Recency-Bounded Verification of Dynamic Database-Driven Systems

Parosh Aziz Abdulla
Uppsala University
parosh@it.uu.se

C. Aiswarya
Uppsala University
aiswarya@cmi.ac.in

Mohamed Faouzi Atig
Uppsala University
mohamed_faouzi.atig@it.uu.se

Marco Montali
Free Univ. of Bozen/Bolzano
montali@inf.unibz.it

Othmane Rezine
Uppsala University
othmane.rezine@it.uu.se

ABSTRACT

We propose a formalism to model database-driven systems, called database manipulating systems (DMS). The actions of a DMS modify the current instance of a relational database by adding new elements into the database, deleting tuples from the relations and adding tuples to the relations. The elements which are modified by an action are chosen by (full) first-order queries. DMS is a highly expressive model and can be thought of as a succinct representation of an infinite state relational transition system, in line with similar models proposed in the literature. We propose monadic second order logic (`MSO-FO`) to reason about sequences of database instances appearing along a run. Unsurprisingly, the linear-time model checking problem of DMS against `MSO-FO` is undecidable. Towards decidability, we propose under-approximate model checking of DMS, where the under-approximation parameter is the "bound on recency". In a k-recency-bounded run, only the most recent k elements in the current active domain may be modified by an action. More runs can be verified by increasing the bound on recency. Our main result shows that recency-bounded model checking of DMS against `MSO-FO` is decidable, by a reduction to the satisfiability problem of MSO over nested words.

CCS Concepts

•Theory of computation → Verification by model checking; Logic and databases;

Keywords: database driven dynamic systems, data aware business processes, relational transition systems, model checking, under-approximation.

1. INTRODUCTION

In the last 15 years, research in business process management (BPM) and workflow technology has progressively shifted its emphasis from a purely control-flow, activity-centric perspective to a more holistic approach that considers also how data are manipulated and evolved by the process [25]. In particular, two lines of research emerged at the intersection of database theory, BPM and formal methods: one focused on modeling languages and technologies for specifying and enacting data-aware business processes [22], and the other tailored to their analysis and verification [10].

The first line of research gave birth to a plethora of new languages and execution platforms, culminating in the so-called *object-centric* [19] and *artifact-centric* paradigms [24], respectively exemplified by frameworks like PHILharmonicFlows [18] and IBM GSM (Guard-Stage-Milestone) [14]. Notably, GSM became the core of the recently published CMMN OMG standard on (adaptive) case management[1]. In this paper, we will use *dynamic database-driven systems* as an umbrella term for all such platforms.

The second line of research focused on understanding the boundaries of decidability and complexity for the verification of dynamic database-driven systems. Two main trends can be identified along this line. The first trend was initiated in the late 1990s with the introduction of relational transducers [2], and continued with new results over progressively richer variants of the initial model, such as systems equipped with arithmetic [15, 13], systems decomposed into interacting web services [16], and systems operating over XML databases [9]. The modelling formalisms introduced in this direction operate over a read-only, input database that is fixed during the system evolution, and use quantifier-free FO formulae to query such a database. The obtained answers can be stored into a read-write state database, whose size is fixed a-priori. Verification problems include control-state reachability [9], or model checking [15, 13] against formulae expressed in FO variants of temporal logics with a limited form of FO quantification across state. Furthermore, verification is *input-parametric*, that is, studied independently from the configuration of data in the initial input database.

In contrast, the second trend studies dynamic systems where the initial state is known. Hence their execution semantics can be captured by means of a single *relational transition system* (RTS), that is, a (possibly) infinite-state transition system whose states are labeled with database instances [27]. Further, the actions allow for bulk read-write operations over the database, possibly injecting fresh values taken from an infinite domain. The injection of such values accounts for the input of new information from the external

[1] http://www.omg.org/spec/CMMN/

PODS'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2902251.2902300

environment (e.g., through user interaction or communication with external systems/services), or the insertion of globally unique identifiers (GUIDS).

Verification of dynamic database-driven systems is challenging due to the infinite state-space generated. Several works [8, 6, 26, 11, 7, 5] succeeded in obtaining decidability by imposing restrictions that yielded finite-state abstractions of the entire system. In [21], decidability is obtained for unbounded-state dynamic database-driven systems in the restrictive case where the database schema contains a single unary relation.

In this paper we propose an under-approximation based on recency of the elements, which allows unbounded state-space. With this restriction we show decidability for the model checking problem against monadic second-order logic over sequences of database instances.

More specifically, we introduce *database-manipulating systems* (DMSs) to model dynamic database-driven systems. Salient features of DMS include guarding every action using (unrestricted) first-order queries on the current database, addition and deletion of tuples in the database, and addition of new elements in the database (which results in a growing active domain).

On top of this model, we study linear-time model checking, using *monadic second-order logic over runs* (MSO-FO) to reason about sequences of database instances appearing along the DMS runs. MSO-FO employs FO queries as its atomic formulae, and supports FO data-quantifications across distinct time points. This powerful logic can express popular verification problems such as reachability, repeated reachability, fairness, liveness, safety, FO-LTL, etc. For example, the property that "every enrolled student eventually graduates" can be formalized in MSO-FO as:

$$\forall x \forall u. \mathit{Enrolled}(u)@x \Rightarrow \exists y. y > x \wedge \mathit{Graduated}(u)@y$$

where x and y are *position* variables, used to predicate about the different time points encountered along a run, while u is a data variable, which matches with values stored in the databases present at these time points. This property corresponds to the FO-LTL formula $\forall u.\mathbf{G}\mathit{Enrolled}(u) \Rightarrow \mathbf{F}\mathit{Graduated}(u)$. More sophisticated properties can be encoded by leveraging the expressive power of MSO-FO, such as that between the enrolment of a student to a course and the moment in which the student passes that course, there is an even number of times in which the student fails that course.

As a first result we show that, unsurprisingly, already propositional reachability turns out to be undecidable to check, even for extremely limited DMSs. Instead of attacking this negative result by limiting the expressive power of the DMS specification formalism, we consider *under-approximate verification*, restricting our attention only to those runs that satisfy a given criterion. In particular, we consider as the under-approximation parameter the *bound on recency*. In a **b**-recency-bounded run, only the most recent **b** elements in the current active domain may be modified (i.e., updated or deleted) by an action, but the behavior of the action may be influenced by the entire content of the database. More runs are verified by increasing the bound on recency. In particular, model checking of safety properties converges to exact model checking in the limit.

Our main result shows that *recency-bounded model checking of DMS against MSO-FO is decidable.* Towards a proof, we encode runs of a recency-bounded DMS as an (infinite) nested word [3]. We then show that the correctness of the encoding can be expressed in MSO over nested words, consequently isolating those runs that correspond to the actual possible behaviors induced by the DMS. At the same time, we describe how to translate the MSO-FO property of interest into a corresponding MSO formula over nested words. In this way, we are able to reduce recency-bounded model checking of DMS against MSO-FO to the satisfiability problem of MSO over nested words, which is known to be decidable [3].

2. PRELIMINARIES

We start by introducing the preliminaries necessary for the development of our framework and results.

Databases. We fix a (data) domain Δ, which is a countably infinite set of data values, acting as standard names. A relational schema $\mathcal{R}$ is a finite set $\{R_1/a_1, \ldots, R_n/a_n\}$ of relation names R_i, each coming with its own arity a_i. A database instance I over schema $\mathcal{R}$ and domain Δ is the union set $\cup_{i:1\leq i\leq n} R_i^I$, where $R_i^I \subseteq \{R_i\} \times \Delta^{a_i}$ represents the content of relation R_i in the database instance I. If I contains a tuple (or a *fact*) $\langle R_i, e_1, \ldots, e_{a_i}\rangle$, we write $R_i(e_1, \ldots, e_{a_i}) \in I$. A nullary relation $p/0$ (also known as proposition) can be either instantiated as the singleton set $\{p()\}$ or the empty set $\emptyset$. In the former case, we say the proposition is *true*, and write $p \in I$. In the latter case $p \notin I$ and we say p is *false*.

We denote the set of all database instances over $\mathcal{R}$ and Δ by $\mathsf{DB\text{-}Inst\text{-}Set}(\mathcal{R}, \Delta)$. The *active domain* of I, denoted $\textsc{adom}(I)$, is the subset of Δ such that $e \in \textsc{adom}(I)$ if and only if e occurs in some fact in I (i.e. there exist $\langle R_i, e_1, \ldots, e_{a_i}\rangle \in I$ such that $e = e_j$ for some $j : 1 \leq j \leq a_i$). Given two database instances $I_1, I_2 \in \mathsf{DB\text{-}Inst\text{-}Set}(\mathcal{R}, \Delta)$, we define $I_1 + I_2$ to be the database instance $I \in \mathsf{DB\text{-}Inst\text{-}Set}(\mathcal{R}, \Delta)$ obtained by taking the relation-wise union. Similarly we define $I_1 - I_2$ where we take the relation-wise set difference. Simply put, $I_1 + I_2 = I_1 \cup I_2$ and $I_1 - I_2 = I_1 \setminus I_2$.

Queries. We use queries to access databases and extract data values of interest. Queries are expressed in *FOL with equality* over the schema $\mathcal{R}$ ($\mathtt{FOL}(\mathcal{R})$ for short). Let $\mathsf{Vars}_{\mathsf{data}} = \{u, v, u_1, \ldots\}$ be the set of FO data-variables ranging over the data values in Δ. A $\mathtt{FOL}(\mathcal{R})$ *query* is given by the following syntax:

$$Q ::= \mathsf{true} \mid R(u_1, \ldots, u_a) \mid \neg Q \mid Q_1 \wedge Q_2 \mid \exists u.Q \mid u_1 = u_2$$

where $R/a \in \mathcal{R}$, and u, u_i are variables from $\mathsf{Vars}_{\mathsf{data}}$. We use standard abbreviations like $Q_1 \vee Q_2 = \neg(\neg Q_1 \wedge \neg Q_2)$, $\forall u.Q = \neg\exists u.\neg Q$, etc. We also denote with $\mathsf{Free\text{-}Vars}(Q)$ the set of free variables appearing in a query Q.

For a set $V \subseteq \mathsf{Vars}_{\mathsf{data}}$, a *substitution* σ of V is a function that maps every variable in V to a value in Δ (i.e., $\sigma : V \to \Delta$). Given a substitution $\sigma : V \to \Delta$ and set $V' \subseteq V$, we define the restriction of σ on V' as the substitution $\sigma' : V' \to \Delta$ such that $\sigma'(u) = \sigma(u)$ for every $u \in V'$. We denote the restriction of σ to V' by $\sigma|_{V'}$.

Given a database instance I over $\mathcal{R}$ and Δ, a $\mathtt{FOL}(\mathcal{R})$ query Q over $\mathcal{R}$, and a substitution $\sigma : \mathsf{Free\text{-}Vars}(Q) \to \Delta$, we write $I, \sigma \models Q$ if the query Q under the substitution σ *holds* in database I. The semantics are as expected, and can be found for completeness in Appendix A. The set of *answers* of Q over I, denoted $\mathit{ans}(Q, I)$, is the set of all substitutions $\sigma : \mathsf{Free\text{-}Vars}(Q) \to \Delta$ such that $I, \sigma \models Q$. When $\mathsf{Free\text{-}Vars}(Q) = \emptyset$ (i.e., Q is a boolean query), we set $\mathit{ans}(Q, I)$ to be the empty substitution $\{\epsilon\}$ whenever

$I, \{\epsilon\} \models Q$ (or $I \models Q$ for short), and we assign $ans(Q, I)$ to the empty set $\emptyset$ whenever $I, \{\epsilon\} \not\models Q$ (or $I \not\models Q$ for short).

EXAMPLE 2.1. We describe a query $\mathsf{Active}(u)$ with a single free variable u, to check whether u is present in some tuple of some relation, no matter what the other elements of the tuple are:

$$\mathsf{Active}(u) \equiv \bigvee_{R/a \in \mathcal{R}} \exists u_1, \ldots, u_a \bigvee_{1 \leq j \leq a} R(u_1, \ldots, u_{j-1}, u, u_{j+1}, \ldots, u_a)$$

$\mathsf{Active}(u)$ characterises $\textsc{adom}(I)$. In fact, $ans(\mathsf{Active}(u), I)$ is $\{\langle u \mapsto e \rangle \mid e \in \textsc{adom}(I)\}$. ■

Substitutions in database instances. Let $V \subseteq \mathsf{Vars}_{\mathsf{data}}$ be a set of variables. Consider a substitution $\sigma : V \to \Delta$ that assigns each variable to an element from Δ. Let $I \in \mathsf{DB\text{-}Inst\text{-}Set}(\mathcal{R}, V)$ be a database instance over schema $\mathcal{R}$ and the variables V. We define $\mathsf{Substitute}(I, \sigma) \in \mathsf{DB\text{-}Inst\text{-}Set}(\mathcal{R}, \Delta)$ to be the database instance obtained from I by substituting every occurrence of variable u by $\sigma(u)$, for each variable $u \in V$.

3. FRAMEWORK

We introduce our model for dynamic database-driven systems. A *Database-Manipulating System (DMS)* over domain Δ and schema $\mathcal{R}$ is a pair $\mathcal{S} = \langle I_0, \textsc{acts} \rangle$, where:

- $I_0 \in \mathsf{DB\text{-}Inst\text{-}Set}(\mathcal{R}, \Delta)$ is the *initial database instance* over $\mathcal{R}$ and Δ, with $\textsc{adom}(I_0) = \emptyset$. I_0 gives truth-values to the nullary relations (also known as propositions), and has empty non-nullary relations
- $\textsc{acts}$ is a set of *(guarded) actions*. An action α is a tuple $\alpha = \langle \vec{u}, \vec{v}, Q, Del, Add \rangle$, where
  - $\vec{u}$ and $\vec{v}$ are disjoint finite subsets of $\mathsf{Vars}_{\mathsf{data}}$, respectively denoting *action parameters* and *fresh-input variables*.
  - Q is a $\mathtt{FOL}(\mathcal{R})$ query, called the *guard* of α.
  - $\vec{u} = \mathsf{Free\text{-}Vars}(Q)$.
  - $Del \in \mathsf{DB\text{-}Inst\text{-}Set}(\mathcal{R}, \vec{u})$ is a database instance over the variables $\vec{u}$ and the schema $\mathcal{R}$.
  - $Add \in \mathsf{DB\text{-}Inst\text{-}Set}(\mathcal{R}, \vec{u} \uplus \vec{v})$ is a database instance over the variables $\vec{u} \uplus \vec{v}$ and $\mathcal{R}$, with $\vec{v} \subseteq \textsc{adom}(Add)$. The set $\vec{v}$ contains the so-called *fresh-input variables* of α.

Given an action $\alpha = \langle \vec{u}, \vec{v}, Q, Del, Add \rangle$, we refer to: $\vec{u}$ by $\alpha{\cdot}\mathsf{free}$, $\vec{v}$ by $\alpha{\cdot}\mathsf{new}$, Q by $\alpha{\cdot}\mathsf{guard}$, Del by $\alpha{\cdot}Del$, and Add by $\alpha{\cdot}Add$.

Intuitively, a DMS operates as follows. At any instant, it maintains a database instance from $\mathsf{DB\text{-}Inst\text{-}Set}(\mathcal{R}, \Delta)$ and a history-set $H \subseteq \Delta$ of elements encountered along its execution. It starts with the initial database instance I_0, and the empty history-set ($H = \emptyset$). At an instant, the DMS can update the current database instance and the history-set by applying an action. An action is applied in three steps. In the first step the current database is queried using Q to retrieve some elements of interest from its active domain. In the second step, some tuples involving the retrieved elements are removed from the current database, as dictated by the variable-database instance Del . Finally, new tuples may be added to the relations of the current database instance, as dictated by Add . The newly inserted tuples may contain fresh values that were not present in the history-set, and that are injected through the fresh-input variables. We give the formal execution semantics below.

Execution semantics. The execution semantics of a DMS $\mathcal{S} = \langle I_0, \textsc{acts} \rangle$ over $\mathcal{R}$ and Δ is defined in terms of a (possibly infinite) *configuration graph* $\mathcal{C}_{\mathcal{S}}$, which has the form of a relational transition system [27, 6] equipped with additional information about the data values encountered so far. Each configuration is a pair $\langle I, H \rangle$, where $I \in \mathsf{DB\text{-}Inst\text{-}Set}(\mathcal{R}, \Delta)$ is a database instance over $\mathcal{R}$ and Δ, and $H \subseteq \Delta$ is a *history-set*, i.e., the set of values encountered in the history of the current execution of the system.

Let $\langle I, H \rangle$ be a configuration and $\alpha = \langle \vec{u}, \vec{v}, Q, Del, Add \rangle$ be an action. Consider a substitution σ from $\vec{u} \uplus \vec{v}$ to Δ. We say that σ is an *instantiating substitution* for α at $\langle I, H \rangle$ if it satisfies the following:

- for every variable $u_i \in \vec{u}$, $\sigma(u_i) \in \textsc{adom}(I)$ (action parameters are substituted with values from the current active domain);
- for every variable $v_i \in \vec{v}$, $\sigma(v_i) \notin H$ (fresh-input variables are substituted with history-fresh values);
- $\sigma|_{\vec{v}}$ is injective (fresh-input variables are assigned to pairwise distinct values);
- $I, \sigma|_{\vec{u}} \models Q$ (the action guard is satisfied).

For a pair of configurations $\langle I, H \rangle$ and $\langle I', H' \rangle$, an action $\alpha = \langle \vec{u}, \vec{v}, Q, Del, Add \rangle \in \textsc{acts}$, and a substitution σ from $\vec{u} \uplus \vec{v}$ to Δ, we have an edge $\langle I, H \rangle \xrightarrow{\alpha:\sigma} \langle I', H' \rangle$ in $\mathcal{C}_{\mathcal{S}}$, if the following conditions hold:

- σ is an instantiating substitution for α at $\langle I, H \rangle$;
- $I' = (I - \mathsf{Substitute}(Del, \sigma)) + \mathsf{Substitute}(Add, \sigma)$;
- $H' = H \cup \{\sigma(v_i) \mid v_i \in \vec{v}\}$.

An *extended run* $\hat{\rho}$ of $\mathcal{S}$ is an infinite sequence

$$\langle I_0, H_0 \rangle \xrightarrow{\alpha_0:\sigma_0} \langle I_1, H_1 \rangle \xrightarrow{\alpha_1:\sigma_1} \langle I_2, H_2 \rangle \xrightarrow{\alpha_2:\sigma_2} \langle I_3, H_3 \rangle \ldots$$

where I_0 is the initial database instance of $\mathcal{S}$, and $H_0 = \emptyset$. Note that, by definition, $H_i = \cup_{0 \leq k \leq i} \textsc{adom}(I_k)$. The *run* ρ generated by the extended run $\hat{\rho}$ is the sequence $I_0, I_1, I_2 \ldots$ of database instances appearing along $\hat{\rho}$. The *set of all runs* of a DMS $\mathcal{S}$ is denoted by $\mathsf{Runs}(\mathcal{S})$.

EXAMPLE 3.1. Consider a schema $\mathcal{R} = \{p/0, R/1, Q/1\}$, and a domain $\Delta = \{e_1, e_2, \ldots\}$. Consider a DMS over $\mathcal{R}$ and Δ, $\mathcal{S} = \langle I_0 = \{p\}, \textsc{acts} = \{\alpha, \beta, \gamma, \delta\} \rangle$ where

$$\begin{aligned}
\alpha =& \langle \emptyset, \{v_1, v_2, v_3\}, \mathsf{true}, \emptyset, \{R(v_1), R(v_2), Q(v_3), p\} \rangle \\
\beta =& \langle \{u\}, \{v_1, v_2\}, p \wedge R(u), \{p, R(u)\}, \{Q(v_1), Q(v_2)\} \rangle \\
\gamma =& \langle \{u\}, \emptyset, p \wedge \neg Q(u), \{p, R(u)\}, \emptyset \rangle \\
\delta =& \langle \{u_1, u_2\}, \emptyset, \neg p \wedge Q(u_1) \wedge (R(u_2) \vee Q(u_2)), \\
& \{Q(u_1), R(u_2)\}, \emptyset \rangle
\end{aligned}$$

A run of the above system is depicted in Figure 1. Notice that once an element is deleted from the current database instance, it is never re-introduced, due to the history-fresh policy. ■

DMSs are very expressive. The following example, following the artifact-centric paradigm [24, 12, 17], gives a glimpse about their modeling power.

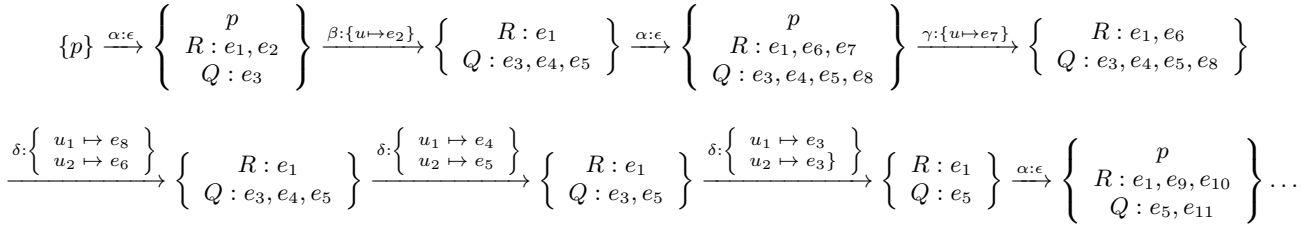


Figure 1: A run of Example 3.1. Recall that the schema is $\{p/0, R/1, Q/1\}$. For ease of readability, we omit the tuple notation $\langle e \rangle$ and simply write e.

EXAMPLE 3.2. Example in the full version of the paper [4] provides the full formalization of a DMS dealing with an agency that advertises restaurant offers and manages the corresponding bookings. Specifically, the process supports B2C interactions where agents select and publish restaurant offers, while customers issue booking requests. The process is centred around the two key business artifacts of *offer* and *booking*. Intuitively, each agent can publish a dinner offer related to some restaurant; if another, more interesting offer is received by the agent, she puts the previous one on hold, so that it will be picked up again later on by the same or another agent (when it will be among the most interesting ones). Each offer can result in a corresponding booking by a customer, or removed by the agent if nobody is interested in it. Offers are customizable, hence each booking goes through a preliminary phase in which the customer indicates who she wants to bring with her to the dinner, then the agent proposes a customized prize for the offer, and finally the customer decides whether to accept it or not. This example is unbounded in many dimensions. On the one hand, unboundedly many offers can be advertised over time. On the other hand, unboundedly many bookings for the same offer can be created (and then canceled), and each such booking could lead to introduce unboundedly many hosts during the drafting stage of the booking. ■

We show in the following that several restrictions of the DMS model can be relaxed without affecting its expressive power, nor compromising our technical results. Such relaxations are essential towards capturing related models in the literature [5, 8, 6], as well as concrete specification languages like IBM GSM [26].

Adding constants to a DMS. We can extend DMS and MSO-FO to take into account a finite subset of distinguished *constant values* $\Delta_0 \subseteq \Delta$ that can be used to specify the content of the initial database instance I_0, and that may be explicitly mentioned in the definition of actions. Given a DMS equipped with constant values Δ_0, we show in [4] how to construct a constant-free DMS over the data domain $\Delta' = \Delta \setminus \Delta_0$, so that the configuration graphs of the two DMSs are isomorphic. The size of the constant-free DMS schema is exponential in the maximum arity of the relations.

Allowing Arbitrary Input Values. The semantics of a DMS requires the input values introduced via fresh variables to not have occurred in the history of the run of the DMS. We prove in Appendix D.2 that this restriction can be lifted, allowing for the input variables to be mapped to any possible value from the data domain.

Non-distinct input values. The semantics of the DMS requires that the fresh variables are injectively mapped to distinct values. We show in Appendix D.1 that this constraint is not restrictive.

Retrieving all answers of a query for bulk action in one step. We have used a *retrieve-one-answer-per-step* semantics rather than a *retrieve-all-answers-per-step* semantics, which would support the modeling of bulk operations over the database, in the style of [6].

Intuitively, in a DMS a bulk operation consists in an action that is applied *for all* the answers of its guard. Such a bulk operation can be simulated by the iterative, non-interruptible application of different standard actions, using special accessory relation to control their execution. In summary, this is done in three phases. In the first phase, the external parameters of the bulk operation are inserted into a dedicated *input relation*, so as to maintain them fixed throughout the other two phases. At the same time, a lock proposition is set, guaranteeing that no other action will interrupt the execution of the next two phases. In the second phase, an "answer accumulation" action is repeatedly executed, incrementally filling an accessory *answer relation* with the answers obtained from the guard of the bulk operation. This is needed because such answers must be computed *before* applying the bulk update. The second phase terminates when all such answers have been transferred into the accessory relation. In the third phase, the actual bulk update is applied in two passes, by iteratively considering each tuple in the answer relation, first applying all deletions, and then all additions. When the third phase terminates, the lock is unset, enabling the possibility of applying other actions. Full details of this construction are given in Appendix D.3.

4. MSO LOGIC FOR DMS: MSO-FO

We propose a powerful logical formalism to reason about the linear runs of a DMS. The formalism, called MSO-FO, combines full monadic second-order logic to reason about the linear-time properties of runs, with atomic formulae consisting of $\texttt{FOL}(\mathcal{R})$ queries, which are used to reason about the content of the encountered database instances.

We use $\mathsf{Vars}_{\mathsf{FO}} = \{x, y, x_1, \dots\}$ to denote first-order position variables, $\mathsf{Vars}_{\mathsf{SO}} = \{X, X_1, \dots\}$ to denote second-order position variables and $\mathsf{Vars}_{\mathsf{data}} = \{u, v, u_1, \dots\}$ to denote first-order data variables. We let $\mathsf{Vars} = \mathsf{Vars}_{\mathsf{FO}} \uplus \mathsf{Vars}_{\mathsf{SO}} \uplus \mathsf{Vars}_{\mathsf{data}}$.

Syntax. Formulae ϕ of MSO-FO over schema $\mathcal{R}$ are given by the following syntax:

$$\phi ::= Q@x \mid x < y \mid x \in X \mid \neg\phi \mid \phi \wedge \phi \mid \exists x.\phi \mid \exists X.\phi \mid \exists^g u.\phi$$

where x, y are first-order position variables, X is a second-order position variable, u is a first-order data variable, and Q is a $\texttt{FOL}(\mathcal{R})$ query. We write $\forall^g u.\phi$ to denote $\neg\exists^g u.\neg\phi$. Further we make use of standard abbreviations: $\forall x.\phi \equiv \neg\exists x.\neg\phi$, $\forall X.\phi \equiv \neg\exists X.\neg\phi$, etc.

The set of free variables of a formula ϕ is denoted $\mathsf{Free\text{-}Vars}(\phi)$. For a set $V \subseteq \mathsf{Vars}$, a substitution σ of V

is a mapping that maps every first-order position variable to a natural number (i.e., $\sigma|_{\mathsf{Vars_{FO}}} : \mathsf{Vars_{FO}} \cap V \to \mathbb{N}$), every second-order position variable to a subset of natural numbers (i.e., $\sigma|_{\mathsf{Vars_{SO}}} : \mathsf{Vars_{SO}} \cap V \to 2^{\mathbb{N}}$) and every data variable to an element from the domain Δ (i.e., $\sigma|_{\mathsf{Vars_{data}}} : \mathsf{Vars_{data}} \cap V \to \Delta$).

Semantics. A run ρ is an infinite sequence of database instances over $\mathcal{R}$ and Δ: $\rho = I_0, I_1, I_2, I_3 \ldots$
The *global active domain* of the run ρ, denoted $\textsc{Gadom}(\rho)$ is the union of all active domains along the run. $\textsc{Gadom}(\rho) = \bigcup_{i \geq 0} \textsc{adom}(I_i)$. An MSO-FO formula ϕ is evaluated over an infinite run $\rho = I_0, I_1, I_2, I_3 \ldots$ under a substitution σ of $\mathsf{Free\text{-}Vars}(\phi)$. If the formula holds in the run ρ under the substitution σ, we write $\rho, \sigma \models \phi$. The semantics is as expected for the standard cases (see Appendix B). For the particular cases, we have:

- $\rho, \sigma \models Q@x$ if $I_i, \sigma' \models Q$, $i = \sigma(x)$ and $\sigma' = \sigma|_{\mathsf{Free\text{-}Vars}(Q)}$
- $\rho, \sigma \models \exists^g u.\phi$ if there exists $e \in \textsc{Gadom}(\rho)$, such that $\rho, \sigma' \models \phi$, where $\textsc{Gadom}(\rho) = \bigcup_{i \geq 0} \textsc{adom}(I_i)$ and $\sigma'(u) = e$, and $\sigma'(\xi) = \sigma(\xi)$ if $\xi \neq u$.

When the formula ϕ is a sentence (i.e, $\mathsf{Free\text{-}Vars}(\phi) = \emptyset$), it can be interpreted on a run ρ under the empty substitution, denoted $\rho \models \phi$.

Example 4.1. Consider the set $\mathsf{Runs}(\mathcal{S})$ of all runs of a DMS $\mathcal{S} = \langle I_0, \text{ACTS} \rangle$. This set is MSO-FO definable by a formula $\varphi_{\mathcal{S}}^{\mathsf{Runs}}$. The formula uses set variable X_α to denote the set of positions where an α action was taken. It can be easily expressed in MSO-FO that the sets $(X_\alpha)_{\alpha \in \text{ACTS}}$ form a partition of $\mathbb{N}$. Further, we need to express the local consistency. For this, we need to say the following: $\forall x \bigwedge_{\alpha \in \text{ACTS}} (x \in X_\alpha \Rightarrow \varphi_\alpha(x))$ where $\varphi_\alpha(x)$ expresses the local consistency by action α. If $\alpha = \langle \vec{u}, \vec{v}, Q, Del, Add \rangle$, then $\varphi_\alpha(x)$ can be expressed as follows, where variables $\xi_i \in \vec{u} \uplus \vec{v}$,

$$\exists^g \vec{u}, \vec{v}.\ \bigwedge_{u \in \vec{u}} \mathsf{Active}(u)@x \wedge \bigwedge_{v \in \vec{v}} (\forall y.\, y \leq x \Rightarrow \neg \mathsf{Active}(v)@y) \wedge Q@x$$

$$\wedge \exists y.\mathsf{succ}(x, y) \wedge \bigwedge_{R/a \in \mathcal{R}} \left(\begin{array}{l} \wedge_{\langle \xi_1 \ldots \xi_a \rangle \in R^{Add}} R(\xi_1 \ldots \xi_a)@(y) \wedge \\ \wedge_{\langle \xi_1 \ldots \xi_a \rangle \in R^{Del} \setminus R^{Add}} \neg R(\xi_1 \ldots \xi_a)@(y) \end{array} \right)$$

In the above, $\mathsf{succ}(x, y)$ states that y is the successor position of x, which can be easily expressed in MSO. ■

Example 4.2. Many standard verification problems on DMS can be expressed in MSO-FO since we can characterise the runs of a DMS (cf. Example 4.1). Of particular interest is the simplest verification problem: propositional reachability. Given a DMS s over $\mathcal{R}$ and Δ and a proposition $p/0 \in \mathcal{R}$, is it possible that an execution of s ever reaches a database instance I with $p^I = \{p\}$? This can be reduced to the satisfiability checking of $\exists \rho.\rho \models (\varphi_{\mathcal{S}}^{\mathsf{Runs}} \wedge \exists x.p@x)$. ■

Model checking. We now present the model checking problem of a DMS against MSO(DMS):

Problem:	MSO/DMS-MC
Input:	A DMS $\mathcal{S}$, a MSO-FO formula ϕ.
Question:	Does $\rho \models \phi$, for every $\rho \in \mathsf{Runs}(\mathcal{S})$?

The next example shows how MSO/DMS-MC can be phrased in such a way that database constraints are incorporated in the analysis of the DMS of interest.

Example 4.3. The presence of database constraints in the dynamic system under study is a key feature, which has been extensively studied in the literature [16, 15, 13, 9, 6]. In our setting, arbitrary FO constraints can be seamlessly added, adopting the semantics, as in [6], that the application of an action is blocked whenever the resulting database instance violates one of the constraints. Given a DMS $\mathcal{S}$, an MSO-FO formula ϕ and a constraint specification on the database instances as a FOL($\mathcal{R}$) sentence ϕ_c, we can reduce the model checking problem of the constrained DMS against ϕ to an unconstrained model checking problem over $\mathcal{S}$, using as formula: $(\forall x.\phi_c@x) \Rightarrow \phi$. ■

The above model checking problem is undecidable, even by considering just simple propositional reachability properties. This is the case with related models of dynamic database-driven systems [5, 8, 6]. A proof of undecidability via a reduction from reachability over 2-counter machines can be found in [4] . Notably, undecidability for propositional reachability holds as soon as the DMS has either: *(i)* a binary predicate in $\mathcal{R}$, with all guards expressed as union of conjunctive queries (UCQ), *(ii)* two unary predicates in $\mathcal{R}$, and FOL guards.

Theorem 4.1. MSO/DMS-MC *is undecidable.*

Towards obtaining decidability of model checking for variants of FO temporal logics, the approaches in the literature either restrict the capabilities of the model, the database schema, the verification logic, or a combination thereof [10]. Here we take a different approach — that of under-approximation.

5. RECENCY-BOUNDEDNESS

As mentioned in the previous section, even propositional reachability is undecidable unless the relational schema of the database is severely restricted. This motivates the study of under-approximate analysis of the DMS. We propose an under-approximation that is parametrised (by an integer **b**) and is exhaustive. That is, more behaviours are captured (in other words, more runs can be analysed) with higher values of **b**, and in the limit it captures all finite behaviours of the DMS. The under-approximate analysis works over arbitrary (unrestricted) schema. Our under approximation is called recency boundedness.

b-restricted actions. In a recency bounded DMS the actions are restricted to act only on the **b** most recent elements in the database. The guards can query the entire database, but the data values that can be retrieved as the result of a query will be only from the **b** recent elements of the database instance. Thus the deletions cannot involve less recent elements. The newly added data values cannot participate in a relation with less recent elements either. This restriction still allows the transitions to reason about all elements in the current database instance. For example, the properties that all elements must satisfy (regardless of their recency), may be stated as a clause in the guard of an action. However, all elements cannot be *acted* on i.e. they cannot be deleted, nor new facts involving them can be added.

The most recent **b** elements are taken relatively to the current database instance. Thus it is possible that an old element which is not in the **b**-recency window eventually enters the **b**-recency window. This happens if more recent elements were deleted from the current database instance, exposing the concerned element.

Sequence numbers. In order to reason about recency, we assume that every element e gets a sequence number seq_no(e) when it is added to the database. An element which is added later/more recently gets a higher sequence number. If there are multiple fresh elements that are added in one action, these elements are given different and unique

sequence numbers in the order in which they appear. Thus, these fresh elements are ordered amongst themselves, and their sequence number is higher than any other sequence number present in the current active domain. Since we have a countably infinite supply of sequence numbers, we do not reuse sequence numbers. That means, even if an element is deleted from the database, its sequence number will not be used by a later element. The sequence numbers may be also thought of as a way of (abstractly) time-stamping elements as they enter the active domain.

$\mathsf{Recent}_\mathbf{b}$. Given a database instance I and a sequence-numbering $\mathtt{seq\_no} : \textsc{adom}(I) \to \mathbb{N}$, we define the **b**-*recent active domain* of I wrt. $\mathtt{seq\_no}$, denoted $\mathsf{Recent}_\mathbf{b}(I, \mathtt{seq\_no})$, to be the *maximal* set $D \subseteq \textsc{adom}(I)$ with $|D| \leq \mathbf{b}$, such that for every (recent) element $e' \in D$ and every (non-recent) element $e \in \textsc{adom}(I) \setminus D$, we have $\mathtt{seq\_no}(e) < \mathtt{seq\_no}(e')$. That is, the set $\mathsf{Recent}_\mathbf{b}(I, \mathtt{seq\_no})$ contains the **b** most-recent elements from $\textsc{adom}(I)$ according to the sequence numbering $\mathtt{seq\_no}$. Notice that, thanks to maximality, $|\mathsf{Recent}_\mathbf{b}(I, \mathtt{seq\_no})| < \mathbf{b}$ if, only if $|\textsc{adom}(I)| < \mathbf{b}$.

We are now ready to formally define the **b**-bounded execution semantics for DMSs.

The b-bounded configuration graph $\mathcal{C}_\mathcal{S}^\mathbf{b}$ of a DMS $\mathcal{S}$ is given as follows. A configuration is a tuple $\langle I, H, \mathtt{seq\_no}\rangle$ where $\mathtt{seq\_no} : H \to \mathbb{N}$ is an injective function assigning sequence numbers to the data values in the history-set. For an action $\alpha = \langle \vec{u}, \vec{v}, Q, Del, Add\rangle \in \textsc{acts}$ and a substitution σ from $\vec{u} \uplus \vec{v}$ to Δ, we write $\langle I, H, \mathtt{seq\_no}\rangle \xrightarrow{\alpha:\sigma}_\mathbf{b} \langle I', H', \mathtt{seq\_no}'\rangle$ if

1. $\langle I, H\rangle \xrightarrow{\alpha:\sigma} \langle I', H'\rangle$ in $\mathcal{C}_\mathcal{S}$.
2. $\sigma(u) \in \mathsf{Recent}_\mathbf{b}(I, \mathtt{seq\_no})$ for each $u \in \vec{u}$ (That is, the values retrieved by the query must be among the b-most recent elements of the current database instance I).
3. $\mathtt{seq\_no}'$ is an injective map from H' to $\mathbb{N}$. It agrees with $\mathtt{seq\_no}$ on all data values in H (note that $H \subseteq H'$). For each fresh-input variable $v \in \vec{v}$, $\mathtt{seq\_no}'(\sigma(v)) > \mathtt{seq\_no}(e)$ for all $e \in H$ (i.e., the fresh elements that are added to the database get higher sequence numbers than the elements in H since they are more recent).
4. If $\vec{v} = \langle v_1, \ldots, v_\eta\rangle$ then for every $1 \leq i < j \leq \eta$, we have $\mathtt{seq\_no}'(v_i) < \mathtt{seq\_no}'(v_j)$. (The sequence number of the fresh elements are ordered according to their appearance in $\vec{v}$.)

Notice that Item 2 is a condition on the substitutions, rather than on transitions. Thus $\mathcal{C}_\mathcal{S}^\mathbf{b}$ has fewer edges than $\mathcal{C}_\mathcal{S}$. In Item 4 what is important is that each fresh element gets a pairwise different sequence number which is higher than that of the entire history. But then, in our decidability proof, we need to guess the order between fresh elements at every step. Fixing an order beforehand simplifies the encoding later on.

A **b**-*bounded extended run* $\hat{\rho}$ of $\mathcal{S}$ is an infinite sequence $\langle I_0, H_0, \mathtt{seq\_no}_0\rangle \xrightarrow{\alpha_0:\sigma_0}_\mathbf{b} \langle I_1, H_1, \mathtt{seq\_no}_1\rangle \xrightarrow{\alpha_1:\sigma_1}_\mathbf{b} \langle I_2, H_2, \mathtt{seq\_no}_2\rangle \xrightarrow{\alpha_2:\sigma_2}_\mathbf{b} \langle I_3, H_3, \mathtt{seq\_no}_3\rangle \ldots$ where I_0 is the initial database instance of $\mathcal{S}$, $H_0 = \emptyset$ and $\mathtt{seq\_no}_0$ is the empty (trivial) sequence-numbering. The **b**-*bounded run* ρ generated by the **b**-bounded extended run $\hat{\rho}$ is the sequence $I_0, I_1, I_2 \ldots$ of database instances appearing along $\hat{\rho}$. The *set of all* **b**-*bounded runs* of a DMS $\mathcal{S}$ is denoted $\mathsf{Runs}_\mathbf{b}(\mathcal{S})$.

Example 5.1. The run depicted in Figure 1 is a 2-recency-bounded run. ■

Example 5.2. Consider the restaurant booking agency example sketched in Section 3 and detailed in [4] . Since the agency has a fixed number of agents/customers, this number indirectly witnesses also how many booking offers can be simultaneously managed by the company. Suppose now that the company works with the following strategy: an agent temporarily freezes the management of an offer because a more interesting (in terms of potential revenue and/or expiration time) offer is received. Furthermore, let us assume that once a booking is closed, it is stored in the database for historical/audit reasons, but never modified in the future courses of execution.

The DMS capturing this example can consequently query the entire (unbounded) logged history of bookings so as, e.g., to check whether a customer finalized at least a given number of bookings in the past. This query can be used to characterize when a customer is gold and, in turn, to tune the actual DMS behavior depending on this. Furthermore, the DMS can manipulate unboundedly many offers over time, following the "last-in first-out" strategy that an offer is picked up or resumed only if the management of all higher-priority offers has been completed, and no higher-priority offer is received.

If we now put a bound on the maximum number of hosts that can be added by a customer to a booking, we can derive a number k_{mb} that indicates how many values need to be simultaneously manipulated in the worst case so as to handle the current, highest-priority offers. This, in turn, tells us that recency-bounded model checking of this unbounded DMS coincides with exact model checking when the bound is $\geq k_{mb}$. ■

Recency-bounded model checking. The problem is parametrised by a bound on recency.

Problem:	Recency-bounded-MSO/DMS-MC
Input:	A DMS $\mathcal{S}$, a MSO-FO formula ϕ, a natural number **b**,
Question:	Does $\rho \models \phi$, for every $\rho \in \mathsf{Runs}_\mathbf{b}(\mathcal{S})$?

Theorem 5.1. Recency-bounded-MSO/DMS-MC *is decidable.*

The proof of Theorem 5.1 is given in the next section.

6. DECIDABILITY OF RECENCY-BOUNDED MODEL CHECKING

We prove the decidability of recency bounded model checking problem by means of a symbolic encoding of runs. The symbolic encoding takes the form of finitely labelled nested words [3]. We show that the set of all valid encodings of recency bounded runs is expressible in monadic second-order logic over nested words. We also show that the MSO-FO specification over runs can be translated syntactically to monadic second-order logic over nested words. Thus we reduce the **b**-recency-bounded model checking problem to satisfiability problem of monadic second-order logic over nested words, which is decidable [3].

The encoding of **b**-bounded runs using nested words and expressing their validity in MSO over nested words is given in Section 6.3 and Section 6.4 respectively. The translation of MSO-FO specifications into MSO over nested word encodings is given in Section 6.5. First we will explain the symbolic abstraction used for the encoding in Section 6.1 and recall nested words in Section 6.2.

6.1 Symbolic abstraction

Consider a **b**-bounded run: $\rho = I_0, I_1, I_2, I_3, \ldots$. Each database instance I_i that appears on this run is potentially unbounded. For the sake of decidability we want our symbolic representation to be a word over finite alphabet.

Towards this we will first consider a **b**-bounded extended run $\hat{\rho} = \langle I_0, H_0, \mathtt{seq\_no}_0\rangle \xrightarrow{\alpha_0:\sigma_0}_{\mathbf{b}} \langle I_1, H_1, \mathtt{seq\_no}_1\rangle \xrightarrow{\alpha_1:\sigma_1}_{\mathbf{b}} \langle I_2, H_2, \mathtt{seq\_no}_2\rangle \dots$ generating ρ, and the sequence of $\langle$action **:** substitution$\rangle$ pairs appearing along $\hat{\rho}$.

A sequence $\mathsf{Gen} = \langle\alpha_0 : \sigma_0\rangle\langle\alpha_1 : \sigma_1\rangle\langle\alpha_2 : \sigma_2\rangle \dots$ of $\langle$action **:** substitution$\rangle$ pairs generates a unique $\hat{\rho}$ (if it exists) by following the semantics. However Gen is also not finitely labelled. The substitutions σ_i maps variables to domain Δ, leaving the set of all such substitutions an infinite set.

Hence we go for the recency-indexing abstraction of a substitution. The recency-indexing abstraction, instead of mapping a variable to an element e, maps it to its relative recency in the current database. The recency-indexing abstraction s of a substitution σ is determined by the current sequence-numbering. We explain this below.

Consider a **b**-bounded extended run $\hat{\rho} =$

$$\left(\langle I_j, H_j, \mathtt{seq\_no}_j\rangle \xrightarrow{\alpha_j:\sigma_j}_{\mathbf{b}} \langle I_{j+1}, H_{j+1}, \mathtt{seq\_no}_{j+1}\rangle\right)_{j\geq 0}$$

For each substitution $\sigma_j : \vec{u}_j \uplus \vec{v}_j \to \Delta$ appearing in $\hat{\rho}$, where $\vec{u}_j = \alpha_j{\cdot}\mathsf{free}$ and $\vec{v}_j = \alpha_j{\cdot}\mathsf{new}$, we have $\sigma_j(u) \in \mathsf{Recent}_{\mathbf{b}}(I_j, \mathtt{seq\_no}_j)$ for all $u \in \vec{u_j}$ thanks to the **b**-boundedness.

The recency-indexing abstraction of σ_j at I_j wrt. the sequence numbering $\mathtt{seq\_no}_j$ is a mapping $s_j : \vec{u}_j \uplus \vec{v}_j \to \{-n, -n+1, \dots, 0, 1, \dots \mathbf{b}-1\}$ where $n = |\vec{v_j}|$ such that

r1. if $\vec{v}_j = \langle v_1, \dots, v_n\rangle$ then $s_j(v_i) = -i$

r2. for $u \in \vec{u}_j$, $s_j(u) \in \{0, 1, \dots, \mathbf{b}-1\}$

r3. for $u \in \vec{u}_j$, $s_j(u)$ represents the recency of $\sigma_j(u)$ at I wrt. the sequence-numbering $\mathtt{seq\_no}_j$. More precisely, $s_j(u) = i$ if $i = |\{e \in \textsc{adom}(I_j) \mid \mathtt{seq\_no}_j(e) > \mathtt{seq\_no}_j(\sigma_j(u))\}|$. For example, if $\sigma_j(u)$ is the most-recent element, then $s_j(u) = 0$.

Notice that given a recency bound **b** and a DMS $\mathcal{S} = \langle I_0, \textsc{acts}\rangle$, the set of all symbolic substitutions s is finite. Let us denote this set by $\mathsf{SymSubs}(\mathcal{S}, \mathbf{b})$. Let $\mathsf{SymSubs}(\alpha, \mathbf{b}) = \{s : \vec{u}\uplus\vec{v} \to \{-n, -n+1, \dots, 0, 1, \dots, \mathbf{b}-1\} \mid \vec{u} = \alpha{\cdot}\mathsf{free}, \vec{v} = \alpha{\cdot}\mathsf{new}, n = |\vec{v}|$ and s satisfies conditions r1 and r2 above$\}$. We have $\mathsf{SymSubs}(\mathcal{S}, \mathbf{b}) = \uplus_{\alpha\in\textsc{acts}}\mathsf{SymSubs}(\alpha, \mathbf{b})$. Let the symbolic alphabet $\mathsf{symAlph}_{\mathcal{S},\mathbf{b}}$ be the finite set $\{\langle\alpha, s\rangle \mid \alpha \in \textsc{acts}$ and $s \in \mathsf{SymSubs}(\alpha, \mathbf{b})\}$.

To every **b**-bounded extended run $\hat{\rho}$, we can identify a corresponding word $w_{\hat{\rho}} \in (\mathsf{symAlph}_{\mathcal{S},\mathbf{b}})^\omega$ by taking the recency-indexing abstraction of the substitutions. Let's denote this correspondence by a mapping Abstr from **b**-bounded extended runs to $(\mathsf{symAlph}_{\mathcal{S},\mathbf{b}})^\omega$. That is $\mathsf{Abstr}(\hat{\rho}) = w_{\hat{\rho}}$. We extend in the natural way the definition of the abstraction function to finite prefixes of **b**-bounded extended runs.

The mapping Abstr is not injective. However, if two **b**-bounded extended runs $\hat{\rho}$ and $\hat{\rho}'$ have the same abstract generating sequence $w = \mathsf{Abstr}(\hat{\rho}) = \mathsf{Abstr}(\hat{\rho}')$, then $\hat{\rho}$ and $\hat{\rho}'$ are equivalent modulo permutations of the data domain Δ (i.e. there exists a bijection $\lambda : \textsc{Gadom}(\hat{\rho}) \to \textsc{Gadom}(\hat{\rho}')$ such that λ is an isomorphism from I_i onto I'_i for every $i \geq 0$, see Appendix C for the detailed proof). This notion of invariance under renaming is very well known in computer science and is discussed in [23]. If we assume a total ordering on the domain Δ, then we can define a canonical $\hat{\rho}$ as the representative of all such equivalent ones. Let the data domain be $\{e_1, e_2, \dots\}$ with the ordering $e_i < e_j$ if $i < j$.

A **b**-bounded extended run $\hat{\rho}$ is canonical if it satisfies the following invariants along the run:

- For every i, for every j, if $e_j \in H_i$ then $\mathtt{seq\_no}_i(e_j) = j$.
- For every i, if $v_k \in \alpha_i{\cdot}\mathsf{new}$ is the k^{th} fresh input variable, then $\sigma(v_k) = e_{n+k}$ where $n = |H_i|$.

The second invariant implies the following:

- For every i, H_i is of the form $\{e_1, \dots, e_n\}$ for some $n \in \mathbb{N}$. That is, there are no gaps in the history.

The mapping Abstr is not surjective either. We will define a partial concretizing function Concr from infinite words $w \in (\mathsf{symAlph}_{\mathcal{S},\mathbf{b}})^\omega$ to **b**-bounded extended runs such that if w is a valid abstraction then $\mathsf{Concr}(w)$ is the canonical extended run $\hat{\rho}$ with $\mathsf{Abstr}(\hat{\rho}) = w$. In order to do so, let us first denote the k-long prefix of w (respectively $\hat{\rho}$) by w_k (respectively $\hat{\rho}_k$). Similarly to the abstraction function, we also extend the concretisation function to finite prefixes of infinite words from $(\mathsf{symAlph}_{\mathcal{S},\mathbf{b}})^\omega$. It is easy to see that w is a valid abstract run if and only if, for every $k \geq 0$, w_k is the prefix of a valid abstract run. In that case, $\hat{\rho} = \mathsf{Concr}(w)$ amounts to the limit of $\mathsf{Concr}(w_k) = \hat{\rho}_k$ when $k \to +\infty$. Also, for $k \geq 0$, if w_k is the prefix of a valid run, then $\hat{\rho}_k = \mathsf{Concr}(w_k)$ is of the form $\left(\langle I_j, H_j, \mathtt{seq\_no}_j\rangle \xrightarrow{\alpha_j:\sigma_j}_{\mathbf{b}} \langle I_{j+1}, H_{j+1}, \mathtt{seq\_no}_{j+1}\rangle\right)_{0\leq j<k}$. We define in what follows $\mathsf{Concr}(w_k)$ by induction on its length k.

For the empty word $w_0 = \epsilon \in (\mathsf{symAlph}_{\mathcal{S},\mathbf{b}})^*$ we define $\mathsf{Concr}(w_0) = \langle I_0, H_0, \mathtt{seq\_no}_0\rangle$ where $H_0 = \emptyset$ and $\mathtt{seq\_no}_0 = \epsilon$, the empty mapping. Suppose $w_{k+1} = w_k\langle\alpha_k, s\rangle$ where $\alpha_k = \langle\vec{u}, \vec{v}, Q, Del, Add\rangle$. $\mathsf{Concr}(w_{k+1})$ is not defined if $\mathsf{Concr}(w_k)$ is not defined. Suppose $\mathsf{Concr}(w_k)$ is defined and is of the form $\left(\langle I_j, H_j, \mathtt{seq\_no}_j\rangle \xrightarrow{\alpha_j:\sigma_j}_{\mathbf{b}} \langle I_{j+1}, H_{j+1}, \mathtt{seq\_no}_{j+1}\rangle\right)_{0\leq j<k}$. $\mathsf{Concr}(w_{k+1})$ is defined if, and only if, the following condition holds:

Condition Cnd **:** There exists a substitution $\sigma : \vec{u} \to \textsc{adom}(I_k)$ such that

- $I_k, \sigma \models Q$ and
- s restricted to $\vec{u}$ is the recency-indexing abstraction of σ at I_k wrt. $\mathtt{seq\_no}_k$.

Assuming condition Cnd holds, w_{k+1} is also the prefix of a valid abstract run and $\mathsf{Concr}(w_{k+1})$ is defined as follows. Let n be the size of H_k, i.e. $n = |H_k|$. We define the substitution $\sigma_k : \vec{u} \uplus \vec{v} \to \Delta$ as follows: $\sigma_k|_{\vec{u}} = \sigma$ and $\sigma_k(v_i) = e_{n+i}$ for every $i : 1 \leq i \leq |\vec{v}|$. Notice that 1) $\sigma_k(u) = \sigma(u) \in \textsc{adom}(I_k)$ for every $u \in \vec{u}$, 2) $\sigma_k(v) \notin H_k$ for every $v \in \vec{v}$, 3) $\sigma_k|_{\vec{v}}$ is injective, and 4) since $I_k, \sigma \models Q$, and $\sigma_k|_{\vec{u}} = \sigma$, we have that $I_k, \sigma_k|_{\vec{u}} \models Q$. Thus, σ_k is an instantiating substitution for α_k at $\langle I_k, H_k\rangle$. Moreover, if we define the set $H_{k+1} = H_k \cup \{e_{n+1}, \dots, e_{n+|\vec{v}|}\}$ and the database instance $I_{k+1} = (I_k - \mathsf{Substitute}(Del, \sigma_k)) + \mathsf{Substitute}(Add, \sigma_k)$, then we have that $\langle I_k, H_k\rangle \xrightarrow{\alpha_k,\sigma_k} \langle I_{k+1}, H_{k+1}\rangle$.

Furthermore, since the restriction of s to $\vec{u}$ is the recency-indexing abstraction of σ at I_k wrt. $\mathtt{seq\_no}_k$, we deduce that $\sigma_k(u) = \sigma(u) \in \mathsf{Recent}_{\mathbf{b}}(I_k, \mathtt{seq\_no}_k)$. Thus, the transition $\langle I_k, H_k\rangle \xrightarrow{\alpha_k,\sigma_k} \langle I_{k+1}, H_{k+1}\rangle$ is also allowed by the **b**-recency

semantics and, assuming that we define $\texttt{seq\_no}_{k+1}$ by $\texttt{seq\_no}_{k+1}|_{H_k} = \texttt{seq\_no}_k$ and by $\texttt{seq\_no}_{k+1}(e_{n+i}) = n+i$ for every $i : 1 \leq i \leq |\vec{v}|$, we have that $\langle I_k, H_k, \texttt{seq\_no}_k\rangle \xrightarrow{\alpha_k,\sigma_k}_{\mathbf{b}} \langle I_{k+1}, H_{k+1}, \texttt{seq\_no}_{k+1}\rangle$ and $\mathsf{Concr}(w_{k+1}) = \left(\langle I_j, H_j, \texttt{seq\_no}_j\rangle \xrightarrow{\alpha_j:\sigma_j}_{\mathbf{b}} \langle I_{j+1}, H_{j+1}, \texttt{seq\_no}_{j+1}\rangle\right)_{0\leq j<k+1}$.

Now, for an infinite word $w \in (\mathsf{symAlph}_{\mathcal{S},\mathbf{b}})^\omega$, $\mathsf{Concr}(w)$ is defined to be the limit of $\hat{\rho}_k = \mathsf{Concr}(w_k)$ for $k \geq 0$. If defined, $\mathsf{Concr}(w)$ is a canonical run. Further, $\mathsf{Abstr}(\mathsf{Concr}(w)) = w$. Furthermore, for every w such that $w = \mathsf{Abstr}(\hat{\rho})$ for $\mathbf{b}$-bounded run $\hat{\rho}$, $\mathsf{Concr}(w)$ is defined, and $\mathsf{Concr}(w)$ and $\hat{\rho}$ are equivalent modulo permutations of the data domain. In particular, if $\hat{\rho}$ is a $\mathbf{b}$-bounded canonical run, then $\hat{\rho} = \mathsf{Concr}(\mathsf{Abstr}(\hat{\rho}))$.

Example 6.1. The abstract generation sequence corresponding to the run in Figure 1 is:
$\langle \alpha : \{v_1 \mapsto -1, v_2 \mapsto -2, v_3 \mapsto -3\}\rangle$ $\langle \beta : \{u \mapsto 1, v_1 \mapsto -1, v_2 \mapsto -2\}\rangle\langle \alpha : \{v_1 \mapsto -1, v_2 \mapsto -2, v_3 \mapsto -3\}\rangle$ $\langle \gamma : \{u \mapsto 1\}\rangle$ $\langle \delta : \{u_1 \mapsto 0, u_2 \mapsto 1\}\rangle$ $\langle \delta : \{u_1 \mapsto 1, u_2 \mapsto 0\}\rangle$ $\langle \delta : \{u_1 \mapsto 1, u_2 \mapsto 1\}\rangle$ $\langle \alpha : \{v_1 \mapsto -1, v_2 \mapsto -2, v_3 \mapsto -3\}\rangle \ldots$ ■

In order to check the consistency of an abstract generating sequence, we need to check that condition Cnd holds at every step of the sequence. To achieve this within a formalism having "decidable theories", we add more structure to the abstract generating sequence by embedding it into a nested word, which we recall in the next section.

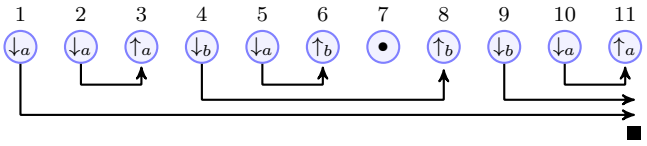
6.2 Nested words

A *visible alphabet* Σ is a finite alphabet partitioned into push letters $\Sigma_\downarrow$, pop letters $\Sigma_\uparrow$ and internal letters Σ_{int}. That is, $\Sigma = \Sigma_\downarrow \uplus \Sigma_\uparrow \uplus \Sigma_{\text{int}}$. Given a word $w = a_1 a_2 \ldots$ over the visible alphabet Σ, we say i is a $\Sigma_\downarrow$ *position* if $a_i \in \Sigma_\downarrow$. Similarly we define $\Sigma_\uparrow$ positions and Σ_{int} positions.

A nested word is a pair $(w, \triangleright)$ where w is a word over a Σ and $\triangleright \subset \{1, \ldots, |w|\}^2$ is the *maximal* binary nesting relation relating $\Sigma_\downarrow$ positions to $\Sigma_\uparrow$ positions such that:

- if $i \triangleright j$ then $i < j$. The nesting relation preserves the linear order.
- if $i \triangleright j$ and $i' \triangleright j'$ are two distinct pairs (either $i \neq i'$ or $j \neq j'$) then $|\{i, i', j, j'\}| = 4$. Two different nesting edges are vertex-disjoint.
- for every $i \triangleright j$ and $i' \triangleright j'$ we do not have $i < i' < j < j'$. The nesting edges must not cross.
- if $i \triangleright j$ and $i < i' < j$ for some $\Sigma_\downarrow$ position i', then there exists j' such that $i' \triangleright j'$. Similarly if $i \triangleright j$ and $i < j' < j$ for some $\Sigma_\uparrow$ position j', then there exists i' such that $i' \triangleright j'$.

Example 6.2. A nested word over the visible alphabet given by $\Sigma_\downarrow = \{\downarrow_a, \downarrow_b\}$, $\Sigma_\uparrow = \{\uparrow_a, \uparrow_b\}$ and $\Sigma_{\text{int}} = \{\bullet\}$ is given below:



■

Note that, given a word w over a visible alphabet Σ, the nesting relation $\triangleright$ is uniquely defined.

Monadic Second-Order Logic ($\texttt{MSO}_{\texttt{NW}}$) over nested words extends MSO over words with an additional binary predicate $\triangleright$ that links a matching push-pop pair. This is in fact the same logic of [20] where the guessed second-order matching variable is built-in in the structure. We assume an unbounded supply of position variables $\{x, y, \ldots\}$ and set variables $\{X, Y, \ldots\}$. The syntax of $\texttt{MSO}_{\texttt{NW}}$ is given by:

$$\varphi := a(x) \mid x < y \mid x \triangleright y \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi \mid \exists X.\varphi$$

Here a ranges over the visible alphabet Σ. The position variables x, y range over positions of the nested word. The set variable X ranges over sets of positions of the nested word. The semantics is as expected.

Example 6.3. Let x and y be two free first-order variables. Suppose we want to state that the first $\downarrow_a$ labelled position after x and the first $\uparrow_b$ labelled position after y are related by a nesting edge. This property can be stated by a formula with two free variables:

$$\varphi_{a,b}(x,y) \equiv \exists x_1 \exists y_1 \downarrow_a(x_1) \wedge \uparrow_b(y_1) \wedge x < x_1 \wedge y < y_1 \wedge x_1 \triangleright y_1 \\ \wedge \forall z\,(x < z < x_1 \Rightarrow \neg\downarrow_a(z)) \wedge (y < z < y_1 \Rightarrow \neg\uparrow_b(z))$$

On Example 6.2, all pairs of positions (i, j) with $2 \leq i \leq 4$ and $1 \leq j \leq 5$ satisfy the above formula. ■

Fact 1 ([3]). *Satisfiability of $\texttt{MSO}_{\texttt{NW}}$ is decidable.*

6.3 Encoding a run as a nested word

Let us fix a DMS $\mathcal{S} = \langle I_0, \text{ACTS}\rangle$, over a set of values Δ and a schema $\mathcal{R}$, and a recency bound $\mathbf{b}$ for the rest of this section. We will first provide the visible alphabet, and then describe the encoding.

Visible alphabet of the encoding. The visible alphabet $\Sigma = \Sigma_{\text{int}} \uplus \Sigma_\uparrow \uplus \Sigma_\downarrow$ where

- $\Sigma_{\text{int}} = \{\textcircled{\alpha : s} \mid \langle \alpha, s\rangle \in \mathsf{symAlph}_{\mathcal{S},\mathbf{b}}\} \cup \{\textcircled{I_0}\}$
- $\Sigma_\uparrow = \{\uparrow_0, \ldots, \uparrow_{\mathbf{b}-1}\}$
- $\Sigma_\downarrow = \{\downarrow_{-\eta}, \ldots, \downarrow_0, \ldots, \downarrow_{\mathbf{b}-1}\}$ where $\eta = \max_{\alpha \in \text{ACTS}} |\alpha{\cdot}\mathsf{new}|$

The internal letters represent the symbolic abstraction described in Section 6.1. Further, we provide a letter $\textcircled{I_0}$ to represent the initial database I_0.

The pop letters and push letters as well as the nesting relation will be used to trace the elements (or datavalues) in an encoding. We explain this more in detail when describing the encoding.

Encoding. As alluded to in Section 6.1, we need to enrich the abstract generating sequences. We go for a richer encoding where each step is followed by an encoding of the effect of the action on the database. The effect of an action involves a) adding some relational tuples to the current database instance; b) deleting some relational tuples from the current database instance. The above two items can induce 1) adding new elements to the current active domain. 2) deleting some elements from the current active domain;

The effects a) and b) are explicitly mentioned in the action α. The number of newly added fresh elements is also explicit in α. Hence the induced effect 1) as well as effects a) and b) can be deduced from the action encoding $\textcircled{\alpha : s}$.

However, the induced effect 2 is not predictable from $\textcircled{\alpha : s}$. The reason is that, even when an element is involved only in deletions, it is not clear whether this element can be removed

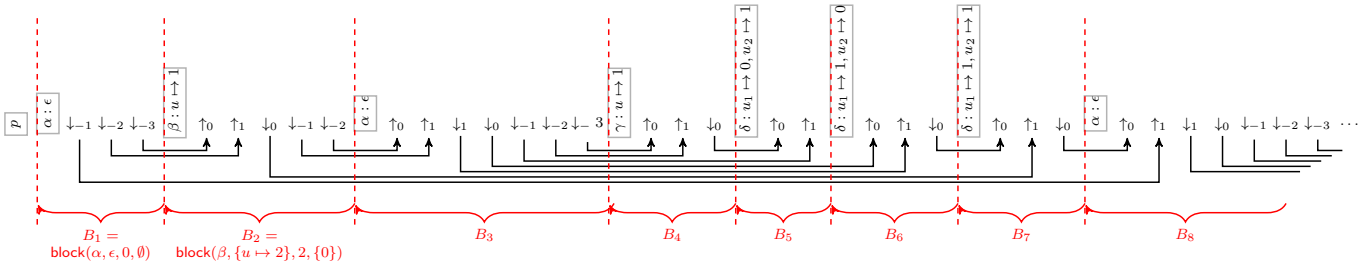


Figure 2: Nested word encoding of the run in Figure 1

from the current active domain since it may be participating in some other relations which were not tested by the action α. Thanks to recency boundedness, we know that if some element is deleted then it must be from the $\mathbf{b}$ most recent elements.

Another subtle problem is that at every configuration, the active domain need not contain $\mathbf{b}$ elements. Let m be $\min\{\mathbf{b}, |\text{ADOM}(I)|\}$, which gives the cardinality of the set $\mathsf{Recent}_{\mathbf{b}}(I, \texttt{seq\_no})$. The value of m at a configuration is not defined from an action encoding $\overline{(\alpha : s)}$. Hence our encoding will also guess the value of m. Later, we will use $\texttt{MSO}_{\texttt{NW}}$ to ensure that our guesses were indeed right.

We will provide an encoding which will "guess" the following: 1) the size of $\mathsf{Recent}_{\mathbf{b}}(I, \texttt{seq\_no},)$ at any configuration and 2) those recent elements which are deleted from the active domain, (or equivalently, it will "guess" those recent elements which are surviving in the active domain).

Suppose that $|\mathsf{Recent}_{\mathbf{b}}(I, \texttt{seq\_no})| = m$ in the current configuration $\langle I, H, \texttt{seq\_no}\rangle$. Consider an action α under an abstract substitution $s : \alpha{\cdot}\mathsf{free} \to \{0, 1, \ldots m-1\}$. Further suppose that the elements with the recency index $J = \{i_1, i_2, \ldots i_\ell\}$ with $J \subseteq \{0, 1, \ldots m-1\}$ are surviving after the action. That means, the elements with recency index in $\{0, 1, \ldots m-1\} \setminus J$ are deleted from the current database. The action along with its effect is encoded by the following visible word, where $n = |\alpha{\cdot}\mathsf{new}|$:

$$\overline{(\alpha : s)} \uparrow_0 \uparrow_1 \ldots \uparrow_{m-1} \downarrow_{i_1} \ldots \downarrow_{i_\ell} \downarrow_{-1} \ldots \downarrow_{-n}$$

with $m - 1 \geq i_1 > \cdots > i_\ell \geq 0$. The above word is parametrised by α, s, m and J. We denote it by $\mathsf{block}(\alpha, s, m, J)$.

Intuitively, we delete all the elements from $\mathsf{Recent}_{\mathbf{b}}(I)$ temporarily, and insert back all the surviving ones (as dictated by J). Notice that the order of the indices of the elements from J make sure that in the later blocks a more-recent element is popped before a less-recent one. Finally, the fresh elements are pushed in.

Our encoding of a $\mathbf{b}$-bounded run is a sequence of such blocks prefixed by $\overline{(I_0)}$:

$$\overline{(I_0)}\,\mathsf{block}(\alpha_1, s_1, m_1, J_1)\,\mathsf{block}(\alpha_2, s_2, m_2, J_2)\cdots$$

The nesting edges are induced on the word due to the visibility of the alphabet. Our encoding has an interesting feature: the number of unmatched pushes in the prefix upto $\overline{(\alpha_j : s_j)}$ is $|\text{ADOM}(I_j)|$ where I_j is the database instance at which α_j is executed. The set $\mathsf{Recent}_{\mathbf{b}}(I_j)$ corresponds to the innermost (rightmost) $|\mathsf{Recent}_{\mathbf{b}}(I_j)|$ unmatched pushes in the prefix. Note that, here an unmatched push in the prefix means it is not matched within the prefix; it may be matched after the prefix.

EXAMPLE 6.4. The nested-word encoding of the run from Figure 1 is depicted in Figure 2. It is 2-recency bounded. The indices 0 and 1 refer to the most recent and second most recent elements. Negative indices refer to freshly added elements.

Notice that in this example B_1 is the only block where $|\mathsf{Recent}_{\mathbf{b}}(I)| < \mathbf{b} = 2$. On all successive blocks $|\mathsf{Recent}_{\mathbf{b}}(I)| = 2$. In block B_1, indices 0 and 1 are not used.

In block B_2, the substitution uses only the second-most recent element, denoted by $u \mapsto 1$. However, the entire $\mathsf{Recent}_{\mathbf{b}}(I)$ is popped. Since the second-most recent element is deleted in B_2, it is not pushed back, but the most recent element is pushed back (denoted by $\downarrow_0$). Hence for B_2, we have $J_2 = \{0\}$.

Action α of block B_3 does not use/modify any element from $\mathsf{Recent}_{\mathbf{b}}(I)$. However, since $\mathsf{Recent}_{\mathbf{b}}(I)$ is non-empty, it is popped entirely and pushed back. Notice the inversion in the order of the sequence of pops and that of pushes. This inversion maintains that less-recent elements are pushed before the more-recent elements.

Notice also that the number of pushes on the left of a block which are not matched on the left correspond to the number of elements in the active domain before the execution of the block. For example, the database instance I_4 just before the execution of block B_5 has 6 elements, and the $\text{ADOM}(I_7)$ has just two elements.

Notice that the abstract substitution need not be injective (cf. block B_7), and need not assign recent values to variables in the order of their recency (cf. blocks B_5 and B_6).

Notice also that the set J is not determined by the action name nor the abstract substitution s. ■

6.3.1 *Conditions for valid encodings*

Consider any nested word W over the visible alphabet $\Sigma = \Sigma_{\text{int}} \uplus \Sigma_\uparrow \uplus \Sigma_\downarrow$ of the form $\overline{(I_0)}\,\mathsf{block}(\alpha_1, s_1, m_1, J_1)\,\mathsf{block}(\alpha_2, s_2, m_2, J_2)\ldots$. Let $w \in (\mathsf{symAlph}_{\mathcal{S},\mathbf{b}})^\omega$ be obtained by the Σ_{int} projection of W. Let W_i denote the prefix of W upto block_i, and w_i be the corresponding projection.

For $i \geq 0$, we say that a prefix W_{i+1} is *good* if $\mathsf{Concr}(w_i)$ is defined. Let $C_i = \langle I_i, H_i, \texttt{seq\_no}_i\rangle$ be the last configuration of $\mathsf{Concr}(w_i)$ in this case. Further we require the following:

1. $m_{i+1} = |\mathsf{Recent}_{\mathbf{b}}(I_i, \texttt{seq\_no}_i)|$;
2. $j \in J_i$ iff, letting e be the element of recency-index j in C_i, there are a relation $R \in \mathcal{R}$ and a tuple t of R involving e such that t is present in I_i but not in instantiated $\alpha_{i+1}{\cdot}Del$, or t is present in instantiated $\alpha_{i+1}{\cdot}Add$; and
3. letting σ_{i+1} be the instantiation of s_{i+1} at C_i, we have $I_i, \sigma_{i+1} \models \alpha_{i+1}{\cdot}\mathsf{guard}$.

We say W is a *valid encoding* of a $\mathbf{b}$-recency bounded run of $\mathcal{S}$ if W_i is good for every $i \geq 0$.

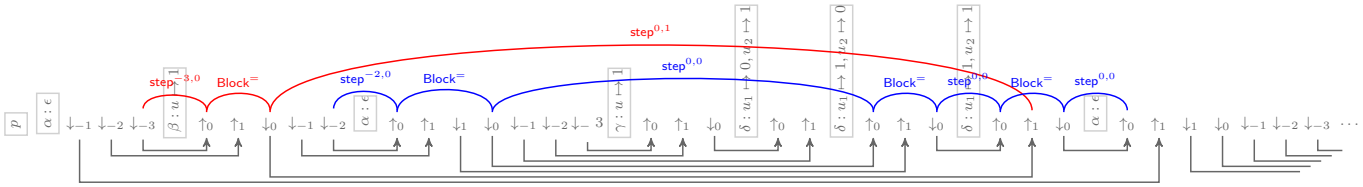


Figure 3: The relation $\mathsf{Eq}^{i,j}$ tracks the occurrences of the same element in the nested word encoding of a run. In a way it is a transitive closure of the $\mathsf{step}^{i,j}$ relations and the $\mathsf{Block}^{=}$ relations. Note that if such a transitive closure path enters a block via the relation $\mathsf{step}^{i,j}$ and exits the block via $\mathsf{step}^{i',j'}$ then $j = i'$.

$$\mathsf{Eq}^{i,j}(x,y) \equiv \forall X_{-\eta} \forall X_{-\eta+1} \ldots \forall X_{\mathbf{b}-1} \left(\left(x \in X_i \wedge \forall x_1 \forall x_2 . \left(\begin{array}{l} \bigwedge_{\eta \le \ell, m \le \mathbf{b}-1} (\mathsf{step}^{\ell,m}(x_1,x_2) \wedge x_1 \in X_\ell) \Rightarrow x_2 \in X_m \\ \wedge \bigwedge_{-\eta \le \ell \le \mathbf{b}-1} (\mathsf{Block}^{=}(x_1,x_2) \wedge x_1 \in X_\ell) \Rightarrow x_2 \in X_\ell \end{array} \right) \right) \Rightarrow y \in X_j \right)$$

Figure 4: Formula $\mathsf{Eq}^{i,j}$ which states that the element indexed by i in the block of the first argument is same as the element indexed by j in the block of the second argument. This is pictorially depicted in Figure 3.

Observe that, if W_i is good then $\mathsf{Concr}(w_i)$ is defined. Hence, if a nested word is not a valid encoding, it can be detected at the first index i such that W_i is not good by observing that conditions (1), (2) or (3) is violated. In this case $\mathsf{Concr}(w_{i-1})$ is defined since W_{i-1} is good. We will exploit this observation to express valid encodings in $\mathtt{MSO}_{\mathtt{NW}}$.

In the remainder of this section we will use the above-set indexing convention for the intuitive explanations. That is, $C_i = \langle I_i, H_i, \mathsf{seq\_no}_i \rangle$ is the last configuration of $\mathsf{Concr}(w_i)$. This means that the previous configuration of block_i (or the configuration where it is being executed) is C_{i-1}.

Remark 6.1. *If W is a valid encoding then, the number of unmatched pushes in the prefix upto block_{j+1} (excluding) is $|\textsc{adom}(I_j)|$. The set $\mathsf{Recent}_{\mathbf{b}}(I_j, \mathsf{seq\_no}_j)$ corresponds to the innermost (rightmost) $|\mathsf{Recent}_{\mathbf{b}}(I_j, \mathsf{seq\_no}_j)|$ unmatched pushes in the prefix. Note that, here an unmatched push in the prefix means it is not matched within the prefix; it may be matched after the prefix.*

We will now provide $\mathtt{MSO}_{\mathtt{NW}}$ formulae stating that these three conditions are satisfied by a nested word over Σ at all of its blocks. The conjunction of the these formulae will characterise $\mathsf{Runs}_{\mathbf{b}}(\mathcal{S})$ (which we denote by $\varphi^{\mathsf{valid}}_{\mathbf{b},\mathcal{S}}$).

6.4 Expressing valid encodings in $\mathtt{MSO}_{\mathtt{NW}}$

We first describe a few $\mathtt{MSO}_{\mathtt{NW}}$ predicates that turn out handy when stating the validity of an encoding in MSO. Such predicates are macros/abbreviation helping towards the readability of the formula describing validity.

6.4.1 Preliminary formulae

We write $\Sigma_{\text{int}}(x)$ as a shorthand for $\bigvee_{a \in \Sigma_{\text{int}}} a(x)$. Similarly we define $\Sigma_{\downarrow}(x) \equiv \bigvee_{a \in \Sigma_{\downarrow}} a(x)$ and $\Sigma_{\uparrow}(x) \equiv \bigvee_{a \in \Sigma_{\uparrow}} a(x)$.

We write $\mathsf{Block}^{=}(x,y)$ to indicate that positions x and y belong to the same block. This is a shorthand for

$$\forall z . ((\neg \Sigma_{\text{int}}(z)) \vee (z \le x \wedge z \le y) \vee (x < z \wedge y < z))$$

Notice that a block has exactly one internal letter, which indicates the action and the abstract substitution. The position labelled by such an internal letter is called *head*, and every block has a unique head. The formula $\mathsf{Block}^{=}(x,y)$ says that x and y must not be separated by an internal letter (or a head).

We now define a unary predicate with a free variable x for each relation name $R/a \in \mathcal{R}$ and choice of a recency indices $i_1, \ldots i_a \in \{0, \ldots, \mathbf{b}-1\}$. The predicate holds at a position if it is the head of a block and its block deletes a tuple $\langle e_1, \ldots e_a \rangle$ from the relation R where e_j is indexed by i_j in its block, for all $j : 1 \le j \le a$. This predicate is denoted $Del(R(i_1, \ldots i_a))@x$.

$$Del(R(i_1, \ldots i_a))@x \equiv \bigvee_{\textcircled{\alpha, s} \in \Gamma} \textcircled{\alpha, s}(x)$$

where $\Gamma = \{ \textcircled{\alpha, s} \mid \alpha \cdot Del$ contains a tuple $R(u_1 \ldots u_a)$ and $s(u_j) = i_j$ for all $1 \le j \le a \}$.

Similarly we define a unary predicate for adding a tuple to a relation as well. However in this case, the indices may refer to the fresh data values as well. Hence we have unary predicate $Add(R(i_1, \ldots i_a))@x$ for each relation name $R/a \in \mathcal{R}$ and choice of a indices $i_1, \ldots i_a \in \{-n, \ldots, 0, \ldots, \mathbf{b}-1\}$, where $n := \max_{\alpha \in \text{ACTS}} |\alpha \cdot \mathsf{new}|$.

$$Add(R(i_1, \ldots i_a))@x \equiv \bigvee_{\textcircled{\alpha, s} \in \Gamma} \textcircled{\alpha, s}(x)$$

where $\Gamma = \{ \textcircled{\alpha, s} \mid \alpha \cdot Add$ contains a tuple $R(\xi_1 \ldots \xi_a)$ and for all $1 \le j \le a$ if $\xi_j \in \alpha \cdot \mathsf{free}$ then $s(\xi_j) = i_j$ and if ξ_j is the kth fresh input variable v_k then $i_j = -k \}$.

Equality between indexed elements of different blocks. Consider the encoding of the run in Figure 2. Notice that the index -2 in the block B_1 and index 1 in the block B_2 refer to the same element (e_2 in the concrete run of Figure 1). Notice also that the element referred to by index -2 in Block B_2 is the same as the element referred to by index 0 in block B_7 (e_5 in the concrete run of Figure 1).

Given two positions x and y and indices i and j, consider following question: I*s the element referred to by index i in the block of x the same as the element referred to by index j in the block of y?* In fact, this property can be expressed in $\mathtt{MSO}_{\mathtt{NW}}$. We define below a binary predicate $\mathsf{Eq}^{i,j}(x,y)$ for the same. Indeed we will define such a predicate for every pair i, j with $-\eta \le i, j \le \mathbf{b}-1$.

Towards this, first notice that the predicate must hold

if there is a $\downarrow_i$-labelled position in the block of x that is $\triangleright$-related to $\uparrow_j$-labelled position in the block of y. This forms the basic step relation towards defining $\mathsf{Eq}^{i,j}(x,y)$.

$$\mathsf{step}^{i,j}(x,y) \equiv \exists z_1 \exists z_2.\mathsf{Block}^{=}(z_1,x) \wedge \mathsf{Block}^{=}(z_2,y) \wedge z_1 \triangleright z_2 \wedge \downarrow_i(z_1) \wedge \uparrow_j(z_2)$$

Recall that $\downarrow_i(x)$ means that the position x is labelled by the letter $\downarrow_i$. Notice that our definition of $\mathsf{step}^{i,j}(x,y)$ is directional, in the sense that x must necessarily be before y for $\mathsf{step}^{i,j}(x,y)$ to hold. The transitive closure of the above step relation gives us the required predicate $\mathsf{Eq}^{i,j}(x,y)$. Suppose the element indexed i in the block of x is e. The element e may appear with different indices at the intermediate steps. Hence we need to take a zig-zag transitive closure. Our formula uses $\mathbf{b}+\eta$ second-order position variables. Intuitively the set X_k contains the set of positions such that the element e is indexed by k in its block. Using the universal quantifier, we require that the minimal of such sets which are closed under the zig-zag transitive closure must contain y in the set X_j. The formula $\mathsf{Eq}^{i,j}(x,y)$ is depicted in Figure 4.

Notice that since step is directional, so is $\mathsf{Eq}^{i,j}(x,y)$. I.e, if $\mathsf{Eq}^{i,j}(x,y)$ then necessarily $x \leq y$ or $\mathsf{Block}^{=}(x,y)$.

Recent elements participating in a relation. Consider a relation R of arity a. We define a predicate $\mathsf{Rel\text{-}R}(x_1,i_1,x_2,i_2,\dots x_a,i_a)@y^{\ominus}$ which holds iff the database instance *before* the execution of the block of y has the tuple $\langle e_1,e_2,\dots e_a\rangle$ in relation R where, the element e_j is indexed by i_j in the block of x_j for all $j: 1 \leq j \leq a$. This predicate can be expressed in MSO, as given below.

$$\exists x\, x < y \wedge \neg \mathsf{Block}^{=}(x,y) \wedge \bigvee_{-\eta \leq \ell_1,\dots \ell_a \leq \mathbf{b}-1} Add(R(\ell_1,\dots \ell_a))@x) \wedge \bigwedge_{1\leq j\leq a} \mathsf{Eq}^{\ell_j,i_j}(x,x_j) \wedge \forall z \neg(x \leq z \wedge z < y \wedge \neg \mathsf{Block}^{=}(z,y) \wedge \bigvee_{0\leq m_1,\dots m_a \leq \mathbf{b}-1} (Del(R(m_1,\dots m_a))@z \wedge \bigwedge_{1\leq j\leq a} \mathsf{Eq}^{\ell_j,m_j}(x,z)))$$

The formula essentially says that the relation tuple has been added to the database instance at some point in the past of y, and since then it has not been deleted.

Similarly we define $\mathsf{Rel\text{-}R}(x_1,i_1,x_2,i_2,\dots x_n,i_n)@y^{\oplus}$ which holds iff the database instance *after* the execution of the block of y has a tuple in relation R as before. It can be expressed as given follows:

$$\exists x\, x \leq y \wedge \bigvee_{-\eta \leq \ell_1,\dots \ell_a \leq \mathbf{b}-1} Add(R(\ell_1,\dots \ell_a))@x \wedge \bigwedge_{1\leq j\leq a} \mathsf{Eq}^{\ell_j,i_j}(x,x_j) \wedge \forall z \neg(x \leq z \wedge z \leq y \wedge \bigvee_{0\leq m_1,\dots,m_a \leq \mathbf{b}-1} (Del(R(m_1,\dots m_a))@z \wedge \bigwedge_{1\leq j\leq a} \mathsf{Eq}^{\ell_j,m_j}(x,z)))$$

6.4.2 *Expressing valid encodings in* $\mathtt{MSO}_{\mathtt{NW}}$

We now show that the conditions given in Section 6.3.1 can be expressed in $\mathtt{MSO}_{\mathtt{NW}}$.

0. Well-formedness We need to check the local consistency of each block appearing in the word, which means s_i must not assign a variable to a recency index higher than or equal to m_i, and that $J_i \subseteq \{0,\dots,m_i\}$. Further it must of the form described in Section 6.3.1. This is a syntactic check inside a block and can be easily stated in $\mathtt{MSO}_{\mathtt{NW}}$.

1. Consistency of m. We write a formula $\varphi_m^{\text{Recent}}(x)$ to state that, just before executing the block of x, the current database I has at least $m+1$ elements in $\text{ADOM}(I)$. Thanks to Remark 6.1, this can be expressed in $\mathtt{MSO}_{\mathtt{NW}}$ by saying that there are $m+1$ distinct pushes before the block of x which are not popped until x.

$$\varphi_m^{\text{Recent}}(x) \equiv \exists y\, \mathsf{Block}^{=}(x,y) \wedge \Sigma_{\text{int}}(y) \wedge \exists x_1,\dots x_m \bigwedge_{i\neq j} x_i \neq x_j \wedge \bigwedge_i \left(\Sigma_{\downarrow}(x_i) \wedge x_i < y \wedge \forall z(x_i \triangleright z \Rightarrow y < z)\right)$$

Now, consistency of m can be stated as:

$$\forall x \bigwedge_{0\leq i\leq \mathbf{b}-1} \neg\varphi_i^{\text{Recent}}(x) \vee \exists y\,(\uparrow_i(y) \wedge \mathsf{Block}^{=}(x,y))$$

2. Consistency of J. Towards this we first need to write a formula $\mathsf{live}(x,i)$ which holds only if the element with recency index i in the block of x is in $\text{ADOM}(I)$ after the execution of the block of x. This is expressed similarly to the formula $\mathsf{Active}(u)$ of Example 2.1. $\mathsf{live}(x,i) \equiv \bigvee_{R/a\in\mathcal{R}} \exists x_1,\dots,x_a \bigwedge_{1\leq j\leq a} x_i \leq x \wedge \bigvee_{-\eta\leq i_1,i_2,\dots,i_a\leq \mathbf{b}-1} \bigvee_{1\leq j\leq a} \mathsf{Rel\text{-}R}(\dots,x_{j-1},i_{j-1},x,i,x_{j+1},i_{j+1},\dots)@x^{\oplus}$ where $\eta = \max_{\alpha\in act} |\alpha{\cdot}\mathsf{new}|$.

Now the consistency of J can be stated by saying that a recency index is pushed in a block iff it is live:

$$\forall x \bigwedge_{0\leq i\leq \mathbf{b}-1} \mathsf{live}(x,i) \Leftrightarrow \left(\exists y \downarrow_i(y) \wedge \mathsf{Block}^{=}(x,y)\right)$$

3. Consistency of action guards. We first present a syntactic translation of an $\mathtt{FOL}(\mathcal{R})$ formula into an $\mathtt{MSO}_{\mathtt{NW}}$ formula. The translation also depends on the current block (a block is represented by the head position of the block which we denote by a free position variable x) as well as the action α and the abstract substitution s used in the block. The translation of an $\mathtt{FOL}(\mathcal{R})$ formula Q at x wrt. s and α is a $\mathtt{MSO}_{\mathtt{NW}}$ formula denoted $\lfloor Q \rfloor_{\alpha,s,x}$.

In our translation, a first-order data variable u is represented by the position x^u and an index i^u, which is an number between $-\eta$ and $\mathbf{b}-1$ where $\eta = \max_{\alpha\in\text{ACTS}} |\alpha{\cdot}\mathsf{new}|$. Intuitively, instead of reasoning about an element in the domain, we reason about it symbolically by means of a (past) position where it is live and its recency index at that position. Given α, s and x, we distinguish between variables belonging to $\alpha{\cdot}\mathsf{free}$ and the other variables. For a variable $u \in \alpha{\cdot}\mathsf{free}$ we set $x^u = x$ and $i^u = s(u)$.

The translation is defined inductively as follows: (In the following $x^u = x$ and $i^u = s(u)$ if $u \in \alpha{\cdot}\mathsf{free}$, and $\eta = \max_{\alpha\in\text{ACTS}} |\alpha{\cdot}\mathsf{new}|$)

- $\lfloor R(u_1,\dots u_n)\rfloor_{\alpha,s,x} \equiv \mathsf{Rel\text{-}R}(x^{u_1},i^{u_1},\dots,x^{u_n},i^{u_n})@x^{\ominus}$
- $\lfloor u_1 = u_2\rfloor_{\alpha,s,x} \equiv \mathsf{Eq}^{i^{u_1},i^{u_2}}(x^{u_1},x^{u_2})$
- $\lfloor \exists u.\, Q\rfloor_{\alpha,s,x} \equiv \exists x^u.\, x^u < x \wedge \bigvee_{-\eta\leq i^u\leq \mathbf{b}-1} \lfloor Q\rfloor_{\alpha,s,x}$
- $\lfloor Q_1 \wedge Q_2\rfloor_{\alpha,s,x} \equiv \lfloor Q_1\rfloor_{\alpha,s,x} \wedge \lfloor Q_2\rfloor_{\alpha,s,x}$
- $\lfloor \neg Q\rfloor_{\alpha,s,x} \equiv \neg\lfloor Q\rfloor_{\alpha,s,x}$

Now we are ready to express the consistency of action guards with the run. It can be expressed by the following formula: $\forall x \bigwedge_{(\alpha:s)\in\Sigma_{\text{int}}} (\alpha,s)(x) \Rightarrow \lfloor \alpha{\cdot}\mathsf{guard}\rfloor_{\alpha,s,x}$.

Let $\varphi^{\mathsf{valid}}_{\mathbf{b},\mathcal{S}}$ be the conjunction of the four conditions listed above. It characterises valid encodings of $\mathbf{b}$-bounded runs of a DMS $\mathcal{S}$.

6.5 Translating MSO-FO specifications into MSO_{NW} specifications

Here we provide a translation of the specifications in MSO-FO to an equivalent one in MSO_{NW} over valid encodings of runs. This is similar in spirit to the translation described for action guards. As done there, we represent a first-order data variable u by the pair x^u, i^u. We extend this representation to incorporate the globally quantified data variables as well. We distinguish neither between free or bound variables, nor between globally quantified and locally quantifies variables: A data variable u is represented by the pair x^u, i^u.

A first-order position variable of MSO-FO will correspond to a block in the nested word encoding. A block is represented by its head. Thus in our translation, a first-order position variable of MSO-FO corresponds to a first-order position variable of MSO_{NW} which varies over heads of blocks. A set variable of MSO-FO corresponds to a set variable in MSO_{NW} relativized to head positions. The translation of a MSO-FO formula φ is denoted $\lfloor\varphi\rfloor$, and is defined inductively as follows:

- $\lfloor Q@x\rfloor \equiv \Sigma_{\text{int}}(x) \wedge \bigvee_{\overbrace{\alpha:s}\in\Sigma_{\text{int}}} \overbrace{\alpha:s}(x) \Rightarrow \lfloor Q\rfloor_{\alpha,s,x}$
- $\lfloor \exists x.\, \varphi\rfloor \equiv \exists x.\, \Sigma_{\text{int}}(x) \wedge \lfloor\varphi\rfloor$
- $\lfloor \exists X.\, \varphi\rfloor \equiv \exists X.\, (\forall x.\, x \in X \Rightarrow \Sigma_{\text{int}}(x)) \wedge \lfloor\varphi\rfloor$
- $\lfloor \exists^g u.\varphi\rfloor \equiv \exists x^u.\, \Sigma_{\text{int}}(x^u) \wedge \bigvee_{-\eta\le i^u\le \mathbf{b}-1} \lfloor\varphi\rfloor$ where $\eta = \max_{\alpha\in\text{ACTS}} |\alpha{\cdot}\mathsf{new}|$
- $\lfloor x < y\rfloor \equiv x < y$
- $\lfloor x \in X\rfloor \equiv x \in X$
- $\lfloor \varphi_1 \wedge \varphi_2\rfloor \equiv \lfloor\varphi_1\rfloor \wedge \lfloor\varphi_2\rfloor$
- $\lfloor \neg\varphi\rfloor \equiv \neg\lfloor\varphi\rfloor$

6.6 Concluding the reduction

We reduce the recency bounded model checking problem to the satisfiability checking of MSO_{NW}. Given a DMS $\mathcal{S}$, recency bound $\mathbf{b}$ and an MSO-FO specification ψ, we construct $\varphi^{\mathsf{valid}}_{\mathbf{b},\mathcal{S}}$ and $\lfloor\psi\rfloor$ as described in the previous subsections. The $\mathbf{b}$ bounded model checking problem reduces to the non-satisfiability checking of $\varphi^{\mathsf{valid}}_{\mathbf{b},\mathcal{S}} \wedge \neg\lfloor\psi\rfloor$. The satisfiability checking of $\text{MSO}_{\text{NW}}$is decidable [3], and hence by our reduction RECENCY-BOUNDED-MSO/DMS-MC is decidable.

The construction of the MSO_{NW} formula $\varphi^{\mathsf{valid}}_{\mathbf{b},\mathcal{S}} \wedge \neg\lfloor\psi\rfloor$ takes time $\mathcal{O}((\mathbf{b} + |\mathcal{R}| + |\text{ACTS}|)^{\mathcal{O}(\mathfrak{a}+\mathfrak{n})})$ where $|\mathcal{R}|$ denotes the number of relations in $\mathcal{R}$, $|\text{ACTS}|$ denotes the number of actions of $\mathcal{S}$, $\mathfrak{a}$ is the maximum arity of the relations in the schema (i.e., $\mathfrak{a} = \max_{R/a\in\mathcal{R}} a$) and $\mathfrak{n}$ is the number of data-variables appearing in the action guards and ψ.

7. RELATED WORK

As pointed out in the introduction, DMSs belong to the series of works on the verification of dabase-driven dynamic systems whose initial state is known [11, 7, 5, 8, 6, 26]. In [8], *artifact-centric multi-agent systems* are proposed to simultaneously account for business artifacts and for the specification of agents operating over them. Building on [11, 7], decidability of verification of FO-CTLK properties with active domain FO quantification is obtained, under the assumption that the size of the databases maintained by agents and artifacts never exceeds a pre-defined bound. This notion of *state-boundedness* is thoroughly studied in [6] on top of the framework of *data-centric dynamic systems* (DCDSs). There, decidability of verification is obtained for a sophisticated variant of FO μ-calculus with active domain FO quantification, in which the possibility of quantifying over individual objects across time is limited to those objects that persist in the active domain of the system. Our logic MSO-FO differs to those used in [8, 6] since it captures linear properties over runs, as opposed to branching properties over RTSs. Furthermore, it leverages the full power of MSO to express sophisticated temporal properties, and is equipped with unrestricted FO quantification across positions of the run, as well as the possibility of quantifying over the objects present in the whole run, as opposed to only those present in the active domain of the current state.

Both in [8] and [6], decidability is obtained by constructing a faithful, finite-state abstraction that preserves the properties to be verified. This shows that state-bounded dynamic systems are an interesting class of *essentially finite-state systems* [1]. On the other hand, state-boundedness is a too restrictive requirement when dealing with systems such as that of Example 3.2. In fact, allowing for unboundedly many tuples to be stored in the database is required to deal with *history-dependent dynamic systems*, whose behavior is influenced by the presence of certain patterns in the (unbounded) history of the system (cf. the definition of *gold customer* in Example 3.2). It is also essential to capture *last-in first-out dynamic systems*, where the currently executed task may be interrupted by a task with a higher-priority, and so on, resuming the execution of the original task only when the (unbounded) chain of higher-priority tasks is completed. See, e.g., the pre-emptive offer handling adopted in Example 5.2. Notably, as argued in Example 5.2, such classes of unbounded systems can all be subject to *exact* MSO-FO model checking, by choosing a sufficiently large bound for recency.

8. CONCLUSION

We have proposed an under-approximation of dynamic database-driven systems that allows unbounded state-space, under which we have shown decidability of model checking against MSO-FO. The decidability is obtained by a reduction to the satisfiability checking of MSO over nested words. The complexity of our model checking procedure is non-elementary in the size of the specification and the DMS. A fine-grained analysis of complexity with respect to various input parameters such as arity of the relations, size of schema, number of variables in the queries etc. is left for future work. It is interesting to study whether one can obtain model checking algorithms with elementary complexity by using other specification formalisms, like temporal logics. Expressing valid runs in temporal logic would be important in this case, and it is interesting problem on its own. Another direction for future work would be to identify other meaningful under-approximation parameters. For example, does bounding the most recently *accessed* elements as opposed to most recently added elements yield decidability? We also aim at applying under-approximation techniques in the case where the initial database is not known, and model checking is studied for every possible initial database, in the style of [15, 13, 9].

Acknowledgments

We acknowledge the support of the *Uppsala Programming for Multicore Architectures Research Center* (UPMARC), the *Programming Platform for Future Wireless Sensor Networks Project* (PROFUN), and the EU project *Optique - Scalable End-user Access to Big Data* (FP7-IP-318338).

9. REFERENCES

[1] P. A. Abdulla. Well (and better) quasi-ordered transition systems. *Bulletin of Symbolic Logic*, 16(4):457–515, 2010.

[2] S. Abiteboul, V. Vianu, B. Fordham, and Y. Yesha. Relational transducers for electronic commerce. *JCSS*, 61(2):236–269, 2000.

[3] R. Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

[4] P. Aziz Abdulla, C. Aiswarya, M. Faouzi Atig, M. Montali, and O. Rezine. Recency-Bounded Verification of Dynamic Database-Driven Systems (Extended Version). http://arxiv.org/abs/1604.03413, Apr. 2016.

[5] B. Bagheri Hariri, D. Calvanese, G. De Giacomo, R. De Masellis, and P. Felli. Foundations of relational artifacts verification. In *BPM*, 2011.

[6] B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. In *PODS*, 2013.

[7] F. Belardinelli, A. Lomuscio, and F. Patrizi. Verification of deployed artifact systems via data abstraction. In *ICSOC*, 2011.

[8] F. Belardinelli, A. Lomuscio, and F. Patrizi. An abstraction technique for the verification of artifact-centric systems. In *KR*, 2012.

[9] M. Bojanczyk, L. Segoufin, and S. Torunczyk. Verification of database-driven systems via amalgamation. In *PODS*, 2013.

[10] D. Calvanese, G. De Giacomo, and M. Montali. Foundations of data-aware process analysis: A database theory perspective. In *PODS*. ACM Press, 2013.

[11] P. Cangialosi, G. De Giacomo, R. De Masellis, and R. Rosati. Conjunctive artifact-centric services. In *ICSOC*, 2010.

[12] D. Cohn and R. Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Engineering Bullettin*, 32(3), 2009.

[13] E. Damaggio, A. Deutsch, and V. Vianu. Artifact systems with data dependencies and arithmetic. In *ICDT*, 2011.

[14] E. Damaggio, R. Hull, and R. Vaculín. On the equivalence of incremental and fixpoint semantics for business artifacts with guard-stage-milestone lifecycles. In *BPM*, 2011.

[15] A. Deutsch, R. Hull, F. Patrizi, and V. Vianu. Automatic verification of data-centric business processes. In *ICDT*, pages 252–267, 2009.

[16] A. Deutsch, L. Sui, and V. Vianu. Specification and verification of data-driven web applications. *JCSS*, 73(3):442–474, 2007.

[17] R. Hull. Artifact-centric business process models: Brief survey of research results and challenges. In *OTM Confederated Int. Conf.*, 2008.

[18] V. Künzle and M. Reichert. Philharmonicflows: towards a framework for object-aware process management. *Journal of Software Maintenance*, 23(4):205–244, 2011.

[19] V. Künzle, B. Weber, and M. Reichert. Object-aware business processes: Fundamental requirements and their support in existing approaches. *Int. J. of Information System Modeling and Design*, 2(2):19–46, 2011.

[20] C. Lautemann, T. Schwentick, and D. Thérien. Logics for context-free languages. In *CSL*, volume 933 of *LNCS*, pages 205–216. Springer, 1995.

[21] A. Lomuscio and J. Michaliszyn. Model checking unbounded artifact-centric systems. In *KR*. AAAIP, 2014.

[22] A. Meyer, S. Smirnov, and M. Weske. Data in business processes. Technical Report 50, Hasso-Plattner-Institut for IT Systems Engineering, Universität Potsdam, 2011.

[23] M. Montali and D. Calvanese. Soundness of data-aware, case-centric processes. *International Journal on Software Tools for Technology Transfer*, pages 1–24, 2016.

[24] A. Nigam and N. S. Caswell. Business artifacts: An approach to operational specification. *IBM Systems Journal*, 42(3), 2003.

[25] M. Reichert. Process and data: Two sides of the same coin? In *OTM*, volume 7565 of *LNCS*, pages 2–19. Springer, 2012.

[26] D. Solomakhin, M. Montali, S. Tessaris, and R. De Masellis. Verification of artifact-centric systems: Decidability and modeling issues. In *ICSOC*, 2013.

[27] M. Y. Vardi. Model checking for database theoreticians. In *ICDT*, volume 3363 of *LNCS*, pages 1–16. Springer, 2005.

APPENDIX

A. SEMANTICS OF $\mathtt{FOL}(\mathcal{R})$ QUERIES

Given a database instance I over $\mathcal{R}$ and Δ, a $\mathtt{FOL}(\mathcal{R})$ query Q over $\mathcal{R}$, and a substitution $\sigma : \mathsf{Free\text{-}Vars}(Q) \to \Delta$, we write $I, \sigma \models Q$ if the query Q under the substitution σ holds in database I. This is defined inductively:

- $I, \sigma \models \mathsf{true}$
- $I, \sigma \models R(u_1, \ldots, u_a)$ if $(e_1, \ldots, e_a) \in R^I$ where $e_i = \sigma(u_i)$ for all $i : 1 \leq i \leq a$.
- $I, \sigma \models u_i = u_j$ if $\sigma(u_i) = \sigma(u_j)$.
- $I, \sigma \models \neg Q$ if $I, \sigma \not\models Q$.
- $I, \sigma \models Q_1 \wedge Q_2$ if $I, \sigma \models Q_1$, and $I, \sigma \models Q_2$.
- $I, \sigma \models \exists u.Q$ if there exists $e \in \textsc{adom}(I)$ such that $I, \sigma' \models Q$ where σ' is obataind from σ as follows: σ' is defined on u and $\sigma'(u) = e$ and $\sigma'(u') = \sigma(u')$ if $u' \neq u$.

B. SEMANTICS OF MSO-FO

A run ρ is an infinite sequence of database instances over $\mathcal{R}$ and Δ:

$$\rho = I_0, I_1, I_2, I_3 \ldots$$

An MSO formula ϕ is interpreted over a run ρ under a substitution σ of the free variables $\mathsf{Free\text{-}Vars}(\phi)$. If the formula holds in the run ρ under the substitution σ, we write $\rho, \sigma \models \phi$. The semantics is defined inductively:

- $\rho, \sigma \models Q@x$ if $I_i, \sigma' \models Q$ where $i = \sigma(x)$ and $\sigma' = \sigma|_{\mathsf{Free\text{-}Vars}(Q)}$
- $\rho, \sigma \models x < y$ if $\sigma(x) < \sigma(y)$

- $\rho, \sigma \models x \in X$ if $\sigma(x) \in \sigma(X)$
- $\rho, \sigma \models \neg\phi$ if $\rho, \sigma \not\models \phi$
- $\rho, \sigma \models \phi_1 \wedge \phi_2$ if $\rho, \sigma \models \phi_1$, and $\rho, \sigma \models \phi_2$
- $\rho, \sigma \models \exists x.\phi$ if there exists $i \in \mathbb{N}$, such that $\rho, \sigma[x \mapsto i] \models \phi$ where $\sigma[x \mapsto i](x) = i$, and $\sigma[x \mapsto i](\xi) = \sigma(\xi)$ if $\xi \neq x$
- $\rho, \sigma \models \exists X.\phi$ if there exists $J \subseteq \mathbb{N}$, such that $\rho, \sigma[X \mapsto J] \models \phi$ where $\sigma[X \mapsto J](X) = J$, and $\sigma[X \mapsto J](\xi) = \sigma(\xi)$ if $\xi \neq X$
- $\rho, \sigma \models \exists^g u.\phi$ if there exists $e \in \textsc{Gadom}(\rho)$, such that $\rho, \sigma[u \mapsto e] \models \phi$ where $\sigma[u \mapsto e](u) = e$, and $\sigma[u \mapsto e](\xi) = \sigma(\xi)$ if $\xi \neq u$

The semantics of database query $(I, \sigma \models Q)$ is as expected (see Appendix A). The substitution of free variables is always restricted to the active domain of I in this case. That is, $\mathsf{Image}(\sigma) \subseteq \textsc{adom}(I)$ is necessary; just having $\mathsf{Image}(\sigma) \subseteq \textsc{Gadom}(\rho)$ is not sufficient.

Intuitively, $Q@x$ evaluates the $\mathtt{FOL}(\mathcal{R})$ query Q over the database instance present at position x in the run. Formula $x < y$ asserts that position x comes before y along the run. Formula $x \in X$ states that position x belongs to the set X of positions. Formula $\exists x.\phi$ states that there exists a position x in the run where ϕ holds, whereas formula $\exists X.\phi$ models that there exists a set X of positions in the run where ϕ holds. Finally, $\exists^g u.\phi$ states that there exists a data value u that is active in some database instance of the run and that makes ϕ true. In this light, the quantifier $\exists^g$ ranges over the *global active domain* of the run, obtained by composing all active domains of the database instances encountered therein.

C. RUNS WHICH ARE EQUIVALENT MODULO PERMUTATIONS

LEMMA C.1. *Two runs match on their abstraction, if and only if they are equivalent modulo permutations of the data domain.*

Let $\hat{\rho} = \left(\langle I_j, H_j, \mathtt{seq\_no}_j\rangle \xrightarrow{\alpha_j:\sigma_j}_{\mathtt{b}} \langle I_{j+1}, H_{j+1}, \mathtt{seq\_no}_{j+1}\rangle\right)_{j\geq 0}$ and $\hat{\rho}' = \left(\langle I'_j, H'_j, \mathtt{seq\_no}'_j\rangle \xrightarrow{\alpha'_j:\sigma'_j}_{\mathtt{b}} \langle I'_{j+1}, H'_{j+1}, \mathtt{seq\_no}'_{j+1}\rangle\right)_{j\geq 0}$ be two extended runs such that $\mathsf{Abstr}(\hat{\rho}) = \mathsf{Abstr}(\hat{\rho}') = (\langle \alpha_j : s_j\rangle)_{j\geq 0}$. That implies that $\alpha_j = \alpha'_j$ for every $j \geq 0$.

We prove in what follows the existence of a bijection $\lambda : \textsc{Gadom}(\hat{\rho}) \to \textsc{Gadom}(\hat{\rho}')$ such that, for every $i \geq 0$, λ is an isomorphism from I_i onto I'_i.

Notice that the global active domain of a given run $\hat{\rho}$ amounts to the set that contains all fresh input elements introduced in the database all along the run. That is $\textsc{Gadom}(\hat{\rho}) = \cup_{i\geq 0} I_i = \cup_{i\geq 0} \{\sigma_i(v) |\ v \in \alpha_i{\cdot}\mathsf{new}\}$ and $\textsc{Gadom}(\hat{\rho}) = \cup_{i\geq 0} I'_i = \cup_{i\geq 0} \{\sigma'_i(v) |\ v \in \alpha'_i{\cdot}\mathsf{new}\} = \cup_{i\geq 0} \{\sigma'_i(v) |\ v \in \alpha_i{\cdot}\mathsf{new}\}$ (since $\alpha'_i = \alpha_i$).

Definition of λ. Let $e \in \Delta$. We have that $e \in \textsc{Gadom}(\hat{\rho})$ if and only if $\exists i \geq 0$ and $\exists v \in \alpha_i{\cdot}\mathsf{new}$ such that $\sigma_i(v) = e$. Since $\alpha'_i = \alpha_i$, $\sigma'_i(v) \in \textsc{Gadom}(\hat{\rho}')$ is also well defined. We set $\lambda(e) = \sigma'_i(v)$. **Injectivity.** Let $e_1, e_2 \in \textsc{Gadom}(\hat{\rho})$ such that $e_1 \neq e_2$. That means that there are $i_1, i_2 \geq 0$, $v_1 \in \alpha_{i_1}{\cdot}\mathsf{new}$ and $v_2 \in \alpha_{i_2}{\cdot}\mathsf{new}$ such that $\sigma_{i_1}(v_1) = e_1$ and $\sigma_{i_2}(v_2) = e_2$. We have that $\lambda(e_1) = \sigma'_{i_1}(v_1)$ and $\lambda(e_1) = \sigma'_{i_2}(v_2)$. Notice that, since $e_1 \neq e_2$, we can not have $i_1 = i_2$ and $v_1 = v_2$ at the same time. That, the fact that substitutions have to be injective when applied to the fresh variables and together with the freshness condition for the newly input values, imply that $\lambda(e_1) \neq \lambda(e_2)$. Thus, λ is injective. **Surjectivity.** Let $e' \in \textsc{Gadom}(\hat{\rho}')$. That implies the existence of $i \geq 0$ and $v \in \alpha'_i{\cdot}\mathsf{new}$ such that $\sigma'_i(v) = e'$. Since $\alpha_i = \alpha'_i$, $\sigma_i(v)$ is also well defined and we set $e = \sigma_i(v)$. We have that $\lambda(e) = e'$. Thus, λ is surjective.

Thus, λ is a bijection. In a similar fashion, we can define yet another bijection from $\{\mathtt{seq\_no}_i(e) |\ e \in \textsc{Gadom}(\hat{\rho})\}$ onto $\{\mathtt{seq\_no}'_i(e) |\ e \in \textsc{Gadom}(\hat{\rho}')\}$ that we denote by β and such that $\beta(\mathtt{seq\_no}_i(e)) = \mathtt{seq\_no}'_i(\lambda(e))$. Moreover, we can show that β is monotonic.

Now that we have defined λ (and β), we shall prove by induction on $i \in \mathbb{N}$ that λ is an isomorphism from I_i onto I'_i.

For the base case ($i = 0$), we trivially have that $I_0 = I'_0$ and $I_0 \cap \textsc{Gadom}(\hat{\rho}) = \emptyset = I_0 \cap \textsc{Gadom}(\hat{\rho}')$. Assume now that for some $i \in \mathbb{N}$ we have that λ is an isomorphism from I_i onto I'_i. Let's prove that λ is also an isomorphism from I_{i+1} onto I'_{i+1}. Let $R(\sigma_i(u_1), \ldots, \sigma_i(u_n))$ (with $R \in \mathcal{R}$ and $u_1, \ldots, u_n \in \alpha_i{\cdot}\mathsf{free} \uplus \alpha_i{\cdot}\mathsf{new}$) be one of the facts that will be added to I_i in order to obtain I_{i+1}. Since $\alpha_i = \alpha'_i$, the fact $R(\sigma'_i(u_1), \ldots, \sigma'_i(u_n))$ is also added to I'_i in order to obtain I'_{i+1}. If $u_j \in \alpha_i{\cdot}\mathsf{new}$ then we have that $\sigma'_i(u_j) = \lambda(\sigma_i(u_j))$. In the case where $u_j \in \alpha_i{\cdot}\mathsf{free}$, since we have that i) the symbolic substitutions s_i and s'_i are equal, which free variables parts are respectively defined by the sequence numbering functions $\mathtt{seq\_no}_i$ and $\mathtt{seq\_no}'_i$, and that ii) $\beta(\mathtt{seq\_no}_i(e)) = \mathtt{seq\_no}'_i(\lambda(e))$ for every $e \in \textsc{adom}(I_i)$, we deduce that $\sigma'_i(u_j) = \lambda(\sigma_i(u_j))$. Thus, the added fact amounts to $R(\lambda(\sigma_i(u_1)), \ldots, \lambda(\sigma_i(u_n)))$. The case of deleted facts is simpler, and the inverse reasoning (from I'_i to I_i) is identical. Thus λ is also an isomorphism from I_{i+1} onto I'_{i+1}.

D. GENERALITY OF THE MODEL

We discuss the generality of the model. In particular, we consider variants of DMSs that supports:

- constant values;
- (SQL-like) *standard variable substitution* for guards and input variables, consequently tackling the repetition of matching values;
- Weakening the freshness requirement for input values, to support the possibility of matching input variables with already existing values.

In summary, we show that all such variants can be reduced back to the standard model presented in Section 3, while preserving the original system behavior.

D.1 DMSs with Possibly Overlapping Inputs

According to the DMS execution semantics, the application of an action is done by substituting the guard answer variables and by *injectively* substituting the fresh input variables with corresponding values. We show here that this is not a limitation of the approach, which can in fact seamlessly account for standard variable substitution, possibly mapping multiple fresh variables with the same value.

The algorithm in Figure 5 shows precisely how to build a set of injective actions, i.e. actions where fresh variables are mapped to different values, from a set of non-injective

1: **procedure** STANDARD-SUBSTITUTION(ACTS)
2: **input:** Set ACTS of actions, **output:** Set ACTS_{std} of actions
3: $\text{ACTS}_{std} := \emptyset$
4: **for all** $\langle Q(\vec{u}), Del(\vec{u}), Add(\vec{u}, \vec{v})\rangle \in$ ACTS **do**
5: **for all** $\vec{p} = \langle \vec{s}_1, \ldots, \vec{s}_{|p|}\rangle$ partition of $\vec{v}$ **do**
6: $Add'(\vec{u}, \langle v_i^{'} | 1 \leq i \leq |p|\rangle) :=$
$Add(\vec{u}, \vec{v})[v/v_i^{'}$ **if** $v \in \vec{s}_i | 1 \leq i \leq |p|]$
7: $\text{ACTS}_{std} :=$ ACTS_{std}
$\cup \{\langle Q(\vec{u}), Del(\vec{u}), Add'(\vec{u}, \langle v_i^{'} | 1 \leq i \leq |p|\rangle)\rangle\}$

Figure 5: Procedure for turning a set of actions into another set of actions that simulates standard variable substitutions. Notation $E(\vec{u})[\vec{u}/\vec{z}]$, where E is a formula or a set of facts, indicates E where variables $\vec{u}$ are consistently replaced with corresponding variables/constants $\vec{z}$. For every partition set element $\vec{s}_i$, all variables belonging to a $\vec{s}_i$ are replaced with the new fresh variable $v_i^{'}$.

actions, i.e. actions where fresh variables can be mapped to the same value. A demonstration of the algorithm is given in Example D.1.

EXAMPLE D.1. Given action

$\alpha = \langle \{u_1, u_2\}, \{v_1, v_2, v_3\}, R(u_1, u_2), \{Q(u_2)\}, \{R(u_2, v_1), R(u_2, v_2), R(u_1, v_3)\}\rangle$, we get:

$$\left\{ \begin{array}{l} \alpha_1 = \langle \{u_1, u_2\}, \{v_1', v_2', v_3'\}, R(u_1, u_2), \{Q(u_2)\}, \\ \quad \{R(u_2, v_1^{'}), R(u_2, v_2^{'}), R(u_1, v_3^{'})\}\rangle, \\ \alpha_2 = \langle \{u_1, u_2\}, \{v_1', v_2'\}, R(u_1, u_2), \{Q(u_2)\}, \\ \quad \{R(u_2, v_1^{'}), R(u_2, v_2^{'}), R(u_1, v_2^{'})\}\rangle, \\ \alpha_3 = \langle \{u_1, u_2\}, \{v_1', v_2'\}, R(u_1, u_2), \{Q(u_2)\}, \\ \quad \{R(u_2, v_2^{'}), R(u_2, v_2^{'}), R(u_1, v_1^{'})\}\rangle, \\ \alpha_4 = \langle \{u_1, u_2\}, \{v_1', v_2'\}, R(u_1, u_2), \{Q(u_2)\}, \\ \quad \{R(u_2, v_2^{'}), R(u_2, v_1^{'}), R(u_1, v_2^{'})\}\rangle, \\ \alpha_5 = \langle \{u_1, u_2\}, \{v_1'\}, R(u_1, u_2), \{Q(u_2)\}, \\ \quad \{R(u_2, v_1^{'}), R(u_2, v_1^{'}), R(u_1, v_1^{'})\}\rangle \end{array} \right\}$$

to interpret the original action using standard variable substitutions for fresh input variables v_1, v_2, v_3. In action α_2 for instance, v_1' replaces the subset $\langle v_1\rangle$, while v_2' replaces the subset $\langle v_2, v_3\rangle$. This replacement corresponds to the partition $\vec{p} = \langle \vec{s}_1 = \langle v_1\rangle, \vec{s}_2 = \langle v_2, v_3\rangle\rangle$. ■

D.2 Weakening Freshness

One may argue that inputs provided to a DMS may not necessarily be fresh. In fact, there may be cases in which a DMS action is meant to establish new relations among already existing values, but still interacting with the external world to decide which. We call *arbitrary-input DMS* a DMS that does not necessarily require the input variables to be assigned to fresh values. We provide in what follows a proof and an example illustrating this remark.

PROOF. Let $\mathcal{S} = \langle I_0, \text{ACTS}\rangle$ be an arbitrary-input DMS defined over the data domain Δ and the schema $\mathcal{R}$. We produce a corresponding standard DMS $\mathcal{S}_{fresh} = \langle I_0, \text{ACTS}'\rangle$, defined over the same data domain and over an extended schema $\mathcal{R}'$, such that:

- the schema of the $\mathcal{R}' = \mathcal{R} \cup \{Hist/1\}$ is the extension of the schema of $\mathcal{S}$ by adding the unary relation *Hist*, which role is to store all the values seen in during the run of the DMS,
- the set of actions ACTS$'$ of the standard DMS $\mathcal{S}'$ is defined as the smallest set satisfying the following property: for every arbitrary-input action $\left\langle \vec{u}, \vec{i}, Q(\vec{u}), Del(\vec{u}), Add\left(\vec{u}, \vec{i}\right)\right\rangle \in$ ACTS with abitrary-input variables $\vec{i}$, and for every possible binary partition $\vec{h} \uplus \vec{f}$ of the set of input variables $\vec{i}$, ACTS$'$ contains the standard DMS action $\langle \vec{u} \uplus \vec{h}, \vec{f}, Q'(\vec{u}, \vec{h}), Del(\vec{u}), Add'(\vec{u}, \vec{h}, \vec{f})\rangle$, where
  - $Q'(\vec{u}, \vec{h}) = Q(\vec{u}) \wedge \bigwedge_{h \in \vec{h}} Hist(h)$ and
  - $Add'(\vec{u}, \vec{h}, \vec{v}) = Add(\vec{u}, \vec{h} \uplus \vec{f}) \cup \left\{Hist(f) | \ f \in \vec{f}\right\}$

Thus, every action with arbitrary-input variables $\vec{i}$ is translated into $2^{|\vec{i}|}$ actions, each one handling the case in which a subset of the uniform input variables are mapped to fresh values, while the remaining ones are bound to values present in the history of the run. It is easy to see that the configuration graph obtained from $\mathcal{S}$ by removing the requirement that input variables must match with fresh values, and the standard configuration graph of $\mathcal{S}'$, indeed coincide. □

EXAMPLE D.2. Given schema $\mathcal{R} = \{R/2, Q/1\}$, we replace the arbitrary input action

$$\langle \{u_1, u_2\}, \{i_1, i_2\}, R(u_1, u_2), \{Q(u_2)\}, \{R(u_2, i_1), R(u_2, i_2)\}\rangle$$

with the standard (fresh) input set of actions

$$\left\{ \begin{array}{l} \alpha_1 = \langle \{u_1, u_2\}, \{f_1, f_2\}, R(u_1, u_2), \\ \quad \{Q(u_2)\}, \{R(u_2, f_1), R(u_2, f_2), Hist(f_1), Hist(v_2)\}\rangle \\ \alpha_2 = \langle \{u_1, u_2, h_1\}, \{f\}, R(u_1, u_2) \wedge Hist(h), \\ \quad \{Q(u_2)\}, \{R(u_2, h), R(u_2, f), Hist(f)\}\rangle \\ \alpha_3 = \langle \{u_1, u_2, h_1, h_2\}, \emptyset, R(u_1, u_2) \wedge Hist(h_1) \wedge Hist(h_1), \\ \quad \{Q(u_2)\}, \{R(u_2, h_1), R(u_2, h_2)\}\rangle \end{array} \right\}$$

■

D.3 Simulating Bulk Operations

We show how *bulk actions* can be simulated by standard DMSs. Recall that DMSs adopt a *retrieve-one-answer-per-step* semantics, i.e., a DMS action $\langle \vec{u}, \vec{v}, Q, Del, Add\rangle$ is applied by nondeterministically grounding its action parameters $\vec{u}$ with *one* answer to Q. Bulk actions, instead, require the adoption of a *retrieve-all-answers-per-step* semantics, where the update specified by *Del* and *Add* would be enforced by simultaneously considering *all* answers to Q. Let $\beta = \langle \vec{u}, \vec{v}, Q, Del, Add\rangle$ be a bulk action, i.e., an action where the action parameters $\vec{u}$ are implicitly universally quantified. We show how the bulk update induced by β can be simulated in a standard DMS though a complex sequence of actions and the introduction of accessory relations.

The following accessory relations are used: *(i)* A proposition $Lock_\beta$, used to "lock" the sequence of actions simulating β, guaranteeing that it is not interrupted by other actions. *(ii)* A relation $FreshInput_\beta$ with arity $|\vec{v}|$, used to store (in a single tuple) the selected substitution for the fresh-input variables of β, enabling the consistent usage of such a substitution when reconstrucing the bulk update of β. *(iii)* A relation $ParMatch_\beta$ with arity $|\vec{u}| + 1$, used to incrementally store all answers to $\beta \cdot \mathsf{guard}$, then exhaustively considering them when reconstructing the bulk update induced by β. The last argument of $ParMatch_\beta$ is used to "flag" tuples that have been already considered for the corresponding deletion of tuples within the bulk update (more details are given below). *(iv)* Two propositions $DelPhase_\beta$ and $AddPhase_\beta$, identifying those portions of the sequence of actions respectively dealing with the bulk deletion/addition of β.

Whenever β is eligible for execution, those accessory relations are all empty. At the completion of the sequence of actions simulating β, such accessory relations will be empty again. To ensure the non-interruptibility of the sequence of actions used to simulate bulk actions, all actions of the DMS of interest must be modified so as to incorporate the negation of all lock propositions (like $Lock_\beta$ above), denoted in the following Φ_{NoLock}.

The simulation of β is done by structuring the sequence in three phases. The first consists of the application of a single initialization action Init_β, executable when $\beta{\cdot}\mathsf{guard}$ admits at least one answer. Init_β sets the lock, and stores the selected substitution for the fresh-input variables. Specifically, Init_β has fresh-input variables $\vec{v}$, and is defined as:

- $\mathsf{Init}_\beta{\cdot}\mathsf{guard} = (\exists \vec{u}.\beta{\cdot}\mathsf{guard}(\vec{u})) \wedge \Phi_{NoLock}$;
- $\mathsf{Init}_\beta{\cdot}Del = \emptyset$;
- $\mathsf{Init}_\beta{\cdot}Add = \{Lock_\beta, FreshInput_\beta(\vec{v})\}$.

The second phase deals with the computation of all answers to $\beta{\cdot}\mathsf{guard}$, storing them into the corresponding accessory relation. Such a phase is identified by the presence of the lock for β, and by the absence of the flags marking the bulk deletion and addition of tuples. The computation of the answers to $\beta{\cdot}\mathsf{guard}$ is handled by iteratively executing action $\mathsf{CompAns}_\beta$. This action is executable when there is at least one answer to $\beta{\cdot}\mathsf{guard}$ that has not yet been transferred. If this is the case, it nondeterministically picks one such answers, and transfers it into the accessory relation. Specifically, $\mathsf{CompAns}_\beta$ has exactly $\vec{u}$ as action parameters, no fresh-input variable, and is defined as follows:

- $\mathsf{CompAns}_\beta{\cdot}\mathsf{guard} = Lock_\beta \wedge \neg DelPhase_\beta \wedge \neg AddPhase_\beta \wedge \beta{\cdot}\mathsf{guard}(\vec{u}) \wedge \neg ParMatch_\beta(\vec{u})$;
- $\mathsf{CompAns}_\beta{\cdot}Del = \emptyset$;
- $\mathsf{CompAns}_\beta{\cdot}Add = \{ParMatch_\beta(\vec{u}, 0)\}$.

When inserting an answer tuple into the accessory relation, the last, additional argument of $ParMatch_\beta$ is set to 0. This witnesses that such an answer tuple has still to be considered when reconstructing the deletions induced by β.

Action $\mathsf{CompAns}_\beta$ becomes non-executable when the FO sentence $\Phi_\beta^{AllSub} = \forall \vec{u}.\beta{\cdot}\mathsf{guard}(\vec{u}) \rightarrow ParMatch_\beta(\vec{u})$ holds in the current database. We consequently insert a dedicated action $\mathsf{EnableU}_\beta$, marking the end of reiterated application of $\mathsf{CompAns}_\beta$. This action is executable when all answer tuples have been transferred to the accessory relation, and has the effect of indicating that it is now time to apply the bulk update induced by β. The execution semantics of DMSs dictates that additions have priority over deletions. For this reason, the bulk update first requires to consider all deletions, and then all additions. Hence, $\mathsf{EnableU}_\beta$ raises flag $DelPhase_\beta$. Specifically, $\mathsf{EnableU}_\beta$ has no action parameters nor fresh-input variables, and is defined as follows:

- $\mathsf{EnableU}_\beta{\cdot}\mathsf{guard} = Lock_\beta \wedge \neg DelPhase_\beta \wedge \neg AddPhase_\beta \wedge \Phi_\beta^{AllSub}$;
- $\mathsf{EnableU}_\beta{\cdot}Del = \emptyset$;
- $\mathsf{EnableU}_\beta{\cdot}Add = \{DelPhase_\beta\}$.

The introduction of flag $DelPhase_\beta$ marks the beginning of the third phase, which deals with the actual bulk update. As mentioned before, this phase is split into two sub-phases: a first sub-phase dealing with deletions, a second sub-phase dealing with additions. Both sub-phases consists of the iterative application of one dedicated action, which deals with the tuples to be deleted/added due to a specific answer tuple to $\beta{\cdot}\mathsf{guard}$. As for deletion, the iteratively executed action is $\mathsf{ApplyDel}_\beta$. This action nondeterministically picks an answer tuple for $\beta{\cdot}\mathsf{guard}$ that still has to be considered for deletion. This is done by extracting a tuple from $ParMatch_\beta$, checking that the last argument of such a tuple corresponds to 0. In addition, since the deletion may depend on the fresh-input of β as well, $\mathsf{ApplyDel}_\beta$ also needs to extract the single tuple present in the input accessory relation $FreshInput_\beta$. Specifically, $\mathsf{ApplyDel}_\beta$ has $\vec{u}$ as action parameters, no fresh-input variables, and is defined as follows:

- $\mathsf{ApplyDel}_\beta{\cdot}\mathsf{guard} = DelPhase_\beta \wedge ParMatch_\beta(\vec{u}, 0)$;
- $\mathsf{ApplyDel}_\beta{\cdot}Del = \beta{\cdot}Del \cup \{ParMatch_\beta(\vec{u}, 0)\}$;
- $\mathsf{ApplyDel}_\beta{\cdot}Add = \{ParMatch_\beta(\vec{u}, 1)\}$.

Notice that the update over $ParMatch_\beta$ changing the last argument of the tuple $\vec{u}$ is used to track that the selected tuple has been already processed for deletion. $\mathsf{ApplyDel}_\beta$ cannot be applied anymore when all tuples in $ParMatch_\beta$ are marked with 1. This situation indicates that no more deletions have to be considered, and that bulk addition must now be handled. Such a transition is captured by the dedicated action $\mathsf{DelToAdd}_\beta$, which does not have action parameters nor fresh-input variables, and is defined as follows:

- $\mathsf{DelToAdd}_\beta{\cdot}\mathsf{guard} = DelPhase_\beta \wedge \forall \vec{u}, m.ParMatch_\beta(\vec{v}, m) \rightarrow m = 1$;
- $\mathsf{DelToAdd}_\beta{\cdot}Del = \{DelPhase_\beta\}$;
- $\mathsf{DelToAdd}_\beta{\cdot}Add = \{AddPhase_\beta\}$.

The second sub-phase simulating the bulk update is captured by the iterative application of action $\mathsf{ApplyAdd}_\beta$, which closely resembles $\mathsf{ApplyDel}_\beta$, with three differences: *(i)* it requires to consider not only the answers to $\beta{\cdot}\mathsf{guard}$ (stored in $ParMatch_\beta$), but also the selected matching for the input variables (stored in $FreshInput_\beta$); *(ii)* it handles the insertion of tuples, hence refers to $\beta{\cdot}Add$; *(iii)* it removes a tuple from $ParMatch_\beta$ to mark that it has been considered for addition. Specifically, $\mathsf{ApplyAdd}_\beta$ has $\vec{u}$ and $\vec{v}$ as action parameters, no fresh-input variables, and is defined as:

- $\mathsf{ApplyAdd}_\beta{\cdot}\mathsf{guard} = AddPhase_\beta \wedge ParMatch_\beta(\vec{u}, 1) \wedge FreshInput_\beta(\vec{v})$;
- $\mathsf{ApplyDel}_\beta{\cdot}Del = \{ParMatch_\beta(\vec{u}, 1)\}$;
- $\mathsf{ApplyDel}_\beta{\cdot}Add = \beta{\cdot}Add$.

It is easy to see that this last sub-phase ends where there is no more tuple in $ParMatch_\beta$. This marks the end of the addition loop, and triggers the execution of the last action of the sequence, namely $\mathsf{Finalize}_\beta$. This last action mirrors Init_β, and has in fact a twofold effect: releasing the lock(s), and emptying the content of $FreshInput_\beta$ by removing its single tuple (recall, in fact, that $ParMatch_\beta$ is already empty). Specifically, $\mathsf{Finalize}_\beta$ has $\vec{v}$ as action parameters (since it needs to match those against the $FreshInput_\beta$ relation), has no fresh-input variable, and is defined as:

- $\mathsf{Finalize}_\beta{\cdot}\mathsf{guard} = FreshInput_\beta(\vec{v}) \wedge \neg\exists \vec{u}, m.ParMatch_\beta(\vec{u}, m)$;
- $\mathsf{Finalize}_\beta{\cdot}Del = \{FreshInput_\beta(\vec{v})\}$;
- $\mathsf{Finalize}_\beta{\cdot}Add = \emptyset$.