arXiv:1411.4516v1 [cs.AI] 17 Nov 2014

Verification of Relational Multiagent Systems with Data Types (Extended Version)

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 full-fledged relational database, and 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. Notably, 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 invocation of web services, or interaction with 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, 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. Notably, this allows us to model and verify state-bounded RMASs that include coordination mechanisms such as ticket-based mutual exclusion protocols.

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

Copyright © 2014, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

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:

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

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 $BF = \langle \langle \mathbb{B}, \{=\} \rangle, \mathsf{true} \rangle$ to refer to two base facets for agent and specification names respectively.

Example 2.2. An Enumeration $\mathtt{s_1}, \ldots, \mathtt{s_n}$ over string values can be modeled as facet $\langle \langle \mathbb{S}, \{=\} \rangle, \bigvee_{i \in \{1,\ldots,n\}} x = \mathtt{s_i} \rangle$. This also accounts for the type of boolean, which can be captured by $Bool = \langle \langle \mathbb{S}, \{=\} \rangle, x = \text{“}\mathtt{t}\text{”} \vee x = \text{“}\mathtt{f}\text{”} \rangle$. ■

Example 2.3. $\langle \langle \mathbb{R}, \{>, =\} \rangle, (x > \mathtt{0} \wedge \mathtt{18} > x) \vee x > \mathtt{65} \rangle$ 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.

In the following, we denote the i-th component of R as $R[i]$, and write $\textsc{type}_\mathcal{D}(R[i])$ to indicate the type associated by $\mathcal{D}$ to $R[i]$. We also denote the tuple of types associated by $\mathcal{D}$ to all components of R as $\textsc{type}_\mathcal{D}(R)$. To simplify readability, we also seldomly use notation $\overline{R}(F_1, \ldots, F_n)$ as a shortcut for $\overline{R} = \langle R/n, \langle F_1, \ldots, F_n \rangle \rangle$.

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

Giving 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

[1]Being infinite does not lead to a loss of generality, thanks to the notion of facet defined below.

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 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 2.4. 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 $PF = \langle\langle \mathbb{Q}, \{<, =\}\rangle, x > 0\rangle$. ■

Example 2.5. Given facet $AF = \langle\langle \mathbb{A}, \{=\}\rangle, \mathsf{true}\rangle$, service $\overline{\mathbf{getN}} = \langle \mathbf{getN}/0, \emptyset, AF\rangle$ returns agent names. ■

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 evoles 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. We assume that the schema is always equipped with a special unary relation $\overline{MyName}$, whose unique component is typed with AF, and that is used to keep track of the global name associated to the agent in the system.
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 superscript the name of the specification 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 $\textsc{live}_T(x)$ 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 $\textsc{live}$ 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 effects 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 "delete" 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-send) **on** $M(\vec{x})$ **to** t **if** $Q(\vec{y}_1)$ **then** $\alpha(\vec{y}_2)$, with $\vec{y}_1 \cup \vec{y}_2 \subseteq \vec{x} \cup \{t\}$, or
- (on-receive) **on** $M(\vec{x})$ **from** s **if** $Q(\vec{y}_1)$ **then** $\alpha(\vec{y}_2)$, with $\vec{y}_1 \cup \vec{y}_2 \subseteq \vec{x} \cup \{s\}$,

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 $\vec{y}_1$ and data objects in $\Delta_{0,\mathcal{F}}$; *(iii)* α is an update action in $\mathcal{A}$, whose parameters are bound to variables $\vec{y}_2$.

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 every other agent. Still, we assume that the institutional agent has special duties, such as in particular handling the creation of agents and their removal from the system, and maintainance of agent-related information, like the set of names for 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)* $\langle Spec/1, \langle BF\rangle\rangle$, to store specification names; *(iii)* $\langle hasSpec/2, \langle AF, BF\rangle\rangle$, 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 of dynamically creating and removing other agents. This can be encoded by readapting (Montali, Calvanese, and De Giacomo 2014). Details are given in the online appendix.

Agent creation/removal. Two actions are employed by the institutional agent to insert or remove an agent into/from the system. Their respective action schemas are $\textsc{newAg}(BF)$ and $\textsc{remAg}(AF)$. As for creation, inst employs the service call introduced in Example 2.5 to introduce a name into the $Agent$ relation, attaching to it the specification name passed as input. However, some additional modeling effort is needed so as to ensure that the introduced name is indeed new:

$$\textsc{newAg}(\mathsf{s}) : \left\{ \begin{array}{r} OldAg(a) \rightsquigarrow \textbf{del}\ \{OldAg(a)\} \\ FreshAg(a) \rightsquigarrow \textbf{del}\ \{FreshAg(a)\} \\ \mathsf{true} \rightsquigarrow \textbf{add} \left\{ \begin{array}{l} FreshAg(\mathbf{getN}()), \\ Agent(\mathbf{getN}()), \\ Spec(\mathbf{getN}(), \mathsf{s}) \end{array} \right\} \\ Agent(a) \rightsquigarrow \textbf{add}\{OldAg(a)\} \end{array} \right\}$$

Intuitively, apart from adding the new agent and attaching the corresponding specification, the action updates the two accessory agent relations $OldAd$ and $FreshAg$, which are assumed to be part of $\mathcal{D}^{\mathtt{ispec}}$, ensuring that in the next state $OldAd$ contains the set of agent names that were present in the immediately preceding state, and that $FreshAg$ contains the newly injected name. Freshness can then be guaranteed by adding a dedicated constraint to $\Gamma^{\mathtt{ispec}}$:

$$\forall a. OldAg(a) \wedge FreshAg(a) \rightarrow \mathsf{false}$$

Removal of an agent is instead simply modelled as:

$$\textsc{remAg}(\mathsf{a}) : \left\{ hasSpec(\mathsf{a}, s) \rightsquigarrow \textbf{del} \left\{ \begin{array}{l} Agent(\mathsf{a}), \\ hasSpec(\mathsf{a}, s) \end{array} \right\} \right\}$$

Update rules that employ these special actions obviously depend on the domain, by including specific on-send and on-receive rules in I.

2.5 Well-Formed Specifications

In an RMAS, every piece of information is typed. This immediately calls for a suitable notion of *well-formedness* that checks the compatibility of types in all agent specifications. 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 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.

We now formalize this intuition. Let $\mathcal{F}$ be a set facets, and $\mathcal{D}$ be a $\mathcal{F}$-typed database schema. Let Q be a FO query over $\mathcal{D}$ and $\mathcal{R}_{\mathcal{F}}$ that uses only constants in $\Delta_{0,\mathcal{F}}$. We say that Q is *$\mathcal{D}$-compatible* if: *(i)* whenever a data object from $\Delta_{0,\mathcal{F}}$ appears in component $R[i]$ inside Q, then it belongs to

$\textsc{type}_{\mathcal{D}}(R[i])$; *(ii)* whenever the same variable x appears in two components $R_1[i_1]$ and $R_2[i_2]$, then $\textsc{type}_{\mathcal{D}}(R_1[i_1]) = \textsc{type}_{\mathcal{D}}(R_2[i_2])$.

By definition of compatibility, each free variable of a $\mathcal{D}$-compatible query is associated to a single facet/data type. This allows us to characterize the "output types" of a query, that is, the types associated to its free variables (and hence also the types of its answer components). Given an $\mathcal{F}$-typed database schema $\mathcal{D}$ and a well-formed FO query $Q(\vec{x})$ over $\mathcal{D}$ and $\mathcal{R}_{\mathcal{F}}$ that uses only constants in $\Delta_{0,\mathcal{F}}$, the *output-type of $x_i \in \vec{x}$ according to Q*, written $\textsc{out-type}_Q(x_i)$, is the unique data type in $\mathcal{F}$ to which x_i is associated by Q, where $\mathcal{T}$ is the set of data types on which $\mathcal{F}$ is defined. We extend the notion of output-type to a tuple of variables $\vec{x}' = \langle x_i, \ldots, x_k \rangle \subseteq \vec{x}$ with $1 \leq i \leq k \leq n$, writing $\textsc{out-type}_Q(\vec{x}')$ as a shortcut for the tuple $\langle \textsc{out-type}_Q(x_i), \ldots, \textsc{out-type}_Q(x_k) \rangle$. We also write $\textsc{out-type}(Q)$, as a shortcut for $\textsc{out-type}_Q(\vec{x})$. Notice that, when applied to an atomic query, this notion corresponds exactly to the typing of the corresponding relation, according to its schema.

Given an RMAS $\mathcal{X} = \langle \mathcal{T}, \mathcal{F}, \Delta_{0,\mathcal{F}}, \mathcal{S}, \mathcal{M}, \mathcal{G}, I \rangle$ and an agent specification $\mathcal{N} = \langle \mathtt{n}, \mathcal{D}, \Gamma, D_0, \mathcal{C}, \mathcal{A}, \mathcal{U} \rangle$ in $\mathcal{G} \cup \{I\}$, we say that:

- $\mathcal{C}$ is *well-formed* if each of its communicative rules $Q(t, \vec{x})$ **enables** $M(\vec{x})$ **to** t is such that *(i)* $\textsc{out-type}_Q(t) = \mathbb{A}$ (i.e., Q binds t to an agent name), and *(ii)* $\textsc{out-type}(Q) = \textsc{type}_{\mathcal{M}}(M)$ (i.e., the payload is instantiated by Q in a compatible way with the types of message M).
- $\mathcal{A}$ is *well-formed* if each of its actions is well-formed. We say in turn that action $\langle \overline{\alpha}, \alpha_{spec} \rangle$ is *well-formed* if every effect $Q(\vec{\mathsf{p}}, \vec{x}) \rightsquigarrow$ **add** A, **del** D in α_{spec} is such that:
  - Q is $\mathcal{D}$-compatible.
  - Whenever a parameter p is mentioned in Q, the type to which p is assigned by $\textsc{out-type}(Q)$ is the same to which p is assigned by $\overline{\alpha}$.
  - For every n-ary typed relation $\overline{R} \in \mathcal{D}$, every fact F of R appearing in D, and for each $i \in \{1, \ldots, n\}$: *(i)* if the i-th position of F contains a data object, then such data object belongs to the domain of $\textsc{type}_{\mathcal{D}}(R[i])$; *(ii)* if the i-th position of F contains a variable $y \in \vec{x}$, then $\textsc{out-type}_Q(y) = \textsc{type}_{\mathcal{D}}(R[i])$.
  - For every n-ary typed relation $\overline{R} \in \mathcal{D}$, every fact F of R appearing in A, and for each $i \in \{1, \ldots, n\}$: *(i)* if the i-th position of F contains a data object, then such data object belongs to the domain of $\textsc{type}_{\mathcal{D}}(R[i])$; *(ii)* if the i-th position of F contains a variable $y \in \vec{x}$, then $\textsc{out-type}_Q(y) = \textsc{type}_{\mathcal{D}}(R[i])$; *(iii)* if the i-th position of F contains a k-ary service call $f(\vec{y})$ with $\vec{y} \subseteq \vec{x}$ and $\langle f/k, \mathcal{F}^{in}, F^{out} \rangle \in \mathcal{S}$, then $\textsc{out-type}_Q(\vec{y}) = \mathcal{F}^{in}$ and $F^{out} = \textsc{type}_{\mathcal{D}}(R[i])$.
- $\mathcal{U}$ is *well-formed* if all its update rules are well-formed. We discuss the case of on-send rules - the definition of well-formedness is identical for on-receive rules. An on-send rule in $\mathcal{U}$ of the form **on** $M(\vec{x})$ **to** t **if** $Q(\vec{y}_1)$ **then** $\alpha(\vec{y}_2)$, with $\vec{y}_1 \cup \vec{y}_2 \subseteq \vec{x} \cup \{t\}$, is *well-formed* if the following conditions hold: *(i)* if $t \in \vec{y}_1$, then $\textsc{out-type}_Q(t) = \mathbb{A}$; *(ii)* if t appears in the i-th component of α, then $\overline{\alpha}$ assigns type $\mathbb{A}$ to its i-th parameter; *(iii)* for each variable $x \in \vec{x} \cap \vec{y}_1$, such that x appears in the i-th component of M, we have that $\textsc{type}_{\mathcal{M}}(M[i]) = \textsc{out-type}_Q(x)$; *(iv)* for each variable $x \in \vec{x} \cap \vec{y}_2$, such that x appears in the i-th component of M and in the j-th component of α, we have that $\overline{\alpha}$ assigns type $\textsc{type}_{\mathcal{M}}(M[i])$ to its j-th parameter.
- $\mathcal{N}$ itself is *well-formed* if $\mathcal{C}$, $\mathcal{A}$ and $\mathcal{U}$ are all well-formed.

Finally, we say that the entire RMAS $\mathcal{X}$ is *well-formed* if all agents specifications in $\mathcal{G} \cup \{I\}$ are well-formed.

It is easy to see that checking whether an RMAS is well-formed requires linear time in the size of the specification.

From now on, we always assume that RMASs are well-formed. It is important to 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 managed at runtime, by dynamically handling facet violations (cf. Section 4).

3 Modeling with RMAS

We briefly show 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). 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, namely how to model a form of *contract net protocol* in RMASs, is provided in Section 3.2. 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, since message queues for asynchronous communication can be modelled as special typed relations in the agent databases:

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

Proof. We consider a form of reliable, asynchronous communication based on message buffers. In particular, the model works as follows:

- Messages sent by an agent to itself are processed immediately (in fact, there is no effective communication in this case).
- Whenever a sender agent emits a message with payload targeting another agent, the message is atomically inserted into a message buffer attached to the target agent.
- The target agent asynchronously reacts to the message by extracting it from the buffer (this could happen much later).
- We consider two variations of this general model: one in which the buffer is ordered (i.e., it is a queue), and one in which the buffer is just a set of messages. Both models are interesting, because they reflect different assumptions on the asynchronous communication model. In fact, the

first guarantees that the order in which messages are processed by the target follows the order in which messages where emitted (possibly by different agents). We call this communication model *asynchronous, ordered* (AO for short), and use acronym AO-RMAS for an RMAS adopting the AO communication model. Contrariwise, the second model accommodates the situation in which the order in which messages are received (i.e., processed) by a target agent does not necessarily reflect the order in which such messages were emitted. We call this communication model *asynchronous, disordered* (AD for short), and use acronym AD-RMAS for an RMAS adopting the AD communication model.

We prove that these asynchronous communication models can be both accommodated by a synchronous RMAS that employs accessory data structures in the agent schemas, specifically tailored to buffer messages and decouple the emission of a message from its processing by the target agent.

Given an AD-RMAS/AO-RMAS $\mathcal{X} = \langle \mathcal{T}, \mathcal{F}, \Delta_{0,\mathcal{F}}, \mathcal{S}, \mathcal{M}, \mathcal{G}, I \rangle$, we convert it into a standard, synchronous RMAS $\mathcal{X}_s = \langle \mathcal{T}, \mathcal{F}, \Delta_{0,\mathcal{F}}, \mathcal{S}_s, \mathcal{M}_s, \mathcal{G}_s, I_s \rangle$, where $\mathcal{M}_s$ and $\mathcal{S}_s$ just extend $\mathcal{M}$ and $\mathcal{S}$ with an additional message/service as illustrated below, and where each agent specification $\mathcal{N} = \langle \mathtt{n}, \mathcal{D}, \Gamma, D_0, \mathcal{C}, \mathcal{A}, \mathcal{U} \rangle$ in $\mathcal{G} \cup \{I\}$ becomes a corresponding agent specification $\mathcal{N}_s = \langle \mathtt{n}, \mathcal{D}_s, \Gamma_s, D_0, \mathcal{C}_s, \mathcal{A}_s, \mathcal{U}_s \rangle$ in $\mathcal{G}_s \cup \{I_s\}$, according to the translation mechanism illustrated in the following (notice that we are interested here in the correctness of the encoding, not in its efficient implementation; effective ways of realizing the translation can be provided by using this encoding as a basis).

Let us first focus on the database schema of the agent specification. We set $\mathcal{D}_s = \mathcal{D} \cup \{\overline{\mathit{MBuffer}}, \overline{\mathit{NewM}}, \overline{\mathit{OldM}}\}$, where $\overline{\mathit{MBuffer}}$ is a global buffer tracking incoming messages that have been received by the agent but still needs to be (asynchronously) processed, while $\overline{\mathit{NewM}}$ and $\overline{\mathit{OldM}}$ are unary accessory relations used to manage the generation of new identifiers for messages to be enqueued. The management of such identifiers closely resembles name management as discussed for the institutional agent.

Specifically, $\overline{\mathit{MBuffer}}$ contains a numeric primary key, and internalizes the payload schemas of all message relations in $\mathcal{M}$, plus an additional component to track the sender agent, and a boolean component indicating whether the payload has a valid content. A tuple in $\overline{\mathit{MBuffer}}$ contains a message identifier and sets exactly one of such boolean components to true, leaving the others false. This indicates what is the type of the buffered message, and that the corresponding payload/sender components contain the actual message payload and sender agents, whereas all other payload/sender components contain meaningless values. For this latter aspect, we assume, without loss of generality, that all data types are equipped with an $\mathtt{undefined}$ data object.

Technically, we fix an ordering over $\mathcal{M}$, that is, a bijection

$$msg : \{1, \dots, |\mathcal{M}|\} \longrightarrow \mathcal{M}$$

and fix the function $index = msg^{-1}$. We set the arity of $\overline{\mathit{MBuffer}}$ to $1 + \sum_{i=1}^{|\mathcal{M}|}(2 + a_i)$, where a_i is the arity of relation $msg(i)$. We make use of the following three specific types: *(i)* the Bool facet (cf. Example 2.2), *(ii)* the RF facet, defined as $\langle\langle \mathbb{R}, \{=, <\}\rangle, \mathsf{true}\rangle$, and *(iii)* the facet AF for agent names. Specifically, we type component $\mathit{MBuffer}[1]$ (the relation primary key) with RF. For each $i \in \{1, \dots, |\mathcal{M}|\}$, we type component $\mathit{MBuffer}\left[2 + \sum_{j=1}^{i-1}(2 + a_j)\right]$ with Bool, and component $\mathit{MBuffer}\left[3 + \sum_{j=1}^{i-1}(2 + a_j)\right]$ with AF, where a_i is the arity of $msg(i)$. Furthermore, for each $i \in \{1, \dots, |\mathcal{M}|\}$ and for every $k \in \{1, \dots, a_i\}$ (a_i being the arity of $msg(i)$), we set the type of component $\mathit{MBuffer}\left[3 + \sum_{j=1}^{i-1}(2 + a_j) + k\right]$ to be the same as the type of component $msg(i)[k]$.

Unary relations $\overline{\mathit{NewM}}$ and $\overline{\mathit{OldM}}$ are respectively used to store newly created or already existing message identifiers. Their unique component is consequently typed with RF.

Let us now consider the database constraints. We set $\Gamma_s = \Gamma \cup \{\Phi_{msgId}\}$, where Φ_{msgId} is a constraint ensuring that new message identifiers do not clash with already existing identifiers, and whose specific shape depend on whether the original RMAS is asynchronous ordered or unordered. In particular:

- if $\mathcal{X}$ is an AD-RMAS, then Φ_{newMsg} is
  $$\forall id_n, id_o. \mathit{NewM}(id_n) \wedge \mathit{OldM}(id_o) \rightarrow id_o \neq id_n$$
  (where $id_o \neq id_n$ is an abbreviation for $\neg(id_o = id_n)$).
- if $\mathcal{X}$ is an AO-RMAS, then Φ_{newMsg} is
  $$\forall id_n, id_o. \mathit{NewM}(id_n) \wedge \mathit{OldM}(id_o) \rightarrow id_o < id_n$$
  In fact, for an ordered RMAS, a newly created message must be enqueued after all pending messages that were enqueued before.

We now focus on the behavior of $\mathcal{N}_s$, that is, on how the rules of $\mathcal{N}$ are translated into corresponding rules in $\mathcal{N}_s$ so as to simulate the asynchronous communication model on top of a synchronous communication model. Since asynchronous communication requires to decouple the emission of a message from the reaction of the target agent, $\mathcal{U}_s$ only maintains the on-send rules of $\mathcal{U}$, replacing the on-receive rules with other on-receive rules. This new on-receive rules are organized in two groups. The first group of rules is just used to insert message received from other agents into the buffer. In particular, for each $i \in \{1, \dots, |\mathcal{M}|\}$, $\mathcal{U}_s$ contains a rule of the form

$$\textbf{on } M_i(\vec{x}) \textbf{ from } s \textbf{ if } \neg \mathit{MyName}(s) \textbf{ then } \textsc{buffer}_{M_i}(\vec{x}, s)$$

where M_i is the name of relation $msg(i)$, and $\textsc{buff}_{M_i}$ is a specific update action in $\mathcal{A}_s$, dedicated to insert the payload and sender agent of a message M_i into the buffer. In particular, $\textsc{buff}_{M_i}(\vec{\mathsf{x}}, \mathsf{s})$ is defined as:

$$\left\{\begin{array}{l} \mathit{OldM}(m) \rightsquigarrow \textbf{del}\{\mathit{OldM}(m)\} \\ \mathit{NewM}(m) \rightsquigarrow \textbf{del}\{\mathit{NewM}(m)\} \\ \mathsf{true} \rightsquigarrow \textbf{add} \left\{\begin{array}{l} \mathit{NewM}(\mathbf{getRN}()), \\ \mathit{MBuffer}(\mathbf{getRN}(), \dots, \underbrace{\text{“}\mathtt{t}\text{”}}_{\text{i-th component}}, \mathsf{p}, \vec{\mathsf{x}}, \dots) \end{array}\right\} \\ \mathit{MBuffer}(m, \_, \dots, \_) \rightsquigarrow \textbf{add}\{\mathit{OldM}(m)\} \end{array}\right\}$$

where $\overline{\mathbf{getRN}}$ is a service that returns a RF data object, and in the addition of the tuple $\mathit{MBuffer}(\mathbf{getRN}(), \ldots, \text{“}\mathtt{t}\text{”}, \mathsf{p}, \vec{\mathsf{x}}, \ldots)$, attributes “t”, $\mathsf{p}, \vec{\mathsf{x}}$ are inserted in those positions corresponding to the boolean component, sender agent component, and payload components dedicated to $msg(i)$, while all the other boolean components are set to “f”, and all remaining components are set to undef.

The processing of a buffered message is triggered by a special communicative rule that is contained in $\mathcal{C}_s$ together with all the original rules in $\mathcal{C}$. The purpose of the communicative rule is to extract a message from the buffer, triggering the agent to process it whenever the original specification contained on-receive rules dedicated to this. This is done by self-sending a message $nextM$ Specifically:

- If $\mathcal{X}$ is an AD-RMAS, the message extraction rule is:

  $\mathit{MyName}(a)$
  $\wedge\ \mathit{MBuffer}(m, \_, \ldots, \_)$ **enables** $nextM(m)$ **to** a

  Indeed, for a disordered RMAS, the order in which messages are received is non-deterministic. This rule mimics such a nondeterminism, since the agent nondeterministically picks one of the buffered messages.
- If $\mathcal{X}$ is an AO-RMAS, the message extraction rule is:

  $\mathit{MyName}(a) \wedge \mathit{MBuffer}(m, \_, \ldots, \_)$
  $\wedge\neg(\exists m_2.\mathit{MBuffer}(m_2, \_, \ldots, \_) \wedge m_2 < m)$
  enables $nextM(m)$ **to** a

  Indeed, for an ordered RMAS, messages are deterministically received according to the order in which they have been sent. This rule mimics such a determinism by following a FIFO policy, picking the first message in the queue. Recall that, for AO-RMAS, whenever a new message is inserted into the queue, its primary key is greater than the primary keys of already enqueued messages.

The last dimension to be covered is the agent reaction to a message to be processed. This is done by suitably reformulating the original on-receive rules present in $\mathcal{U}$. Specifically, for each on-receive rule

on $M(\vec{x})$ **from** s **if** $Q(\vec{y}_1)$ **then** $\alpha(\vec{y}_2)$

in $\mathcal{U}$, with $\vec{y}_1 \cup \vec{y}_2 \subseteq \vec{x} \cup \{s\}$, $\mathcal{U}_s$ contains a corresponding on-receive rule (which, by construction of $\mathcal{X}_s$, is triggered only by the agent itself)

on $nextM(m)$ **from** a
if $\mathit{MyName}(a) \wedge \Phi_M(m, \vec{y}_1, \vec{y}_2) \wedge Q(\vec{y}_1)$ **then** $\alpha(m, \vec{y}_2)$

where $\Phi_M(m, s, \vec{x})$ is a query that: *(i)* checks whether the identifier m points to a tuple in the buffer that actually refers to a message of type M (this can be done by checking whether the boolean component in position $index(M)$ is set to “t”); *(ii)* if so, extracts the sender of message m, and its payload $\vec{x}$. Technically, the query is simply formulated as:

$$\Phi_M(m, s, \vec{x}) = \mathit{MBuffer}(m, \_, \ldots, \underbrace{\text{“}\mathtt{t}\text{”}}_{index(M)\text{-th component}}, s, \vec{x}, \ldots)$$

A final, additional update rule that always triggers when a $nextM$ message is received is needed to properly update the buffer, by removing the processed message:

on $nextM(m)$ **from** a **if** $\mathit{MyName}(a)$ **then** REMOVEM(m)

where:

REMOVEM(m) : $\{\mathit{MBuffer}(\mathsf{m}, \vec{x}) \rightsquigarrow$ **del**$\{\mathit{MBuffer}(\mathsf{m}, \vec{x})\}\}$

By putting everything together, if we project away the accessory relations $\mathit{MBuffer}$, OldM and NewM, we obtain that the asynchronous execution semantics of $\mathcal{X}$ under both the ordered and disordered assumption exactly corresponds to that of $\mathcal{X}_s$ under the standard synchronous semantics, as precisely defined in Figure 1. □

The proof of Theorem 3.1 already gives a glimpse about the modelling power of RMASs equipped with ordered types. We next discuss how these features can be exploited to easily capture mutual exclusion protocols based on tickets.

3.1 Ticket-Based Mutual Exclusion Protocols

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 inst, 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 $\mathit{Assigned}$ relation is empty (see below):

on ASKTICKET() **from** a
if $\neg \mathit{HasTicket}(a, \_) \wedge \neg \mathit{Assigned}(\_, \_)$ **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 $\mathit{Assigned}$, tracing that the ticket has been assigned but the corresponding agent still needs to be informed.

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

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:

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

An assigned ticket must be sent to the requestor agent:

$\mathit{Assigned}(t, a)$ **enables** GIVETICKET(t) **to** a

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

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

$$\text{BINDTICKET}(\mathsf{a}, \mathsf{t}) : \left\{\begin{matrix} \mathsf{true} \rightsquigarrow \textbf{del}\ \{\mathit{Assigned}(\mathsf{a}, \mathsf{t})\} \\ \mathsf{true} \rightsquigarrow \textbf{add}\{\mathit{hasTicket}(\mathsf{a}, \mathsf{t})\} \end{matrix}\right\}$$

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 $\textsc{cMsg}(\vec{p}, t)$ **from** a
if $hasTicket(a,t) \land \neg(\exists a', t'.hasTicket(a',t') \land t > t')$
then $\textsc{cAct}(a, \vec{p})$

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

3.2 Contract Net

We show how the classical contract net protocol (Smith 1980) can be easily accommodated in our framework. This can be considered as an example of a "price-based" protocol, and therefore indirectly shows how different kinds of auctions could be modelled as well, as, e.g., done in (Belardinelli 2014).

An RMAS that incorporates the contract net protocol contains two agent specifications (that can be obviously enriched and extended on a per-domain basis): the specification of an *initiator* agent, and the specification of a *participant* agent. The first specification is embodied by an agent that is interested in delegating the execution of a task to another agent, so as to achieve a desired goal. The second specification is embodied by agents that have the capabilities and the interest in executing the task, provided that they get back a reward.

The system employs the following FIPA-like messages:

- $\overline{cfp}(SF)$ (from the initiator to participants) – a call-for-proposal related to the execution of the provided task (for simplicity, we use strings to represent tasks, and we assume that the task name is used also as a *conversation identifier*);
- $\overline{propose}(SF, PF)$ (from a participant to the initiator), with PF as in Example 2.4 – a proposal to execute the task indicated in the first parameter, for the price indicated in the second parameter;
- $\overline{reject}(SF)$ (from the initiator to a participant) – rejection of all proposals for the specified task;
- $\overline{accept}(SF, PF)$ (from the initiator to a participant) – acceptance of a proposal;
- $\overline{inform}(SF)$ (from a participant to the initiator) – notification that the task has been executed.
- $\overline{failure}(SF)$ (from a participant to the initiator) – notification that the task execution failed.

Let us focus on the realization of the protocol from the point of view of inst, which acts as the initiator. We first introduce the relations used by inst to run the protocol:

- $\overline{Agent}(AF)$ lists the (names of) agents known to the initiator agent; if the initiator agent is inst, then it already holds all agents present in the system, otherwise the initiator agent can engage in a preliminary interaction with inst and/or other agents to collect such names.
- $\overline{Task}(SF, StateF)$ lists the task names that the initiator agent is interested to assign, i.e., those that can become the subject of an instance of the contract net protocol. $StateF = \langle\langle \mathbb{S}, \{=\}\rangle, x = \text{"}\mathtt{todo}\text{"} \lor x = \text{"}\mathtt{assigned}\text{"} \lor x = \text{"}\mathtt{done}\text{"}\rangle$ is an enumerative facet used to track the state of each task – the three states are self-explanatory.
- $\overline{Contacted}(AF, SF)$ lists those agents that have been already contacted for a given task.
- $\overline{PropPrice}(AF, SF, PF)$ lists those agents that answered to a proposal with a certain price.
- $\overline{AssignedTo}(AF, SF, PF)$ lists those tasks that have been assigned to an agent for a given price.

We have now all the ingredients to model the behavioral rules of the initiator agent. First of all, the initiator agent can issue a call-for-proposal for any task in the "todo" state, directed towards an eligible agent. This is captured by the communicative rule:

$Task(t, \text{"}\mathtt{todo}\text{"}) \land Agent(a)$
$\land\, \Phi_{sui}(a,t) \land \neg Contacted(a,t)$ **enables** $cfp(t)$ **to** a

where $\Phi_{sui}(a,t)$ is a boolean query that checks whether a is a suitable agent for executing t, and that does so by possibly involving additional relations maintained by the initiator agent for this specific purpose. An agent is considered eligible if it is suitable and has not been already contacted for the selected task.

The initiator agent reacts to this message by indicating that agent a has been contacted for task t:

on $cfp(t)$ **to** a **if** true **then** $\textsc{markContacted}(a,t)$

where

$\textsc{markContacted}(\mathsf{a},\mathsf{t}) : \{\mathsf{true} \rightsquigarrow \textbf{add}\{Contacted(\mathsf{a},\mathsf{t})\}\}$

When a proposer agent sends back a proposal, the initiator agent stores it into the $\overline{PropPrice}$ relation:

on $propose(t,p)$ **from** s **if** true **then** $\textsc{setProposal}(s,t,p)$

where

$\textsc{setProposal}(\mathsf{s},\mathsf{t},\mathsf{p}) : \{\mathsf{true} \rightsquigarrow \textbf{add}\{PropPrice(\mathsf{s},\mathsf{t},\mathsf{p})\}\}$

Notice that this formalization seamlessly enables the same agent to make different proposals for the same task, but can be easily modified so as to account for the situation where only one proposal per agent can be accepted.

The presence of at least one registered proposal enables the initiator to assign the task to some agent, provided that such an agent made the best proposal, i.e., that with the lowest price. Notice that the initiator is free to choose *when* to accept, and can decide to contact further agents before actually selecting the best proposal.

$PropPrice(a,t,p) \land \neg(\exists p_2.PropPrice(\_,t,p_2) \land p_2 < p)$
enables $accept(t,p)$ **to** a

When the initiator decides to actually accept the best offer, it reacts by tracking to which agent the task has been assigned (and with wich price), taking also care of properly updating the task state, as well as to clean the $PropPrice$ relation. This is done through two different rules. The task assignment is handled by rule

on $accept(t,p)$ **to** a **if** true **then** $\textsc{markAssigned}(a,t,p)$

where $\textsc{markAssigned}(\mathsf{a},\mathsf{t},\mathsf{p})$:

$$\left\{\begin{array}{ll}\mathsf{true} & \rightsquigarrow \textbf{add}\,\{\mathit{AssignedTo}(\mathsf{a},\mathsf{t},\mathsf{p})\}\\ \mathit{PropPrice}(\mathsf{a},\mathsf{t},p_a) & \rightsquigarrow \textbf{del}\ \ \{\mathit{PropPrice}(\mathsf{a},\mathsf{t},p_a)\}\end{array}\right\}$$

The task state update is instead managed by rule

on $accept(t,p)$ **to** a **if** true **then** $\textsc{setState}(t,$ "assigned")

where $\textsc{setState}(\mathsf{t},\mathsf{state})$ is a generic state-update action formalized as follows:

$$\left\{\begin{array}{ll}\mathit{Task}(\mathsf{t},\mathsf{oldstate}) \rightsquigarrow & \textbf{del}\ \ \{\mathit{Task}(\mathsf{t},\mathsf{oldstate})\}\\ & \textbf{add}\ \{\mathit{Task}(\mathsf{t},\mathsf{state})\}\end{array}\right\}$$

The acceptance of an offer enables the initiator to send a rejection to all the agents that made an offer but were not selected:

$\mathit{PropPrice}(a,t,\_) \land \neg(\mathit{AssignedTo}(a,t,\_))$
enables $reject(t)$ **to** a

To track that a rejection has been sent, the initiator reacts to the rejection message by removing all proposals registered for the corresponding agent and task:

on $reject(t)$ **to** a **if** true **then** $\textsc{remProps}(a,t)$

where $\textsc{remProps}(\mathsf{a},\mathsf{t})$:

$$\big\{\mathit{PropPrice}(\mathsf{a},\mathsf{t},p) \rightsquigarrow \textbf{del}\{\mathit{PropPrice}(\mathsf{a},\mathsf{t},p)\}\big\}$$

Finally, an assigned task is marked as "done" whenever the corresponding agent informs the initiator that the task has been executed, or brought back to the "todo" state if the agent signals a failure. These two cases are respectively handled by the two on-receive rules

on $inform(t)$ **from** a
if $\mathit{AssignedTo}(a,t,\_)$ **then** $\textsc{setState}(t,$ "done")

and

on $inform(t)$ **from** a
if $\mathit{AssignedTo}(a,t,\_)$ **then** $\textsc{setState}(t,$ "todo")

which reuse the action SETSTATE as defined above. The case of a failure allows the initiator agent to restart a contract net protocol for the non-executed task.

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 n of an agent active in s, returns the database of 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 name of nspecification adopted by 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 given in Figure 1. 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 a is nondeterministically picked selecting its name from $s.db(\mathsf{inst})$. (3) The communicative rules of 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, it 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 $\mu Ł_p^@$ Verification Logic. To specify sophisticated properties over RMASs we employ the $\mu Ł_p^@$ logic. This logic combines the salient features of those introduced in (Bagheri Hariri et al. 2013) and (Montali, Calvanese, and De Giacomo 2014). $\mu Ł_p^@$ 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, $\mu Ł_p^@$ 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, $\mu Ł_p^@$ adopts a controlled form of FO quantification across time: quantification is limited to those objects that *persist* in the system:

$$\Phi ::= Q_\ell \mid \neg\Phi \mid \Phi_1 \land \Phi_2 \mid \exists x.\textsc{live}_T(x) \land \Phi \mid Z \mid \mu Z.\Phi \mid$$
$$\textstyle\bigwedge_{i\in\{1,\ldots,n\}} \textsc{live}_{T_i}(\vec{x_i}) \land \langle - \rangle\Phi \mid \bigwedge_{i\in\{1,\ldots,n\}} \textsc{live}_{T_i}(\vec{x_i}) \land [-]\Phi$$

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 $\langle - \rangle$ and $[-]$ 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 $\mu Ł_p^@$ is defined over a relational transition system similarly to the semantics of $\mu Ł_p$ in (Bagheri Hariri et al. 2013). The most peculiar aspect is

```
**procedure** BUILD-TS($\mathcal{X}$)
**input:** RMAS $\mathcal{X} = \langle \mathcal{T}, \mathcal{F}, \Delta_{0,\mathcal{F}}, \mathcal{S}, \mathcal{M}, \mathcal{G}, I \rangle$, **output:** Transition system $\Upsilon_{\mathcal{X}} = \langle \Delta_T, \Sigma, s_0, \rightarrow \rangle$
  $AS_0 := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec_n}) \in D_0^{\mathsf{inst}}\}$    ▷ Initial agents with their specifications
  **for all** $\langle \mathtt{n}, \mathtt{spec_n} \rangle \in AS_0$ **do** $s_0.db(\mathtt{n}) := D_0^{\mathsf{spec_n}}$    ▷ Specify the initial state by extracting the initial DBs from the agent specs
  $\Sigma := \{s_0\}, \rightarrow := \emptyset$
  **while** true **do**
    **pick** $s \in \Sigma$    ▷ Nondepickterministically pick a state
    $CurAS := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{a}, \mathtt{spec_n}) \in s.db(\mathsf{inst})\}$    ▷ Get currently active agents with their specifications
    **pick** $\langle \mathtt{a}, \mathtt{spec_a} \rangle \in CurAS$    ▷ Nondeterministically pick an active agent $\mathtt{a}$, elected as “sender”
    $EMsg :=$ GET-MSGS$(\mathcal{C}^{\mathsf{spec_a}}, s.db(\mathtt{a}), CurAS)$    ▷ Get the enabled messages with target agents
    **if** $EMsg \neq \emptyset$ **then**
      **pick** $\langle M(\vec{\mathtt{o}}), \mathtt{b} \rangle \in EMsg$, with $\langle \mathtt{b}, \mathtt{spec_b} \rangle \in CurAS$ ▷ Pick a message+target agent and trigger message exchange and reactions
      $ACT_{\mathtt{a}} := \emptyset$, $ACT_{\mathtt{b}} := \emptyset$    ▷ Get the actions with actual parameters to be applied by $\mathtt{a}$ and $\mathtt{b}$
      **for all** matching on-send rules “**on** $M(\vec{x})$ **to** $t$ **if** $Q(t, \vec{x})$ **then** $\alpha(t, \vec{x})$” in $\mathcal{U}^{\mathsf{spec_a}}$ **do**
        **if** $ans(Q(\mathtt{b}, \vec{\mathtt{o}}), s.db(\mathtt{a}))$ and $\alpha(\mathtt{b}, \vec{\mathtt{o}})$ conforms to $\overline{\alpha} \in \mathcal{A}^{\mathsf{a}}$ **then** $ACT_{\mathtt{a}} := ACT_{\mathtt{a}} \cup \alpha(\mathtt{b}, \vec{\mathtt{o}})$
      **for all** matching on-receive rules “**on** $M(\vec{x})$ **from** $s$ **if** $Q(s, \vec{x})$ **then** $\alpha(s, \vec{x})$” in $\mathcal{U}^{\mathsf{spec_b}}$ **do**
        **if** $ans(Q(\mathtt{a}, \vec{\mathtt{o}}), s.db(\mathtt{b}))$ and $\alpha(\mathtt{a}, \vec{\mathtt{o}})$ conforms to $\overline{\alpha} \in \mathcal{A}^{\mathsf{b}}$ **then** $ACT_{\mathtt{b}} := ACT_{\mathtt{b}} \cup \alpha(\mathtt{a}, \vec{\mathtt{o}})$
      $\langle ToDel^{\mathsf{a}}, ToAdd_s^{\mathsf{a}} \rangle :=$ GET-FACTS$(\mathcal{X}, s.db(\mathtt{a}), ACT_{\mathtt{a}})$, $\langle ToDel^{\mathsf{b}}, ToAdd_s^{\mathsf{b}} \rangle :=$ GET-FACTS$(\mathcal{X}, s.db(\mathtt{b}), ACT_{\mathtt{b}})$
      $DB_s^{\mathsf{a}} := (s.db(\mathtt{a}) \setminus ToDel^{\mathsf{a}}) \cup ToAdd_s^{\mathsf{a}}$    ▷ Calculate new $\mathtt{a}$’s DB, still with service calls to be issued
      $DB_s^{\mathsf{b}} := (s.db(\mathtt{b}) \setminus ToDel^{\mathsf{b}}) \cup ToAdd_s^{\mathsf{b}}$    ▷ Calculate new $\mathtt{b}$’s DB, still with service calls to be issued
      **if** for each $\mathtt{f}(\vec{\mathtt{o}}) \in$ CALLS$(DB_s^{\mathsf{a}} \cup DB_s^{\mathsf{b}})$ with $\overline{\mathtt{f}} = \langle \mathtt{f}/n, \mathcal{F}^{in}, F^{out} \rangle \in \mathcal{S}$, $\vec{\mathtt{o}}$ conforms to $\mathcal{F}^{in}$ **then** ▷ Check service input types
        **pick** $\sigma \in \{\theta \mid (i)\ \theta$ is a total function, $(ii)\ \theta : SCalls \rightarrow \Delta_{\mathcal{T}}$, $(iii)$ for each $\mathtt{f}(\vec{\mathtt{o}}), \mathtt{f}(\vec{\mathtt{o}})\theta$ conforms to $F^{out}\}$
        $DB_{cand}^{\mathsf{a}} := DB_s^{\mathsf{a}}\sigma$, $DB_{cand}^{\mathsf{b}} := DB_s^{\mathsf{b}}\sigma$    ▷ Obtain new candidate DBs by substituting service calls with results
        **if** $DB_{cand}^{\mathsf{a}}$ conforms to $\mathcal{D}^{\mathsf{a}}) \wedge (DB_{cand}^{\mathsf{a}}$ satisfies $\Gamma^{\mathsf{a}})$ **then** $DB_{\mathtt{a}} := DB_{cand}^{\mathsf{a}}$    ▷ Update $\mathtt{a}$’s DB
        **else** $DB_{\mathtt{a}} := s.db(\mathtt{a})$    ▷ Rollback $\mathtt{a}$’s DB
        **if** $DB_{cand}^{\mathsf{b}}$ conforms to $\mathcal{D}^{\mathsf{b}}) \wedge (DB_{cand}^{\mathsf{b}}$ satisfies $\Gamma^{\mathsf{b}})$ **then** $DB_{\mathtt{b}} := DB_{cand}^{\mathsf{b}}$    ▷ Update $\mathtt{b}$’s DB
        **else** $DB_{\mathtt{b}} := s.db(\mathtt{b})$    ▷ Rollback $\mathtt{b}$’s DB
        **pick** fresh state $s'$    ▷ Create new state
        $NewAS := \emptyset$    ▷ Determine the (possibly changed) set of active agents and their specs
        **if** $\mathtt{a} = \mathsf{inst}$ **then** $NewAS := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec_n}) \in DB_{\mathtt{a}}\}$
        **else if** $\mathtt{b} = \mathsf{inst}$ **then** $NewAS := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec_n}) \in DB_{\mathtt{b}}\}$
        **else** $NewAS := CurAS$    ▷ No change if $\mathsf{inst}$ is not involved in the interaction or must reject the update
        **for all** $\langle \mathtt{n}, \mathtt{spec_n} \rangle \in NewAS$ **do**    ▷ Do the update for each active agent
          **if** $\mathtt{n} = \mathtt{a}$ **then** $s'.db(\mathtt{n}) := DB_{\mathtt{a}}$    ▷ Case of sender agent
          **else if** $\mathtt{n} = \mathtt{b}$ **then** $s'.db(\mathtt{n}) := DB_{\mathtt{b}}$    ▷ Case of target agent
          **else if** $\mathtt{n} \notin CurAS$ **then**    ▷ Case of newly created agent
            $s'.db(\mathtt{n}) := D_0^{\mathsf{spec_n}} \cup \{MyName(\mathtt{n})\}$    ▷ $\mathtt{n}$’s initial DB gets the initial data fixed by its specification, plus its name
          **else** $s'.db(\mathtt{n}) := s.db(\mathtt{n})$    ▷ Default case: persisting agent not affected by the interaction
        **if** $\exists s'' \in \Sigma$ s.t. $s''.db(\mathsf{inst}) = s'.db(\mathsf{inst})$ and for each $\langle \mathtt{n}, \_ \rangle \in CurAS$, $s''.db(\mathtt{n}) = s'.db(\mathtt{n})$ **then**
          $\rightarrow := \rightarrow \cup \langle s, s'' \rangle$    ▷ State already exists: connect $s$ to that state
        **else** $\Sigma := \Sigma \cup \{s'\}, \rightarrow := \rightarrow \cup \langle s, s' \rangle$    ▷ Add and connect new state
**function** GET-MSGS($\mathcal{C}$,$DB$,$CurAS$)    ▷ Evaluate communicative rules $\mathcal{C}$ on DB $DB$, and return the enabled messages with targets
  $EMsg := \emptyset$
  **for all** communicative rules “$Q(t, \vec{x})$ **enables** $M(\vec{x})$ **to** $t$” in $\mathcal{C}$ **do**
    **for all** $\theta \in ans(Q, DB)$ **do**    ▷ $\theta$ provides an actual payload and target agent
      **if** $t\theta \in \{\mathtt{n} \mid \langle \mathtt{n}, \_ \rangle \in CurAS\}$ and $M(\vec{x})\theta$ conforms to $\overline{M} \in \mathcal{M}$ **then**    ▷ $\theta$ is well-typed and has an active agent as target
        $EMsg := EMsg \cup \langle M(\vec{x}, t)\theta, t\theta \rangle$    ▷ Add the ground event and target agent to the set of enabled events
  **return** $EMsg$
**function** GET-FACTS($\mathcal{X}$, $DB$, $ACT$)    ▷ Applies actions $ACT$ on DB $DB$, and returns facts to be added and deleted
  $ToAdd_s := \emptyset$; $ToDel := \emptyset$    ▷ $ToAdd_s$: facts with embedded service calls, to be added; $ToDel$: facts to be deleted
  **for all** instantiated actions $\alpha(\vec{\mathtt{v}}) \in ACT$ **do**
    **for all** effects “$Q(\vec{p}, \vec{x}) \rightsquigarrow$ **add** $A$, **del** $D$” in the definition of $\alpha$ **do**
      **for all** $\theta \in ans(Q(\vec{\mathtt{v}}, \vec{x}), D)$ **do**    ▷ Get an answer from the left-hand side
        $ToAdd_s := ToAdd_s \cup A\theta[\vec{p}/\vec{\mathtt{v}}]$    ▷ Get facts to add (with embedded service calls)
        $ToDel := ToDel \cup D\theta[\vec{p}/\vec{\mathtt{v}}]$    ▷ Get facts to delete
  **return** $\langle ToDel, ToAdd_s \rangle$    ▷ Recall: facts to be added still contain service calls - to be substituted with actual results
```

Figure 1: Procedure for constructing a transition system describing the execution semantics of an RMAS; given a set F of facts, CALLS(F) returns the ground service calls contained in F

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:

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

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

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 $\mu Ł_p^@$ 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, possibly, 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). Similarly, from Theorem 3.1 we obtain that when an RMAS is state-bounded, asynchronous communication can be modelled only by putting a threshold on the size of each message queue.

(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 $\mu Ł_p^@$ property Φ, we have $\Upsilon_\mathcal{X} \models \Phi$ if and only if $\Upsilon_{\widehat{\mathcal{X}}} \models \Phi$.*

Proof. Let $\mathcal{X} = \langle \mathcal{T}, \mathcal{F}, \Delta_{0,\mathcal{F}}, \mathcal{S}, \mathcal{M}, \mathcal{G}, I \rangle$. We construct $\widehat{\mathcal{X}} = \langle \mathcal{T}, \widehat{\mathcal{T}}, \Delta_{0,\mathcal{F}}, \widehat{\mathcal{S}}, \widehat{\mathcal{M}}, \widehat{\mathcal{G}}, \widehat{I} \rangle$ as follows:

- $\widehat{\mathcal{T}}$ is the set of base facets constructed starting from the types in $\mathcal{T}$.
- $\widehat{\mathcal{S}}$ and $\widehat{\mathcal{M}}$, are obtained from $\mathcal{S}$ and $\mathcal{M}$ by substituting the facet attached to each component with the corresponding base facet: whenever a component is originally typed with facet $\langle T, \varphi(x) \rangle \in \mathcal{F}$, the corresponding component is typed with the base facet $\langle T, \mathsf{true} \rangle \in \widehat{\mathcal{T}}$.
- $\widehat{\mathcal{S}}$ and $\widehat{\mathcal{M}}$, are obtained from $\mathcal{S}$ and $\mathcal{M}$ by substituting the facet attached to each component with the corresponding base facet: whenever a component is originally typed with facet $\langle T, \varphi(x) \rangle \in \mathcal{F}$, the corresponding component is typed with the base facet $\langle T, \mathsf{true} \rangle \in \widehat{\mathcal{T}}$.
- Each agent specification $\mathcal{N} = \langle \mathtt{n}, \mathcal{D}, \Gamma, D_0, \mathcal{C}, \mathcal{A}, \mathcal{U} \rangle$ in $\mathcal{G} \cup \{I\}$ becomes a corresponding agent specification $\widehat{\mathcal{N}} = \langle \mathtt{n}, \widehat{\mathcal{D}}, \widehat{\Gamma}, D_0, \widehat{\mathcal{C}}, \widehat{\mathcal{A}}, \widehat{\mathcal{U}} \rangle$ in $\widehat{\mathcal{G}} \cup \{\widehat{I}\}$. The database schema $\widehat{\mathcal{D}}$ transforms $\mathcal{D}$ similarly to how $\widehat{\mathcal{S}}$ and $\widehat{\mathcal{M}}$ transform $\mathcal{S}$ and $\mathcal{M}$: for every n-ary typed relation $\overline{R} \in \mathcal{D}$, a corresponding n-ary relation $\overline{R}$ is included in $\widehat{\mathcal{D}}$, such that, for every $i \in \{1, \ldots, n\}$, $\textsc{type}_{\widehat{\mathcal{D}}}(R'[i]) = \langle T, \mathsf{true} \rangle$ if and only if $\textsc{type}_\mathcal{D}(R[i]) = \langle T, \varphi(x) \rangle$. In addition, for every typed service call $\overline{f}(\langle T_1, \varphi_1(x) \rangle, \ldots, \langle T_n, \varphi_n(x) \rangle)$ in $\mathcal{S}$, $\widehat{\mathcal{D}}$ contains a relation $\overline{Input_f}(\langle T_1, \varphi_1(x) \rangle, \ldots, \langle T_n, \varphi_n(x) \rangle)$, whose use is explained below.

The other elements of $\widehat{\mathcal{N}}$ ensure that the type checks of $\mathcal{N}$ are properly recreated in the form of special queries and constraints. In particular:

- For every communicative rule "$Q(t, \vec{x})$ **enables** $M(\vec{x})$ **to** t" in $\mathcal{C}$, with $|\vec{x}| = n$, $\widehat{\mathcal{C}}$ contains the corresponding rule

$$Q(t, \vec{x}) \wedge \bigwedge_{i \in \{1,\ldots,n\}, \langle T_i, \varphi_i(x) \rangle = \textsc{type}_\mathcal{M}(M[i])} \varphi(x_i) \textbf{ enables } M(\vec{x})$$

  This guarantees that the filter criterion applied on lines 45-47 of Figure 1 is properly reconstructed, so that $\mathcal{X}$ and $\widehat{\mathcal{X}}$ produce the same sets of enabled messages.
- A similar approach is applied to the update rules in $\mathcal{U}$, incorporating into each condition the facet expressions of the facets attached to the corresponding action components, in such a way that the filter criterion applied on lines 15 and 17 of Figure 1 is properly reconstructed. This ensures that $\mathcal{X}$ and $\widehat{\mathcal{X}}$ produce the same sets of instantiated actions.
- Actions $\mathcal{A}$ need to be translated by ensuring that the types of relations in $\mathcal{D}$ and those of the service call input/outputs in $\mathcal{S}$ are properly checked. The typing of relation components is guaranteed by augmenting the set Γ of constraints. Specifically, beside all the original constraints in Γ, for each n-ary typed relation $\overline{R}$ in $\mathcal{D}$ and every $i \in \{1, \ldots, n\}$, we insert into $\widehat{\Gamma}$ a dedicated constraint

$$\forall x_i.R(\_, \ldots, x_i, \ldots, \_) \rightarrow \varphi_i(x_i)$$

  where φ_i is the facet formula of $\textsc{type}_\mathcal{D}(R[i])$. This technique guarantees that $\mathcal{X}$ and $\widehat{\mathcal{X}}$ equivalently evaluate the conditions on lines 24 and 26 of Figure 1 ($\widehat{\mathcal{X}}$

always satisfies the conformance test, and lifts the original conformance test of $\mathcal{X}$ as a test on the satisfaction of database constraints, expressed in the second conjunct of lines 24 and 26). Finally, the tests expressed on lines 21 and 22 of Figure 1, which respectively check whether the service calls involved in an action application have inputs and outpus conforming to their respective facets, is reformulated using the technique illustrated in the following. For every action $\overline{\alpha} \in \mathcal{A}$, $\widehat{\mathcal{A}}$ contains an action $\overline{\alpha}'$, constructed by properly manipulating the set of facts in the **add**-set. Specifically, for each effect "$Q(\vec{\mathsf{p}}, \vec{x}) \rightsquigarrow$ **add** A, **del** D" in the specification of α, α' contains a corresponding effect "$Q(\vec{\mathsf{p}}, \vec{x}) \rightsquigarrow$ **add** A', **del** D", where:

$$\begin{aligned} A' = A \cup \{ & \mathit{Input}_f(\vec{x}) \mid F \in A \text{ and } \mathbf{f}(\vec{x}) \text{ appears in } F\} \\ & \{\mathit{Output}_f(\mathbf{f}(\vec{x})) \mid F \in A \text{ and } \mathbf{f}(\vec{x}) \text{ appears in } F\} \end{aligned}$$

Intuitively, A' adds a fact for relation Input_f/n and a fact for relation $\mathit{Output}_f/1$ for every n-ary service call $\mathbf{f}$ appearing in A, in such a way that the contect of these two facts respectively correspond to the input and output of $\mathbf{f}$. Since it is not important that such facts are persisted in the agent database, but it is only important that they are present after the action is applied, the specification of each action in $\widehat{\mathcal{A}}$ also contains the following effects:

$$\left\{ \mathit{Input}_{f_i}(\vec{x}) \rightsquigarrow \textbf{del}\{\mathit{Input}_{f_i}(\vec{x})\} \;\middle|\; \overline{f_i} \in \mathcal{S} \right\}$$

The conformance with the service input facets can then be reformulated similarly to the case of relations in $\mathcal{D}$, that is, by further augmenting the set $\widehat{\Gamma}$ of constraints. Specifically, for each n-ary service call $\overline{\mathbf{f}} = \langle \mathbf{f}/n, \mathcal{F}^{in}, F^{out} \rangle$ in $\mathcal{S}$, we insert two dedicated constraints in $\widehat{\Gamma}$:

1. by denoting with φ_i the facet formula of the i-th component of $\mathcal{F}^{in}$,

$$\forall x_1, \ldots, x_n . \mathit{Input}_f(x_1, \ldots, x_n) \rightarrow \bigwedge_{i \in \{1, \ldots, n\}} \varphi_i(x_i)$$

2. by denotwing with ψ the facet formula of F^{out},

$$\forall x . \mathit{Output}_f(x) \rightarrow \psi(x)$$

This mechanism lifts the checks applied for $\mathcal{X}$ on lines 21 and 22 of Figure 1 (which is trivially true for $\widehat{\mathcal{X}}$) as additional constraint checks on lines 24 and 26, where the satisfaction of database constraints is tested.

The translation mechanism ensures that the execution semantics of $\widehat{\mathcal{X}}$ suitably reconstructs that of $\mathcal{X}$, i.e., if we project away the accessory relations used for the service call inputs, we have that $\Upsilon_{\widehat{\mathcal{X}}}$ is equivalent to $\Upsilon_{\mathcal{X}}$. □

As a consequence of Theorem 5.1, we have that, for shallow-typed RMASs, the transition system construction can be simplified as shown in the BUILD-TS-SHALLOW procedure of Figure 2.

5.1 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.*

Proof. The proof is by reduction from the halting problem of two-counter machines. A *counter* is a memory register that stores a (non-negative) integer. Notice that the proof works in the same way even if we substitute $\mathbb{Z}$ with $\mathbb{Q}$ or $\mathbb{R}$, provided that they are equipped with the successor relation.

Given two positive integers $n, m \in \mathbb{N}^+$, an m-*counter machine* $\mathfrak{C}$ with counters $c_1, \ldots, c_m$ is a program constituted by a (numbered) sequence of n instructions:

$$1 : \mathit{CMD}_1; \quad 2 : \mathit{CMD}_2; \quad \ldots \quad n : \mathsf{HALT};$$

where the n-th instruction indicates that $\mathfrak{C}$ halts, while for every $k \in \{1, \ldots, n-1\}$, instruction $k : \mathit{CMD}_k$ has one of the two following forms:

- (*increment command* for counter i) $\mathit{CMD}_k = \mathsf{INC}(i, k')$, with $i \in \{1, \ldots, m\}$ and $k' \in \{1, \ldots, n\}$, which increases the counter c_i of one unit, and then jumps to instruction number k':

$$k : c_i := c_i + 1; \ \mathsf{GOTO}\ k';$$

- (*conditional decrement command* for counter i) $\mathit{CMD}_k = \mathsf{CDEC}(i, k', k'')$, with $i \in \{1, \ldots, m\}$ and $k', k'' \in \{1, \ldots, n\}$, which tests whether the value of counter i is zero. If so, it jumps to instruction k'; otherwise, it decreases counter i of one unit, and then jumps to instruction k'':

$$\begin{aligned} k : \ \ \textbf{if}\ c_i == 0 \ \ & \textbf{then}\ \mathsf{GOTO}\ k'; \\ & \textbf{else}\ \{c_i := c_i - 1; \ \mathsf{GOTO}\ k''; \} \end{aligned}$$

An *input* for an m-counter machine is an m-tuple of values $\langle d_1, \ldots, d_m \rangle$ (such that $d_i \in \mathbb{N}$), used to initialize its counters. Given an m-counter machine $\mathfrak{C}$ and an input I of size m, we say that $\mathfrak{C}$ *halts on input* I if the execution of $\mathfrak{C}$ with counter initial values set by I eventually reaches the last, HALT command.

It is well-known that checking whether a 2-counter machine halts on a given input is undecidable (Minsky 1967), and that undecidability still holds when checking whether the 2-counter machine halts on input $\langle 0, 0 \rangle$.

We show how to encode a 2-counter machine into a state-bounded, shallow-typed RMAS containing a single agent specification that work over unary relations only. Specifically, given a 2-counter machine $\mathfrak{C}$ with n instructions, we construct RMAS $\mathcal{X}_{\mathfrak{C}} = \langle \{AT, ZT\}, \{AF, ZF\}, \{\mathtt{0}, \ldots, \mathtt{k}\}, \{\overline{\mathbf{input}}\}, \{go\}, \emptyset, I_{\mathfrak{C}} \rangle$, where $k = max\{2, n\}$, and:

- $AT = \langle \mathbb{A}, \{=\} \rangle$ is the agent type (just used to keep track of the inst name), $ZT = \langle \mathbb{Z}, \{<, =, \mathsf{succ}\} \rangle$ is the integer type (but, as specified above, $\mathbb{Z}$ can be seamlessly substituted by $\mathbb{Q}$ or $\mathbb{R}$).

```
procedure BUILD-TS-SHALLOW($\widehat{\mathcal{X}}$)
input: Shallow-typed RMAS $\widehat{\mathcal{X}} = \langle \mathcal{T}, \widehat{\mathcal{T}}, \Delta_{0,\mathcal{F}}, \widehat{\mathcal{S}}, \widehat{\mathcal{M}} \rangle$, output: Transition system $\Upsilon_{\mathcal{X}} = \langle \Delta_T, \Sigma, s_0, \rightarrow \rangle$
  $AS_0 := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec_n}) \in D_0^{\mathsf{inst}}\}$    ▷ Initial agents with their specifications
  for all $\langle \mathtt{n}, \mathtt{spec_n} \rangle \in AS_0$ do $s_0.db(\mathtt{n}) := D_0^{\mathsf{spec_n}}$    ▷ Specify the initial state by extracting the initial DBs from the agent specs
  $\Sigma := \{s_0\}, \rightarrow := \emptyset$
  while true do
    pick $s \in \Sigma$    ▷ Nondeterministically pick a state
    $CurAS := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{a}, \mathtt{spec_n}) \in s.db(\mathsf{inst})\}$    ▷ Get currently active agents with their specifications
    pick $\langle \mathtt{a}, \mathtt{spec_a} \rangle \in CurAS$    ▷ Nondeterministically pick an active agent a, elected as "sender"
    $EMsg :=$ GET-MSGS($\widehat{\mathcal{C}}^{\mathsf{spec_a}}, s.db(\mathtt{a}), CurAS$)    ▷ Get the enabled messages with target agents
    if $EMsg \neq \emptyset$ then
      pick $\langle M(\vec{\mathtt{o}}), \mathtt{b} \rangle \in EMsg$, with $\langle \mathtt{b}, \mathtt{spec_b} \rangle \in CurAS$ ▷ Pick a message+target agent and trigger message exchange and reactions
      $ACT_{\mathtt{a}} := \emptyset$, $ACT_{\mathtt{b}} := \emptyset$    ▷ Get the actions with actual parameters to be applied by a and b
      for all matching on-send rules "on $M(\vec{x})$ to $t$ if $Q(t, \vec{x})$ then $\alpha(t, \vec{x})$" in $\widehat{\mathcal{U}}^{\mathsf{spec_a}}$ do
        if $ans(Q(\mathtt{b}, \vec{\mathtt{o}}), s.db(\mathtt{a}))$ then $ACT_{\mathtt{a}} := ACT_{\mathtt{a}} \cup \alpha(\mathtt{b}, \vec{\mathtt{o}})$
      for all matching on-receive rules "on $M(\vec{x})$ from $s$ if $Q(s, \vec{x})$ then $\alpha(s, \vec{x})$" in $\widehat{\mathcal{U}}^{\mathsf{spec_b}}$ do
        if $ans(Q(\mathtt{a}, \vec{\mathtt{o}}), s.db(\mathtt{b}))$ then $ACT_{\mathtt{b}} := ACT_{\mathtt{b}} \cup \alpha(\mathtt{a}, \vec{\mathtt{o}})$
      $\langle ToDel^{\mathsf{a}}, ToAdd_s^{\mathsf{a}} \rangle :=$ GET-FACTS($\widehat{\mathcal{X}}, s.db(\mathtt{a}), ACT_{\mathtt{a}}$), $\langle ToDel^{\mathsf{b}}, ToAdd_s^{\mathsf{b}} \rangle :=$ GET-FACTS($\widehat{\mathcal{X}}, s.db(\mathtt{b}), ACT_{\mathtt{b}}$)
      $DB_s^{\mathsf{a}} := (s.db(\mathtt{a}) \setminus ToDel^{\mathsf{a}}) \cup ToAdd_s^{\mathsf{a}}$    ▷ Calculate new a's DB, still with service calls to be issued
      $DB_s^{\mathsf{b}} := (s.db(\mathtt{b}) \setminus ToDel^{\mathsf{b}}) \cup ToAdd_s^{\mathsf{b}}$    ▷ Calculate new b's DB, still with service calls to be issued
      pick $\sigma \in \{\theta \mid (i)\ \theta$ is a total function, $(ii)\ \theta : SCalls \rightarrow \Delta_{\mathcal{T}}$, $(iii)$ for each $\mathbf{f}(\vec{\mathtt{o}}), \mathbf{f}(\vec{\mathtt{o}})\theta$ conforms to $F^{out}\}$
      $DB_{cand}^{\mathsf{a}} := DB_s^{\mathsf{a}}\sigma, DB_{cand}^{\mathsf{b}} := DB_s^{\mathsf{b}}\sigma$    ▷ Obtain new candidate DBs by substituting service calls with results
      if $DB_{cand}^{\mathsf{a}}$ satisfies $\widehat{\Gamma}^{\mathsf{a}}$ then $DB_{\mathtt{a}} := DB_{cand}^{\mathsf{a}}$    ▷ Update a's DB
      else $DB_{\mathtt{a}} := s.db(\mathtt{a})$    ▷ Rollback a's DB
      if $DB_{cand}^{\mathsf{b}}$ satisfies $\widehat{\Gamma}^{\mathsf{b}}$ then $DB_{\mathtt{b}} := DB_{cand}^{\mathsf{b}}$    ▷ Update b's DB
      else $DB_{\mathtt{b}} := s.db(\mathtt{b})$    ▷ Rollback b's DB
      pick fresh state $s'$    ▷ Create new state
      $NewAS := \emptyset$    ▷ Determine the (possibly changed) set of active agents and their specs
      if $\mathtt{a} = \mathsf{inst}$ then $NewAS := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec_n}) \in DB_{\mathtt{a}}\}$
      else if $\mathtt{b} = \mathsf{inst}$ then $NewAS := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec_n}) \in DB_{\mathtt{b}}\}$
      else $NewAS := CurAS$    ▷ No change if inst is not involved in the interaction or must reject the update
      for all $\langle \mathtt{n}, \mathtt{spec_n} \rangle \in NewAS$ do    ▷ Do the update for each active agent
        if $\mathtt{n} = \mathtt{a}$ then $s'.db(\mathtt{n}) := DB_{\mathtt{a}}$    ▷ Case of sender agent
        else if $\mathtt{n} = \mathtt{b}$ then $s'.db(\mathtt{n}) := DB_{\mathtt{b}}$    ▷ Case of target agent
        else if $\mathtt{n} \notin CurAS$ then    ▷ Case of newly created agent
          $s'.db(\mathtt{n}) := D_0^{\mathsf{spec_n}} \cup \{MyName(\mathtt{n})\}$    ▷ n's initial DB gets the initial data fixed by its specification, plus its name
        else $s'.db(\mathtt{n}) := s.db(\mathtt{n})$    ▷ Default case: persisting agent not affected by the interaction
      if $\exists s'' \in \Sigma$ s.t. $s''.db(\mathsf{inst}) = s'.db(\mathsf{inst})$ and for each $\langle \mathtt{n}, \_ \rangle \in CurAS$, $s''.db(\mathtt{n}) = s'.db(\mathtt{n})$ then
        $\rightarrow := \rightarrow \cup \langle s, s'' \rangle$    ▷ State already exists: connect $s$ to that state
      else $\Sigma := \Sigma \cup \{s'\}, \rightarrow := \rightarrow \cup \langle s, s' \rangle$    ▷ Add and connect new state
```

Figure 2: Simplification of BUILD-TS dealing with shallow-typed RMASs

- AF and ZF are the base facets defined starting from AT and ZT respectively.
- $\overline{\mathbf{input}} = \langle \mathbf{input}/0, \langle\rangle, ZF \rangle$ is a 0-ary service that returns integer values.
- go is a message sent by inst to itself so as to trigger the processing of the next instruction.
- $I_{\mathfrak{C}}$ is a specification for the institutional agent that mimics the program of $\mathfrak{C}$.

Specifically, $I_{\mathfrak{C}} = \langle \mathsf{instspec}, \mathcal{D}_{\mathfrak{C}}, \Gamma_{\mathfrak{C}}, D_0^{\mathsf{inst}}, \mathcal{C}_{\mathfrak{C}}, \mathcal{A}_{\mathfrak{C}}, \mathcal{U}_{\mathfrak{C}} \rangle$, where:

- $\mathcal{D}_{\mathfrak{C}} = \begin{Bmatrix} \overline{C_1}(ZF), \overline{C_1^p}(ZF), \overline{C_2}(ZF), \overline{C_2^p}(ZF), \\ \overline{PC}(ZF), \overline{Op}(ZF), \overline{Target}(ZF), \overline{Halted}() \\ \overline{Agent}(AF), \overline{MyName}(AF) \end{Bmatrix}$

  where:
  - $\overline{C_1}$ and $\overline{C_2}$ store the current values of the two counters,
  - $\overline{C_1^p}$ and $\overline{C_2^p}$ store their previous values,
  - $\overline{PC}$ stores the program counter (i.e., the number of the instruction to be processed),
  - $\overline{Op}$ indicates the nature of the operator to be applied (0 means increment, while 1 means decrement),
  - $\overline{Target}$ indicates the target counter, that is, the counter to which the operation must be applied (1 means the first counter, 2 the second),
  - $\overline{Halted}$ is a proposition indicating that the agent finished the execution (i.e., reached the last instruction of $\mathfrak{C}$).
- $\Gamma_{\mathfrak{C}}$ contains constraints that encode the semantics of operations. In particular:
  - In the case of increment, the target counter must have a

current value that is successor of the previous value:

$$\begin{array}{l} Op(\mathsf{0}) \wedge \mathit{Target}(\mathsf{1}) \\ \quad \rightarrow (\forall x_p, x.C_1(x) \wedge C_1^p(x_p) \rightarrow \mathsf{succ}(x, x_p)) \\ Op(\mathsf{0}) \wedge \mathit{Target}(\mathsf{2}) \\ \quad \rightarrow (\forall x_p, x.C_2(x) \wedge C_2^p(x_p) \rightarrow \mathsf{succ}(x, x_p)) \end{array}$$

– In the case of decrement, the opposite holds, i.e., the target counter must have a current value that is precedessor of the previous value:

$$\begin{array}{l} Op(\mathsf{1}) \wedge \mathit{Target}(\mathsf{1}) \\ \quad \rightarrow (\forall x_p, x.C_1(x) \wedge C_1^p(x_p) \rightarrow \mathsf{succ}(x_p, x)) \\ Op(\mathsf{1}) \wedge \mathit{Target}(\mathsf{2}) \\ \quad \rightarrow (\forall x_p, x.C_2(x) \wedge C_2^p(x_p) \rightarrow \mathsf{succ}(x_p, x)) \end{array}$$

- The initial database of inst initializes the two counters to 0, and the program counter to the first instruction:

$$D_0^{\mathsf{inst}} = \{\mathit{Agent}(\mathsf{inst}), \mathit{MyName}(\mathsf{inst}), C_1(\mathsf{0}), C_2(\mathsf{0}), PC(\mathsf{1})\}$$

- $\mathcal{C}_{\mathfrak{C}}$ contains just a single rule, which enables inst to send a *go* message to itself if it is not halted:

$$\mathit{MyName}(a) \wedge \neg \mathit{Halted}\ \textbf{enables}\ go()\ \textbf{to}\ a$$

- $\mathcal{A}_{\mathfrak{C}}$ contains the following actions:
  – $\overline{\text{SET-PC}}(ZF)$ updates the program counter to the value passed as parameter:

$$\text{SET-PC}(\mathsf{next}) : \left\{ \begin{array}{r} PC(x) \rightsquigarrow \textbf{del}\ \{PC(x)\}, \\ \mathsf{true} \rightsquigarrow \textbf{add}\{PC(\mathsf{next})\} \end{array} \right\}$$

  – $\overline{\text{SET-OP}}(ZF, ZF)$ sets the operation, i.e., the operation type and the target counter, to the passed parameters:

$$\text{SET-OP}(\mathsf{o}, \mathsf{t}) : \left\{ \begin{array}{r} Op(x) \rightsquigarrow \textbf{del}\ \{Op(x)\}, \\ \mathit{Target}(x) \rightsquigarrow \textbf{del}\ \{\mathit{Target}(x)\}, \\ \mathsf{true} \rightsquigarrow \textbf{add}\{Op(\mathsf{o})\} \\ \mathsf{true} \rightsquigarrow \textbf{add}\{\mathit{Target}(\mathsf{t})\} \end{array} \right\}$$

  – $\overline{\text{U-C}}(ZF)$ updates the value of the counter whose index is passed as parameter, and at the same time remembers the current value moving it to the "previous" counter relation:

$$\text{U-C}(\mathsf{c}) : \left\{ \begin{array}{ll} \mathsf{c} = 1 \wedge C_1^p(x) & \rightsquigarrow \textbf{del}\ \{C_1^p(x)\} \\ \mathsf{c} = 1 \wedge C_1(x) & \rightsquigarrow \textbf{del}\ \{C_1(x)\}, \textbf{add}\{C_1^p(x)\} \\ \mathsf{c} = 1 & \rightsquigarrow \textbf{add}\{C_1(\textbf{input}())\} \\ \mathsf{c} = 2 \wedge C_2^p(x) & \rightsquigarrow \textbf{del}\ \{C_2^p(x)\} \\ \mathsf{c} = 2 \wedge C_2(x) & \rightsquigarrow \textbf{del}\ \{C_2(x)\}, \textbf{add}\{C_2^p(x)\} \\ \mathsf{c} = 2 & \rightsquigarrow \textbf{add}\{C_2(\textbf{input}())\} \end{array} \right\}$$

  It is worth noting that the action nondeterministically updates the content of the first or second counter, depending on the value of the parameter. However, by considering the constraints modelled in $\Gamma_{\mathfrak{C}}$, only the successor state that has picked exactly the successor or precedessor value of the current one will be selected, depending on what the current operation is.

  – $\overline{\text{STOP}}()$ is an action without parameters that just sets the *Halted* flag to true:

$$\text{STOP}() : \{\mathsf{true} \rightsquigarrow \textbf{add}\{\mathit{Halted}\}\}$$

- $\mathcal{U}_{\mathfrak{C}}$ constains a set of rules that mirror the instructions of $\mathfrak{C}$, according from the following translation schema:
  – For instruction $k : \mathsf{INC}(i, k')$ (with $i \in \{1, 2\}$), we get:

$$\begin{array}{l} \textbf{on}\ go\ \textbf{if}\ PC(\mathtt{k})\ \textbf{then}\ \text{SET-PC}(\mathtt{k'}) \\ \textbf{on}\ go\ \textbf{if}\ PC(\mathtt{k})\ \textbf{then}\ \text{SET-OP}(\mathtt{0}, \mathtt{i}) \\ \textbf{on}\ go\ \textbf{if}\ PC(\mathtt{k})\ \textbf{then}\ \text{U-C}(\mathtt{i}) \end{array}$$

  The first rule handles the update of the program counter. The second rule indicates that counter `i` must be subject to operation with code `0`. The third rule indicates that the instruction require to update the content of counter `i`.

  – For instruction $k : \mathsf{CDEC}(i, k', k'')$ (with $i \in \{1, 2\}$), we get:

$$\begin{array}{l} \textbf{on}\ go\ \textbf{if}\ PC(\mathtt{k}) \wedge C_{\mathtt{i}}(\mathtt{0})\ \textbf{then}\ \text{SET-PC}(\mathtt{k'}) \\ \textbf{on}\ go\ \textbf{if}\ PC(\mathtt{k}) \wedge \neg C_{\mathtt{i}}(\mathtt{0})\ \textbf{then}\ \text{SET-PC}(\mathtt{k''}) \\ \textbf{on}\ go\ \textbf{if}\ PC(\mathtt{k}) \wedge \neg C_{\mathtt{i}}(\mathtt{0})\ \textbf{then}\ \text{SET-OP}(\mathtt{1}, \mathtt{i}) \\ \textbf{on}\ go\ \textbf{if}\ PC(\mathtt{k}) \wedge \neg C_{\mathtt{i}}(\mathtt{0})\ \textbf{then}\ \text{U-C}(\mathtt{i}) \end{array}$$

  The formalization is specular to the case of increment, with the proviso that the manipuation of the counter is triggered only if the counter is not `0`.

  – For instruction $n : \mathsf{HALT}$, we simply get:

$$\textbf{on}\ go\ \textbf{if}\ PC(\mathtt{n})\ \textbf{then}\ \text{HALT}()$$

It is now apparent that $\mathfrak{C}$ halts on input $\langle 0, 0\rangle$ if and only if $\Upsilon_{\mathcal{X}_{\mathfrak{C}}} \models \mu Z.(\mathit{Halted}) \vee \langle - \rangle Z$

□

5.2 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 $\mu Ł_p^{@}$ properties over state-bounded, densely-ordered RMASs is decidable, and reducible to conventional, finite-state model checking.*

Let $\mathcal{X} = \langle \mathcal{T}, \mathcal{F}, \Delta_{0,\mathcal{F}}, \mathcal{S}, \mathcal{M}, \mathcal{G}, I\rangle$ be an RMAS, and Φ be a closed $\mu Ł_p^{@}$ property. Notice that, by hypothesis, $\mathcal{T}$ is constituted by a set $\mathcal{T}_u$ of data types equipped with domain-specific equality only, and a set $\mathcal{T}_o$ of data types equipped also with a dense total order: $\mathcal{T} = \mathcal{T}_u \uplus \mathcal{T}_o$.

The proof is quite involved, so we separate it into several steps and intermediate lemmas.

The first step consists in reformulating the input RMAS $\mathcal{X}$ into the equivalent, shallow-typed version $\widehat{\mathcal{X}} = \langle \mathcal{T}, \widehat{\mathcal{T}}, \Delta_{0,\mathcal{F}}, \widehat{\mathcal{S}}, \widehat{\mathcal{M}}\rangle$, as defined in the proof of Theorem 5.1. By Theorem 5.1, we have that $\Upsilon_{\mathcal{X}} \models \Phi$ if and only if $\Upsilon_{\widehat{\mathcal{X}}} \models \Phi$.

As a second step, we consider the infinite-state transition system $\Upsilon_{\widehat{\mathcal{X}}}$, and seek a faithful (sound and complete) finite-state abstraction of it, suitably extending the technique in (Bagheri Hariri et al. 2013) so as to consider types, and dense orders in particular. Since $\mathcal{X}$ is state-bounded, two sources of infinity are possibly present in $\Upsilon_{\mathcal{X}}$ and $\Upsilon_{\widehat{\mathcal{X}}}$:

```
1: procedure BUILD-FB-TS-SHALLOW($\widehat{\mathcal{X}}$)
2: input: Shallow-typed RMAS $\widehat{X} = \langle \mathcal{T}, \widehat{\mathcal{T}}, \Delta_{0,\mathcal{F}}, \widehat{\mathcal{S}}, \widehat{\mathcal{M}} \rangle$, with $\mathcal{T} = \{T_u^1, \ldots, T_u^n\} \cup \{T_o^1, \ldots, T_o^m\}$, output: TS $\Upsilon_{\mathcal{X}} = \langle \Delta_T, \Sigma, s_0, \rightarrow \rangle$
3:   $AS_0 := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec_n}) \in D_0^{\mathsf{inst}}\}$    ▷ Initial agents with their specifications
4:   for all $\langle \mathtt{n}, \mathtt{spec_n} \rangle \in AS_0$ do $s_0.db(\mathtt{n}) := D_0^{\mathsf{spec_n}}$    ▷ Specify the initial state by extracting the initial DBs from the agent specs
5:   $\Sigma := \{s_0\}$, $\rightarrow := \emptyset$
6:   while true do
7:     pick $s \in \Sigma$    ▷ Nondeterministically pick a state
8:     $CurAS := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{a}, \mathtt{spec_n}) \in s.db(\mathsf{inst})\}$    ▷ Get currently active agents with their specifications
9:     pick $\langle \mathtt{a}, \mathtt{spec_a} \rangle \in CurAS$    ▷ Nondeterministically pick an active agent a, elected as "sender"
10:    $EMsg :=$ GET-MSGS($\widehat{\mathcal{C}}^{\mathsf{spec_a}}, s.db(\mathtt{a}), CurAS$)    ▷ Get the enabled messages with target agents
11:    if $EMsg \neq \emptyset$ then
12:      pick $\langle M(\vec{\mathtt{o}}), \mathtt{b} \rangle \in EMsg$, with $\langle \mathtt{b}, \mathtt{spec_b} \rangle \in CurAS$ ▷ Pick a message+target agent and trigger message exchange and reactions
13:      $ACT_\mathtt{a} := \emptyset$, $ACT_\mathtt{b} := \emptyset$    ▷ Get the actions with actual parameters to be applied by a and b
14:      for all matching on-send rules "on $M(\vec{x})$ to $t$ if $Q(t, \vec{x})$ then $\alpha(t, \vec{x})$" in $\widehat{\mathcal{U}}^{\mathsf{spec_a}}$ do
15:        if $ans(Q(\mathtt{b}, \vec{\mathtt{o}}), s.db(\mathtt{a}))$ then $ACT_\mathtt{a} := ACT_\mathtt{a} \cup \alpha(\mathtt{b}, \vec{\mathtt{o}})$
16:      for all matching on-receive rules "on $M(\vec{x})$ from $s$ if $Q(s, \vec{x})$ then $\alpha(s, \vec{x})$" in $\widehat{\mathcal{U}}^{\mathsf{spec_b}}$ do
17:        if $ans(Q(\mathtt{a}, \vec{\mathtt{o}}), s.db(\mathtt{b}))$ then $ACT_\mathtt{b} := ACT_\mathtt{b} \cup \alpha(\mathtt{a}, \vec{\mathtt{o}})$
18:      $\langle ToDel^\mathtt{a}, ToAdd_s^\mathtt{a} \rangle :=$ GET-FACTS($\widehat{\mathcal{X}}, s.db(\mathtt{a}), ACT_\mathtt{a}$), $\langle ToDel^\mathtt{b}, ToAdd_s^\mathtt{b} \rangle :=$ GET-FACTS($\widehat{\mathcal{X}}, s.db(\mathtt{b}), ACT_\mathtt{b}$)
19:      $DB_s^\mathtt{a} := (s.db(\mathtt{a}) \setminus ToDel^\mathtt{a}) \cup ToAdd_s^\mathtt{a}$    ▷ Calculate new a's DB, still with service calls to be issued
20:      $DB_s^\mathtt{b} := (s.db(\mathtt{b}) \setminus ToDel^\mathtt{b}) \cup ToAdd_s^\mathtt{b}$    ▷ Calculate new b's DB, still with service calls to be issued
21:      for all data type $T \in \mathcal{T}$ do    ▷ Fetch the active domain and service calls for each type
22:        $ADom_s(T) := \begin{array}{l}\{\mathtt{d} \mid \mathtt{d} \in \Delta_T \cap \Delta_{0,\mathcal{F}}\} \\ \cup \{\mathtt{d} \mid \mathtt{d} \in \Delta_T \cap \text{ADOM}(s)\} \\ \cup \{\mathbf{f}(\vec{\mathtt{o}}) \mid \mathbf{f}(\vec{\mathtt{o}}) \in \text{CALLS}(DB_s^\mathtt{a} \cup DB_s^\mathtt{b}) \text{ and } \overline{\mathbf{f}} = \langle \mathbf{f}/n, \mathcal{F}^{in}, F^{out} \rangle \in \widehat{\mathcal{S}} \text{ with } F^{out} = \langle T, \mathsf{true} \rangle\}\end{array}$
23:      pick $\mathfrak{H} \in \left\{ \langle \mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{H}_1, \ldots, \mathcal{H}_m \rangle \,\middle|\, \begin{array}{l}\mathcal{P}_i \text{ is a } T_u^i\text{-equality commitment on } ADom_s(T_u^i) \text{ for } i \in \{1, \ldots, n\}, \\ \mathcal{H}_j \text{ is a } T_o^j\text{-densely ordered commitment on } ADom_s(T_o^j) \text{ for } j \in \{1, \ldots, m\}\end{array} \right\}$
24:      $\sigma := \left\{ \mathbf{f}(\vec{\mathtt{o}}) \mapsto \mathtt{d} \mid \mathbf{f}(\vec{\mathtt{o}}) \in SCalls \text{ and } \text{ASSIGN-RES}_{\mathfrak{H}}^{\Delta_\mathcal{T}}(s, \mathbf{f}(\vec{\mathtt{o}})) = \mathtt{d} \right\}$
25:      $DB_{cand}^\mathtt{a} := DB_s^\mathtt{a}\sigma$, $DB_{cand}^\mathtt{b} := DB_s^\mathtt{b}\sigma$    ▷ Obtain new candidate DBs by substituting service calls with results
26:      if $DB_{cand}^\mathtt{a}$ satisfies $\widehat{\Gamma}^\mathtt{a}$ then $DB_\mathtt{a} := DB_{cand}^\mathtt{a}$    ▷ Update a's DB
27:      else $DB_\mathtt{a} := s.db(\mathtt{a})$    ▷ Rollback a's DB
28:      if $DB_{cand}^\mathtt{b}$ satisfies $\widehat{\Gamma}^\mathtt{b}$ then $DB_\mathtt{b} := DB_{cand}^\mathtt{b}$    ▷ Update b's DB
29:      else $DB_\mathtt{b} := s.db(\mathtt{b})$    ▷ Rollback b's DB
30:      pick fresh state $s'$    ▷ Create new state
31:      $NewAS := \emptyset$    ▷ Determine the (possibly changed) set of active agents and their specs
32:      if $\mathtt{a} = \mathsf{inst}$ then $NewAS := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec_n}) \in DB_\mathtt{a}\}$
33:      else if $\mathtt{b} = \mathsf{inst}$ then $NewAS := \{\langle \mathtt{n}, \mathtt{spec_n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec_n}) \in DB_\mathtt{b}\}$
34:      else $NewAS := CurAS$    ▷ No change if inst is not involved in the interaction or must reject the update
35:      for all $\langle \mathtt{n}, \mathtt{spec_n} \rangle \in NewAS$ do    ▷ Do the update for each active agent
36:        if $\mathtt{n} = \mathtt{a}$ then $s'.db(\mathtt{n}) := DB_\mathtt{a}$    ▷ Case of sender agent
37:        else if $\mathtt{n} = \mathtt{b}$ then $s'.db(\mathtt{n}) := DB_\mathtt{b}$    ▷ Case of target agent
38:        else if $\mathtt{n} \notin CurAS$ then    ▷ Case of newly created agent
39:          $s'.db(\mathtt{n}) := D_0^{\mathsf{spec_n}} \cup \{MyName(\mathtt{n})\}$    ▷ n's initial DB gets the initial data fixed by its specification, plus its name
40:        else $s'.db(\mathtt{n}) := s.db(\mathtt{n})$    ▷ Default case: persisting agent not affected by the interaction
41:      if $\exists s'' \in \Sigma$ s.t. $s''.db(\mathsf{inst}) = s'.db(\mathsf{inst})$ and for each $\langle \mathtt{n}, \_ \rangle \in CurAS$, $s''.db(\mathtt{n}) = s'.db(\mathtt{n})$ then
42:        $\rightarrow := \rightarrow \cup \langle s, s'' \rangle$    ▷ State already exists: connect $s$ to that state
43:      else $\Sigma := \Sigma \cup \{s'\}$, $\rightarrow := \rightarrow \cup \langle s, s' \rangle$    ▷ Add and connect new state
```

Figure 3: Procedure for constructing a transition system that is a finite-branching, faithful abstraction of the transition system constructed by BUILD-TS-SHALLOW

1. infinite branching, that is, presence of a state with infinitely many successors due to the injection of data through service calls;
2. infinite runs, that is, runs that visit infinitely many different agent databases.

We can get rid of the infinite-branching in $\Upsilon_{\widehat{\mathcal{X}}}$ by suitably pruning it:

Lemma 5.4. *For every shallow-typed RMAS $\widehat{\mathcal{X}}$, there exists a transition system $\Lambda_{\widehat{\mathcal{X}}}$ that obeys the following properties:*

(i) $\Lambda_{\widehat{\mathcal{X}}}$ *is finite-branching;*
(ii) *for every closed $\mu Ł_p^@$ 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. 2012; Bagheri Hariri et al. 2013). Equality commitments are used to abstractly describe how the result of a service call relates through (in)equality to the values present in the active domain of the system, and to those returned by other service calls issued in the same moment, without considering their actual, specific results. Technically, we adapt the definition of equality commitment in (Bagheri Hariri et al. 2012) to the case of RMASs, taking into account that: *(i)* differently from DCDSs, data objects are typed, and *(ii)* some data objects could be compared not only with equality, but also with a domain-specific total, dense relation.

Consider a data type $T_u \in \mathcal{T}_u$, and a set S made up of data objects in Δ_{T_u} and of ground service calls built by applying a service call $\overline{\mathbf{f}} \in \mathcal{S}$ to input data objects, such that the return type of $\overline{\mathbf{f}}$ is compatible with T_u. A T_u*-equality commitment* $\mathcal{P}$ on S is a partition of S, that is, a set of disjoint subsets of S, called *cells*, such that the union of the cells in $\mathcal{P}$ is exactly S. Each cell contains at most one data object (but arbitrarily many ground service calls). For any $e \in \mathcal{P}$, $[e]_{\mathcal{P}}$ denotes the cell to which e belongs.

The intention of $\mathcal{H}$ is to abstractly characterize how the elements in S are related to each other via the domain-specific relation $=_{T_u}$ of T_u. In particular, $\mathcal{P}$ is used to capture equality and non-equality commitments on the members of S in the following sense: for every $e_1, e_2 \in S$, we have $e_1 =_{T_u} e_2$ if and only if $[e_1]_{\mathcal{H}} =_{T_u} [e_2]_{\mathcal{H}}$.

Consider now a data type $T_o \in \mathcal{T}_o$, and a set S as before. A T_o*-densely ordered commitment* $\mathcal{H}$ on S is a pair $\langle \mathcal{P}, pos \rangle$, where:

- $\mathcal{P}$ is a T_o-equality commitment over S;
- pos is an ordering over $\mathcal{P}$ that is compatible with S, i.e., pos is a bijection $\{1, \ldots, |\mathcal{P}|\} \longrightarrow \mathcal{P}$ that obeys to the following property: for every $P_1, P_2 \in \mathcal{P}$, whenever P_1 contains a data object $\mathtt{d}_1 \in T$ and P_2 contains a data object $\mathtt{d}_2$ in Δ_T , we have $pos(P_1) <_{\mathbb{N}} pos(P_2)$ if and only if $\mathtt{d}_1 <_{T_o} \mathtt{d}_2$, where $<_{\mathbb{N}}$ denotes the total order relation on natural numbers.

The intention of $\mathcal{H}$ is to abstractly characterize how the elements in S are related to each other via the domain-specific relations $=_{T_o}$ and $<_{T_o}$ of T. Specifically, $\mathcal{P}$ covers equality, while pos accounts for $<$, and orders the members of S in the following sense: for every $e_1, e_2 \in S$, we have $e_1 <_{T_o} e_2$ if and only if $pos([e_1]_{\mathcal{P}}) <_{\mathbb{N}} pos([e_2]_{\mathcal{P}})$.

We now exploit commitments to change the BUILD-TS algorithm, shown in Figure 1 and used to construct $\Upsilon_{\widehat{\mathcal{X}}}$. In particular, we start from the TS-BUILD-SHALLOW procedure, and modify the function that nondeterministically selects the results returned by service calls. First of all, we assume the existence of a pre-defined function ASSIGN-RES, parameterized by a tuple of commitments, which substitutes a service call with a corresponding result that is in accordance with the cell to which the service call belongs. In particular, let $\mathcal{T}_u = \{T_u^1, \ldots, T_u^n\}$ and $\mathcal{T}_o = \{T_o^1, \ldots, T_o^m\}$. Let $\langle S_u^1, \ldots, S_u^n, S_o^1, \ldots, S_o^m \rangle$ be a tuple of sets, each containing data objects from the corresponding type, and possibly also service calls whose return type matches with that type. Let $\mathfrak{H} = \langle \mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{H}_1, \ldots, \mathcal{H}_m \rangle$ be a tuple of commitments, where each $\mathcal{P}_i$ is a T_u^i-equality commitment built over S_u^i, and where each $\mathcal{H}_j$ is a T_o^j-densely ordered commitment built over S_o^j.

Specifically, given a data domain Δ, we define

$$\textsc{assign-res}^{\Delta}_{\mathfrak{H}} : \Sigma \times \textsc{calls}(\bigcup_{i \in \{1,\ldots,n\}} S_u^i \cup \bigcup_{j \in \{1,\ldots,m\}} S_o^j) \longrightarrow \Delta$$

where, by fixing a state $s \in \Sigma$, $\textsc{assign-res}^{\Delta}_{\mathfrak{H}}$ obeys to the following properties:

- For $i \in \{1, \ldots, n\}$, for every service call $\mathbf{f}(\vec{\mathtt{o}}) \in S_u^i$ and for every data object $\mathtt{d} \in S_u^i$: $\textsc{assign-res}^{\Delta}_{\mathfrak{H}}(s, \mathbf{f}(\vec{\mathtt{o}})) =_{T_u^i} \mathtt{d}$ iff $[\mathbf{f}(\vec{\mathtt{o}})]_{\mathcal{P}_i} =_{T_u^i} [\mathtt{d}]_{\mathcal{P}_i}$.
- For $i \in \{1, \ldots, n\}$ and for every two service calls $\mathbf{f_1}(\vec{\mathtt{o_1}}), \mathbf{f_2}(\vec{\mathtt{o_2}}) \in S_u^i$: $\textsc{assign-res}^{\Delta}_{\mathfrak{H}}(s, \mathbf{f_1}(\vec{\mathtt{o_1}})) =_{T_u^i} \textsc{assign-res}^{\Delta}_{\mathfrak{H}}(s, \mathbf{f_2}(\vec{\mathtt{o_2}}))$ iff $[\mathbf{f_1}(\vec{\mathtt{o_1}})]_{\mathcal{P}_i} =_{T_u^i} [\mathbf{f_2}(\vec{\mathtt{o_2}})]_{\mathcal{P}_i}$.
- For $j \in \{1, \ldots, m\}$ with $\mathcal{H}_j = \langle \mathcal{P}'_j, pos_j \rangle$, for every service call $\mathbf{f}(\vec{\mathtt{o}}) \in S_o^j$ and for every data object $\mathtt{d} \in S_o^j$: $\textsc{assign-res}^{\Delta}_{\mathfrak{H}}(s, \mathbf{f}(\vec{\mathtt{o}})) =_{T_u^i} \mathtt{d}$ iff $[\mathbf{f}(\vec{\mathtt{o}})]_{\mathcal{P}'_j} \Delta =_{T_u^i} [\mathtt{d}]_{\mathcal{P}'_j}$.
- For $\mathcal{H}_j = \langle \mathcal{P}'_j, pos_j \rangle$ ($j \in \{1, \ldots, m\}$), and for every two service calls $\mathbf{f_1}(\vec{\mathtt{o_1}}), \mathbf{f_2}(\vec{\mathtt{o_2}}) \in S_o^j$: $\textsc{assign-res}^{\Delta}_{\mathfrak{H}}(s, \mathbf{f_1}(\vec{\mathtt{o_1}})) =_{T_o^j} \textsc{assign-res}^{\Delta}_{\mathfrak{H}}(s, \mathbf{f_2}(\vec{\mathtt{o_2}}))$ iff $[\mathbf{f_1}(\vec{\mathtt{o_1}})]_{\mathcal{P}'_j} =_{T_o^j} [\mathbf{f_2}(\vec{\mathtt{o_2}})]_{\mathcal{P}'_j}$.
- For $\mathcal{H}_j = \langle \mathcal{P}'_j, pos_j \rangle$ ($j \in \{1, \ldots, m\}$), and for every two service calls $\mathbf{f_1}(\vec{\mathtt{o_1}}), \mathbf{f_2}(\vec{\mathtt{o_2}}) \in S_o^j$:
  - $\textsc{assign-res}^{\Delta}_{\mathfrak{H}}(s, \mathbf{f_1}(\vec{\mathtt{o_1}})) =_{T_o^j} \textsc{assign-res}^{\Delta}_{\mathfrak{H}}(s, \mathbf{f_2}(\vec{\mathtt{o_2}}))$ iff $[\mathbf{f_1}(\vec{\mathtt{o_1}})]_{\mathcal{P}'_j} =_{T_o^j} [\mathbf{f_2}(\vec{\mathtt{o_2}})]_{\mathcal{P}'_j}$;
  - $\textsc{assign-res}^{\Delta}_{\mathfrak{H}}(s, \mathbf{f_1}(\vec{\mathtt{o_1}})) <_{T_o^j} \textsc{assign-res}^{\Delta}_{\mathfrak{H}}(s, \mathbf{f_2}(\vec{\mathtt{o_2}}))$ iff $pos([\mathbf{f_1}(\vec{\mathtt{o_1}})]_{\mathcal{P}'_j}) <_{\mathbb{N}} pos([\mathbf{f_2}(\vec{\mathtt{o_2}})]_{\mathcal{P}'_j})$.

Intuitively, this function is used to select a *single*, representative combination of service call results that obey to the constraints imposed by a given commitment.

Figure 3 shows the revised version of the algorithm in Figure 2. Instead of picking any combination of service call results, the BUILD-FB-TS-SHALLOW algorithm picks an equality/densely-ordered commitment for each type of the input RMAS, constructed over the current active domain for that type, where the current active domain for type T is obtained by considering:

- the initial data objects for T;
- the current data objects for T;
- the service calls that must be issued now, and whose return facet is defined over type T.

The combination of service call results for each type is then obtained by applying the pre-defined ASSIGN-RES function.

Let $\Lambda_{\widehat{\mathcal{X}}}$ be the transition system obtained by the application of the BUILD-FB-TS-SHALLOW procedure over the shallow-typed RMAS $\widehat{\mathcal{X}}$. We first argue that $\Lambda_{\widehat{\mathcal{X}}}$ is finite-branching, differently from $\Upsilon_{\widehat{\mathcal{X}}}$, for which the function GET-CALL-RES may return infinitely many combinations of service call results. In fact, given the current active domain $ADom_s(T)$ of a type T, there are only finitely many commitments that can be constructed over that set. More specifically, when T is an unordered type their number is bounded by the Bell number of $|ADom_s(T)|$, wherease when T is an ordered type their number is bounded by the Bell number of $|ADom_s(T)|$, multiplied by the factorial of $|ADom_s(T)|$ (so as to account for the permutation of data objects). Since the ASSIGN-RES function assigns a single combination of results for each commitment, there are only finitely many combination of service call results, and consequently only finitely many successor states of a given state can be present in $\Lambda_{\widehat{\mathcal{X}}}$.

To show that $\Upsilon_{\widehat{\mathcal{X}}}$ and $\Lambda_{\widehat{\mathcal{X}}}$ satisfy the same set of $\mu Ł_p^@$ formulae, one needs to follow step-by-step the proof of (Bagheri Hariri et al. 2012; Bagheri Hariri et al. 2013), noticing that the notion of densely-ordered commitment covers the case of formulae of the form $x < y$, which is the only one not already tackled by (Bagheri Hariri et al. 2012; Bagheri Hariri et al. 2013). This concludes the proof of Lemma 5.4.

We now observe that $\Lambda_{\widehat{\mathcal{X}}}$ may still contain runs visiting infinitely many different states. The third phase of our proof consequently consists of showing that it is possible to produce a "folded" folded transition system $\Theta_{\widehat{\mathcal{X}}}$ that is finite-state, and such that for every closed $\mu Ł_p^@$ property Φ, $\Lambda_{\widehat{\mathcal{X}}} \models \Phi$ if and only if $\Theta_{\widehat{\mathcal{X}}} \models \Phi$.

Before showing how this can be done, we define a variant of BUILD-FB-TS-SHALLOW that, instead of employing the domain-specific (rigid) ordering relations, relies on additional "comparison tables" that are suitably manipulated state by state. The algorithm is shown in Figure 4. The construction algorithm exploits a specific database (indexed in the state by symbol $<$) to store the projection of the ordering relations of types in $\mathcal{T}_o$, where only actively persisting data objects are considered. Such database employs a relation $lessThan_{T_o}$ for each densely-ordered data type $T_o \in \mathcal{T}_o$. In order to make the input RMAS insisting on such relations instead of the domain-specific ones, we introduce the FLATTEN operator, which takes an RMAS or one of its components, and substitutes every occurrence of a query of the form $x <_{\mathcal{T}_o} y$ with the corresponding atomic query $lessThan_{T_o}(x, y)$.

Such a database is initialized by computing, for each data type $T_o^i \in \mathcal{T}_o$, the transitive closure of the $<_{T_o^i}$ relation on the initial data domain for T_o^i, and by inserting all extracted pairs into the dedicated $lessThan_{T_o^i}$ binary relation. It is then used whenever a query is issued over an agent database, so as to complement it with the explicit listing of all $lessThan$ relations. Finally, it is updated state-by-state:

- on the one hand by considering the issued service calls, in accordance with the pos relation of the established densely-ordered commitments (cf. line 36 in Figure 4);
- on the other hand by filtering away those tuples that involve a data object that is not persisting when performing a transition from the current to the next state (cf. line 53 in Figure 4).

Let $\Lambda_{\widehat{\mathcal{X}}}^{flat}$ be the transition system produced by BUILD-FB-TS-SHALLOW-FLAT$(\widehat{\mathcal{X}})$. We have that:

Lemma 5.5. *For every shallow-typed RMAS $\widehat{\mathcal{X}}$ and every closed $\mu Ł_p^@$ property Φ:*

$$\Lambda_{\widehat{\mathcal{X}}} \models \Phi \textit{ if and only if } \Lambda_{\widehat{\mathcal{X}}}^{flat} \models \text{FLATTEN}(\Phi)$$

The lemma can be proven by induction on the construction of the two transition systems, recalling that:

- Every execution step of an RMAS is triggered by issuing domain-independent queries over the current database of one of its agents, and therefore comparisons can only be applied to data objects actively present in that databse.
- $\mu Ł_p^@$ queries can only compare data objects that are present in the current active domain of the system, or that were present in the immediately previous state. This is suitably handled, for FLATTEN(Φ), in line 53 of Figure 4, where all comparisons between data objects present in the previous or current states are explicitly maintained.

It is also important to observe that $\Lambda_{\widehat{\mathcal{X}}}^{flat}$ does not alter the state-boundedness of $\Lambda_{\widehat{\mathcal{X}}}$, because it only adds relations on data objects that are present in the current or previous active domains, while comparisons between old data objects are filtered away.

However, the crucial property of the construction of $\Lambda_{\widehat{\mathcal{X}}}^{flat}$, is that apart from data objects present in the initial data domain, *the comparison database is not based on the domain-specific ordering relations, but is constructed starting from the picked densely-ordered commitments*, as shown in line 36 of Figure 4. We combine this crucial property with the inability of $\mu Ł_p^@$, due to its persistent nature, of comparing currently active data objects with objects that were encountered in the past, but are not active anymore. In particular, we can directly apply the data recycling technique in (Bagheri Hariri et al. 2012; Bagheri Hariri et al. 2013), reusing old, forgotten data objects in place of fresh ones.

Figure 5 shows the construction algorithm with recycling of data objects. Let $\Theta_{\widehat{\mathcal{X}}}$ be the transition system produced by such an algorithm. Due to the fact, argued before, that during the system construction comparisons are stored by analyzing densely-ordered commitments, and not domain-specific ordering relations, correctness is obtained by adapting the proof in (Bagheri Hariri et al. 2012; Bagheri Hariri et al. 2013). In particular, we obtain that, when the original 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, that is, before the set $Passive$ is guaranteed to contain sufficiently many used but non-active data objects. This implies that the construction algorithm of Figure 5 terminates, and in turn that $\Theta_{\widehat{\mathcal{X}}}$ is finite-state, and represents at the same time a sound and complete abstraction of the original system.

By putting everything together, we obtain in fact that, for every state-bounded, densely-ordered RMAS $\mathcal{X}$, and for every $\mu Ł_p^@$ property Φ:

1. $\Theta_{\widehat{\mathcal{X}}}$ can be effectively constructed using the procedure BUILD-TS-ABSTRACT of Figure 5;
2. $\Theta_{\widehat{\mathcal{X}}}$ has a finite number of states;
3. $\Upsilon_{\mathcal{X}} \models \Phi$ if and only if $\Theta_{\widehat{\mathcal{X}}} \models$ FLATTEN(Φ).

This concludes the proof. □

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, a source of complexity that 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 presents connections to (Belardinelli 2014), which extends the framework in (Belardinelli, Lomuscio, and Patrizi 2012) with types so as to model and verify auctions. The two settings are incomparable w.r.t. both the framework and the verification logic, and it would be interesting to study cross-transfer of results between the two settings.

References

[Bagheri Hariri et al. 2012] Bagheri Hariri, B.; Calvanese, D.; De Giacomo, G.; Deutsch, A.; and Montali, M. 2012. Verification of relational data-centric dynamic systems with external services. CoRR Technical Report arXiv:1203.0024, arXiv.org e-Print archive.

[Bagheri Hariri et al. 2013] 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 et al. 2014] 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 and Katoen 2008] Baier, C., and Katoen, J.-P. 2008. *Principles of Model Checking*. The MIT Press.

[Belardinelli, Lomuscio, and Patrizi 2012] 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 2014] 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, Gerber, and Pugh 1999] 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.

[Cavada et al. 2014] 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 *Lecture Notes in Computer Science*, 334–342. Springer.

[Chopra and Singh 2013] 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.

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

[Lomuscio, Qu, and Raimondi 2009] 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 *Lecture Notes in Computer Science*, 682–688. Springer.

[Minsky 1967] Minsky, M. L. 1967. *Computation: Finite and Infinite Machines*. Prentice-Hall, Inc.

[Montali, Calvanese, and De Giacomo 2014] 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 and Pistore 2005] 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 *Lecture Notes in Computer Science*, 1–28. Springer.

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

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

[Smith 1980] Smith, R. G. 1980. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers* 29(12):1104–1113.

```
procedure BUILD-FB-TS-SHALLOW-FLAT($\widehat{\mathcal{X}}$)
input: Shallow-typed, RMAS $\widehat{X} = \langle \mathcal{T}, \widehat{\mathcal{T}}, \Delta_{0,\mathcal{F}}, \widehat{\mathcal{S}}, \widehat{\mathcal{M}} \rangle$, with $\mathcal{T} = \{T_u^1, \dots, T_u^n\} \cup \{T_o^1, \dots, T_o^m\}$
output: transition system $\Upsilon_{\mathcal{X}} = \langle \Delta_T, \Sigma, s_0, \rightarrow \rangle$
  $D_0^< := \emptyset$    ▷ Initial DB incorporating the domain-specific $<$ relations for data objects in $\Delta_{0,\mathcal{F}}$
  for all $i \in \{1, \dots, m\}$ do
    for all $\mathtt{d}_1, \mathtt{d}_2 \in \Delta_{0,\mathcal{F}} \cap \Delta_{T_o^m}$ do
      if $\mathtt{d}_1 <_{T_o^m} \mathtt{d}_2$ then $D_0^< := D_0^< \cup \{\mathit{lessThan}_{T_o^m}(\mathtt{d}_1, \mathtt{d}_2)\}$
  $AS_0 := \{\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \mid \mathit{hasSpec}(\mathtt{n}, \mathtt{spec}_\mathtt{n}) \in D_0^{\mathsf{inst}}\}$    ▷ Initial agents with their specifications
  for all $\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \in AS_0$ do $s_0.db(\mathtt{n}) := D_0^{\mathsf{spec}_\mathtt{n}}$    ▷ Specify the initial state by extracting the initial DBs from the agent specs
  $s_0.db(<) := D_0^<$    ▷ Insert the special less-than DB
  $\Sigma := \{s_0\}$, $\rightarrow := \emptyset$
  while true do
    pick $s \in \Sigma$    ▷ Nondeterministically pick a state
    $\mathit{CurAS} := \{\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \mid \mathit{hasSpec}(\mathtt{a}, \mathtt{spec}_\mathtt{n}) \in s.db(\mathsf{inst})\}$    ▷ Get currently active agents with their specifications
    pick $\langle \mathtt{a}, \mathtt{spec}_\mathtt{a} \rangle \in \mathit{CurAS}$    ▷ Nondeterministically pick an active agent a, elected as "sender"
    $\mathit{EMsg} := \textsc{get-msgs}(\textsc{flatten}(\widehat{\mathcal{C}}^{\mathsf{spec}_\mathtt{a}}), s.db(\mathtt{a}) \cup s.db(<), \mathit{CurAS})$    ▷ Get the enabled messages with target agents
    if $\mathit{EMsg} \neq \emptyset$ then
      pick $\langle M(\vec{\mathtt{o}}), \mathtt{b} \rangle \in \mathit{EMsg}$, with $\langle \mathtt{b}, \mathtt{spec}_\mathtt{b} \rangle \in \mathit{CurAS}$ ▷ Pick a message+target agent and trigger message exchange and reactions
      $ACT_\mathtt{a} := \emptyset$, $ACT_\mathtt{b} := \emptyset$    ▷ Get the actions with actual parameters to be applied by a and b
      for all matching on-send rules "on $M(\vec{x})$ to $t$ if $Q(t, \vec{x})$ then $\alpha(t, \vec{x})$" in $\textsc{flatten}(\widehat{\mathcal{U}}^{\mathsf{spec}_\mathtt{a}})$ do
        if $\mathit{ans}(Q(\mathtt{b}, \vec{\mathtt{o}}), s.db(\mathtt{a}) \cup s.db(<))$ then $ACT_\mathtt{a} := ACT_\mathtt{a} \cup \alpha(\mathtt{b}, \vec{\mathtt{o}})$
      for all matching on-receive rules "on $M(\vec{x})$ from $s$ if $Q(s, \vec{x})$ then $\alpha(s, \vec{x})$" in $\textsc{flatten}(\widehat{\mathcal{U}}^{\mathsf{spec}_\mathtt{b}})$ do
        if $\mathit{ans}(Q(\mathtt{a}, \vec{\mathtt{o}}), s.db(\mathtt{b}) \cup s.db(<))$ then $ACT_\mathtt{b} := ACT_\mathtt{b} \cup \alpha(\mathtt{a}, \vec{\mathtt{o}})$
      $\langle \mathit{ToDel}^\mathsf{a}, \mathit{ToAdd}_s^\mathsf{a} \rangle := \textsc{get-facts}(\textsc{flatten}(\widehat{\mathcal{X}}), s.db(\mathtt{a}) \cup s.db(<), ACT_\mathtt{a})$
      $\langle \mathit{ToDel}^\mathsf{b}, \mathit{ToAdd}_s^\mathsf{b} \rangle := \textsc{get-facts}(\textsc{flatten}(\widehat{\mathcal{X}}), s.db(\mathtt{b}) \cup s.db(<), ACT_\mathtt{b})$
      $DB_s^\mathsf{a} := (s.db(\mathtt{a}) \setminus \mathit{ToDel}^\mathsf{a}) \cup \mathit{ToAdd}_s^\mathsf{a}$    ▷ Calculate new a's DB, still with service calls to be issued
      $DB_s^\mathsf{b} := (s.db(\mathtt{b}) \setminus \mathit{ToDel}^\mathsf{b}) \cup \mathit{ToAdd}_s^\mathsf{b}$    ▷ Calculate new b's DB, still with service calls to be issued
      for all data type $T \in \mathcal{T}$ do    ▷ Fetch the active domain and service calls for each type
                             $\{\mathtt{d} \mid \mathtt{d} \in \Delta_T \cap \Delta_{0,\mathcal{F}}\}$
        $ADom_s(T) :=$ $\cup\{\mathtt{d} \mid \mathtt{d} \in \Delta_T \cap \textsc{adom}(s)\}$
                             $\cup\{\mathbf{f}(\vec{\mathtt{o}}) \mid \mathbf{f}(\vec{\mathtt{o}}) \in \textsc{calls}(DB_s^\mathsf{a} \cup DB_s^\mathsf{b})$ and $\overline{\mathbf{f}} = \langle \mathbf{f}/n, \mathcal{F}^{in}, F^{out} \rangle \in \widehat{\mathcal{S}}$ with $F^{out} = \langle T, \mathsf{true} \rangle\}$
      pick $\mathfrak{H} \in \left\{ \langle \mathcal{P}_1, \dots, \mathcal{P}_n, \mathcal{H}_1, \dots, \mathcal{H}_m \rangle \,\middle|\, \begin{array}{l} \mathcal{P}_i \text{ is a } T_u^i\text{-equality commitment on } ADom_s(T_u^i) \text{ for } i \in \{1, \dots, n\}, \\ \mathcal{H}_j \text{ is a } T_o^j\text{-densely ordered commitment on } ADom_s(T_o^j) \text{ for } j \in \{1, \dots, m\} \end{array} \right\}$
      $\sigma := \left\{ \mathbf{f}(\vec{\mathtt{o}}) \mapsto \mathtt{d} \mid \mathbf{f}(\vec{\mathtt{o}}) \in \mathit{SCalls} \text{ and } \textsc{assign-res}_{\mathfrak{H}}^{\Delta_\mathcal{T}}(s, \mathbf{f}(\vec{\mathtt{o}})) = \mathtt{d} \right\}$
      $D^< := \emptyset$    ▷ Recalculate the $\mathit{lessThan}$ relations by considering the current active domains and the picked commitments
      for all $i \in \{1, \dots, m\}$, with $\mathcal{H}_i = \langle \mathcal{P}'_i, \mathit{pos}_i \rangle$ do
        for all $\mathtt{d}_1, \mathtt{d}_2 \in \mathcal{P}'_i \sigma$ do
          if $\mathit{pos}_i([\mathtt{d}_1]_{\mathcal{P}'_i \sigma}) <_\mathbb{N} \mathit{pos}_i([\mathtt{d}_2]_{\mathcal{P}'_i \sigma})$ then
            $D^< := D^< \cup \{\mathit{lessThan}_{T_o^i}(\mathtt{d}_1, \mathtt{d}_2)\}$
      $DB_{cand}^\mathsf{a} := DB_s^\mathsf{a} \sigma$, $DB_{cand}^\mathsf{b} := DB_s^\mathsf{b} \sigma$    ▷ Obtain new candidate DBs by substituting service calls with results
      if $DB_{cand}^\mathsf{a}$ satisfies $\textsc{flatten}(\widehat{\Gamma}^\mathsf{a})$ then $DB_\mathtt{a} := DB_{cand}^\mathsf{a}$    ▷ Update a's DB
      else $DB_\mathtt{a} := s.db(\mathtt{a})$    ▷ Rollback a's DB
      if $DB_{cand}^\mathsf{b}$ satisfies $\textsc{flatten}(\widehat{\Gamma}^\mathsf{b})$ then $DB_\mathtt{b} := DB_{cand}^\mathsf{b}$    ▷ Update b's DB
      else $DB_\mathtt{b} := s.db(\mathtt{b})$    ▷ Rollback b's DB
      pick fresh state $s'$    ▷ Create new state
      $\mathit{NewAS} := \emptyset$    ▷ Determine the (possibly changed) set of active agents and their specs
      if $\mathtt{a} = \mathsf{inst}$ then $\mathit{NewAS} := \{\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \mid \mathit{hasSpec}(\mathtt{n}, \mathtt{spec}_\mathtt{n}) \in DB_\mathtt{a}\}$
      else if $\mathtt{b} = \mathsf{inst}$ then $\mathit{NewAS} := \{\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \mid \mathit{hasSpec}(\mathtt{n}, \mathtt{spec}_\mathtt{n}) \in DB_\mathtt{b}\}$
      else $\mathit{NewAS} := \mathit{CurAS}$    ▷ No change if inst is not involved in the interaction or must reject the update
      for all $\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \in \mathit{NewAS}$ do    ▷ Do the update for each active agent
        if $\mathtt{n} = \mathtt{a}$ then $s'.db(\mathtt{n}) := DB_\mathtt{a}$    ▷ Case of sender agent
        else if $\mathtt{n} = \mathtt{b}$ then $s'.db(\mathtt{n}) := DB_\mathtt{b}$    ▷ Case of target agent
        else if $\mathtt{n} \notin \mathit{CurAS}$ then    ▷ Case of newly created agent
          $s'.db(\mathtt{n}) := D_0^{\mathsf{spec}_\mathtt{n}} \cup \{\mathit{MyName}(\mathtt{n})\}$    ▷ n's initial DB gets the initial data fixed by its specification, plus its name
        else $s'.db(\mathtt{n}) := s.db(\mathtt{n})$    ▷ Default case: persisting agent not affected by the interaction
      $D_+^< := \{\mathit{lessThan}_{T_o}(\mathtt{d}_1, \mathtt{d}_2) \mid \mathit{lessThan}_{T_o}(\mathtt{d}_1, \mathtt{d}_2) \in D^<$ and $\mathtt{d}_1, \mathtt{d}_2 \in \textsc{adom}(s) \cup \textsc{adom}(s')\}$    ▷ Filter $\mathit{lessThan}$
      $s'.db(<) := D_+^<$    ▷ Keep the explicit $\mathit{lessThan}$ relation only for persisting objects
      if $\exists s'' \in \Sigma$ s.t. $s''.db(\mathsf{inst}) = s'.db(\mathsf{inst})$ and for each $\langle \mathtt{n}, \_ \rangle \in \mathit{CurAS}$, $s''.db(\mathtt{n}) = s'.db(\mathtt{n})$ then
        $\rightarrow := \rightarrow \cup \langle s, s'' \rangle$    ▷ State already exists: connect $s$ to that state
      else $\Sigma := \Sigma \cup \{s'\}$, $\rightarrow := \rightarrow \cup \langle s, s' \rangle$    ▷ Add and connect new state
```

Figure 4: Procedure for constructing a transition system that is equivalent to that of BUILD-FB-TS-SHALLOW, but incorporates the ordering relations as special database facts

```
procedure BUILD-ABSTRACT-TS($\widehat{\mathcal{X}}$)
input: Shallow-typed, RMAS $\widehat{X} = \langle \mathcal{T}, \widehat{\mathcal{T}}, \Delta_{0,\mathcal{F}}, \widehat{\mathcal{S}}, \widehat{\mathcal{M}} \rangle$, with $\mathcal{T} = \{T_u^1, \ldots, T_u^n\} \cup \{T_o^1, \ldots, T_o^m\}$
output: transition system $\Upsilon_{\mathcal{X}} = \langle \Delta_T, \Sigma, s_0, \rightarrow \rangle$
  $D_0^< := \emptyset$    ▷ Initial DB incorporating the domain-specific $<$ relations for data objects in $\Delta_{0,\mathcal{F}}$
  for all $i \in \{1, \ldots, m\}$ do
    for all $\mathtt{d}_1, \mathtt{d}_2 \in \Delta_{0,\mathcal{F}} \cap \Delta_{T_o^m}$ do
      if $\mathtt{d}_1 <_{T_o^m} \mathtt{d}_2$ then $D_0^< := D_0^< \cup \{lessThan_{T_o^m}(\mathtt{d}_1, \mathtt{d}_2)\}$
  $AS_0 := \{\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec}_\mathtt{n}) \in D_0^{\mathsf{inst}}\}$    ▷ Initial agents with their specifications
  for all $\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \in AS_0$ do $s_0.db(\mathtt{n}) := D_0^{\mathtt{spec}_\mathtt{n}}$    ▷ Specify the initial state by extracting the initial DBs from the agent specs
  $s_0.db(<) := D_0^<$    ▷ Insert the special less-than DB
  $\Sigma := \{s_0\}, \rightarrow := \emptyset$
  $UsedObj := \Delta_{0,\mathcal{F}}$    ▷ Initialization of the container of used data objects
  while true do
    pick $s \in \Sigma$    ▷ Nondeterministically pick a state
    $CurAS := \{\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \mid hasSpec(\mathtt{a}, \mathtt{spec}_\mathtt{n}) \in s.db(\mathsf{inst})\}$    ▷ Get currently active agents with their specifications
    pick $\langle \mathtt{a}, \mathtt{spec}_\mathtt{a} \rangle \in CurAS$    ▷ Nondeterministically pick an active agent a, elected as “sender”
    $EMsg :=$ GET-MSGS(FLATTEN($\widehat{\mathcal{C}}^{\mathtt{spec}_\mathtt{a}}$), $s.db(\mathtt{a}) \cup s.db(<)$, $CurAS$)    ▷ Get the enabled messages with target agents
    if $EMsg \neq \emptyset$ then
      pick $\langle M(\vec{\mathtt{o}}), \mathtt{b} \rangle \in EMsg$, with $\langle \mathtt{b}, \mathtt{spec}_\mathtt{b} \rangle \in CurAS$ ▷ Pick a message+target agent and trigger message exchange and reactions
      $ACT_\mathtt{a} := \emptyset$, $ACT_\mathtt{b} := \emptyset$    ▷ Get the actions with actual parameters to be applied by a and b
      for all matching on-send rules “on $M(\vec{x})$ to $t$ if $Q(t, \vec{x})$ then $\alpha(t, \vec{x})$” in FLATTEN($\widehat{\mathcal{U}}^{\mathtt{spec}_\mathtt{a}}$) do
        if $ans(Q(\mathtt{b}, \vec{\mathtt{o}}), s.db(\mathtt{a}) \cup s.db(<))$ then $ACT_\mathtt{a} := ACT_\mathtt{a} \cup \alpha(\mathtt{b}, \vec{\mathtt{o}})$
      for all matching on-receive rules “on $M(\vec{x})$ from $s$ if $Q(s, \vec{x})$ then $\alpha(s, \vec{x})$” in FLATTEN($\widehat{\mathcal{U}}^{\mathtt{spec}_\mathtt{b}}$) do
        if $ans(Q(\mathtt{a}, \vec{\mathtt{o}}), s.db(\mathtt{b}) \cup s.db(<))$ then $ACT_\mathtt{b} := ACT_\mathtt{b} \cup \alpha(\mathtt{a}, \vec{\mathtt{o}})$
      $\langle ToDel^{\mathtt{a}}, ToAdd_s^{\mathtt{a}} \rangle :=$ GET-FACTS(FLATTEN($\widehat{\mathcal{X}}$), $s.db(\mathtt{a}) \cup s.db(<)$, $ACT_\mathtt{a}$)
      $\langle ToDel^{\mathtt{b}}, ToAdd_s^{\mathtt{b}} \rangle :=$ GET-FACTS(FLATTEN($\widehat{\mathcal{X}}$), $s.db(\mathtt{b}) \cup s.db(<)$, $ACT_\mathtt{b}$)
      $DB_s^{\mathtt{a}} := (s.db(\mathtt{a}) \setminus ToDel^{\mathtt{a}}) \cup ToAdd_s^{\mathtt{a}}$    ▷ Calculate new a’s DB, still with service calls to be issued
      $DB_s^{\mathtt{b}} := (s.db(\mathtt{b}) \setminus ToDel^{\mathtt{b}}) \cup ToAdd_s^{\mathtt{b}}$    ▷ Calculate new b’s DB, still with service calls to be issued
      for all data type $T \in \mathcal{T}$ do    ▷ Fetch the active domain and service calls for each type
                         $\{\mathtt{d} \mid \mathtt{d} \in \Delta_T \cap \Delta_{0,\mathcal{F}}\}$
        $ADom_s(T) := \cup\{\mathtt{d} \mid \mathtt{d} \in \Delta_T \cap$ ADOM$(s)\}$
                       $\cup\{\mathbf{f}(\vec{\mathtt{o}}) \mid \mathbf{f}(\vec{\mathtt{o}}) \in$ CALLS$(DB_s^{\mathtt{a}} \cup DB_s^{\mathtt{b}})$ and $\overline{\mathbf{f}} = \langle \mathbf{f}/n, \mathcal{F}^{in}, F^{out} \rangle \in \widehat{\mathcal{S}}$ with $F^{out} = \langle T, \mathsf{true} \rangle\}$
      $PassiveObj := UsedObj \setminus$ ADOM$(s)$    ▷ Calculate passive objects, i.e., data objects used in the past but not active now
      pick $\mathfrak{H} \in \left\{ \langle \mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{H}_1, \ldots, \mathcal{H}_m \rangle \,\middle|\, \begin{array}{l} \mathcal{P}_i \text{ is a } T_u^i\text{-equality commitment on } ADom_s(T_u^i) \text{ for } i \in \{1, \ldots, n\}, \\ \mathcal{H}_j \text{ is a } T_o^j\text{-densely ordered commitment on } ADom_s(T_o^j) \text{ for } j \in \{1, \ldots, m\} \end{array} \right\}$
      $\Delta := \Delta_{\mathcal{T}}$    ▷ By default, service calls are substitued with data objects arbitrarily taken from $\Delta_{\mathcal{T}}$
      if $\left| \bigcup_{\mathcal{P} \in \{\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{P}'_1, \ldots, \mathcal{P}'_m\}} \{ec \in \mathcal{P} \mid \text{there is no } \mathtt{d} \in ec\} \right| \leq |PassiveObj|$ then    ▷ Sufficiently many passive objects
        $\Delta := PassiveObj$    ▷ Pick the fresh results by recycling objects in $PassiveObj$
      $\sigma := \left\{ \mathbf{f}(\vec{\mathtt{o}}) \mapsto \mathtt{d} \mid \mathbf{f}(\vec{\mathtt{o}}) \in SCalls \text{ and } \text{ASSIGN-RES}_{\mathfrak{H}}^{\Delta}(s, \mathbf{f}(\vec{\mathtt{o}})) = \mathtt{d} \right\}$    ▷ Get fresh or recycled values
      $D^< := \emptyset$    ▷ Recalculate the $lessThan$ relations by considering the current active domains and the picked commitments
      for all $i \in \{1, \ldots, m\}$, with $\mathcal{H}_i = \langle \mathcal{P}'_i, pos_i \rangle$ do
        for all $\mathtt{d}_1, \mathtt{d}_2 \in \mathcal{P}'_i\sigma$ do
          if $pos_i([\mathtt{d}_1]_{\mathcal{P}'_i\sigma}) <_{\mathbb{N}} pos_i([\mathtt{d}_2]_{\mathcal{P}'_i\sigma})$ then
            $D^< := D^< \cup \{lessThan_{T_o^i}(\mathtt{d}_1, \mathtt{d}_2)\}$
      $DB_{cand}^{\mathtt{a}} := DB_s^{\mathtt{a}}\sigma$, $DB_{cand}^{\mathtt{b}} := DB_s^{\mathtt{b}}\sigma$    ▷ Obtain new candidate DBs by substituting service calls with results
      if $DB_{cand}^{\mathtt{a}}$ satisfies FLATTEN($\widehat{\Gamma}^{\mathtt{a}}$) then $DB_\mathtt{a} := DB_{cand}^{\mathtt{a}}$    ▷ Update a’s DB
      else $DB_\mathtt{a} := s.db(\mathtt{a})$    ▷ Rollback a’s DB
      if $DB_{cand}^{\mathtt{b}}$ satisfies FLATTEN($\widehat{\Gamma}^{\mathtt{b}}$) then $DB_\mathtt{b} := DB_{cand}^{\mathtt{b}}$    ▷ Update b’s DB
      else $DB_\mathtt{b} := s.db(\mathtt{b})$    ▷ Rollback b’s DB
      pick fresh state $s'$    ▷ Create new state
      $NewAS := \emptyset$    ▷ Determine the (possibly changed) set of active agents and their specs
      if $\mathtt{a} = \mathsf{inst}$ then $NewAS := \{\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec}_\mathtt{n}) \in DB_\mathtt{a}\}$
      else if $\mathtt{b} = \mathsf{inst}$ then $NewAS := \{\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \mid hasSpec(\mathtt{n}, \mathtt{spec}_\mathtt{n}) \in DB_\mathtt{b}\}$
      else $NewAS := CurAS$    ▷ No change if inst is not involved in the interaction or must reject the update
      for all $\langle \mathtt{n}, \mathtt{spec}_\mathtt{n} \rangle \in NewAS$ do    ▷ Do the update for each active agent
        if $\mathtt{n} = \mathtt{a}$ then $s'.db(\mathtt{n}) := DB_\mathtt{a}$    ▷ Case of sender agent
        else if $\mathtt{n} = \mathtt{b}$ then $s'.db(\mathtt{n}) := DB_\mathtt{b}$    ▷ Case of target agent
        else if $\mathtt{n} \notin CurAS$ then    ▷ Case of newly created agent
          $s'.db(\mathtt{n}) := D_0^{\mathtt{spec}_\mathtt{n}} \cup \{MyName(\mathtt{n})\}$    ▷ n’s initial DB gets the initial data fixed by its specification, plus its name
        else $s'.db(\mathtt{n}) := s.db(\mathtt{n})$    ▷ Default case: persisting agent not affected by the interaction
      $D_+^< := \{lessThan_{T_o}(\mathtt{d}_1, \mathtt{d}_2) \mid lessThan_{T_o}(\mathtt{d}_1, \mathtt{d}_2) \in D^<$ and $\mathtt{d}_1, \mathtt{d}_2 \in$ ADOM$(s) \cup$ ADOM$(s')\}$    ▷ Filter $lessThan$
      $s'.db(<) := D_+^<$    ▷ Keep the explicit $lessThan$ relation only for persisting objects
      if $\exists s'' \in \Sigma$ s.t. $s''.db(\mathsf{inst}) = s'.db(\mathsf{inst})$ and for each $\langle \mathtt{n}, \_ \rangle \in CurAS$, $s''.db(\mathtt{n}) = s'.db(\mathtt{n})$ then
        $\rightarrow := \rightarrow \cup \langle s, s'' \rangle$    ▷ State already exists: connect $s$ to that state
      else $\Sigma := \Sigma \cup \{s'\}$, $\rightarrow := \rightarrow \cup \langle s, s' \rangle$    ▷ Add and connect new state
```

Figure 5: Procedure for constructing a sound and complete abstraction of the transition system constructed with the BUILD-FB-TS-SHALLOW-FLAT procedure, by recycling non-persisting data objects