mathsf{a}, \theta \models R@b(\vec{x})$, if:[2]

$$\begin{cases} \sigma.abox(\mathsf{a}) \text{ is defined and } (\mathcal{T}, \sigma.abox(\mathsf{a})) \models R(\vec{x})\theta, & \text{if } b\theta = \mathsf{self} \\ \sigma.abox(b\theta) \text{ is defined and } (\mathcal{T}, \sigma.abox(b\theta)) \models R(\vec{x})\theta, & \text{if } b\theta \neq \mathsf{self} \end{cases}$$

Note that the semantics supports a sort of *dynamic binding* of location arguments, using θ to substitute a variable location argument with an agent name. This relation extends in the natural way to UCQ_ℓ and ECQ_ℓ, considering that quantification ranges over the active domain ADOM(σ) of σ, which is defined as the union of the active domains of the ABoxes maintained by the agents present in σ. This, in turn, allows us to define the *certain answers to* Q *obtained by agent* a *in state* σ, denoted $\text{ANS}_\ell(Q, \mathcal{T}, \sigma, \mathsf{a})$, as the set of substitutions θ for the free variables in Q such that Q holds in state σ from the perspective of a, i.e.,

$$\text{ANS}_\ell(Q, \mathcal{T}, \sigma, \mathsf{a}) = \{\theta \mid \mathcal{T}, \sigma, \mathsf{a}, \theta \models Q\}.$$

```
procedure BUILD-TS
input: DACMAS $\mathcal{S} = \langle \mathcal{T}, \mathcal{E}, \mathcal{X}, \mathcal{I}, \mathcal{C}, \mathcal{B} \rangle$ and initial state $\sigma_0$
output: Transition system $\langle \Delta, \mathcal{T} \cup \mathcal{B}, \Sigma, \sigma_0, \Rightarrow \rangle$
  $\Sigma := \{\sigma_0\}, \Rightarrow := \emptyset$
  while true do
    pick $\sigma \in \Sigma$ and $\mathsf{a} \in \{\mathsf{ag} \mid \mathsf{Agent}(\mathsf{ag}) \in \sigma.abox(\mathsf{inst})\}$
    Fetch all current behavioural rules for $\mathsf{a}$
    Calculate the enabled events for $\mathsf{a}$ and their receiver
    if There exists at least an enabled event then
      pick an enabled event $\text{EV}(\vec{\mathsf{e}})$ for $\mathsf{a}$ with receiver $\mathsf{b}$
      $\mathcal{A}_i := \text{APPLY}(\mathcal{S}, \sigma, \mathsf{inst}, \mathsf{a}, \mathsf{b}, \text{EV}(\vec{\mathsf{e}}))$        ▷ New inst ABox
      $\Sigma := \Sigma \cup \{\sigma'\}$        ▷ Tentatively add a new state $\sigma'$
      for all $\mathsf{x} \in \{\mathsf{ag} \mid \mathsf{Agent}(\mathsf{ag}) \in \mathcal{A}_i\}$ do
        $\sigma'.abox(\mathsf{x}) := \text{APPLY}(\mathcal{S}, \sigma, \mathsf{x}, \mathsf{a}, \mathsf{b}, \text{EV}(\vec{\mathsf{e}}))$
      if for every $\mathsf{x} \in \{\mathsf{ag} \mid \mathsf{Agent}(\mathsf{ag}) \in \mathcal{A}_i\}$, $\langle \mathcal{T}, \sigma'.abox(\mathsf{x}) \rangle$ is satisfiable then $\Rightarrow := \Rightarrow \cup \langle \sigma, \sigma' \rangle$
      else $\Sigma := \Sigma \setminus \{\sigma'\}$        ▷ Inconsistent execution step
function APPLY($\mathcal{S}, \sigma, x, a, b, \text{EV}(\vec{e})$)
output: new ABox for $x$ after reacting to $\text{EV}(\vec{e})$ from $a$ to $b$
  if $x \notin \{\mathsf{inst}, a, b\}$ then return $\sigma.abox(x)$
  Fetch all current behavioural rules for $x$
  if $x = a$ then        ▷ $x$ is the sender agent
    Fetch all on-send and "self" on-receive rules and compute actions with actual params
  if $x = b$ then        ▷ $x$ is the receiver agent
    Fetch all on-receive and "self" on-send rules and compute actions with actual param
  if $x = \mathsf{inst}$ then        ▷ $x$ is the institutional agent
    Fetch all matching and "any-event" on-exchange rules and compute actions with actual param
  $TOADD := \emptyset$, $TODEL := \emptyset$
  for all $\alpha(\vec{\mathsf{v}}) \in ACT$ do        ▷ $ACT$ = set of fetched actions
    $TOADDSK := \emptyset$
    for all effect "$[q^+(\vec{p}, \vec{x})] \wedge Q^-(\vec{p}, \vec{x}) \rightsquigarrow$ add $A$, del $D$" in the definition of $\alpha$ do
      for all $\theta \in \text{ANS}_\ell([q^+(\vec{\mathsf{v}}, \vec{x})] \wedge Q^-(\vec{\mathsf{v}}, \vec{x}), \mathcal{T}, \sigma, x)$ do
        $TOADDSK := TOADDSK \cup A\theta[\vec{p}/\vec{\mathsf{v}}]$
        $TODEL := TODEL \cup D\theta[\vec{p}/\vec{\mathsf{v}}]$
    pick a substitution $\theta_{sk}$ of the Skolem terms with data
    $TOADD := TOADD \cup TOADDSK\theta_{sk}$
  if $x = \mathsf{inst}$ then $TOADD := TOADD \cup \{\mathsf{Agent}(\mathsf{inst})\}$
  return $(\sigma.abox(x) \setminus TODEL) \cup TOADD$
```

Figure 1: Transition system construction

The construction of the transition system $\Upsilon_{\mathcal{S}}^{\sigma_0}$ is given in Figure 1. From a given state, a successor state is nondeterministically computed as follows: *(i)* a sender agent is picked from the set of active agents; *(ii)* the communicative rules of the sender agent are evaluated, determining the set of enabled (ground) events with destinations; *(iii)* an enabled event with destination is picked from such set; *(iv)* the sender, destination and institutional agents react to the picked event by evaluating their update rules, and by executing the corresponding actions. In general, the resulting transition system is *infinite-branching*, due to the agent choices when injecting new data into their ABoxes, and contains *infinite runs*, due to the possibly infinitely many different facts stored in such ABoxes over time.

5. VERIFICATION OF DACMAS

To specify dynamic properties over DACMASs, we use a first-order variant of μ-calculus [13, 6]. μ-calculus is virtually the most powerful temporal logic used for model checking of finite-state transition systems, and is able to express both linear time logics

[2] We assume that θ is the identity on data items (including the special constants self and inst).

such as LTL and PSL, and branching time logics such as CTL and CTL* [9]. Technically, μ-calculus separates local properties, asserted on the current state or on states that are immediate successors of the current one, from properties talking about states that are arbitrarily far away from the current one [13]. The latter are expressed through the use of fixpoints.

In our variant of μ-calculus, local properties are expressed as ECQ_ℓ queries over the current state of the DACMAS. At the same time we allow for a controlled form of first-order quantification across states, inspired by [2, 6], where the quantification ranges over data items across time only as long as such items persist in the active domain. Formally, we define the logic $\mu\mathcal{L}_p^{ECQ_\ell}$ as:

$$\begin{aligned}\Phi ::= {} & Q_\ell \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\textsc{live}(x) \wedge \Phi \mid \\ & \textsc{live}(\vec{x}) \wedge \langle - \rangle\Phi \mid \textsc{live}(\vec{x}) \wedge [-]\Phi \mid Z \mid \mu Z.\Phi\end{aligned}$$

where Q is a (possibly open) ECQ_ℓ query, in which the only constants that may appear are those in the initial state of the system, Z is a second order predicate variable (of arity 0), and $\textsc{live}(x_1, \ldots, x_n)$ abbreviates $\bigwedge_{i \in \{1,\ldots,n\}} \textsc{live}(x_i)$. For $\mu\mathcal{L}_p^{ECQ_\ell}$, the following assumption holds: in $\textsc{live}(\vec{x}) \wedge \langle - \rangle\Phi$ and $\textsc{live}(\vec{x}) \wedge [-]\Phi$, the variables $\vec{x}$ are exactly the free variables of Φ, once we substitute to each bounded predicate variable Z in Φ its bounding formula $\mu Z.\Phi'$. We adopt the usual abbreviations, including $\nu Z.\Phi$ for greatest fixpoints.

Intuitively, the use of $\textsc{live}(\cdot)$ in $\mu\mathcal{L}_p^{ECQ_\ell}$ ensures that data items are only considered if they persist along the system evolution, while the evaluation of a formula with data that are not present in the current state trivially leads to false or true. This is in line with DACMASs, where the evolution of a commitment instance persists until the commitment is discharged or canceled, and where an agent name is meaningful only while it persists in the system: when an agent leaves the system and its name a is canceled by inst, inst could reuse a in the future to identify another agent.

The formula $\mu Z.\Phi$ denotes the least fixpoint of the formula Φ. As usual in μ-calculus, formulae of the form $\mu Z.\Phi$ must obey to the *syntactic monotonicity* of Φ w.r.t. Z, which states that every occurrence of the variable Z in Φ must be within the scope of an even number of negation symbols. This ensures that the least fixpoint $\mu Z.\Phi$ always exists.

The semantics of $\mu\mathcal{L}_p^{ECQ_\ell}$ formulae is defined over a possibly infinite transition system $\Upsilon = \langle \Delta, \mathcal{T} \cup \mathcal{B}, \Sigma, \sigma_0, \Rightarrow \rangle$ (cf. Section 4), assuming that ECQ_ℓ queries are posed from the point of view of inst. This does not prevent the possibility to query the ABoxes of the other agents, thanks to the dynamic binding for location arguments. Since $\mu\mathcal{L}_p^{ECQ_\ell}$ contains formulae with both individual and predicate free variables, we introduce an *individual variable valuation* v, i.e., a mapping from individual variables x to Δ, and a *predicate variable valuation* V, i.e., a mapping from the predicate variables Z to subsets of Σ. With these three notions in place, we assign meaning to formulae by associating to Υ, v, and V an *extension function* $(\cdot)_{v,V}^{\Upsilon}$, which maps formulae to subsets of Σ. Formally, the extension function $(\cdot)_{v,V}^{\Upsilon}$ is defined inductively as shown in Figure 2.

When Φ is a closed formula, $(\Phi)_{v,V}^{\Upsilon}$ does not depend on v or V, and we denote the extension of Φ simply by $(\Phi)^{\Upsilon}$. A closed formula Φ holds in a state $s \in \Sigma$ if $s \in (\Phi)^{\Upsilon}$. In this case, we write $\Upsilon, s \models \Phi$. Given DACMAS $\mathcal{S}$, an initial state σ_0 and a $\mu\mathcal{L}_p^{ECQ_\ell}$ formula Φ, we are interested in the following *verification* problem: $\Upsilon_{\mathcal{S}}^{\sigma_0}, \sigma_0 \models \Phi$.

$$\begin{aligned}
(Q_\ell)_{v,V}^{\Upsilon} &= \{\sigma \in \Sigma \mid \mathcal{T}, \sigma, \mathsf{inst}, v \models Q_\ell\} \\
(\neg\Phi)_{v,V}^{\Upsilon} &= \Sigma \setminus (\Phi)_{v,V}^{\Upsilon} \\
(\Phi_1 \wedge \Phi_2)_{v,V}^{\Upsilon} &= (\Phi_1)_{v,V}^{\Upsilon} \cap (\Phi_2)_{v,V}^{\Upsilon} \\
(\exists x.\textsc{live}(x) \wedge \Phi)_{v,V}^{\Upsilon} &= \{\sigma \in \Sigma \mid \exists \mathsf{d} \in \textsc{adom}(\sigma).\sigma \in (\Phi)_{v[x/\mathsf{d}],V}^{\Upsilon}\} \\
(\textsc{live}(\vec{x}) \wedge \langle - \rangle\Phi)_{v,V}^{\Upsilon} &= \{\sigma \in \Sigma \mid \vec{x}/\vec{\mathsf{d}} \in v \text{ implies } \vec{\mathsf{d}} \subseteq \textsc{adom}(\sigma) \\
&\qquad \text{and } \exists \sigma'.\sigma \Rightarrow \sigma' \text{ and } \sigma' \in (\Phi)_{v,V}^{\Upsilon}\} \\
(\textsc{live}(\vec{x}) \wedge [-]\Phi)_{v,V}^{\Upsilon} &= \{\sigma \in \Sigma \mid \vec{x}/\vec{\mathsf{d}} \in v \text{ implies } \vec{\mathsf{d}} \subseteq \textsc{adom}(\sigma) \\
&\qquad \text{and } \forall \sigma'.\sigma \Rightarrow \sigma' \text{ implies } \sigma' \in (\Phi)_{v,V}^{\Upsilon}\} \\
(Z)_{v,V}^{\Upsilon} &= V(Z) \\
(\mu Z.\Phi)_{v,V}^{\Upsilon} &= \bigcap\{\mathcal{E} \subseteq \Sigma \mid (\Phi)_{v,V[Z/\mathcal{E}]}^{\Upsilon} \subseteq \mathcal{E}\}
\end{aligned}$$

Figure 2: Semantics of $\mu\mathcal{L}_p^{ECQ_\ell}$

EXAMPLE 5.1. Consider the contract of Example 3.4. Assume that $\mathcal{T}$ contains the axiom MyGoldCust $\sqsubseteq$ MyCust, where MyGoldCust denotes the gold customers of a seller. The following $\mu\mathcal{L}_p^{ECQ_\ell}$ property models that, for every delivery base-level commitment instance a seller has towards one of its gold customers, there must exist a run where the instance persists in the system until it is eventually satisfied.

$$\begin{aligned}&\nu Z.(\forall s, c, i.\mathsf{DeliveryC}(s, c, \mathsf{active}, i) \wedge \mathsf{MyGoldCust@}s(c) \\ &\ \rightarrow \mu Y.(\mathsf{DeliveryC}(s, c, \mathsf{sat}, i)) \vee (\textsc{live}(s, c, i) \wedge \langle - \rangle Y)) \wedge [-]Z\end{aligned}$$

5.1 State Boundedness and Decidability

The number of states of $\Upsilon_{\mathcal{S}}^{\sigma_0}$ is in general infinite, and verification of (even propositional) temporal properties of simple forms (e.g., reachability) turns out to be undecidable [2, 6]. This calls for identifying interesting classes of DACMASs for which verification is decidable. Recently, the notion of *state-bounded* system has been proposed in the context of both data-aware business processes [2] and MASs [3], as an interesting condition that ensures decidability of verification for rich first-order temporal properties, while reflecting naturally occurring working assumptions in real-world systems. Intuitively, state-boundedness allows for encountering infinitely many different data during the evolution of the system, provided that such data do not accumulate in a single state.

We take this general notion and adapt it to DACMASs. In particular, a DACMAS is *state-bounded* if, for every agent active in the system, there exists a bound on the number of data items simultaneously stored in its ABox. Since the ABox of inst stores the names of the active agents, this implicitly bounds also the number of simultaneously active agents. Observe, however, that the overall number of data items (and hence also agents) encountered across and along the runs of the system can still be infinite. With this notion in place, we are able to prove the key result of this paper:

THEOREM 5.1. *Verifying state-bounded DACMASs against $\mu\mathcal{L}_p^{ECQ_\ell}$ properties is decidable and reducible to finite-state model checking.*

PROOF SKETCH. The crux is to encode a state-bounded DACMAS $\mathcal{S}$ into a state-bounded Data-Centric Dynamic System (DCDS) $\mathcal{D}$ [2], and the $\mu\mathcal{L}_p^{ECQ_\ell}$ property Φ of interest into a $\mu\mathcal{L}_p$ property [2]. The encoding is done by flattening $\mathcal{S}$ as follows:

1. We compile away the TBox $\mathcal{T}$ by *(i)* rewriting all ECQ_ℓ contained in $\mathcal{S}$ and in the formula using the inclusion assertions of $\mathcal{T}$ [4, 5] (maintaining the location arguments unaltered), and *(ii)* rewriting the disjointness and key assertions in $\mathcal{T}$ into denial constraints to be checked over the data [5], and adding these to $\mathcal{D}$.
2. We unify all the ABoxes at the different agents into a single relational database where every relation has an additional parameter that denotes the name of the agent. Accordingly, we reformulate

the queries obtained at the previous step into this new vocabulary. Φ is in this way translated into a property Φ' that belongs to $\mu\mathcal{L}_p$.
3. We use events with parameters of $\mathcal{S}$ as actions for $\mathcal{D}$. The communicative rules will formulate the process, which controls when actions can be executed and over which parameters, while the update rules will formulate the action specifications. For each event, we group the effects involved in the corresponding update rules, reformulating each effect

$$[q^+] \wedge Q^- \rightsquigarrow \textbf{add}\{A_1, \dots, A_n\}, \textbf{del}\{D_1, \dots, D_m\}$$

into the equivalent set of "singleton" effects

$$[q^+] \wedge Q^- \wedge Q_r \rightsquigarrow \textbf{add}\{A_1\} \cdots [q^+] \wedge Q^- \wedge Q_r \rightsquigarrow \textbf{add}\{A_n\}$$
$$[q^+] \wedge Q^- \wedge Q_r \rightsquigarrow \textbf{del}\{D_1\} \cdots [q^+] \wedge Q^- \wedge Q_r \rightsquigarrow \textbf{del}\{D_m\}$$

where Q_r is the condition contained in the update rule from which the effect is extracted. We then translate all the obtained singleton effects into corresponding DCDS effects. The main difficulty is dealing with add and delete lists, instead of a complete reconstruction of the resulting data store required by DCDS. As for deletions, for every relation R in the vocabulary of the database, each singleton deletion effect e_i related to R is fetched, where

$$e_i = [q_i^+(\vec{p},\vec{x})] \wedge Q_i^-(\vec{p},\vec{x}) \wedge Q_r(\vec{p}) \rightsquigarrow \textbf{del}\{R(\vec{y})\}$$

and $\vec{y}$ is contained in $\vec{p} \cup \vec{x}$. Notice that, in $\mathcal{D}$, the free variables of Q_r correspond by construction to the parameters $\vec{p}$ of the action. From such rules, the following unique effect is produced, to maintain all those tuples in R that are not deleted by any of the deletion effects (resembling a frame axiom for R):

$$[R(\vec{z})] \wedge \neg \bigvee_{e_i} (q_i^+(\vec{p},\vec{x}') \wedge Q_i^+(\vec{p},\vec{x}') \wedge Q_r(\vec{p}) \wedge eq'(\vec{z},\vec{y})) \rightsquigarrow R(\vec{z})$$

where *(i)* $eq(\vec{z},\vec{y})$ denotes the transitive symmetric closure of the component-wise equality between $\vec{z}$ and $\vec{y}$, *(ii)* the vector of variables $\vec{x}'$ is obtained from $\vec{x}$ by replacing each variable x with the variable of $\vec{z}$ that is equated to x via $eq(\vec{z},\vec{y})$, and *(iii)* $eq'(\vec{z},\vec{y})$ is the conjunction of equalities obtained from $eq(\vec{z},\vec{y})$ by filtering away those equalities that contain a variable not in $\vec{z} \cup \vec{p}$. Singleton addition effects are instead simply kept unaltered in the obtained specification. This respects the fact that in DACMAS additions have priority over deletions: if the same fact is asserted to be deleted and added at the same time, the deletion DCDS effect does not transfer the tuple, but the addition DCDS effect does it.
The encoding is correct, in the sense that $\mathcal{D}$ faithfully reproduces the execution semantics of $\mathcal{S}$. Specifically, $\mathcal{S}$ and σ_0 verify Φ if and only if $\mathcal{D}$ (and the initial state corresponding to σ_0) verify Φ'. Also, $\mathcal{S}$ is state-bounded if and only if $\mathcal{D}$ is state-bounded. [2] shows that verification of $\mu\mathcal{L}_p$ properties over state-bounded DCDSs is decidable, and reducible to conventional, finite-state model checking. This is done by constructing a faithful, abstract transition system that verifies the same $\mu\mathcal{L}_p$ properties of the original one. Thanks to the correctness of the encoding, this abstraction can be used to faithfully verify properties of $\mathcal{S}$ as well. □

6. CONCLUSION

DACMASs are readily implementable in standard technologies such as JADE (which supports dynamic agent creation) and lightweight ontologies. Observe that a system execution requires polynomial time at each step (actually logspace w.r.t. the data, as any system based on relational databases). Only offline verification of the system is (as usual) exponential in the representation. Our framework complements that of [7], which employs data-aware commitments to monitor a system execution and track the state of commitment instances, but cannot be exploited for static analysis.

We have shown here how to encode DACMASs into DCDSs, so as to lift the key decidability results in [2] on verification of state-bounded DCDSs, to the case of state-bounded DACMASs. State-boundedness is a semantic condition that is undecidable to check [2]. Sufficient syntactic conditions over the action effects that ensure state boundedness are proposed in [2]. Such conditions can, in principle, be lifted to DACMASs as well. This is matter of ongoing work. We also consider extending our framework with the possibility of checking epistemic properties, in the line of [3]. Notice that, if instead of relying on the μ-calculus, we rely on CTL, we can relax the persistence requirement in the logic, as in [3].

Acknowledgements. This research has been partially supported by the EU IP Project Optique (*Scalable End-user Access to Big Data*), grant agreement n. FP7-318338, and by the Sapienza Award 2013 "SPIRITLETS: SPIRITLET–based Smart spaces".

7. REFERENCES

[1] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[2] B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. In *Proc. of the 32nd ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems (PODS)*, 2013.

[3] F. Belardinelli, A. Lomuscio, and F. Patrizi. An abstraction technique for the verification of artifact-centric systems. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*, pages 319–328, 2012.

[4] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. EQL-Lite: Effective first-order query processing in description logics. In *Proc. of the 20th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 274–279, 2007.

[5] D. Calvanese, G. De Giacomo, D. Lembo, M. Lenzerini, and R. Rosati. Data complexity of query answering in description logics. *Artificial Intelligence*, 195:335–360, 2013.

[6] D. Calvanese, G. De Giacomo, M. Montali, and F. Patrizi. Verification and synthesis in description logic based dynamic systems. In *Proc. of the 7th Int. Conf. on Web Reasoning and Rule Systems (RR)*, pages 50–64. Springer, 2013.

[7] F. Chesani, P. Mello, M. Montali, and P. Torroni. Representing and monitoring social commitments using the event calculus. *J. of Autonomous Agents and Multi-Agent Systems*, 27(1):85–130, 2013.

[8] A. K. Chopra and M. P. Singh. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*, chapter Agent Communication, pages 101–141. The MIT Press, 2013.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. The MIT Press, 1999.

[10] A. Poggi, D. Lembo, D. Calvanese, G. De Giacomo, M. Lenzerini, and R. Rosati. Linking data to ontologies. *J. on Data Semantics*, X:133–173, 2008.

[11] M. P. Singh. Formalizing communication protocols for multiagent systems. In *Proc. of the 20th Int. Joint Conf. on Artificial Intelligence (IJCAI)*, pages 1519–1524, 2007.

[12] M. P. Singh, A. K. Chopra, and N. Desai. Commitment-based service-oriented architecture. *IEEE Computer*, 42(11):72–79, 2009.

[13] C. Stirling. *Modal and Temporal Properties of Processes*. Springer, 2001.