Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

Verification of Relational Multiagent Systems with Data Types

Diego Calvanese **Marco Montali**
Free University of Bozen-Bolzano
Piazza Domenicani 3, 39100 Bolzano, Italy
{calvanese,montali}@inf.unibz.it

Giorgio Delzanno
University of Genova
Via Dodecaneso 35, 16146 Genova, Italy
giorgio.delzanno@unige.it

Abstract

We study the extension of relational multiagent systems (RMASs), where agents manipulate full-fledged relational databases, with data types and facets equipped with domain-specific, rigid relations (such as total orders). Specifically, we focus on design-time verification of RMASs against rich first-order temporal properties expressed in a variant of first-order μ-calculus with quantification across states. We build on previous decidability results under the state-bounded assumption, i.e., in each single state only a bounded number of data objects is stored in the agent databases, while unboundedly many can be encountered over time. We recast this condition, showing decidability in presence of dense, linear orders, and facets defined on top of them. Our approach is based on the construction of a finite-state, sound and complete abstraction of the original system, in which dense linear orders are reformulated as non-rigid relations working on the active domain of the system only. We also show undecidability when including a data type equipped with the successor relation.

1 Introduction

We study *relational multiagent systems* (RMASs), taking inspiration from the recently defined framework of data-aware commitment-based multiagent systems (DACMASs) (Chopra and Singh 2013; Montali, Calvanese, and De Giacomo 2014). Broadly speaking, an RMAS is constituted by agents that maintain data in an internal fully fledged relational database, apply proactive and reactive rules to update their own data, and exchange messages with other agents. Messages have an associated payload, which is used to move data from one agent to another. When updating their internal database, agents may also inject fresh data into the system, by invoking external services. This abstraction serves as a metaphor for any kind of interaction with the external world, such as interaction with web services, or humans.

From the data perspective, previous research has mainly focused on a single, countably infinite data domain, whose elements can only be compared for equality and inequality. This assumption is highly restrictive, since data types used in applications are typically equipped with domain-specific, rigid relations (such as total orders), and might be specialized through the use of *facets* (ISO/IEC 11404:2007 2007; Savkovic and Calvanese 2012).

The focus of this work is on design-time verification of RMASs against rich first-order temporal properties, allowing for quantification across states. By considering only a countably infinite domain with equality, it has been shown in (Belardinelli, Lomuscio, and Patrizi 2012; Bagheri Hariri et al. 2013; Montali, Calvanese, and De Giacomo 2014) that decidability of verification holds for variants of first-order temporal logics under the assumption that the system is state-bounded, i.e., unboundedly many data objects can be encountered over time, provided that in each single state only a bounded number of them is stored in the agent databases (Bagheri Hariri et al. 2014). We recast this condition by considering different options for the data types. Specifically, by exploiting an encoding of two-counter machines, we show that decidability of verification even of propositional reachability properties is lost when one of the data types is equipped with the successor relation. Our main technical result is showing decidability for a variant of first-order μ-calculus in presence of dense, linear orders, and facets defined on top of them. In this case, we provide an explicit technique to construct a finite-state, faithful abstraction of the original system, in which dense linear orders are reformulated as non-rigid relations working on the active domain of the system only. This allows us to model and verify state-bounded RMASs that include coordination mechanisms such as ticket-based mutual exclusion protocols.

Major details about the framework and its execution semantics, as well as full proofs and examples, can be found in an extended version of this work (Calvanese, Delzanno, and Montali 2014).

2 Relational Multiagent Systems

RMASs are data-aware multiagent systems constituted by agents that exchange and update data. Beside generic agents, an RMAS is equipped with a so-called *institutional agent*, which exists from the initial system state, and can be contacted by the other agents as a sort of "white-page" agent, i.e., to: *(i)* get information about the system as a whole; *(ii)* obtain names of other agents so as to establish an interaction with them; and *(iii)* create and remove agents.

At a surface level, RMASs and DACMASs share many aspects. There are however two key differences in the way they model data. On the one hand, while DACMASs consider

only a single, abstract data domain equipped with equality only, in RMASs data are typed and enriched with domain-specific relations. This deeply impacts the modeling power of the system (see Section 3). On the other hand, while agents in DACMASs operate with incomplete knowledge about the data, and use a description logic ontology as a semantic interface for queries, RMASs employ standard relational technology for storage and querying services. This is done to simplify the treatment and isolate the core issues that arise when incorporating data types and facets, but we believe our results can be transferred to DACMASs as well.

An *RMAS* $\mathcal{X}$ is a tuple $\langle \mathcal{T}, \mathcal{F}, \Delta_{0,\mathcal{F}}, \mathcal{S}, \mathcal{M}, \mathcal{G}, I \rangle$, where: (1) $\mathcal{T}$ is a finite set of *data types*; (2) $\mathcal{F}$ is a finite set of *facets* over $\mathcal{T}$; (3) $\Delta_{0,\mathcal{F}}$ is the initial data domain of $\mathcal{X}$; (4) $\mathcal{S}$ is a finite set of *$\mathcal{F}$-typed service calls*; (5) $\mathcal{M}$ is a finite set of *$\mathcal{F}$-typed relations* denoting messages with payload; (6) $\mathcal{G}$ is a finite set of *$\mathcal{F}$-typed agent specifications*; and (7) I is the $\mathcal{F}$-typed specification of the *institutional agent*.

2.1 Data Types and Their Facets

Data types and facets provide the backbone for modeling real-world objects manipulated by the RMAS agents. A *data type* T is a pair $\langle \Delta_T, \mathcal{R}_T \rangle$, where Δ_T is an infinite set[1], and $\mathcal{R}_T$ is a set of relation schemas. Each relation schema $R/n \in \mathcal{R}_T$ with name R and arity n is associated with an n-ary predicate $R^T \subseteq \Delta_T^n$. Given a set $\mathcal{T}$ of data types, we denote by $\mathcal{R}_\mathcal{T}$ all domain-specific relations mentioned in $\mathcal{T}$. Similarly, $\Delta_\mathcal{T}$ groups all the (pairwise disjoint) data domains of the data types in $\mathcal{T}$. The interaction between data types is orthogonal to our work and is left for the future.

Example 2.1. We consider the following, well-known data domains, whose relations retain the usual meaning:
- Dense total orders such as $\langle \mathbb{Q}, \{<, =\} \rangle$ and $\langle \mathbb{R}, \{<, =\} \rangle$.
- Total orders with successor, like: $\langle \mathbb{Z}, \{<, =, \mathsf{succ}\} \rangle$. ■

We assume that every RMAS has two special datatypes: *(i)* $\langle \mathbb{A}, \{=\} \rangle$ for *agent names* that, as in mobile calculi, behave as pure names (Needham 1989; Montanari and Pistore 2005) and can only be tested for (in)equality. *(ii)* $\langle \mathbb{B}, \{=\} \rangle$ for *agent specification names* (see Section 2.4).

Facets are introduced to restrict data types. A *facet* F is a pair $\langle T, \varphi(x) \rangle$ where $T = \langle \Delta_T, \mathcal{R}_T \rangle$ is a data type, and $\varphi(x)$ is a monadic *facet formula* built as:

$$\varphi(x) := \mathsf{true} \mid P(\vec{v}) \mid \neg\varphi(x) \mid \varphi_1(x) \vee \varphi_2(x)$$

where $P(\vec{v})$ is a relation whose schema belongs to $\mathcal{R}_T$, and whose terms $\vec{v}$ are either variable x or data objects in Δ_T. We use the standard abbreviations false and $\varphi_1(x) \wedge \varphi_2(x)$. Given a set $\mathcal{F}$ of facets, we use $\mathcal{R}_\mathcal{F}$ and $\Delta_\mathcal{F}$ as a shortcut for $\mathcal{R}_\mathcal{T}$ and $\Delta_\mathcal{T}$ respectively, where $\mathcal{T}$ is the set of data types on which facets in $\mathcal{F}$ are defined.

Given a *facet* $F = \langle T, \varphi(x) \rangle$ with $T = \langle \Delta_T, \mathcal{R}_T \rangle$, a data object $\mathtt{d}$ *belongs to* F if: *(i)* $\mathtt{d} \in \Delta_T$; *(ii)* $\varphi(x)$ holds in F under substitution $[x/\mathtt{d}]$, written $F, [x/\mathtt{d}] \models \varphi(x)$. In turn, given substitution $\sigma = [x/\mathtt{d}]$, relation $F, \sigma \models \varphi(x)$ is inductively defined as follows:

$$\begin{array}{ll} F, \sigma \models \mathsf{true} & \\ F, \sigma \models R(\vec{v})\sigma & \text{if } R(\vec{v})\sigma \text{ is true in } T \\ F, \sigma \models \neg\varphi(x) & \text{if } F, \sigma \not\models \varphi(x) \\ F, \sigma \models \varphi_1(x) \wedge \varphi_2(x) & \text{if } F, \sigma \models \varphi_1(x) \text{ and } F, \sigma \models \varphi_2(x) \end{array}$$

Notice that a *base facet* that simply ranges over all data objects of a data type can be encoded with true as its facet formula. In particular, we use $AF = \langle \langle \mathbb{A}, \{=\} \rangle, \mathsf{true} \rangle$ and $SF = \langle \langle \mathbb{B}, \{=\} \rangle, \mathsf{true} \rangle$ to refer to two base facets for agent and specification names respectively.

Example 2.2. $\langle \langle \mathbb{R}, \{>, =\} \rangle, (x > \mathtt{0} \wedge \mathtt{18} > x) \vee x > \mathtt{65} \rangle$ denotes ages of junior or senior people. ■

Facets are used as relation types. Given a set $\mathcal{F}$ of facets, an *$\mathcal{F}$-typed relation schema* $\overline{R}$ is a pair $\langle R/n, \mathcal{F}_R \rangle$, where R/n is a relation schema with name R and arity n, and $\mathcal{F}_R$ is an n-tuple $\langle F_1, \ldots, F_n \rangle$ of facets in $\mathcal{F}$.

An *$\mathcal{F}$-typed database schema* $\mathcal{D}$ is a finite set of $\mathcal{F}$-typed relation schemas, such that no two typed relations in $\mathcal{D}$ share the same name. Obviously, since relations are typed, it is important to define when their tuples agree with their facets. Let $\overline{R} = \langle R/n, \mathcal{F}_R \rangle$ be a relation schema. We say that a fact $R(\mathtt{o_1}, \ldots, \mathtt{o_n})$ *conforms to* $\overline{R}$ if for every $i \in \{1, \ldots, n\}$, we have that $\mathtt{o_i}$ belongs to F_i. Let $\mathcal{F}$ be a set of facets, and $\mathcal{D}$ be an $\mathcal{F}$-typed database schema. A database instance I *conforms to* $\mathcal{D}$ if every tuple $R(\mathtt{o_1}, \ldots, \mathtt{o_n}) \in I$ conforms to its corresponding relation schema $\overline{R} \in \mathcal{D}$.

2.2 Initial Data Domain

Given a data type $T = \langle \Delta_T, \mathcal{R}_T \rangle$, we isolate a *finite* subset $\Delta_{0,T} \subset \Delta_T$ of *initial data objects* for T. This subset explicitly enumerates those data objects that can be used in the initial states of the agent specifications (cf. Section 2.4), plus specific "control data objects" that are explicitly mentioned in the agent specifications themselves, and consequently contribute to determine the possible executions.

We extend this notion to cover also those objects used in the definition of facets. Giving a facet $F = \langle T, \varphi(x) \rangle$ with $T = \langle \Delta_T, \mathcal{R}_T \rangle$, the set of *initial data objects* for F is a finite subset of Δ_T that contains all data objects explicitly mentioned in $\varphi(x)$. The *initial data domain* of an RMAS with set $\mathcal{F}$ of facets, written $\Delta_{0,\mathcal{F}}$, is then defined as the (disjoint) union of initial data objects for each facet in $\mathcal{F}$.

2.3 Typed Service Calls

Typed service calls provide an abstract mechanism for agents to incorporate new data objects when updating their own databases. As argued in (Bagheri Hariri et al. 2013; Montali, Calvanese, and De Giacomo 2014; Bagheri Hariri et al. 2014), this is crucial to make the system "open" to the external world, and accounts for a variety of interaction modes, such as interaction with services or humans. We exploit this mechanism to model in particular the agent ability to inject new data according to internal decisions taken by the agent itself, but still external to its specification.

Given a set $\mathcal{F}$ of facets, an *$\mathcal{F}$-typed service* $\overline{\mathbf{f}}$ is a triple $\langle \mathbf{f}/n, \mathcal{F}^{in}, F^{out} \rangle$, where *(i)* $\mathbf{f}/n$ is a function schema with name $\mathbf{f}$ and arity n; *(ii)* $\mathcal{F}^{in}$ is an n-tuple $\langle F_1, \ldots, F_n \rangle$ of

[1] Facets can be used to model finite sets (cf. below).

facets in $\mathcal{F}$ representing the *input types* of the service call; *(iii)* F^{out} is a facet in $\mathcal{F}$ representing the *output facet* of the service call. As for typed relations, in $\mathcal{S}$ there are no two typed services that share the same name. Intuitively, when invoked with a tuple of ground data objects belonging to their input facets, the service non-deterministically returns a data object that belongs to the output facet.

Example 2.3. Service $\overline{\mathbf{getPrice}} = \langle \mathbf{getPrice}/0, \{SF\}, PF \rangle$ gets a string in $SF = \langle \langle \mathbb{S}, \{=\} \rangle, \mathsf{true} \rangle$ referring to a product, and returns a rational price $PF = \langle \langle \mathbb{Q}, \{<, =\} \rangle, x > 0 \rangle$. ■

2.4 Agent Specifications

In RMASs, agent specifications consist of three main components. The first is the data component, whose intensional part is a typed database schema with constraints; every agent adopting the same specification starts with the same initial extensional data, but during the execution it autonomously evolves by interacting with other agents and services. The second is a proactive behavior, constituted by a set of condition-action communicative rules that determine which messages can be emitted by the agent, together with their actual payload and target agent. The third is a reactive behavior, constituted by ECA-like update rules that determine how the agent updates its own data when a certain message with payload is received from or sent to another agent.

Given a set $\mathcal{F}$ of facets with initial data domain $\Delta_{0,\mathcal{F}}$, an *$\mathcal{F}$-typed agent specification* is a tuple $\langle \mathtt{n}, \mathcal{D}, \Gamma, D_0, \mathcal{C}, \mathcal{A}, \mathcal{U} \rangle$, where: (1) $\mathtt{n} \in \mathbb{B} \cap \Delta_{0,\mathcal{F}}$ is the *specification name*, which is assumed to be also part of the initial data domain. (2) $\mathcal{D}$ is an $\mathcal{F}$-typed database schema. (3) Γ is a finite set of database constraints over $\mathcal{D}$, i.e., of domain-independent first-order formulae over $\mathcal{D}$ and $\mathcal{R}_{\mathcal{F}}$, using only constants from $\Delta_{0,\mathcal{F}}$. (4) D_0 is the *initial agent state*, i.e., a database instance that conforms to $\mathcal{D}$, satisfies all constraints in Γ, and uses only constants from D_0. (5) $\mathcal{C}$ is a set of *communicative rules*, defined below. (6) $\mathcal{A}$ and $\mathcal{U}$ are sets of *update actions* and *update rules*, defined below. When clear from the context, we use the name of a component with the specification name as superscript, to extract that component from the specification tuple. For example, $\mathcal{D}^{\mathtt{n}}$ denotes the database schema above.

Communicative rules. These rules are used to determine which messages with payload are enabled to be sent by the agent to other agents, depending on the current configuration of the agent database. When multiple ground messages with payload are enabled, the agent nondeterministically chooses one of them, according to an internal, black-box policy.

A *communicative rule* is a rule of the form

$$Q(t, \vec{x}) \textbf{ enables } M(\vec{x}) \textbf{ to } t$$

where: *(i)* Q is a domain-independent FO query over $\mathcal{D}$ and $\mathcal{R}_{\mathcal{F}}$, whose terms are variables t and $\vec{x}$, as well as data objects in $\Delta_{0,\mathcal{F}}$; *(ii)* $M(\vec{x})$ is a message, i.e., a typed relation whose schema belongs to $\mathcal{M}$.

Let $\mathcal{F}$ be a set facets, $\mathcal{D}$ a $\mathcal{F}$-typed database schema, D a database instance that conforms to $\mathcal{D}$, and $Q(x_1, \ldots, x_n)$ a FO query over $\mathcal{D}$ and $\mathcal{R}_{\mathcal{F}}$ that uses only constants in $\Delta_{0,\mathcal{F}}$. The *answer* $ans(Q, D)$ *to* Q *over* D is the set of assignments θ from the free variables $\vec{x}$ of Q to data objects in $\Delta_{0,\mathcal{F}}$, such that $D \models Q\theta$. We treat $Q\theta$ as a boolean query, and we say $ans(Q\theta, D) \equiv \mathsf{true}$ if and only if $D \models Q\theta$.

In the following, we use the special query $\text{LIVE}_T(x)$ as a shortcut for the query that returns all data objects in the current active domain that belong to data type T. Given schema $\mathcal{D}$, such a query can be easily expressed as the union of conjunctive queries checking whether x belongs to a component of some relation in $\mathcal{D}$, such that the component has type T. In this respect, notice that any query can be relativized to the active domain through LIVE atoms.

We also make use to the anonymous variable "_" to signify an existentially quantified variable not used elsewhere.

Update actions. These are parametric actions used to update the agent current database instance, possibly injecting new data objects by interacting with typed services.

An *update action* is a pair $\langle \overline{\alpha}, \alpha_{spec} \rangle$, where: *(i)* $\overline{\alpha}$ is the *action schema*, i.e., a typed relation accounting for the action name and for the number of action parameters, together with their types; *(ii)* α_{spec} is the action specification and has the form $\alpha(\vec{\mathsf{p}}) : \{e_1, \ldots, e_n\}$, where $\{e_1, \ldots, e_n\}$ are update effects. Each update effect has the form

$$Q(\vec{\mathsf{p}}, \vec{x}) \rightsquigarrow \textbf{add } A, \textbf{del } D$$

where *(i)* Q is a domain-independent FO query over $\mathcal{D}$ and $\mathcal{R}_{\mathcal{F}}$, whose terms are parameters $\vec{\mathsf{p}}$, variables $\vec{x}$, and data objects in $\Delta_{0,\mathcal{F}}$; *(ii)* A is a set of "add" facts over $\mathcal{D}$ that include as terms: free variables $\vec{x}$ of Q, parameters $\vec{\mathsf{p}}$ and terms $\mathbf{f}(\vec{x}, \vec{\mathsf{p}})$, with $\overline{\mathbf{f}}$ in $\mathcal{S}$; *(iii)* D is a set of "del" facts that include as terms free variables $\vec{x}$ and parameters $\vec{\mathsf{p}}$.

An update action is applied by grounding its parameters $\vec{\mathsf{p}}$ with data objects $\vec{\mathsf{o}}$. This results in partially grounding each of its effects. The effects are then applied in parallel over the agent database, as follows. For each partially grounded effect $Q(\vec{\mathsf{o}}, \vec{x}) \rightsquigarrow \textbf{add } A, \textbf{del } D$, $Q(\vec{\mathsf{o}}, \vec{x})$ is evaluated over the current database and for each obtained answer θ, the fully ground facts $A\theta$ (resp., $D\theta$) are obtained. All the ground facts in $D\theta$ are deleted from the agent database. Facts in $A\theta$, instead, could contain (ground) typed service calls. In this case, every service call is issued, obtaining back a (possibly fresh) data object belonging to the output facet of the service. The instantiated facts in $A\theta$ obtained by replacing the ground service calls with the corresponding results are then added to the current database, giving priority to additions.

Update rules. These are conditional, ECA-like rules used by the agent to invoke an update action on its own data when a message with payload is exchanged with another agent.

An *update rule* is a rule of the form

- **on** $\text{MSG}(\vec{x})$ **to** t **if** $Q(t, \vec{x})$ **then** $\alpha(d, \vec{x})$ (on-send), or
- **on** $\text{MSG}(\vec{x})$ **from** s **if** $Q(t, \vec{x})$ **then** $\alpha(s, \vec{x})$ (on-receive)

where: *(i)* $M(\vec{x})$ is a message, i.e., a typed relation whose schema belongs to $\mathcal{M}$; *(ii)* Q is a FO query over $\mathcal{D}$, whose terms are variables t (resp., s) and $\vec{x}$, as well as constants in $\Delta_{0,\mathcal{F}}$; *(iii)* α is an update action in $\mathcal{A}$, whose parameters are bound to variables $\vec{x}$ and t (resp., s).

Institutional Agent Specification. In an RMAS, an *institutional* agent is dedicated to the management of the system as a whole. Differently from DACMASs (Montali, Calvanese,

and De Giacomo 2014), we do not assume here that the institutional agent has full visibility of the messages exchanged by all agents acting into the system. It is simply an agent that is always active in the system and whose name, inst in the following, is known by all agents. Still, we assume that the institutional agent has special duties, such as in particular management of agent creation and removal from the system, and maintainance of agent-related information, like names names of all active agents together with their specifications.

Technically, the institutional agent specification I is a standard agent specification named ispec, partially grounded as follows. To keep track of agents and their specifications, $\mathcal{D}_i$ contains three dedicated typed relations: *(i)* $\langle Agent/1, \langle AF \rangle \rangle$, to store agent names; *(ii)* $\langle Spec/1, \langle SF \rangle \rangle$, to store specification names; *(iii)* $\langle hasSpec/2, \langle AF, SF \rangle \rangle$, to store the relationship between agents and their specifications. Given these special relations, inst can also play the role of *agent registry*, supporting agents in finding names of other agents to communicate with. Additional system-level relations, such as agent roles, duties, commitments (Montali, Calvanese, and De Giacomo 2014), can be insterted into $\mathcal{D}_{\mathsf{inst}}$ depending on the specific domain under study. To properly enforce that $hasSpec/2$ relates agent to specification names, foreign keys can be added to $\Gamma^{\mathtt{ispec}}$. Futhermore, we properly initialize D_0^{inst} as follows: *(i)* $Agent(\mathsf{inst}) \in D_0^{\mathtt{ispec}}$; *(ii)* $Spec(s_i) \in D_0^{\mathtt{ispec}}$ for every agent specification that is part of the RMAS, i.e., for specification name ispec and all specification names mentioned in $\mathcal{G}$; *(iii)* $hasSpec(\mathsf{inst}, \mathtt{instSpec}) \in D_0^{\mathtt{ispec}}$. Obviously, inst may have other initial data, and specific rules and actions. Of particular interest is the possibility for inst to dynamically create and remove agents. This can be encoded by readapting (Montali, Calvanese, and De Giacomo 2014), as shown in (Calvanese, Delzanno, and Montali 2014).

Well-Formed Specifications. In an RMAS, every piece of information is typed. This calls for a suitable notion of *well-formedness* that checks the compatibility of types inside agent specifications (see (Calvanese, Delzanno, and Montali 2014)). Intuitively, an RMAS $\mathcal{X}$ is well-formed if: (1) every query appearing in $\mathcal{X}$ consistently use variables, that is, if a variable appears in multiple components, they all have the same data type; (2) every proactive rule instantiates the message payload with compatible data objects, and the destination agent with an agent name; (3) every reactive rule correctly relates the data types of the message payload with those of the query and of the update action; (4) each action effect uses parameters in a compatible way with the action type; (5) each action effect instantiates the facts in the head in a compatible way with their types; (6) each service call correctly binds its inputs and output. From now on, we always assume well-formedness. Notice that well-formedness does not guarantee that the restrictions imposed by facets are always satisfied, but only that the agent specification consistently use data types. Consistency with facets is dynamically managed at runtime (cf. Section 4).

3 Modeling with RMAS

We sketch how RMASs can be easily accommodate complex data-aware interaction protocols, leveraging on data types. We take inspiration from ticket-based mutual exclusion protocols (Bultan, Gerber, and Pugh 1999; Baier and Katoen 2008; Delzanno and Podelski 1999). This can be used, in our setting, to guarantee the possibility for an agent to engage in a complex, critical interaction with the institutional agent. Another interesting example, modeling a form of *contract net* in RMASs, is provided in (Calvanese, Delzanno, and Montali 2014). The interested reader can also refer to (Montali, Calvanese, and De Giacomo 2014) for commitment-based interactions.

From now on, we assume that interaction in RMAS is synchronous. This assumption is without loss of generality. Asynchronous communication can be simulated via an encoding of (un)ordered buffer operations as queries and updates on typed relations.

Theorem 3.1. *Asynchronous RMASs based on message queues/buffers can be simulated by synchronous RMASs.*

The idea behind ticket-based mutual exclusion protocols is that, when a process wants to access a critical section, it must get a ticket, and wait until its turn arrives. We model tickets using the base facet $RF = \langle \langle \mathbb{R}, \{<, =\} \rangle, \mathsf{true} \rangle$ for real numbers, and exploit the domain-specific relation $<$ to compare agent tickets. In our formulation, the critical section consists of a (possibly complex) interaction with the inst, excluding the possibility for other agents to concurrently engage in the same kind of interaction with inst.

We focus on the realization of the inst agent, in such a way that mutual exclusion is guaranteed no matter how the other agents behave. First of all, inst gives top priority to handle ticket requests by the agents. A ticket request is issued by another agent using a 0-ary message ASKTICKET. Agent inst reacts by invoking a ticket generation action, provided that the sender agent is not already owner of a ticket, and the $Assigned$ relation is empty (see below):

on ASKTICKET() **from** a
if $\neg HasTicket(a, \_) \wedge \neg Assigned(\_, \_)$ **then** GENTICKET(a)

Action GENTICKET takes as input an agent name, and uses a typed service $\overline{\mathbf{getTicket}} = \langle \mathbf{getTicket}/0, \emptyset, RF \rangle$ to get a numerical ticket. The result is stored in the temporary relation $Assigned$, tracing that the ticket has been assigned but the corresponding agent still needs to be informed.

GENTICKET(a) : {true $\rightsquigarrow$ **add**$\{Assigned(\mathsf{a}, \mathbf{getTicket}())\}$}

To guarantee that every agent will have the possibility of engaging the critical interaction with inst, every time a ticket is assigned to an agent, inst must ensure that such agent will be served *after* those already possessing a ticket. This is enforced through the following database constraint, which leverages on the domain-specific relation $>$ for tickets:

$$\forall t_{new}, t. Assigned(\_, t_{new}) \wedge HasTicket(\_, t) \rightarrow t_{new} > t$$

An assigned ticket must be sent to the requestor agent:

$$Assigned(t, a) \textbf{ enables } \text{GIVETICKET}(t) \textbf{ to } a$$

to which inst itself reacts by moving the tuple from the temporary relation $Assigned$ to $hasTicket$:

on GIVETICKET(t) **to** a **if** true **then** BINDTICKET(a, t)

$$\text{BINDTICKET}(\mathtt{a},\mathtt{t}) : \begin{Bmatrix} \mathsf{true} \rightsquigarrow \textbf{del}\ \{Assigned(\mathtt{a},\mathtt{t})\} \\ \mathsf{true} \rightsquigarrow \textbf{add}\{hasTicket(\mathtt{a},\mathtt{t})\} \end{Bmatrix}$$

Now, let CMSG be a critical message. To engage in the critical interaction with inst triggered by message CMSG, the agent provides the payload and the ticket. Agent inst positively react to the request provided that the ticket indeed corresponds to the agent, and that the ticket is now to be served (i.e., it is smaller than any other ticket):

on CMSG$(\vec{p}, t)$ **from** a
if $hasTicket(a, t) \wedge \neg(\exists a', t'.hasTicket(a', t') \wedge t > t')$
then CACT$(a, \vec{p})$

This pattern can be replicated for any other critical interaction. Additional state relations can be added to discipline the orderings among critical message exchanges.

4 Verification

We now focus on the verification of RMASs against rich first-order temporal properties. The execution semantics of RMAS $\mathcal{X} = \langle \mathcal{T}, \mathcal{F}, \Delta_{0,\mathcal{F}}, \mathcal{S}, \mathcal{M}, \mathcal{G}, I \rangle$ is captured by a *relational transition system* $\Upsilon_{\mathcal{X}} = \langle \Delta_{\mathcal{T}}, \mathcal{D}_{\mathcal{X}}, \Sigma, s_0, db, \rightarrow \rangle$, where: *(i)* $\mathcal{D}_{\mathcal{X}}$ is the union of typed schemas in the specifications of $\mathcal{G}$ and I; *(ii)* Σ is a possibly infinite sets of *states*; *(iii)* $s_0 \in \Sigma$ is the *initial state*; *(iv)* db is a function that, given a state $s \in \Sigma$ and the name $\mathtt{n}$ of an agent active in s, returns the database of $\mathtt{n}$ in state s, written $s.db(\mathtt{n})$, which must be $\mathcal{D}^{\mathtt{spec}_\mathtt{n}}$-conformant (where $\mathtt{spec}_\mathtt{n}$ is the specification of $\mathtt{n}$). *(v)* $\rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation between states.

The full $\Upsilon_{\mathcal{X}}$ construction starting from the initial state is detailed in (Calvanese, Delzanno, and Montali 2014). We report the main steps in the following. The initial state s_0 is constructed by assigning $s_0.db(\mathsf{inst})$ to the initial database instance $D_0^{\mathtt{ispec}}$ of I, and the initial database of each agent mentioned in $D_0^{\mathtt{ispec}}$ taking from its specification. The construction then proceeds by mutual induction over Σ and $\rightarrow$, repeating the following steps forever: (1) A state s is picked from Σ. (2) An active agent $\mathtt{a}$ is nondeterministically picked selecting its name from $s.db(\mathsf{inst})$. (3) The communicative rules of $\mathtt{a}$ are evaluated, extracting all enabled messages with their ground payloads and destination agents. (4) An enabled messages is nondeterministically picked. (5) The on-send/on-receive rules of the two involved agents are triggered, fetching all actions to be applied. (6) The actions are applied over the respective databases. If there are service calls involved, they are nondeterminstically substituted with resulting data objects, consistently with the service output facets. (7) Each agent updates its own database provided that the database resulting from the parallel application of the actions is compatible with the schema and satisfies all constraints. Otherwise the old database is maintained, so as to model a sort of "transaction rollback". (8) If one of the involved agents is inst and the update leads to the introduction of a new agent into the system, its database is initialized in accordance to its specification. (9) The global state so obtained is declared to be successor of the state picked at step 1.

Interestingly, $\Upsilon_{\mathcal{X}}$ is in general *infinite-branching*, because of the substitution of service calls with their results, and *infinite runs*, because of the storage of such data objects in time.

The $\mu\mathcal{L}_p^{@}$ Verification Logic. To specify sophisticated properties over RMASs we employ the $\mu\mathcal{L}_p^{@}$ logic. This logic combines the salient features of those introduced in (Bagheri Hariri et al. 2013) and (Montali, Calvanese, and De Giacomo 2014). $\mu\mathcal{L}_p^{@}$ supports the full μ-calculus to predicate over the system dynamics. Recall that the μ-calculus is virtually the most expressive temporal logics: it subsumes LTL and CTL*. To query possibly different agent databases, $\mu\mathcal{L}_p^{@}$ adopts FO queries extended with location arguments (Montali, Calvanese, and De Giacomo 2014), which are dynamically bound to agents. Furthermore, to track the temporal evolution of data objects, $\mu\mathcal{L}_p^{@}$ adopts a controlled form of FO quantification across time: quantification is limited to those objects that *persist* in the system:

$$\Phi ::= Q_\ell \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\text{LIVE}_T(x) \wedge \Phi \mid Z \mid \mu Z.\Phi \mid \bigwedge_{i\in\{1,\ldots,n\}} \text{LIVE}_{T_i}(\vec{x_i}) \wedge \langle\!\rightarrow\!\rangle\Phi \mid \bigwedge_{i\in\{1,\ldots,n\}} \text{LIVE}_{T_i}(\vec{x_i}) \wedge [\!\rightarrow\!]\Phi$$

where Q_ℓ is a (possibly open) FO query with location arguments, in which the only constants that may appear are those in $\Delta_{0,\mathcal{F}}$, and Z is a second order predicate variable (of arity 0). Furthermore, the following assumption holds: in the $\langle\!\rightarrow\!\rangle$ and $[\!\rightarrow\!]$ cases, the variables $x_1, \ldots, x_n$ 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. Notice that the usage of LIVE can be safely substituted by an atomic positive query.

The semantics of $\mu\mathcal{L}_p^{@}$ is defined over a relational transition system similarly to the semantics of $\mu\mathcal{L}_p$ in (Bagheri Hariri et al. 2013). The most peculiar aspect is constituted by Q_ℓ, which allows one to dynamically inspect the databases maintained by active agents. In particular, Q_ℓ is a standard (typed) FO query, whose atoms have the form $R(\vec{x})@a$, where R is a (typed) relation, and a denotes an agent name. The evaluation of the atomic query $R(\vec{x})@a$ over a relational transition system Υ with substitution θ returns those states s of Υ such that:

- $a\theta$ is an active agent in s, that is, $Agent(a\theta) \in s.db(\mathsf{inst})$;
- the atomic query $R(\vec{x})\theta$ evaluates to true in the database instance that agent $a\theta$ has in state s, i.e., $ans(R(\vec{x})\theta, s.db(a\theta)) \equiv \mathsf{true}$.

Example 4.1. Consider the protocol in Section 3, assuming that inst uses a unary typed relation $inCritical$ to store the agent that is currently in the critical interaction. Given:

$$First(a) = \exists t.hasTicket@\mathsf{inst}(a, t) \wedge \neg\exists a', t'.hasTicket@\mathsf{inst}(a', t') \wedge a' \neq a \wedge t' < t,$$

$$\nu Z.(\forall a.Agent@\mathsf{inst}(a) \wedge First(a) \rightarrow \mu Y.(inCritical@\mathsf{inst}(a) \vee (Agent@\mathsf{inst}(a) \wedge \langle\!\rightarrow\!\rangle Y)) \wedge [\!\rightarrow\!]Z$$

models that when an agent is "first", there will be a run in which it persists into the system until it enters the critical interaction. ■

5 Decidability of Verification

We now study different aspects of the following *verification problem*: given a closed $\mu\mathcal{L}_p^{\mathbb{Q}}$ property Φ and an RMAS $\mathcal{X}$, check whether Φ holds over the relational transition system $\Upsilon_{\mathcal{X}}$, written $\Upsilon_{\mathcal{X}} \models \Phi$. Unsurprisingly, this problem in general is undecidable. In a recent series of works, verification of data-aware dynamic systems has been studied under the notion of *state-boundedness* (Bagheri Hariri et al. 2014), which, in the context of RMASs, can be phrased as follows. An RMAS $\mathcal{X}$ is *state-bounded* if, for every state s of $\Upsilon_{\mathcal{X}}$, the number of data objects stored in each agent database does not exceed a pre-defined bound.

As shown in previous work, state-boundedness still allows one to model systems that encounter infinitely many different data objects (and even agents) along their runs, provided that they do not accumulate in the same state. In our setting, this means that infinitely many different agents can interact, provided that at each time point only a bounded number of them is active (Montali, Calvanese, and De Giacomo 2014). In the asynchronous case, the boundedness assumption requires to put a threshold also on the maximum size of each message queue/buffer. In fact, by Theorem 3.1 we know that message queues/buffers can be encoded through special accessory relations in the agent databases, and state boundedness applies to such accessory relations as well.

(Montali, Calvanese, and De Giacomo 2014) have shown that verification of state-bounded DACMASs is decidable. We study now how data types impact on this.

Compilation of Facets. Facets can be eliminated, getting a *shallow-typed* RMAS, i.e., one using base facets only.

Theorem 5.1. *For every RMAS $\mathcal{X}$, there exists a corresponding shallow-typed RMAS $\widehat{\mathcal{X}}$ such that, for every $\mu\mathcal{L}_p^{\mathbb{Q}}$ property Φ, we have $\Upsilon_{\mathcal{X}} \models \Phi$ if and only if $\Upsilon_{\widehat{\mathcal{X}}} \models \Phi$.*

The crux of the proof is as follows. Whenever there is an n-ary typed relation $\overline{R}$ whose i-th component is typed with facet $F = \langle T, \varphi(x)\rangle$, we proceed by reducing F to the base facet $\langle T, \mathsf{true}\rangle$, and *compiling* the typing into a dedicated constraint: $\forall x_i.R(\_, \ldots, x_i, \ldots, \_) \rightarrow \varphi(x_i)$.

RMASs with the Successor Relation. We now show that including at least one data type with the successor relation compromises decidability:

Theorem 5.2. *Verification of a propositional reachability property over state-bounded, shallow-typed RMASs that use a single data type equipped with the successor relation is undecidable, even when the RMAS contains a single agent that uses unary relations only.*

The proof is by reduction from the halting problem of two-counter machines. Intuitively, a counter can be encoded in a simple state-bounded RMAS that employs a unary, typed relation $\overline{C}$ to store the integer value of the counter, and a 0-ary service call $\overline{\mathbf{input}}$ to insert a new value into the counter. Increment can be captured by the update action:

$$\textsc{inc-c}() : \left\{ \begin{array}{r} C(x) \rightsquigarrow \mathbf{del}\{C(x)\}, \mathbf{add}\{C_{old}(x), C(\mathbf{input}())\} \\ Op(x) \rightsquigarrow \mathbf{del}\{Op(x)\}, \mathbf{add}\{Op(0)\} \end{array} \right\}$$

that copies the old counter into relation C_{old}, and inserts into C the result of a service call. $Op(0)$ signifies that the current operation is an increment. Increment is ensured by adding the following constraint: $\forall x, y.Op(0) \wedge C_{old}(x) \wedge C(y) \rightarrow \mathsf{succ}(y, x)$. Decrement can be captured symmetrically, encoding the counter program as self-directed rules.

Densely-Ordered RMASs. Given the previous undecidability result, we focus on dense orders. A *densely-ordered* RMAS only relies on data types equipped with domain-specific equality $=$ and, possibly, total dense orders, as well as corresponding facets. For this class of RMASs, we have:

Theorem 5.3. *Verification of closed $\mu\mathcal{L}_p^{\mathbb{Q}}$ properties over state-bounded, densely-ordered RMASs is decidable, and reducible to conventional, finite-state model checking.*

We informally discuss the proof. First, we reformulate the input RMAS $\mathcal{X}$ into the shallow-typed version $\widehat{\mathcal{X}}$ of Theorem 5.1. We then consider the infinite-state transition system $\Upsilon_{\widehat{\mathcal{X}}}$, and seek a faithful (sound and complete) finite state abstraction of it. To do so, we proceed in two phases.

First of all, we get rid of the infinite-branching in $\Upsilon_{\widehat{\mathcal{X}}}$, by introducing a pruned transition system $\Lambda_{\widehat{\mathcal{X}}}$ that obeys the following properties: *(i)* $\Lambda_{\widehat{\mathcal{X}}}$ is finite-branching; *(ii)* for every closed $\mu\mathcal{L}_p^{\mathbb{Q}}$ property Φ, $\Upsilon_{\widehat{\mathcal{X}}} \models \Phi$ if and only if $\Lambda_{\widehat{\mathcal{X}}} \models \Phi$. To produce $\Lambda_{\widehat{\mathcal{X}}}$, we extend the notion of *equality commitment* exploited in (Bagheri Hariri et al. 2013) so as to account for dense linear orders. We call the extension *densely-ordered commitment*. The intuition is as follows. Consider e.g., an agent database with two active data objects 1 and 2.3, with the usual ordering $1 < 2.3$. Now consider the insertion of a service call result d. Noticing that d is obtained nondeterministically, and that $<$ is a dense total order, we have 5 possible *densely-ordered commitments*, according to all possible domain-specific relations holding between d and the two existing active values: (1) d is such that $\mathtt{d} < 1$; (2) $\mathtt{d} = 1$; (3) d is such that $1 < \mathtt{d} < 2.3$ (this is always possible, thanks to density); (4) $\mathtt{d} = 2.3$; (5) d is such that $2.3 < \mathtt{d}$. If more service calls are issued at the same time, all combinations must be considered, also accounting for how the service call results relate to each other. Obviously, there are in general infinitely many different concrete results that fall inside the same densely-ordered commitment. However, due to the fact that RMASs are constructed using domain-independent queries, and the observation that $\mu\mathcal{L}_p^{\mathbb{Q}}$ properties only quantify over the active domain, such properties are not able to distinguish different configurations of data objects that fall inside the same densely-ordered commitment. In the example above, no $\mu\mathcal{L}_p^{\mathbb{Q}}$ property would distinguish the cases where the service returns 1.5 from the one where the result is 2: both cases would fall inside the third commitment. Since in a given state the number of densely-ordered commitments is bounded by the agent specification and data, $\Lambda_{\widehat{\mathcal{X}}}$ is finite-branching.

Notice that $\Lambda_{\widehat{\mathcal{X}}}$ may still contain runs visiting infinitely many different states. The second phase consists therefore in producing a folded transition system $\Theta_{\widehat{\mathcal{X}}}$ that is finite-state, and such that for every closed $\mu\mathcal{L}_p^{\mathbb{Q}}$ property Φ, $\Lambda_{\widehat{\mathcal{X}}} \models \Phi$ if and only if $\Theta_{\widehat{\mathcal{X}}} \models \Phi$. The idea behind the construction of $\Theta_{\widehat{\mathcal{X}}}$ resembles the notion of *value recycling* introduced in (Bagheri Hariri et al. 2013). In spite of the fact that $\mu\mathcal{L}_p^{\mathbb{Q}}$

can employ domain-specific relations, it still can only apply them over data objects present in the active domain of the system. Furthermore, due to the persistent nature of $\mu\mathcal{L}_p^{\mathbb{Q}}$, it is also not possible to compare currently active data objects with objects that were encountered in the past, but are not active anymore. Therefore, starting from $\Lambda_{\widehat{\mathcal{X}}}$ we can collapse each rigid, domain-specific dense total order $<$ into a non-rigid $lessThan$ relation defined over the active data objects only, ensuring that all the initial finitely many data objects in $\Delta_{0,\mathcal{F}}$ are explicitly accounted and maintained active during the system execution. At the same time, when possible, a previously encountered data object `old` that is *not active anymore* can be recycled and used in place of a fresh data object `new`, but defining $lessThan$ over `old` as if it was `new`. This makes $\mu\mathcal{L}_p^{\mathbb{Q}}$ unable to distinguish `old` from `new` in the current state. By applying the recycling technique in (Bagheri Hariri et al. 2013) it can be shown that, when the RMAS is state-bounded, then only a bounded number of new data objects must be inserted before recycling makes it not necessary anymore to consider fresh values. Hence, the procedure always terminates, producing the finite transition system $\Theta_{\widehat{\mathcal{X}}}$ that satisfies the same $\mu\mathcal{L}_p^{\mathbb{Q}}$ properties as $\Upsilon_{\mathcal{X}}$.

6 Conclusion

RMASs constitute a very rich modeling framework for data-aware multiagent systems. The presence of concrete data types and their facets greatly empowers its modeling capabilities, making it, e.g., apt to capture mutual exclusion protocols, asynchronous interactions with bounded queues, and price-based protocols. Our key result, namely that densely-order, state-bounded RMASs are verifiable with standard model checking techniques, paves the way towards concrete verification algorithms for this class of systems (Lomuscio, Qu, and Raimondi 2009; Cavada et al. 2014). In this respect, a major obstacle is the exponentiality in the data slots that can be changed over time, which is inherent in all data-aware dynamic systems (Deutsch, Sui, and Vianu 2007). We intend to attack this by proposing data modularization techniques to decompose the system into smaller components.

From a foundational perspective, our work is related to (Belardinelli 2014), which extends the framework in (Belardinelli, Lomuscio, and Patrizi 2012) with types, so as to model and verify auctions. Our setting is incomparable with (Belardinelli 2014) w.r.t. the framework and the verification logic, and we intend to cross-transfer results between them. Finally, we plan to investigate connections between the considered framework and parameterized verification techniques for concurrent systems with data as those studied in (Delzanno and Rosa-Velardo 2013).

Acknowledgments

This research has been partially supported by the EU FP7-318338 IP project Optique (*Scalable End-user Access to Big Data*), and by the project *MAGIC: Managing Completeness of Data*, funded by the province of Bozen-Bolzano.

References

Bagheri Hariri, B.; Calvanese, D.; De Giacomo, G.; Deutsch, A.; and Montali, M. 2013. Verification of relational data-centric dynamic systems with external services. In *Proc. of the 32nd ACM SIGACT SIGMOD SIGAI Symp. on Principles of Database Systems (PODS)*, 163–174.

Bagheri Hariri, B.; Calvanese, D.; Deutsch, A.; and Montali, M. 2014. State-boundedness in data-aware dynamic systems. In *Proc. of the 14th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*. AAAI Press.

Baier, C., and Katoen, J.-P. 2008. *Principles of Model Checking*. The MIT Press.

Belardinelli, F.; Lomuscio, A.; and Patrizi, F. 2012. 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)*, 319–328.

Belardinelli, F. 2014. Model checking auctions as artifact systems: Decidability via finite abstraction. In *Proc. of the 21st Eur. Conf. on Artificial Intelligence (ECAI)*, 81–86.

Bultan, T.; Gerber, R.; and Pugh, W. 1999. Model-checking concurrent systems with unbounded integer variables: Symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems* 21(4):747–789.

Calvanese, D.; Delzanno, G.; and Montali, M. 2014. Verification of relational multiagent systems with data types (extended version). CoRR Technical Report arXiv:1411.4516, arXiv.org e-Print archive.

Cavada, R.; Cimatti, A.; Dorigatti, M.; Griggio, A.; Mariotti, A.; Micheli, A.; Mover, S.; Roveri, M.; and Tonetta, S. 2014. The nuXmv symbolic model checker. In *Proc. of the 26th Int. Conf. on Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, 334–342. Springer.

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

Delzanno, G., and Podelski, A. 1999. Model checking in CLP. In *Proc. of the Int. Conf. on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 223–239.

Delzanno, G., and Rosa-Velardo, F. 2013. On the coverability and reachability languages of monotonic extensions of petri nets. *TCS* 467:12–29.

Deutsch, A.; Sui, L.; and Vianu, V. 2007. Specification and verification of data-driven web applications. *J. of Computer and System Sciences* 73(3):442–474.

ISO/IEC 11404:2007. 2007. Information technology: General-Purpose Datatypes (GPD). Technical report, ISO/IEC, CH-1211 Geneva 20, Switzerland.

Lomuscio, A.; Qu, H.; and Raimondi, F. 2009. MCMAS: A model checker for the verification of multi-agent systems. In *Proc. of the 21st Int. Conf. on Computer Aided Verification (CAV)*, volume 5643 of *LNCS*, 682–688. Springer.

Montali, M.; Calvanese, D.; and De Giacomo, G. 2014. Verification of data-aware commitment-based multiagent systems. In *Proc. of the 13th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS)*, 157–164.

Montanari, U., and Pistore, M. 2005. History-dependent automata: An introduction. In *Proc. of the 5th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-Moby)*, volume 3465 of *LNCS*, 1–28. Springer.

Needham, R. 1989. *Distributed Systems*. Addison Wesley Publ. Co. chapter Names, 89–101.

Savkovic, O., and Calvanese, D. 2012. Introducing datatypes in *DL-Lite*. In *Proc. of the 20th Eur. Conf. on Artificial Intelligence (ECAI)*.