Verification of Data-Aware Commitment-Based Multiagent System

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

Diego Calvanese  
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 \( cc(d, c, q_p, q_d) \) 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.


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 \( \Delta \)) are inserted.

The main result of this paper is that, when a DACMAS is state-bound, 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 \( \mu \)-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:

\[ C \rightarrow N \mid P[\vec{x}] \mid C \cap C \quad R \rightarrow P \mid P[i_1, \ldots, i_h] \]

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, \( \cap \) denotes concept conjunction, while \( P[\vec{i}] \) 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 \( I = (\Delta^I, \cdot)^I \), where \( \Delta^I \) is the nonempty interpretation domain, and \( \cdot^I \) is an interpretation function, assigning to each concept C a subset \( \Delta^C \) of \( \Delta^I \) to each constant \( a \) an object \( a^I \) of \( \Delta^I \). The semantics of an ontology is given by stating when an intensional knowledge, and an eventually quantified variables \( \ell \)-th component of tuple \( \vec{a}^I \). Also, \( \vec{a}^I \) assigns to each constant a an object \( a^I \) of \( \Delta^I \). We adopt the unique name assumption, i.e., \( a_1 \neq a_2 \) implies \( a_1^I \neq a_2^I \).

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

**3. FRAMEWORK**

We introduce now our framework for modeling DACMASs. Formally, a DACMAS is a tuple \( (\mathcal{T}, \mathcal{X}, \mathcal{I}, \mathcal{T}, \mathcal{C}, \mathcal{B}) \), where: (i) \( \mathcal{T} \) is a global DLR-Lite TBox; (ii) \( \mathcal{X} \) 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) \( \mathcal{I} \) 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) \( \mathcal{B} \) 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): hasSpec[i] \( \sqsubseteq \text{Agent} \), hasSpec[2] \( \sqsubseteq \text{Spec} \).

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 $\text{R@a}(\vec{x})$ to denote an atomic query returning the extension of $\text{R}$ in the ABox of agent $a$. If $a$ does not point to (the name of) an agent currently in the system, then $\text{R@a}(\vec{x})$ evaluates to empty. Beside the special constant $\text{self}$, we also use $\text{self}$ to implicitly refer to the agent that is posing the query (similarly to “this” in object-oriented programming). When an agent specifications react to events, through proactive and reactive rules. Such rules instead of full visibility of all exchanged messages, so as to properly communicated. All agents but the institutional one are only aware of the location of the other participants.

### 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 rule could result in the insertion of new data items (from the countably infinite domain $\Delta$), 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 $(\text{sn}, \Pi)$, where $\text{sn}$ is the specification name, and $\Pi$ 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

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

where $Q$ is an $\text{ECQ}_j$, and $\text{ev}(\vec{x})$ is an event supported by the system, i.e., predicate $\text{ev}/[\vec{x}]$ 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 $\theta$ returned by the query, using it to determine the event receiver and its payload. This states that the ground event $\text{ev}(\vec{x})\theta$ can be sent by the agent to $\text{self}$, provided that $\text{self}$ 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:

\[
\text{Spec@inst}(\text{sel}, \text{seller}) \text{ enables } \text{REQ_REG to } \text{sel}
\]

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

\[
\text{MyCust}(m) \land \text{Item}(i) \text{ enables } \text{PROPOSE}(i) \text{ to } m
\]

Update rules are ECA-like rules of the form:

- **on** $\text{ev}(\vec{x})$ **to** $r$ **if** $\text{Q}(r, \vec{x})$ **then** $\alpha(r, \vec{x})$ (on-send)
- **on** $\text{ev}(\vec{x})$ **from** $s$ **if** $\text{Q}(s, \vec{x})$ **then** $\alpha(s, \vec{x})$ (on-receive)
  where $\text{ev}/[\vec{x}]$ is an event type from $\mathcal{E}$, $Q$ is an $\text{ECQ}_j$, and $\alpha$ 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 $\alpha$ using the actual event payload and receiver/sender. Action $\alpha$ 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 $\alpha$ and by a list $\vec{p}$ of parameters), and $\{e_1, \ldots, e_n\}$ are update effects, each of which has the form

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

- $q^+$ is an $\text{UCQ}_j$, and $Q^-$ is an $\text{ECQ}$, 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. During the execution, the effect is applied with a ground substitution $\beta$ for the action parameters, and for every answer $\theta$ to the query $q^+((\vec{d}, \vec{x})) \land Q^-((\vec{d}, \vec{x}))$.
- $A$ 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 $\text{skolem}(A)$ to denote all Skolem terms mentioned in $A$. At runtime, whenever a ground Skolem term is produced by applying $\theta$ to $A$, the agent autonomously substitutes it with a possibly new data item taken from $\Delta$. This mechanism is exploited by the agent to inject new data into the system. The ground set of facts so obtained is $\text{added}$ 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 $\theta$, it is $\text{removed}$ 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:

\[
\text{on } \text{ASK_REG from } c \text{ if } \text{true then } \text{makeCust}(c)
\]

where $\text{makeCust}(x) : \{[\text{true} \rightsquigarrow \text{add } \{\text{MyCust}(x)\}]\}$. Assume now that the seller maintains the item cart for a customer, using relation $\text{InCart}(i,c)$ to model that item $i$ is in the cart of $c$. The seller reaction to an “empty cart” request is modeled as:

\[
\text{on } \text{EMPTY_CART_REQ from } c \text{ if } \text{MyCust}(c) \text{ then } \text{doEmpty}(c)
\]

\[1\] The distinction between $q^+$ and $Q^-$ is needed for technical reasons, borrowed from [2].
where \(dEmpty(c) : \{ [\text{InCart}(i, c)] \rightarrow \text{del}(\text{InCart}(i, c)) \}\). 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 \(p(i)\):

\[
\text{on NEW_ITEM}(i) \ \text{from} \ a \ \text{if} \ \text{true then} \ \text{addItem}(i)
\]

where \(\text{addItem}(i) : \{ [\text{true}] \rightarrow \text{add}(\{\text{item}(i), \text{Price}(i, p(i))\}) \}\).

### 3.3 Institutional Agent Specification

The institutional agent \(\mathcal{I}\) 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:

\[
\text{on EV}(\bar{x}) \ \text{from} \ s \ \text{to} \ r \ \text{if} \ Q(s, r, \bar{x}) \ \text{then} \ \alpha(s, r, \bar{x})
\]

where \(Q\) and \(\alpha\) 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:

\[
\text{on any event from} \ s \ \text{to} \ r \ \text{if} \ Q(s, r) \ \text{then} \ \alpha(s, r)
\]

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 start 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 its initial specification. The axiom \(\text{NewA}[1] \subseteq \neg \text{Agent}\) 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.

\[
\text{on AG_REQ}(s) \ \text{from} \ a \ \text{if} \ \neg (\exists x \exists y. \text{NewA}(x, y)) \ \text{then} \ \text{create}(s) \ \text{create}(s) : \{ [\text{true}] \rightarrow \text{add}(\{\text{NewA}(n(), s)\}) \}\}
\]

Note that axiom \(\text{NewA}[1] \subseteq \neg \text{Agent}\) 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:

\[
\text{NewA}(a, s) \ \text{enables} \ \text{INSERT_AG}(a, s) \ \text{to} \ \text{self}
\]

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

\[
\text{on INSERT_AG}(a, s) \ \text{from} \ \text{self} \ \text{if} \ \text{true then} \ \text{do_ins}(a, s) \ \text{do_ins}(a, s) : \{ [\text{true}] \rightarrow \text{add}(\{\text{Agent}(a), \text{Spec}(a, s)\}), \ \text{del}(\text{NewA}(a, s)) \}\]

### 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:

\[
\text{on EV}(\bar{x}) \ \text{from} \ s \ \text{to} \ r \ \text{if} \ Q_{\mathcal{C}}(s, r, \bar{x}) \ \text{then} \ \text{CC}(s, r, \{q_{\mathcal{C}}^{+}(s, r, \bar{x}, \bar{y}) \land Q_{\mathcal{P}}^{+}(s, r, \bar{x}, \bar{y}), Q_{\mathcal{D}}(s, r, \bar{x}, \bar{y})\})
\]

where \(n\) is the commitment name, the \(\text{ECQ}_{\mathcal{C}}\) \(Q_{\mathcal{C}}\) is the condition for the creation of the conditional commitment, \(\{q_{\mathcal{C}}^{+}, Q_{\mathcal{P}}^{+}\}\) where, as in update effects, \(q_{\mathcal{C}}^{+}\) is an \(\text{UCQ}_{\mathcal{C}}\) and \(Q_{\mathcal{P}}^{+}\) is an \(\text{ECQ}_{\mathcal{C}}\) whose free variables all occur among those of \(q_{\mathcal{C}}^{+}\) is the pre-condition determining the generation of a corresponding base-level commitment, and the \(\text{ECQ}_{\mathcal{C}}\) \(Q_{\mathcal{D}}\) 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 \(\text{GET-CC}()\) to extract the set of unconditional commitments contained in \(\mathcal{C}\).

According to the literature, commitments are manipulated either explicitly via specific events (such as 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 \(\text{GET-CC}()\) 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 \(\text{EV}\) is sent by agent \(a\) to agent \(b\) with payload \(d\), if \(Q_{\mathcal{C}}(a, b, d)\) is satisfied, an instance of the conditional commitment \(n\) is created. The debtor, creditor, and payload of this instance are respectively \(a, b,\) and \(d\).
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 \(\{q_{\mathcal{C}}^{+}(a, b, d, \bar{v}) \land Q_{\mathcal{P}}^{+}(a, b, d, \bar{v})\}\) is satisfied with actual values \(\bar{v}\) for variables \(\bar{y}\), the conditional commitment instance creates a base-level commitment instance with payload \(d\) and \(\bar{v}\). Such base-level instance is put into the active state. The discharge condition for this instance is the instantiation of \(Q_{\mathcal{D}}\) with the involved agents and specific payload, i.e., \(a\) is committed to bring about \(Q_{\mathcal{D}}(a, b, d, \bar{v})\).
3. Also a base-level commitment instance is explicitly or 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 \(Q_{\mathcal{D}}(a, b, d, \bar{v})\) 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 \( \text{Paid}(i, c) \) if \( c \) has paid for \( i \), and that the customer stores a fact \( \text{Owned}(i) \) whenever it owns \( i \), the commitment rule can be specified as:

\[
\text{on ACCEPT_REG from } s \text{ to } c \text{ if } \text{MyCust}\oslash s(c) \text{ then } CC\text{Delivery}(s, c, \{ \text{Item}\oslash s(i) \land \text{Paid}\oslash s(i, c), \text{Owned}\oslash c(i) \})
\]

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 \( B \) is a set of relations used by \( \text{inst} \) 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 \( CC_n (s, r, \{ q_p^+ (s, r, \bar{x}, \bar{y}), Q_p (s, r, \bar{x}, \bar{y}), Q_d (s, r, \bar{x}, \bar{y}) \} \) in \( \text{GET-CC}(C) \) induces two relations in \( B \), on the basis of the commitment name \( n \) and the payloads \( \bar{x} \) and \( \bar{y} \): (i) \( nCC/\text{ar} \), where \( ar = 2 + |\bar{x}| \) for debtor, creditor, and conditional commitment payload; (ii) \( nCC/\text{ar} \), where \( ar = 3 + |\bar{x}| + |\bar{y}| \) for debtor, creditor, state, and base-level commitment payload.

**Example 3.5.** The commitment in Example 3.4 induces the following relations in \( B \): \( \text{DeliveryCC}(\text{debtor}, \text{creditor}) \) and \( \text{DeliveryC}(\text{debtor}, \text{creditor}, \text{state}, \text{item}) \).

### 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 \( \text{inst} \) using the vocabulary of the CBox \( B \), 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 \( \text{GET-CC}(C) \), according to the “templates” described in the remainder of this section. We denote with \( \text{CC-RULES}(C) \) all the commitment manipulation rules produced from \( C \).

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

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

\[
\text{on EV}(\bar{x}) \text{ from } s \text{ to } r \text{ if } Q_\rho (s, r, \bar{x}) \text{ then create_nCC}(s, r, \bar{x}) \text{ create_nCC}(s, r, \bar{x}) : \{ \text{[true]} \rightarrow \text{add}\{nCC(s, r, \bar{x})\} \}
\]

**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 \( \text{DELEGATE_nCC} \) 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:

\[
\text{on DELEGATE_nCC}(c, \bar{x}) \text{ from } d_o \text{ to } d_n \text{ if } nCC(d_o, c, \bar{x}) \text{ then changedeb_nCC}(d_o, d_n, c, \bar{x}) \text{ changedeb_nCC}(d_o, d_n, c, \bar{x}) : \{ \text{[true]} \rightarrow \text{add}\{nCC(d_n, c, \bar{x})\}, \text{del}\{nCC(d_o, c, \bar{x})\} \}
\]

**CC cancellation.** The cancelation of a conditional commitment instance for commitment \( n \) is triggered when the debtor sends to the creditor a \( \text{CANCEL_nCC} \) event, providing the instance payload. If the instance exists, it is removed:

\[
\text{on CANCEL_nCC}(\bar{x}) \text{ from } d \text{ to } c \text{ if } nCC(d, c, \bar{x}) \text{ then delete_nCC}(d, c, \bar{x}) \text{ delete_nCC}(d, c, \bar{x}) : \{ \text{[true]} \rightarrow \text{del}\{nCC(d, c, \bar{x})\} \}
\]

**C creation.** Every conditional commitment instance for relation \( nCC \) creates a base-level commitment instance whenever the precondition (whose variables \( \bar{x} \) are grounded with the instance payload) holds with an answer substitution \( \theta \) for variables \( \bar{y} \). 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:

\[
\text{on any event from } d \text{ to } c \text{ if } \text{true then create}C(d, c) \text{ where, for each commitment } CC_n (s, r, \{ q_p^+ (s, r, \bar{x}, \bar{y}), Q_p (s, r, \bar{x}, \bar{y}), Q_d (s, r, \bar{x}, \bar{y}) \} \text{ in } \text{GET-CC}(C), \text{action } createC(d, c) \text{ contains the following detachment effect:}
\]

\[
\text{[nCC}(d, c, \bar{x}) \land Q_p (d, c, \bar{x}, \bar{y})] \land Q_d (d, c, \bar{x}, \bar{y}) \rightarrow \text{add } \{ nC(d, c, \text{active}, \bar{x}, \bar{y}) \}
\]

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:

\[
\text{on DELEGATE_nCC}(c, \bar{x}, \bar{y}) \text{ from } d_o \text{ to } d_n \text{ if } nC(d_o, c, \text{active}, \bar{x}, \bar{y}) \text{ then changedeb_nCC}(d_o, d_n, c, \bar{x}) \text{ changedeb_nCC}(d_o, d_n, c, \bar{x}) : \{ \text{[true]} \rightarrow \text{add}\{nCC(d_n, c, \text{active}, \bar{x}, \bar{y})\}, \text{del}\{nCC(d_o, c, \text{active}, \bar{x}, \bar{y})\} \}
\]

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

\[
\text{on CANCEL_nCC}(\bar{x}, \bar{y}) \text{ from } d \text{ to } c \text{ if } nC(d, c, \text{active}, \bar{x}, \bar{y}) \text{ then viol_nCC}(d, c, \text{active}, \bar{x}, \bar{y}) \text{ viol_nCC}(d, c, \text{active}, \bar{x}, \bar{y}) : \{ \text{[true]} \rightarrow \text{add}\{nC(d, c, \text{active}, \bar{x}, \bar{y})\}, \text{del}\{nC(d, c, \text{active}, \bar{x}, \bar{y})\} \}
\]

**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:

\[
\text{on any event from } d \text{ to } c \text{ if } \text{true then discharge}C(d, c) \text{ where, for each } CC_n (s, r, \{ q_p^+ (s, r, \bar{x}, \bar{y}), Q_p (s, r, \bar{x}, \bar{y}), Q_d (s, r, \bar{x}, \bar{y}) \} \text{ in } \text{CC}(C), \text{action } dischargeC(d, c) \text{ contains:}
\]

\[
\text{[nC}(d, c, \text{active}, \bar{x}, \bar{y})] \land Q_d (d, c, \bar{x}, \bar{y}) \rightarrow \text{add}\{nC(d, c, \text{sat}, \bar{x}, \bar{y})\}, \text{del}\{nC(d, c, \text{active}, \bar{x}, \bar{y})\}
\]
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):

\[
\text{on any event from } a \to b \text{ if true then removeFinal()}
\]

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

\[
[nC(d, c, s, \vec{x}, \vec{y})] \land (s = \text{sat } \lor s = \text{viol}) \Rightarrow \text{del}\{nC(d, c, s, \vec{x}, \vec{y})\}
\]

Example 3.6. Assume that the only rule in \(C\) is that of Example 3.4. The following CC creation rule is produced

\[
\text{on ACCEPT_REG from } s \to c \text{ if MyCust @ s(c) then create_DeliveryCC(s, c)}
\]

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

\[
\begin{align*}
\text{create}_C(d, c) : \{ & \text{DeliveryCC}(d, c) \land \text{Item@sd}(i) \land \text{Paid@id}(i, c) \} \\
\Rightarrow & \text{add}\{\text{DeliveryCC}(d, c, active, i)\}
\end{align*}
\]

\[
\begin{align*}
\text{discharge}_C(d, c) : \{ & \text{DeliveryC}(d, c, active, i) \land \text{Owns@e}(i) \\
\Rightarrow & \text{add}\{\text{DeliveryC}(d, c, sat, i)\}, \text{del}\{\text{DeliveryC}(d, c, active, i)\}\}
\end{align*}
\]

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 \(S = \langle T, \mathcal{E}, X, T, C, B \rangle\) and an initial state \(\sigma_0\), the execution semantics of \(S\) over \(\sigma_0\) is defined by a transition system \(T^S_0 = \langle \Delta, T \cup B, \Sigma, \sigma_0, \Rightarrow \rangle\), where:

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

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

\[
\forall \theta.\sigma.abox(a) \models R(\vec{x}) \text{, if } b@ = \text{self} \\
\forall \theta.\sigma.abox(b@) \models R(\vec{x}) \text{, if } b@ \neq \text{self}
\]

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

\[
\text{ANS}_i(Q, T, \sigma, a) = \{ \theta \mid T, \sigma, a, \theta \models Q \}
\]

The construction of the transition system \(T^S_0\) 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 \(\mu\)-calculus [13, 6]. \(\mu\)-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
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 EOCQ 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 $\mu L^\text{EOCQ}_p$ as:

$$\Phi ::= Q | \neg \Phi | \Phi_1 \land \Phi_2 | \exists \exists \; \text{LIVE}(x) \land \Phi | \text{LIVE}(\vec{x}) \land (\neg) \Phi | \text{LIVE}(\vec{x}) \land [\neg] \Phi \mid Z \mid \mu Z. \Phi$$

where $Q$ is a (possibly open) EOCQ 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 $\text{LIVE}(x_1, \ldots, x_n)$ abbreviates $\bigwedge_{\ell \in \{1, \ldots, n\}} \text{LIVE}(x_\ell)$. For $\mu L^\text{EOCQ}_p$, the following assumption holds: in $\text{LIVE}(\vec{x}) \land (\neg) \Phi$ and $\text{LIVE}(\vec{x}) \land [\neg] \Phi$, the variables $\vec{x}$ are exactly the free variables of $\Phi$, once we substitute to each bounded predicate variable $Z$ in $\Phi$ its bounding formula $\mu Z. \Phi'$. We adopt the usual abbreviations, including $\nu \exists Z. \Phi$ for greatest fixpoints.

Intuitively, the use of $\text{LIVE}(\cdot)$ in $\mu L^\text{EOCQ}_p$ 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 is canceled by inst, inst could reuse a in the future to identify another agent.

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

The semantics of $\mu L^\text{EOCQ}_p$ formulae is defined over a possibly infinite transition system $\mathcal{T} = \langle \Delta, \mathcal{U} \cup \mathcal{B}, \Sigma, \sigma_0, \Rightarrow \rangle$ (cf. Section 4), assuming that EOCQ 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 $\mu L^\text{EOCQ}_p$ 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 $\Delta$, and a predicate variable valuation $v$, i.e., a mapping from the predicate variables $Z$ to subsets of $\Sigma$. With these three notions in place, we assign meaning to formulae by associating to $\mathcal{T}, v$, and $v$ an extension function $(\cdot)^V_{v, v}$, which maps formulae to subsets of $\Sigma$. Formally, the extension function $(\cdot)^V_{v, v}$ is defined inductively as shown in Figure 2.

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

Example 5.1. Consider the contract of Example 3.4. Assume that $\mathcal{T}$ contains the axiom $\text{MyGoldCust} \subseteq \text{MyCust}$, where

$$(Q_1)^{\mathcal{V}}_{v, v} = \{ \sigma \in \Sigma \mid \mathcal{T}, \sigma, \text{inst}, v \models Q_1 \}$$

$$(\neg\Phi)^{\mathcal{V}}_{v, v} = \Sigma \setminus (\Phi)^{\mathcal{V}}_{v, v}$$

$$(\exists x. \text{LIVE}(x) \land \Phi)^{\mathcal{V}}_{v, v} = \{ \sigma \in \Sigma \mid \exists d \in \text{ADOM}(\sigma). \sigma \in (\Phi)^{\mathcal{V}}_{\mathcal{D}, v} \}$$

$$(\text{LIVE}(\vec{x}) \land (\neg) \Phi)^{\mathcal{V}}_{v, v} = \{ \sigma \in \Sigma \mid \vec{x}/d \in v \text{ implies } d \subseteq \text{ADOM}(\sigma) \}$$

$$(\text{LIVE}(\vec{x}) \land [\neg] \Phi)^{\mathcal{V}}_{v, v} = \{ \sigma \in \Sigma \mid \vec{x}/d \in v \text{ implies } d \subseteq \text{ADOM}(\sigma) \text{ and } \forall \sigma'. \sigma' \models \sigma \}$$

$$(\nu Z. \Phi)^{\mathcal{V}}_{v, v} = \{ \mathcal{E} \subseteq \Sigma \mid (\Phi)^{\mathcal{V}}_{\mathcal{E}, v}[\mathcal{Z}/\mathcal{E}] \subseteq \mathcal{E} \}$$

Figure 2: Semantics of $\mu L^\text{EOCQ}_p$

MyGoldCust denotes the gold customers of a seller. The following $\mu L^\text{EOCQ}_p$ 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.

$$\nu Z. (\forall s, c, \text{Delivery}(s, c, \text{active}, i) \land \text{MyGoldCust}(s, c) \rightarrow \mu Y. (\text{Delivery}(s, c, \text{sat}, i) \lor (\text{LIVE}(s, c) \land (\neg) Y)) \land [\neg] Z$$

5.1 State Boundedness and Decidability

The number of states of $\mathcal{T}^{\mathcal{S}}_0$ 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 in 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 $\mu L^\text{EOCQ}_p$ 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 $\mu L^\text{EOCQ}_p$ property $\Phi$ of interest into a $\mu L_p$ property [2]. The encoding is done by flattening $\mathcal{S}$ as follows:

1. We compile away the TBox $\mathcal{T}$ by (i) rewriting all EOCQ containing 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. \( \Phi \) is in this way translated into a property \( \Phi' \) that belongs to \( \mu \mathcal{L}_{p} \).

3. We use events with parameters of \( S \) as actions for \( D \). 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

\[
[q_{+}] \land Q^{-} \leadsto add\{A_{1}, \ldots, A_{n}\}, del\{D_{1}, \ldots, D_{m}\}
\]

into the equivalent set of “singleton” effects

\[
[q_{+}] \land Q^{-} \land Q_{e} \leadsto add\{A_{1}\} \ldots [q_{+}] \land Q^{-} \land Q_{r} \leadsto add\{A_{n}\} \\
[q_{+}] \land Q^{-} \land Q_{r} \leadsto del\{D_{1}\} \ldots [q_{+}] \land Q^{-} \land Q_{r} \leadsto del\{D_{m}\}
\]

where \( Q_{e} \) 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

\[
e_{i} = [q_{i}^{+}(\vec{p}, \vec{x})] \land Q_{e}^{+}(\vec{p}, \vec{x}) \land Q_{r}(\vec{p}) \leadsto del\{R(\vec{y})\}
\]

and \( \vec{y} \) is contained in \( \vec{p} \cup \vec{x} \). Notice that, in \( D \), 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 \( R \)):

\[
[R(\vec{z})] \land \forall \vec{v} \exists \vec{w} \; [q_{i}(\vec{p}, \vec{x}) \land Q_{e}(\vec{p}, \vec{x}) \land Q_{r}(\vec{p}) \land eq(\vec{z}, \vec{y})] \leadsto R(\vec{z})
\]

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 \( \vec{z} \) 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 \( D \) faithfully reproduces the execution semantics of \( S \). Specifically, \( S \) and \( \sigma_{0} \) verify \( \Phi \) if and only if \( D \) (and the initial state corresponding to \( \sigma_{0} \)) verify \( \Phi' \). Also, \( S \) is state-bounded if and only if \( D \) 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 \( S \) 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 \( \mu \)-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 “SPIRITLTS: SPIRITLET–based Smart spaces”.

7. REFERENCES


