Proceedings of the Fourteenth International Conference on Principles of Knowledge Representation and Reasoning

State-Boundedness in Data-Aware Dynamic Systems

Babak Bagheri Hariri
Diego Calvanese **Marco Montali**
Free University of Bozen-Bolzano, Italy
lastname@inf.unibz.it

Alin Deutsch
University of California, San Diego
San Diego, CA USA
deutsch@cs.ucsd.edu

Abstract

Verification of dynamic systems that manipulate data, stored in a database or ontology, has lately received increasing attention. A plethora of recent works has shown that verification of systems working over unboundedly many data is decidable even for very rich temporal properties, provided that the system is *state-bounded*. This condition requires the existence of an overall bound on the amount of data stored in each single state along the system evolution. In general, checking state-boundedness is undecidable. An open question is whether it is possible to isolate significant classes of dynamic systems for which state-boundedness is decidable. In this paper we provide a strong negative answer, by resorting to a novel connection with variants of Petri nets. In particular, we show undecidability for systems whose data component contains unary relations only, and whose action component queries and updates such relations in a very limited way. To contrast this result, we propose interesting relaxations of the sufficient conditions proposed in the concrete setting of Data-Centric Dynamic Systems, building on recent results on chase termination for tuple-generating dependencies.

1 Introduction

The formalization and verification of dynamic systems that manipulate data, stored in a database or ontology, has lately received an increasing attention from both a foundational and a practical point of view (Calvanese, De Giacomo, and Montali 2013). In particular, business process management has progressively shifted its focus from a purely control-flow perspective that under-specifies (if not completely abstracts away) the data component, to a data-aware perspective where data have the same importance as system dynamics (Dumas 2011). This comprehensive view of dynamic systems has given rise to a series of paradigms, such as data-centric workflows (Vianu 2009) and business artifacts (Hull 2008), where the static (i.e., data-related) and dynamic (i.e., process-related) system aspects are jointly modelled and managed.

The verification of a data-aware dynamic system is notoriously challenging: as soon as it can incorporate (possibly unbounded) data from the external environment, e.g., due to interaction with external services or humans, the transition system representing its execution becomes infinite-state, and verification turns out to be undecidable even for propositional reachability properties (Deutsch et al. 2009; Belardinelli, Lomuscio, and Patrizi 2012; Bagheri Hariri et al. 2013b; 2013a). To mitigate this problem, an extensive amount of research has been devoted to find suitable classes of data-aware dynamic systems for which verification of first-order temporal properties becomes decidable. In particular, a plethora of recent works has shown that verification of data-aware dynamic systems working over unboundedly many data is decidable even for very rich temporal logics, provided that the system is *state-bounded*. For example, Belardinelli, Lomuscio, and Patrizi (2012) show decidability of verification of state-bounded (there called b-bounded) artifact centric multiagent systems (ACMASs) for a first-order, epistemic variant of CTL, with active domain quantification that applies across time points. Bagheri Hariri et al. (2013b) give a key decidability result for the verification of state-bounded Data-Centric Dynamic Systems (DCDSs) against a first-order variant of the μ-calculus with a limited form of quantification across time. Within the research line of reasoning about actions, De Giacomo, Lesperance, and Patrizi (2012) show that (state-)bounded Situation Calculus theories can be verified against a first-order variant of the μ-calculus, without quantification across states. Notably, in all these cases state-boundedness guarantees the existence of a faithful (sound and complete) finite-state abstraction of the system, paving the way for the application of standard model checking tools.

Intuitively, state boundedness requires the existence of an overall bound on the amount of data stored in each single state of the system (a database instance or an ABox) along the system evolution. As shown by Bagheri Hariri et al. (2013b), checking state-boundedness is in general undecidable. The proof is given for the specific framework of DCDSs, but can be straightforwardly applied to other similar frameworks, such as ACMASs. Due to undecidability of state-boundedness, the existing works have either: *(i)* assumed that the system is state-bounded (Belardinelli, Lomuscio, and Patrizi 2012); *(ii)* explicitly enforced state-boundedness by blocking those actions that would lead to exceed the bound (De Giacomo, Lesperance, and Patrizi 2012; Solomakhin et al. 2013); *(iii)* studied sufficient conditions that can be checked over the dynamic component of the system (actions and/or processes) to test whether they guarantee state-boundedness (Bagheri Hariri et al. 2013b).

An open question is whether it is possible to isolate significant classes of dynamic systems for which state-boundedness is decidable. In this paper we study this problem, devising a novel connection between variants of Petri nets (Transfer/Reset nets) (Dufourd, Finkel, and Schnoebelen 1998; Dufourd, Jancar, and Schnoebelen 1999) and DCDSs, and inter-reducing the problem of checking whether a DCDS is state-bounded to the problem of checking whether a Transfer/Reset/Petri net is bounded (i.e., ensures that the number of tokens present in each place of the net never exceeds a given bound). Observe that DCDSs are expressively equivalent to business artifact formalisms for business process specification deployed in industrial settings (Bagheri Hariri et al. 2013b; Solomakhin et al. 2013). Thanks to the connection between DCDSs and Petri nets, we provide a strong negative answer to the aforementioned open question: we show undecidability of DCDSs with unary relations, queried in a very limited way (i.e., without using joins, and with a very limited usage of inequalities). Furthermore, we show that undecidability is mainly related to the possibility for relations to lose their content during the application of a single action, and that decidability is obtained if this is prevented (though with an intractable complexity lower bound). Also in this case, the provided results can be transferred to other similar frameworks, such as ACMASs.

To contrast this negative result, we improve the sufficient (syntactic) conditions proposed by Bagheri Hariri et al. (2013b) to check whether a DCDS is state-bounded, by defining a hierarchy of sufficient conditions that progressively refine the analysis of how data are manipulated by the action component of the DCDS. To do so, we take inspiration from recent results on chase termination for tuple-generating dependencies, and in particular from the *safety* (Meier, Schmidt, and Lausen 2009) and *stratified* (Deutsch, Nash, and Remmel 2008) extensions of *weak-acyclicity* (Fagin et al. 2005).

It is interesting to observe that all our results can be lifted from the pure relational setting we consider here, to a knowledge-based setting as the one presented in (Bagheri Hariri et al. 2013a). This can be immediately done when the data component is a full-fledged ontology expressed in a Description Logic that allows for rewriting queries posed over the ontology as unions of conjunctive queries posed over the data only (Calvanese et al. 2007).

2 Data-Centric Dynamic Systems

We recall the main aspects of Data-Centric Dynamic Systems (DCDSs). For a comprehensive treatment, the reader is referred to (Bagheri Hariri et al. 2013b).

A DCDS $\mathcal{S}$ is a pair $\langle \mathcal{D}, \mathcal{P} \rangle$, where $\mathcal{D}$ is the data component of $\mathcal{S}$, and $\mathcal{P}$ is its process component. The data component $\mathcal{D}$ is a tuple $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, I_0 \rangle$, where: *(i)* $\mathcal{C}$ is a countably infinite set of constants; *(ii)* $\mathcal{R}$ is a *database schema*, i.e., a sets of relation schemas; *(iii)* $\mathcal{E}$ is a set of *equality constraints* over $\mathcal{R}$; *(iv)* I_0 is the *initial database instance* of $\mathcal{S}$, i.e., a database instance conforming to $\mathcal{R}$ and $\mathcal{E}$, and made up of values in $\mathcal{C}$. Given a relation $R \in \mathcal{R}$ and a database instance I conforming to $\mathcal{R}$, we denote with $|R|_I$ the number of R-tuples contained in I.

The process layer $\mathcal{P}$ defines the progression mechanism for the DCDS. It is constituted by a process, which queries the current data maintained by $\mathcal{D}$ and determines which actions are executable; actions, in turn, query and update $\mathcal{D}$, possibly introducing new values from the external environment, by issuing service calls. Technically, $\mathcal{P} = \langle \mathcal{F}, A, \varrho \rangle$, where *(i)* $\mathcal{F}$ is a finite set of *functions*, each representing the interface to a (nondeterministic) *external service*; *(ii)* A is a finite set of *actions*, whose execution updates the data layer, and may involve external service calls; *(iii)* ϱ is a finite set of *condition-action rules* that form the specification of the overall *process*, which tells at any moment which actions can be executed. By considering a finite set $\mathfrak{A}$ of action names, an *action* $\alpha \in A$ is an expression $\alpha(p_1, \ldots, p_n) : \{e_1, \ldots, e_m\}$, where: *(i)* $\alpha(p_1, \ldots, p_n)$ is its *signature*, constituted by a name $\alpha \in \mathfrak{A}$ and a sequence $p_1, \ldots, p_n$ of *parameters*, to be substituted with values when the action is invoked, and *(ii)* $\{e_1, \ldots, e_m\}$, also denoted as $\text{EFFECT}(\alpha)$, is a set of *specifications of effects*, which are assumed to take place simultaneously. Each e_i has the form $q_i^+ \wedge Q_i^- \rightsquigarrow E_i$, where:

- $q_i^+ \wedge Q_i^-$ is a query over $\mathcal{R}$ whose terms are variables, action parameters, and constants from I_0, where q_i^+ is a union of conjunctive queries, and Q_i^- is an arbitrary FO formula whose free variables are among those of q_i^+. Intuitively, q_i^+ selects the tuples to instantiate the effect with, and Q_i^- filters away some of them.
- E_i is the effect, i.e., a set of facts for $\mathcal{R}$, which includes as terms: terms in the active domain of I_0, free variables of q_i^+ and Q_i^- (including action parameters), and Skolem terms formed by applying a function $f \in \mathcal{F}$ to one of the previous kinds of terms. Such Skolem terms involving functions represent external (nondeterministic) service calls and are interpreted as the returned value chosen by an external user/environment when executing the action.

The *process* ϱ is a finite set of *condition-action rules*, of the form $Q \mapsto \alpha$, where α is an action in A and Q is a FO query over $\mathcal{R}$ whose free variables are exactly the parameters of α, and whose other terms can be quantified variables or constants mentioned in I_0.

Execution semantics. The execution semantics of a DCDS $\mathcal{S}$ is a possibly infinite transition system $\Upsilon_{\mathcal{S}}$ whose states are labeled by database instances. It represents all possible computations that the process layer can do on the data layer. Specifically, $\Upsilon_{\mathcal{S}} = \langle \mathcal{C}, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$, where: *(i)* Σ is a set of states; *(ii)* $s_0 \in \Sigma$ is the initial state; *(iii)* db is a function that, given a state $s \in \Sigma$, returns the database of s, which is made up of values in $\mathcal{C}$ and conforms to $\mathcal{R}$; *(iv)* $\Rightarrow \subseteq \Sigma \times \mathfrak{A} \times \Sigma$ is a transition relation over states, labeled by action names.

Given a DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ with $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, I_0 \rangle$ and $\mathcal{P} = \langle \mathcal{F}, A, \varrho \rangle$, the transition system $\Upsilon_{\mathcal{S}}$ is intuitively constructed as follows. Starting from I_0, all condition-action rules in ϱ are evaluated, determining which actions are executable, and with which ground parameter assignments. Nondeterministically, one such action with parameter assignment $\alpha\rho$ is selected and executed over I_0. To do so, every effect of α (partially grounded with the parameter assignment ρ)

is evaluated, by calculating all the answers of its left-hand side, and grounding the right-hand side accordingly. If the right-hand side contains service calls, they are issued, receiving back for each of them a value nondeterministically chosen from $\mathcal{C}$. This value is then used to substitute the service call with the actual result. The overall set of ground facts obtained in this way finally constitutes the next database instance. Notice that upon the execution of $\alpha\rho$, the content of a relation is lost unless it is explicitly maintained through dedicated effects of α. The transition system construction then proceeds by constructing all possible successors, each of which is obtained by selecting one of the executable actions with parameters, and one result for each of the involved service calls. The construction then recursively proceeds over this newly generated states. For a formal description of the execution semantics, see (Bagheri Hariri et al. 2013b).

State-Boundedness. Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, I_0 \rangle$ and $\mathcal{P} = \langle \mathcal{F}, A, \varrho \rangle$. We say that $\mathcal{S}$ is *state-bounded* if every state of $\Upsilon_{\mathcal{S}}$ is labeled with a database instance whose size does not exceed a pre-defined bound. We say that $\mathcal{S}$ is *structurally state-bounded* if, for every database instance I_0' conforming to $\mathcal{R}$ and $\mathcal{E}$, and made up of values in $\mathcal{C}$, $\langle \mathcal{D}', \mathcal{P} \rangle$ is state-bounded, where $\mathcal{D}' = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, I_0' \rangle$.

3 Relating DCDSs and Reset Transfer Nets

In this section we introduce variants of Petri nets and draw a formal correspondence with some specific classes of DCDSs. In Section 4 we leverage on this connection to provide key results related to (structural) state-boundedness of DCDSs. We assume the reader has knowledge of standard Petri nets.[1]

3.1 Reset Transfer Nets

Reset Transfer nets (RT nets) are a restricted form of Reset Post G-nets, introduced for the first time in (Dufourd, Finkel, and Schnoebelen 1998). Intuitively, a (standard) Petri net can consume and produce a fixed amount of tokens when a transition fires, while all the remaining tokens are left untouched. The "reset" dimension of RT nets extends this behavior by allowing to instantaneously emptying the content of some places when a transition fires. The "transfer" dimension of RT nets supports instead the possibility of instantaneously transferring the entire content of some place to another place when a transition fires. These specific classes exactly correspond to those defined in (Dufourd, Finkel, and Schnoebelen 1998). An RT net combines the capabilities of R and T nets.

A (marked) *RT net* is a tuple $N = \langle P, T, F, m_0 \rangle$, where:

- P is a set of *places*;
- T is a set of *transitions* (with $P \cap T = \emptyset$);
- $F : P \times T \cup T \times P \to P \cup \{0, 1\}$ is a *flow function* mapping each pair place/transition (called *pre-arc*) and transition/place (called *post-arc*) to either a place name or the integers 0 or 1,[2] with the following constraints:
  - for each place $p \in P$ and transition $t \in T$, $F(p, t) \in \{0, 1, p\}$;
  - for each transition $t \in T$ and place $p_2 \in P$, if $F(t, p_2) = p$, then the following two conditions hold: *(i)* $F(p, t) = p$, *(ii)* for each place $p_3 \in P$ with $p_3 \neq p_2$, $F(t, p_3) \neq p$.
- m_0 is the *initial marking*, i.e., a multiset of places $m_0 \in \mathbb{B}(P)$, with $\mathbb{B}(P) : P \to \mathbb{N}$ the set of multisets over the set P of places.

Given a place $p \in P$ and a transition $t \in T$, we say that p is an *input place* for t if $F(p, t) = 1$, and that (p, t) is a *reset arc* if $F(p, t) = p$. Symmetrically, given a place $p_2 \in P$ and a transition $t \in T$, we say that p_2 is an *output place* for t if $F(t, p_2) = 1$, and that (t, p_2) is a *copy arc* if $F(t, p_2) = p$ for some $p \in P$. In this latter case, the pre-arc (p, t) and the post-arc (t, p_2) form together a *transfer arc*[3].

An *RT net* $N = \langle P, T, F, m_0 \rangle$ is a:

- *Reset Petri net (R net)* if for each $t \in T$ and $p \in P$, $F(t, p) \in \{0, 1\}$;
- *Transfer Petri net (T net)* if for each $p \in P$ and $t \in T$ such that $F(p, t) = p$, there exists $p_2 \in P$ such that $F(t, p_2) = p$;
- *Petr net* if for each $p \in P$ and $t \in T$, $F(p, t) \in \{0, 1\}$ and $F(t, p) \in \{0, 1\}$.

Execution Semantics. Given a marking m and a place p, notation $m(p) \in \mathbb{N}$ represents the number of tokens assigned by m to p, i.e., the number of times p appears in m. Instrumental to the definition of the net execution semantics is the notion of "token transfer" induced by an arc of the net for a given marking. This is formally captured by the function TRANSF, defined as follows: given an RT net $N = \langle P, T, F, m_0 \rangle$, for every marking m over P, place $p \in P$, and transition $t \in T$, we have

$$\text{TRANSF}_m(p, t) = \begin{cases} m(p) & \text{if } F(p, t) = p \\ F(p, t) & \text{otherwise} \end{cases}$$

$$\text{TRANSF}_m(t, p) = \begin{cases} m(p') & \text{if } F(t, p) = p' \\ F(t, p) & \text{otherwise} \end{cases}$$

Given a marking m and a transition $t \in T$, we say that t is *enabled* (or *fireable*) in m, written $m[t\rangle$, if, for every place $p \in P$, we have that $m(p) \geq \text{TRANSF}_m(p, t)$. Given a transition t and two markings m and m', we say that t fires in m producing m', written $m[t\rangle m'$, if and only if: *(i)* t is enabled in m, *(ii)* m' is such that for every $p \in P$,

$$m'(p) = m(p) - \text{TRANSF}_m(p, t) + \text{TRANSF}_m(t, p)$$

The execution semantics of an RT net $N = \langle P, T, F, m_0 \rangle$ is defined in terms of a transition system $\text{RG}(N) = \langle M, m_0, \to \rangle$ (called *reachability graph*), where $M \subseteq \mathbb{B}(P)$ is a set of markings, and $\to \subseteq M \times T \times M$ is a transition relation labeled with elements from T. Specifically, M and $\to$ are defined by mutual induction as the smallest sets satisfying the following properties: *(i)* $m_0 \in M$; *(ii)* if $m \in M$, then for every transition $t \in T$ and marking m' such that $m[t\rangle m'$, we have $m' \in M$ and $m \xrightarrow{t} m'$. An RT net N is *bounded* if and only if each marking in $\text{RG}(N)$ is bounded or, equivalently, $\text{RG}(N)$ contains a finite number of markings.

[1] Here Petri net is used to identify a Place Transition (P/T) net.

[2] Admitting arbitrary integers does not increase the expressive power of the model.

[3] Remember that, by definition, in this case we have $F(p, t) = p$.

An RT net $N = \langle P, T, F, m_0 \rangle$ is *structurally bounded* if its boundedness does not depend on the initial marking, i.e., for every $m'_0 \in \mathbb{B}(P)$, $\langle P, T, F, m'_0 \rangle$ is bounded.

3.2 From RT Nets to DCDSs

We formally define a translation from RT Nets to DCDSs, and then provide an informal account for it. Specifically, we introduce a translation function N-TO-D that, given an RT net $N = \langle P, T, F, m_0 \rangle$, produces a DCDS $\text{N-TO-D}(N) = \langle \mathcal{D}_N, \mathcal{P}_N \rangle$ whose execution semantics weakly reproduces that of $\text{RG}(N)$. The data layer is $\mathcal{D}_N = \langle \mathcal{C}, \mathcal{R}_N, \emptyset, I_0^N \rangle$, where

- $P_i/1 \in \mathcal{R}_N$ if and only if $p_i \in P$;
- I_0^N is such that $\{P_i(d_1), \ldots, P_i(d_k) \mid d_j \neq d_h \text{ for } j, h \in \{1, \ldots, k\} \text{ and } j \neq h\} \subseteq I_0^N$ if and only if $p^k \in m_0$.

The process layer of $\text{N-TO-D}(N)$ is $\mathcal{P}_N = \langle \mathcal{F}_N, A_N, \varrho_N \rangle$, where every transition $t \in T$ is translated into a condition-action rule $\xi_t \in \varrho_N$ and a dedicated action $\alpha_t \in A_N$, which respectively encode the enablement and firing semantics of t:

$$\xi_t = \bigwedge_{p_i^{in} \in \{p \mid F(p,t)=1\}} P_i^{in}(x_i) \mapsto \alpha_t(x_1, \ldots, x_{|\{p \mid F(p,t)=1\}|})$$

where $\alpha_t(x_1, \ldots, x_n)$ contains the following effects:

1. *(Lossy maintenance of tokens in input places)* For each $p_i^{in} \in P$ such that $F(p_i^{in}, t) = 1$:
$$P_i^{in}(y) \wedge y \neq x_i \rightsquigarrow P_i^{in}(f_i(y)) \in \text{EFFECT}(\alpha_t) \quad (1)$$
2. *(Lossy maintenance of tokens in other places)* For each $p_l \in P$ such that $F(p_l, t) = 0$:
$$P_l(y) \rightsquigarrow P_l(h_l(y)) \in \text{EFFECT}(\alpha_t) \quad (2)$$
3. *(Lossy token generation)* For each $p_j^{out} \in P$ such that $F(t, p_j^{out}) = 1$:
$$\mathsf{true} \rightsquigarrow \{P_j^{out}(g_j())\} \in \text{EFFECT}(\alpha_t) \quad (3)$$
4. *(Lossy token transfer)* For each $p_k^{tout} \in P$ such that $F(t, p_k^{out}) = p_k^{tin}$:
$$P_k^{tin}(y) \rightsquigarrow \{P_k^{tout}(h_k(y))\} \in \text{EFFECT}(\alpha_t) \quad (4)$$
5. No other effect belongs to $\text{EFFECT}(\alpha_t)$.

The intuition behind the translation is to mimic the firing of a transition by means of service calls. In the resulting DCDS, places become unary relations, and tokens become distinct identifiers from $\mathcal{C}$. The condition-action rule ξ_t checks whether each input place for t has a non-empty extension (i.e., contains at least one token) and, if so, non-deterministically selects one token from each of such places. Effects of type (1) and (2) maintain those tokens that are not affected by the firing, i.e., tokens present in non-input places, or tokens contained in input places but not selected by ξ_t. The maintenance is "lossy" because there is no guarantee that the service calls f_i will return different identifiers when called with different values. This notion of *lossiness* resembles that proposed by (Mayr 2003) for lossy counter machines and lossy Petri nets. Effects of type (3) are used to generate the tokens that t puts in its output places upon firing. Such generation is also lossy, because there is no guarantee that service calls g_j will return fresh identifiers, distinct from those already present in the output places. Effects of type (4) lossily deal with the transfer arcs of t, producing one token to be put in the transfer destination for each token present in the transfer source. Note that those places connected to a transition with a reset arc that does not form a transfer arc, do not lead to any effect in the translation. This reflects the "destructive" nature of DCDSs (data not explicitly copied are lost during the application of an action).

The following theorem states that, despite lossiness, the obtained DCDS preserves boundedness of the original net.

Theorem 1. *For every RT net N, N is bounded if and only if $\text{N-TO-D}(N)$ is state-bounded.*

This is proven in two steps. First we show that $\text{N-TO-D}(N)$ obeys to a sort of *monotonicity*. Given two database instances I_1 and I_2 over $\mathcal{R}$, we say that I_1 is *cardinality-contained* in I_2, written $I_1 \trianglelefteq I_2$ if for every relation $R \in \mathcal{R}$, $|R|_{I_1} \leq |R|_{I_2}$. Monotonicity is then defined as follows: for every two states s_1 and s_2 in Σ such that $db(s_1) \trianglelefteq db(s_2)$, and for each state s'_1 such that $s_1 \overset{\alpha_t}{\Longrightarrow} s'_1$, there exists s'_2 such that $s_2 \overset{\alpha_t}{\Longrightarrow} s'_2$ and $db(s'_1) \trianglelefteq db(s'_2)$. Monotonicity implies that, as long as state boundedness is concerned, for every state s in $\Upsilon_{\text{N-TO-D}(N)}$ it is sufficient to consider only its "maximal successors" $\text{MAX-SUCC}_{\alpha_t}(s) = \{s' \in \Sigma \mid s \overset{\alpha_t}{\Longrightarrow} s' \text{ and } \nexists s''.s \overset{\alpha_t}{\Longrightarrow} s'' \text{ and } s' \trianglelefteq s''\}$.

The second step consists in showing that the portion of $\Upsilon_{\text{N-TO-D}(N)}$ that just considers maximal successors faithfully reproduces $\text{RG}(N)$. To compare the states of $\Upsilon_{\text{N-TO-D}(N)}$ with those of $\text{RG}(N)$, we define the following *cardinality-equivalence* relation: given a marking $m \in M$ and a state $s \in \Sigma$, we say that m is cardinality-equivalent to s, written $m \approx s$, if, for each place $p_i \in P$, $m(p) = n$ if and only if $|P_i|_{db(s)} = n$. By definition, $m_0 \approx s_0$. It can then be shown, inductively, that, given $m \in M$ and $s \in \Sigma$ such that $m \approx s$, for every transition $t \in T$ (and corresponding $\alpha_t \in A_N$):

- $m[t\rangle$ if and only if α_t is executable in s;
- whenever $m[t\rangle$, given $m' \in M$ such that $m \xrightarrow{t} m'$, we have $m' \approx s'$ for every $s' \in \text{MAX-SUCC}_s(\alpha_t)$.

This implies the result, since $\text{RG}(N)$ has an unbounded run if and only if $\Upsilon_{\text{N-TO-D}(N)}$ has a state-unbounded run.

As a direct consequence, we obtain:

Corollary 2. *For every RT net N, N is structurally bounded if and only if $\text{N-TO-D}(N)$ is structurally state-bounded.*

3.3 Lossy Reset Transfer DCDSs

We now introduce a class of DCDSs that resembles those obtained from RT nets through the application of the N-TO-D translation function. For this reason, we call such a class *Lossy Reset Transfer (LRT)* DCDSs.

An *LRT DCDS* is a DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$, with $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, I_0 \rangle$ and $\mathcal{P} = \langle \mathcal{F}, A, \varrho \rangle$, that obeys to the following requirements:

- $\mathcal{R} = \{P_1/1, \ldots, P_{|\mathcal{R}|}/1\}$ (only unary relations are used).
- $\mathcal{E} = \emptyset$.
- For each action $\alpha(\vec{x}) \in A$, ϱ contains a single rule of the form $Q(\vec{x}) \mapsto \alpha(\vec{x})$, where Q is a (non-self) cartesian

product of a set of unary relations:

$$Q(x_1, \ldots, x_n) = \bigwedge_{i \in \{1,\ldots,n\},\ P_i \neq P_j \text{ for } i \neq j} P_i(x_i)$$

The set of relations in Q is denoted by $\text{RELS}(Q)$.

- Every action $\alpha(\vec{x}) \in A$, with $Q(\vec{x}) \mapsto \alpha(\vec{x}) \in \varrho$, has the following shape:
  1. for each $P_i \in \text{RELS}(Q)$, α *must* contain a *lossy maintain-rest* effect of the form
  $$P_i(y) \wedge y \neq x_i \rightsquigarrow P_i(f_i(y)) \quad (5)$$
  2. for each $P_l \in \mathcal{R} \setminus \text{RELS}(Q)$, α *may* contain a *lossy maintain* effect of the form
  $$P_l(y) \rightsquigarrow P_l(h_l(y)) \quad (6)$$
  3. for each $P_j \in \mathcal{R} \setminus \text{RELS}(Q)$, α *may* contain *either* a *lossy create* effect of the form
  $$\mathsf{true} \rightsquigarrow P_j(g_j()) \quad (7)$$
  or a *lossy copy* effect of the form
  $$P'_j(y) \rightsquigarrow P_j(h_j(y)) \quad (8)$$
  for some $P'_j \in \mathcal{R} \setminus (\text{RELS}(Q) \cup P_j)$;
  4. no other effect belongs to α.

An LRT DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ with $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \emptyset, I_0 \rangle$ and $\mathcal{P} = \langle \mathcal{F}, A, \varrho \rangle$ is:

- A *Lossy Reset (LR) DCDS* if no action $\alpha \in A$ contains lossy-copy effects of the form (8); the "reset" nature comes from the fact that those relations that are not mentioned in the left-hand side of any effect in $\text{EFFECT}(\alpha)$ loose their current content when α is executed.
- A *Lossy Transfer (LT) DCDS* if, for every action $\alpha \in A$ with $Q(\vec{x}) \mapsto \alpha(\vec{x}) \in \varrho$, and every relation $P_i \in \mathcal{R} \setminus \text{RELS}(Q)$, P_i appears in the left-hand side of at least an effect of α (which must be either a lossy maintain effect of the form (6), or a lossy copy effect of the form (8)); in other words, every relation contributes to determine the content of the state obtained by the application of α.
- A *Lossy Petri (LP) DCDS* if, for every action $\alpha \in A$ with $Q(\vec{x}) \mapsto \alpha(\vec{x}) \in \varrho$, and every relation $P_i \in \mathcal{R} \setminus \text{RELS}(Q)$, P_i appears only in the left-hand side of a lossy maintain effect of the form (6) in α (and possibly in the right-hand side of a lossy create effect).

The following proposition highlights the connection induced by the N-TO-D translation function between the different classes of nets and DCDSs studied in this paper.

Proposition 3. *Given an RT/R/T/Petri net N,* $\text{N-TO-D}(N)$ *produces an LRT/LR/LT/LP DCDS.*

The proof of this proposition can be obtained by comparing the definitions for such different classes, and checking that N-TO-D preserves their properties in the translation.

We conclude this section by pointing out that LRT DCDSs are very weak in terms of their ability of querying and manipulating data. In fact, they only contain unary relations, which are queried without using any joins in the condition-action rules, and only by means of atomic queries in the effects. Furthermore, no negative components are used in such queries, except for a very limited use of inequalities (always employed to select all tuples of a relation except the one matching with one of the action parameters). Also, the content of relations is not guaranteed to be preserved when executing an action: copy and maintain effects always employ service calls to transfer data from one state to the other, which implies that the extension of a relation may change in content, and decrease in size. Finally, the overall size of the entire database instance may maximally increase only of a fixed amount of tuples, determined by the number of lossy create effects present in the applied action.

3.4 From LRT DCDSs to RT Nets

We define a translation function D-TO-N that, given an LRT DCDS $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ with $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \emptyset, I_0 \rangle$ and $\mathcal{P} = \langle \mathcal{F}, A, \varrho \rangle$, produces an RT net $\text{D-TO-N}(\mathcal{S}) = \langle P_\mathcal{S}, T_\mathcal{S}, F_\mathcal{S}, m_0^\mathcal{S} \rangle$ defined as follows:

- $p_i \in P_\mathcal{S}$ if and only if $P_i/1 \in \mathcal{R}$;
- $p_i^k \in m_0^\mathcal{S}$ if and only if $|P_i|_{I_0} = k$;
- $t_\alpha \in T$ if and only if $\alpha \in A$;
- As for pre-arcs, for each $p_i \in P_\mathcal{S}$ and each $t_\alpha \in T$:
  1. if, given $Q(\vec{x}) \mapsto \alpha(\vec{x})$ in ϱ, $P_i \in \text{RELS}(Q)$, then $F_\mathcal{S}(p_i, t_\alpha) = 1$;
  2. else if α contains an effect of the form $P_i(y) \rightsquigarrow P_i(h_i(y))$,[4] then $F_\mathcal{S}(p_i, t_\alpha) = 0$;
  3. else $F_\mathcal{S}(p_i, t_\alpha) = p_i$.
- As for post-arcs, for each $t_\alpha \in T$ and $p_i \in P_\mathcal{S}$:
  1. if α contains an effect of the form $\mathsf{true} \rightsquigarrow P_i(g_i())$, then $F(t_\alpha, p_i) = 1$;
  2. if α contains an effect of the form $P'_i(y) \rightsquigarrow P_i(h_i(y))$ with $P'_i \neq P_i$, then $F(t_\alpha, p_i) = p'_i$;
  3. $F(t_\alpha, p_i) = 0$ otherwise.

Intuitively, the translation works as follows. Every relation mentioned in the condition-action rule for α becomes an input place for t_α, and every relation targeted by a lossy create effect becomes an output place for t_α. Relations which are lossily maintained by α do not contribute as input for t_α, attesting that t will not consume tokens from them. Relations that do not fall in any of these two categories are consumed completely by t; this can happen either because their content is lossily copied, or because they do not appear in the left-hand side of any effect in α (and hence will loose their content when α is applied). Finally, lossy copy effects are translated into a transfer arc passing through t (notice that, in this case, the definition of LRT DCDSs does not allow the target relation to be also targeted by a lossy create effect).

Theorem 4. *The translation functions* N-TO-D *and* D-TO-N *are one the inverse of the other, i.e., (i) for every RT net N, $N = \text{D-TO-N}(\text{N-TO-D}(N))$; (ii) for every LRT DCDS $\mathcal{S}$, $\mathcal{S} = \text{N-TO-D}(\text{D-TO-N}(\mathcal{S}))$.*

The proof of this theorem is obtained by: *(i)* considering each component of an RT net, and showing that the same component is obtained by applying D-TO-N $\circ$ N-TO-D on it, *(ii)* considering each component of an LRT DCDS, and showing that the same component is obtained by applying N-TO-D $\circ$ D-TO-N on it.

Theorems 4 and 1, together with Proposition 3, imply:

[4] Note that the same relation P_i is used in both sides of the effect.

Corollary 5. *Given an LRT/LR/LT/LP DCDS $\mathcal{S}$,* D-TO-N($\mathcal{S}$) *produces an RT/R/T/Petri net.*

Theorem 6. *For every LRT DCDS $\mathcal{S}$, $\mathcal{S}$ is state-bounded if and only if* D-TO-N$\mathcal{S}$ *is bounded.*

As a direct consequence of Theorem 6, we then have:

Corollary 7. *For every LRT DCDS $\mathcal{S}$, $\mathcal{S}$ is structurally state-bounded if and only if* D-TO-N($\mathcal{S}$) *is structurally bounded.*

4 State-Boundedness is Difficult

We now exploit the connection between RT nets (and their subclasses) and LRT DCDSs (and their subclasses) to show that state-boundedness is intrinsically difficult to check for data-aware dynamic systems: either highly-undecidable[5] or intractable even for the low expressive systems expressible with LRT DCDSs. In particular, we have that: *(i)* checking boundedness of R nets reduces to checking state-boundedness of LR DCDSs (cf. Proposition 3 and Theorem 1); *(ii)* checking state-boundedness of LP DCDSs reduces to checking boundedness of Petri nets (cf. Corollary 5 and Theorem 6); *(iii)* checking state-boundedness of LT DCDSs reduces to checking boundedness of T nets (cf. Corollary 5 and Theorem 6). Similar reductions hold for structural state-boundedness, thanks to Proposition 3 and Corollaries 2, 5, and 7.

Since checking boundedness for R nets is Σ^1_1-complete (Dufourd, Finkel, and Schnoebelen 1998; Dufourd, Jancar, and Schnoebelen 1999), we thus get:

Theorem 8. *Checking state-boundedness is highly undecidable for LR (and hence LRT) DCDSs.*

Thanks to the fact that checking boundedness is EXPSPACE-complete for Petri nets (Esparza and Nielsen 1994), and decidable for T nets (Dufourd, Jancar, and Schnoebelen 1999), decidability is interestingly obtained for those LRT DCDSs in which all relations are used in the left-hand side of effects, i.e., all relations contribute to determine the new database instance obtained after the application of an action. The sub-class of LRT DCDSs that enjoy this property is that of LT (which includes LP) DCDSs:

Theorem 9. *Checking state-boundedness is* EXPSPACE*-complete for LP DCDSs.*

Theorem 10. *Checking state-boundedness is decidable and* EXPSPACE*-hard for LT DCDSs.*

Structural state-boundedness pushes the undecidability frontier even further, but can be efficiently checked for the very simple case of LP DCDSs. Specifically, since checking structural boundedness is Π^1_2-complete for R nets, Π^1_1-complete for T nets (Dufourd, Jancar, and Schnoebelen 1999), and decidable in PTIME for Petri nets, (Esparza and Nielsen 1994; Dufourd, Jancar, and Schnoebelen 1999) we get:

Theorem 11. *Checking structural state-boundedness is highly undecidable for LT and LR (and hence LRT) DCDSs.*

Theorem 12. *Checking structural state-boundedness is decidable in* PTIME *for LP DCDSs.*

[5]I.e., at the first level or above in the Analytical Hierarchy (Rogers 1967).

We conclude by observing that a line of reasoning analogus to that of DCDSs can be repeated for similar frameworks, e.g., ACMASs (Belardinelli, Lomuscio, and Patrizi 2012).

5 Sufficient State-Boundedness Conditions

In Section 3.1, we have proved that checking (structural) state-boundedness even for very limited DCDSs is already undecidable. We introduce now sufficient syntactic conditions that guarantee structural state-boundedness. In particular, we discuss different dimensions for relaxation of the GR^+-acyclicity condition originally introduced by Bagheri Hariri et al. (2013b), and based on them we develop our most general sufficient condition, called stratified-$\exists GR^{++}$-acyclicity.

5.1 GR^{++}-Acyclicity

The GR^+-acyclicity condition for structural state-boundedness (Bagheri Hariri et al. 2013b) is a syntactic restriction over the dataflow graph of a DCDS, in which nodes of the dataflow graph represent relation names that occur in the schema of the system, and edges represent the flow of the data between these relations. Here we introduce GR^{++}-acyclicity as a relaxation of GR^+-acyclicity in which the dataflow graph represents the system behavior at finer granularity by using as nodes positions of relations, instead of relations themselves.

We call *refined dataflow graph* of a set A of actions the directed edge-labeled graph $\langle N, G\rangle$ where: *(i)* $N \subseteq \mathcal{R} \times \mathbb{N}^+$ is a set of nodes such that for every R/n occurring in A, $\langle R, i\rangle \in N$ for every $i \in \{1, \ldots, n\}$; *(ii)* G is a 5-tuple $(R_1, \mathsf{id}, R_2, b, a)$, where R_1 and R_2 are two nodes in N, id is a (unique) edge identifier, b is a Boolean flag used to mark *special* edges, and a is flag used to denote the action of A to which the edge belongs. Formally, G is the minimal set such that, for each action α in A, each effect e in α of the form $q^+ \wedge Q^- \rightsquigarrow E$, each $R(r_1, \ldots, r_m)$ in q^+, each $i \in \{1, \ldots, m\}$, each $Q(t_1, \ldots, t_n)$ in E, and each $j \in \{1, \ldots, n\}$, we have that:

- if t_i is a free variable and $r_i = t_j$, then $(\langle R, i\rangle, \mathsf{id}, \langle Q, j\rangle, \mathsf{false}, \alpha) \in G$.
- if t_i is a service call $f(s_1, \ldots, s_l)$, then for each $k \in \{1, \ldots, l\}$, if $r_i = s_k$ then $(\langle R, i\rangle, \mathsf{id}, \langle Q, j\rangle, \mathsf{true}) \in G$.
- if t_i is a nullary service call $f()$, then $(\langle R, i\rangle, \mathsf{id}, \langle Q, j\rangle, \mathsf{true}) \in G$.

in which id is a fresh edge identifier.

Let π_1, π_2, and π_3 be paths in the dataflow graph D of A. We say that $\langle \pi_1, \pi_2, \pi_3\rangle$ is a *generate-recall triple* if

(i) $\pi_1\pi_2\pi_3$ forms a path in D,
(ii) π_1, π_3 are simple cycles, and
(iii) π_2 is a path containing a special edge, with $edges(\pi_2) \setminus edges(\pi_1) \neq \emptyset$.

The generate-recall triple $\langle \pi_1, \pi_2, \pi_3\rangle$ is called *non-accumulating* if there is an edge $e \in edges(\pi_2)$ that does not belong to the same action as any of the edges following e in $\pi_2\pi_3$. Finally, we say that A is GR^{++}*-acyclic* if all its generate-recall triples are non-accumulating.

We say a process layer $\mathcal{P} = \langle \mathcal{F}, A, \varrho\rangle$ is GR^{++}*-acyclic*, if A is GR^{++}-acyclic. A DCDS is GR^{++}-acyclic if its process layer is GR^{++}-acyclic. The GR^+-acyclicity definition by

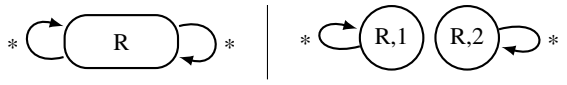

Figure 1: Dataflow and refined dataflow graphs of Example 1

Bagheri Hariri et al. (2013b) differs only in that the dataflow graph's nodes represent entire relations, not their positions (the dataflow graph by Bagheri Hariri et al. corresponds to collapsing all refined dataflow graph nodes modeling positions of relation R into a single node, for every R.)

Example 1. Consider a DCDS D with one action $\alpha = \{R(x,y) \rightsquigarrow \{R(f(x),\ h(y))\}$. It is easy to see that D is *(i)* structural state-bounded, *(ii)* not GR^{+}-acyclic, but *(iii)* GR^{++}-acyclic. For *(i)*, observe that the cardinality of R after executing α is at most the cardinality before execution (potentially strictly lower if the service calls behave non-injectively). For *(ii)*, consult the data flow graph on the left side of Figure 1. The simple cycle π_1 can be the left self-loop on node R, π_2 and π_3 can coincide with the right self-loop. For *(iii)*, consult the refined data flow graph on the right side of Figure 1. Notice now that this graph contains no generate-recall cycle, since in each connected component π_1 and π_2 must coincide. ■

Theorem 13. *GR^{++}-acyclicity is a strict relaxation of GR^{+}-acyclicity that guarantees structural state-boundedness.*

5.2 Safe-GR^{++}-Acyclicity

The main idea of this relaxation is to distinguish different types of special edges in the dataflow graph to define a more refined condition for structural state-boundedness. We borrowed the notion of affected positions, which are an overestimation of the positions in which a new value can be introduced, from Calì, Gottlob, and Kifer (2013), Meier, Schmidt, and Lausen (2009), to distinguish unbounded edges, which is an overestimation of the edges of the dataflow graph, in which unboundedly many different values can flow during the life of a DCDS.

More formally, we call *affected positions* of a set A of actions, denoted with $\text{AFF}(A)$, the smallest set of positions π occurring in A satisfying following conditions:

- If a service call appears in π, then $\pi \in \text{AFF}(A)$.
- If in the head of an effect a variable x appears in π, and occurrences of x in the body of this effect is only limited to the affected positions.

We use affected positions to determine an overestimation of unbounded edges of dataflow graph. Given an effect f, we say that an edge e from a position p in the body of e to a position in the head of e is (potentially) *unbounded* if all the positions in the body of e that using the same variable as p are affected positions. Otherwise, we call e *bounded*. For instance, consider a DCDS with one action as follows: $\{A(x) \wedge B(x) \wedge C(y) \rightsquigarrow A(f(x)), C(h(y))\}$. $\langle A,1\rangle$ and $\langle C,1\rangle$ are the only affected positions. The edges from the position $\langle A,1\rangle$ and $\langle B,1\rangle$ to $\langle A,1\rangle$ are bounded edges, since for each of them there is an unaffected position $\langle B,1\rangle$ in the body of the effect, while the special edge from $\langle C,1\rangle$ to $\langle C,1\rangle$ is an unbounded edge.

We call *safe-refined dataflow graph* of a set A of actions the directed edge-labeled graph $\langle N, G\rangle$ where: *(i)* $N \subseteq \mathcal{R} \times \mathbb{N}^{+}$ is a set of nodes such that for every R/n occurring in A, $\langle R,i\rangle \in N$ for every $i \in \{1,\ldots,n\}$; *(ii)* G is a 6-tuple $(R_1, \mathsf{id}, R_2, f_s, f_b, a)$, where R_1 and R_2 are two nodes in N, id is a (unique) edge identifier, f_s is a flag that specify if the edge is normal with "$-$", or special with "$*$", f_b is a flag that specify if an edge is bounded with "b", or unbounded with "u", and a is flag used to denote the action of A to which the edge belongs. Formally, G is the minimal set satisfying the following condition. For each action α in A, each effect e in α of the form $q^{+} \wedge Q^{-} \rightsquigarrow E$, each $R(r_1,\ldots,r_m)$ in q^{+}, each $i \in \{1,\ldots,m\}$ each $Q(t_1,\ldots,t_n)$ in E, and each $j \in \{1,\ldots,n\}$:

- If t_i is a free variable and $r_i = t_j$,
  - if for each position p in the body of e in which the variable r_i occurs, it holds that $p \in \text{AFF}(A)$, then $(\langle R,i\rangle, \mathsf{id}, \langle Q,j\rangle, \text{-}, u, a) \in G$;
  - otherwise, $(\langle R,i\rangle, \mathsf{id}, \langle Q,j\rangle, \text{-}, b, a) \in G$.
- If t_i is a service call $f(s_1,\ldots,s_l)$ with $l \geq 1$, then for each $k \in \{1,\ldots,l\}$, if $r_i = s_k$, then
  - If for each position p in the body of e in which the variable r_i happens it holds that $p \in \text{AFF}(A)$, then $(\langle R,i\rangle, \mathsf{id}, \langle Q,j\rangle, *, u, a) \in G$;
  - otherwise, $(\langle R,i\rangle, \mathsf{id}, \langle Q,j\rangle, *, b, a) \in G$.
- If t_i is a nullary service call $f()$, then
  - if for each position p in the body of e it holds that $p \in \text{AFF}(A)$ then $(\langle R,i\rangle, \mathsf{id}, \langle Q,j\rangle, *, u) \in G$;
  - otherwise, $(\langle R,i\rangle, \mathsf{id}, \langle Q,j\rangle, *, b) \in G$.

  in which id is a fresh edge identifier.

We define the notion of *generate-recall triple* for a safe-refined dataflow graph exactly as in the case of GR^{++}-acyclicity. Moreover, a generate-recall triple $\langle \pi_1, \pi_2, \pi_3\rangle$ is called *safe* if either there is an edge $e \in edges(\pi_2)$ that does not belong to the same action as any of the edges following e in $\pi_2\pi_3$, or π_3 contains a bounded edge. Finally, we say that A is *safe-GR^{++}-acyclic* if all its generate-recall triples are safe.

Example 2. Consider a DCDS D with one action

$$\alpha = \begin{cases} A(x) \rightsquigarrow \{A(f(x))\} \\ B(x) \rightsquigarrow \{B(x)\} \\ A(x) \rightsquigarrow \{R(x,f(x))\} \\ R(x,y) \wedge B(x) \rightsquigarrow \{R(y,x)\} \end{cases}$$

D is *(i)* structural state-bounded, *(ii)* not GR^{++}-acyclic, but *(iii)* safe-GR^{++}-acyclic.

Given a state I of a DCDS, let's denote with $|R_i|_I$ the number of different values that occurs in I in the position i of the relation R. Consider the states I and I' of D, in which I' is the result of the execution of α over I.

For *(i)* observe that $|A_1|_{I'} \leq |A_1|_I$, $|B_1|_{I'} = |B_1|_I$, $|R_2|_{I'} \leq |A_1|_I + |B_1|_I$, and $|R_1|_{I'} \leq (|A_1|_I + |R_2|_I) \leq (2.|A_1|_I + |B_1|_I)$. Given the fact that the cardinality of $\langle A,1\rangle$ and $\langle B,1\rangle$ is not increasing, and the cardinality of all the positions is linear in the cardinality of these two positions, the size of the resulting instance by application of α cannot

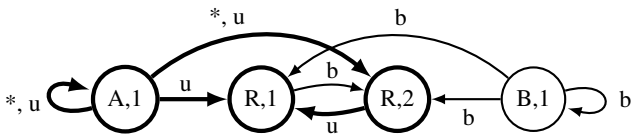

Figure 2: Safe-refined dataflow graph of Example 2. Affected positions and unbounded edges are shown with thicker lines

increase unboundedly. For *(ii)* consult the refined dataflow graph of D given in Figure 2, ignoring the labels for boundedness of edges. The path π_1 can be the self loop in $\langle A, 1\rangle$, π_2 can be the edge from $\langle A, 1\rangle$ to $\langle R, 1\rangle$, and π_3 can be the simple cycle between $\langle R, 1\rangle$ and $\langle R, 1\rangle$. For *(iii)*, consult the safe-refined dataflow graph of Figure 2. $\langle A, 1\rangle$ and $\langle R_2\rangle$ are affected positions because of occurrences of the service call $f(x)$ in them. Moreover, because of the third effect, $\langle R, 1\rangle$ is also an affected position, but $\langle B, 1\rangle$ is not an affected position. Consequently all the outgoing edges from $\langle B, 1\rangle$ are bounded edges. Moreover, because of the occurrences of $\langle B_1\rangle$ with the same variable as the $\langle R, 1\rangle$ in the body of the fourth effect, the outgoing edge from $\langle R, 1\rangle$ to $\langle R, 2\rangle$ is also a bounded edge. Notice now that this graph contains no generate-recall cycle $\pi_1\pi_2\pi_3$ such that all the edges of the recall cycle are unbounded edges. ■

Theorem 14. *Safe-GR^{++}-acyclicity is a strict relaxation of GR^{++}-acyclicity that guarantees structural state-boundedness.*

5.3 Stratified-GR^{++}-Acyclicity

The syntactic restrictions that we have studied until now are defined over the structure of the effects of the actions, and ignore the conditions under which the actions can be executed, i.e., the process of the system. Here, we exploit process to single out larger classes of structural state-bounded DCDSs. Getting insight from stratified relaxation of weak-acyclicity in data exchange (Deutsch, Nash, and Remmel 2008), we specify the potential order of execution of actions of a system, through what we call the control-flow graph of the system, and use this graph to determine maximal set of actions that can violate structural state-boundedness. Remember that actions of DCDSs are not inflationary, and the next state is only determined by applying the effects of the action over the current state. Consequently, by looking to the head of the effects of an action α, we can find a superset of the actions that can be fired after α. We use this idea to relax safe-GR^{++}-acyclicity.

Given an action α and a set of condition-action rules ρ, the set of requirements for firing of α wrt ρ is a set of set of relation names defined as follows: $\text{INPUT}(\rho, \alpha) = \{\{\text{RELATIONS}(Q^+)\} \mid (Q^+ \wedge Q \mapsto \alpha) \in \rho\}$, in which Q^+ is a conjunctive query, and $\text{RELATIONS}(Q)$ is the set of relation names occurring in Q. We denote with $\text{OUTPUT}(\alpha)$ the set of relation names occurring in the head of effects of α. Given a set of condition-action rules ϱ, and two actions α and β, we say that β can (potentially) be fired after α wrt ϱ, written $\alpha <_\varrho \beta$ (or simply $\alpha < \beta$), if there exists a set of relation names $S \in \text{INPUT}(\varrho, \beta)$ such that $S \subseteq \text{OUTPUT}(\alpha)$.

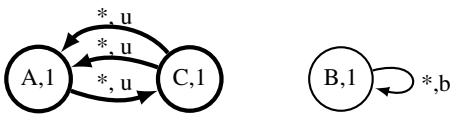

Figure 3: Safe-refined dataflow graph of Example 3. Affected positions and unbounded edges are shown with thicker lines

Given a set of condition-action rules ϱ and a set of actions A, the *controlflow graph* of A is the directed edge-labeled graph $\langle A, E\rangle$ in which for two actions $\alpha, \beta \in \Sigma$ we have that $(\alpha, \beta) \in E$ iff $\alpha < \beta$. We say a DCDS with the set of actions A is *stratified-GR^{++}-acyclic* if for every set of actions S that correspond to a maximal strongly connected component in the controlflow graph of A, the restriction of A to S is safe-GR^{++}-acyclic.

Example 3. Consider a DCDS D with two actions $\alpha(x) = \{A(x) \wedge B(y) \rightsquigarrow \{C(x), B(y)\}$ and $\beta(x, y) = \{C(x) \wedge C(y) \rightsquigarrow \{A(g(x)), A(h(y))\}$ and the set of condition-action rules $\{A(x) \wedge B(x) \mapsto \alpha(x), C(x) \wedge C(y) \mapsto \beta(x, y)\}$. The refined dataflow graph of D is shown in Figure 3. It is easy to see that D is *(i)* structural state-bounded, *(ii)* not safe-GR^{++}-acyclic, but *(iii)* stratified-GR^{++}-acyclic.

For *(i)*, observe that after execution of action α it is only possible to execute action β since the resulting state will not have any atom with the relation name A, which is needed for execution of α. Moreover, after executing β, none of the actions can be fired, since the resulting state will not have any atom with the relation name B, which is needed for execution of α, and not any atom with the relation name of C which is needed for execution of β. For *(ii)*, consult the safe-refined dataflow graph of Figure 3. As a witness of the generate-recall cycle, it is possible to select π_1 as the cycle generated by the top edge from $\langle C, 1\rangle$ and $\langle A, 1\rangle$ and the edge from $\langle A, 1\rangle$ to $\langle C, 1\rangle$, π_2 as the second edge from $\langle C, 1\rangle$ to $\langle A, 1\rangle$, and π_3 as the cycle generated by the edge from $\langle A, 1\rangle$ to $\langle C, 1\rangle$ and the second edge from $\langle C, 1\rangle$ to $\langle A, 1\rangle$.

For *(iii)*, $\text{INPUT}(\varrho, \alpha) = \{\{A, B\}\}$, $\text{INPUT}(\varrho, \beta) = \{\{C\}\}$, $\text{OUTPUT}(\alpha) = \{B, C\}$, $\text{OUTPUT}(\beta) = \{A\}$. Consequently, $\alpha \not< \alpha$, $\alpha < \beta$, $\beta \not< \alpha$, and $\beta \not< \beta$. The controlflow graph has only one edge from α to β. Since there is no strongly connected component in the controlflow graph, D trivially satisfies the condition of stratified-GR^{++}-acyclicity. ■

Theorem 15. *Stratified-GR^{++}-acyclicity is a strict relaxation of safe-GR^{++}-acyclicity that guarantees structural state-boundedness.*

5.4 $\exists$- Acyclicity Relaxations

All the syntactic restrictions presented so far are based on finding cycles in the corresponding dataflow graphs, as part of generate-recall pairs. The graph edges are constructed conservatively, to reflect possible flow of data between relations of the data layer. In this section, we observe that edge construction can be over-conservative, and we show a technique to safely consider fewer edges. This is clearly beneficial, as removing edges reduces the opportunity for forming cycles.

The technique is based on a counterintuitive observation regarding overestimations of an original DCDS D. These are rewritings O of D whose effects generate a superset of the data generated by the effects of D. It turns out that it is possible for the data flow graph of overestimate O to have a subset of the edges of the data flow graph of D. Since structural state-boundedness of O guarantees state-boundedness of D, checking the data flow graph of O leads to a relaxed syntactic restriction for state-boundedness.

One situation in which overestimation of the system results in smaller dataflow graphs is when a nullary service call occurs in heads of the DCDS. In the definition of GR^{++}-acyclicity in Section 5.1, for each nullary service call in the head of an effect we add a special edge from all the positions of the body of the effect to the position in which the service call occurs. Consider an effect $e : A(x) \wedge B(y) \rightsquigarrow C(f())$. In the definition of GR^{++}-acyclicity there are special edges from $\langle A, 1\rangle$ and $\langle B, 1\rangle$ to $\langle C, 1\rangle$ in the dataflow graph. However, since we are looking for sufficient conditions for state-boundedness, we can overestimate the result of e by replacing the effect e with $e_1 : A(x) \wedge B(y) \rightsquigarrow C(f(x))$ or $e_2 : A(x) \wedge B(y) \rightsquigarrow C(f(y))$. Notice although the resulting system is an overestimation of the original system, the resulting dataflow graph has less edges, and consequently, it is more likely to satisfy the condition of GR^{++}-acyclicity.

A similar situation holds for repeated positions with the same variables in the body of the effect. Lets assume to have an effect $f : A(x) \wedge B(x) \rightsquigarrow C(x)$. It is possible to over estimate f with $f1 : A(x) \wedge \exists y B(y) \rightsquigarrow C(x)$ or $f2 : \exists y A(y) \wedge B(x) \rightsquigarrow C(x)$. The resulting dataflow graphs will have less edges, and consequently, it is more likely to satisfy the condition of GR^{++}-acyclicity.

More formally, given a DCDS D, $\text{OVEREST}(D)$ is a set of DCDSs obtained from D by applying the following rules for each of the possible options:

- Replace the nullary service calls in the head of the effects with a unary service call using as variable, one of the variables of the body of the effect. Notice that for each choice of variables for unary service calls, there will be one DCDS in $\text{OVEREST}(D)$.
- Among multiple occurrences of one variable in different positions of the body of an effect, keep one of them, and rename and existentially quantify the others. Also notice that for each possible selection option, there will be a DCDS in $\text{OVEREST}(D)$.

We say that D is $\exists GR^{++}$-acyclic, if there exists a DCDS $D' \in \text{OVEREST}(D)$, such that D' is GR^{++}-acyclic.

Example 4. Consider a DCDS D with one actions $\alpha = \{A(x) \wedge B(x) \wedge B(y) \rightsquigarrow \{A(f()), B(g())\}$. The dataflow graph of D is depicted in Figure 4. It is easy to see that D is *(i)* state-bounded, and *(ii)* not stratified-GR^{++}-acyclic, but *(iii)* $\exists GR^{++}$-acyclic. For *(i)*, observe that $|A_1|_{I'} \leq |A_1|_I$, and $|B_1|_{I'} \leq |B_1|_I$. For *(ii)*, consult the safe-refined dataflow graph of Figure 4. As a witness of the generate-recall cycle consider π_1 as the self loop of $\langle A, 1\rangle$, π_2 as the edge from $\langle A, 1\rangle$ to $\langle B, 1\rangle$, and π_3 as the self loop of $\langle B, 1\rangle$.

For *(iii)*, considering the two options for selecting a position among the positions with the same variable in body of the only effect of α, and four options for overestimating

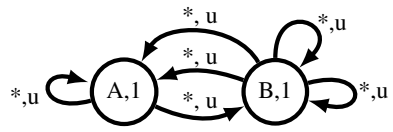

Figure 4: Safe-refined dataflow graph of Example 4

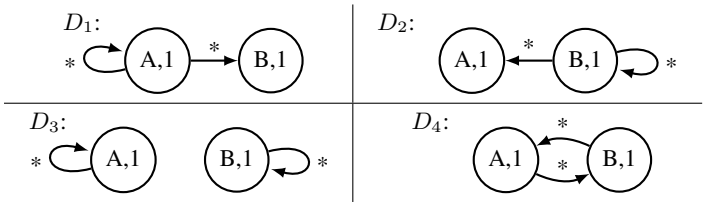

Figure 5: Refined dataflow graphs regarding $\exists GR^{++}$-acyclicity of Example 4

the two nullary service calls in the head of the effect of α to unary service calls using the variables x and y of the body, $\text{OVEREST}(D)$ contains eight different overestimations. Here we represent four of these overestimations $D_1, \ldots, D_4$, each of them with one action, namely, $\alpha_1, \ldots, \alpha_4$ as follows:

$$\alpha_1 : \{A(x) \wedge \exists z\ B(z) \wedge B(y) \rightsquigarrow \{A(f(x), B(g(x))\}\}$$
$$\alpha_2 : \{\exists z\ A(z) \wedge B(x) \wedge B(y) \rightsquigarrow \{A(f(x), B(g(x))\}\}$$
$$\alpha_3 : \{A(x) \wedge \exists z\ B(z) \wedge B(y) \rightsquigarrow \{A(f(x), B(g(y))\}\}$$
$$\alpha_4 : \{A(x) \wedge \exists z\ B(z) \wedge B(y) \rightsquigarrow \{A(f(y), B(g(x))\}\}$$

For *(iii)*, consult the refined dataflow graphs of $D_1, \ldots, D_4$ in Figure 5. For $\exists GR^{++}$-acyclicity, it is enough to have one of the overestimations GR^{++}-acyclic. In this example, all four represented overestimations are GR^{++}-acyclic. ■

Theorem 16. *$\exists GR^{++}$-acyclicity is a strict relaxation of GR^{++}-acyclicity that guarantees state-boundedness.*

A Hierarchy of Acyclicity Classes. The idea for $\exists GR^{++}$-acyclicity applies to generate existential relaxations of safe-GR^{++}-acyclicity and stratified-GR^{++}-acyclicity, called safe-$\exists GR^{++}$-acyclicity and stratified-$\exists GR^{++}$-acyclicity. Figure 6 summarizes our results. Note that stratified-$\exists GR^{++}$-acyclicity is the most relaxed condition, and GR^{+} is the most over-conservative. Moreover, we can prove that safe-GR^{++}-acyclicity and $\exists GR^{++}$-acyclicity are incomparable.

All acyclicity variants we have introduced are purely syntactic notions. The non-existential ones can be checked in coNP since the various versions of dataflow graphs have size polynomial in the size of the process layer specification, and, if there is a violation of acyclicity, then we can show that there is a polynomial-sized one. For existential relaxations, complexity jumps to Σ_2^p, due to the need of guessing the member of $\text{OVEREST}(D)$ to which the coNP check applies.

6 Conclusion

State-boundedness has recently gained a central place in verification of data-aware dynamic systems. This motivated us to study whether there are interesting decidable classes of such systems for which state-boundedness is decidable. Thanks to

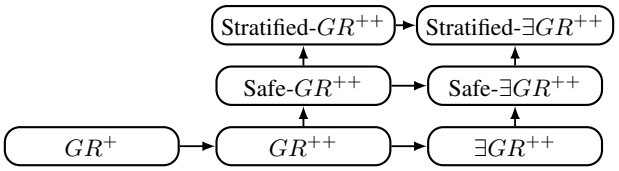

Figure 6: Syntactic restrictions for structural state-boundedness. Arrows show strict containment of classes. Unconnected classes are incomparable.

novel connections with Petri nets, we answer the question in the negative, but at the same time paving the way towards an entire field dedicated to studying sufficient, checkable conditions for state-boundedness. Inspired from sufficient conditions for chase termination in database theory, we show that there are many non-trivial classes of DCDSs that guarantee state-boundedness. The flourishing literature on fine-grained syntactic conditions for chase termination, not considered in this work, points to promising future research.

Acknowledgments. This research has been partially supported by the EU IP project Optique (Scalable End-user Access to Big Data), grant agreement n. FP7-318338.

References

Bagheri Hariri, B.; Calvanese, D.; Montali, M.; De Giacomo, G.; De Masellis, R.; and Felli, P. 2013a. Description logic Knowledge and Action Bases. *Journal of Artificial Intelligence Research* 46.

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

Belardinelli, F.; Lomuscio, A.; and Patrizi, F. 2012. An abstraction technique for the verification of artifact-centric systems. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*. AAAI Press.

Calì, A.; Gottlob, G.; and Kifer, M. 2013. Taming the infinite chase: Query answering under expressive relational constraints. *Journal of Artificial Intelligence Research* 48:115–174.

Calvanese, D.; De Giacomo, G.; Lembo, D.; Lenzerini, M.; and Rosati, R. 2007. Tractable reasoning and efficient query answering in description logics: The *DL-Lite* family. *J. of Automated Reasoning* 39(3):385–429.

Calvanese, D.; De Giacomo, G.; and Montali, M. 2013. Foundations of data-aware process analysis: A database theory perspective. In *Proc. of the 32th Symp. on Principles of Database Systems (PODS)*, 1–12. ACM Press.

De Giacomo, G.; Lesperance, Y.; and Patrizi, F. 2012. Bounded situation calculus action theories and decidable verification. In *Proc. of the 13th Int. Conf. on the Principles of Knowledge Representation and Reasoning (KR)*, 467–477.

Deutsch, A.; Hull, R.; Patrizi, F.; and Vianu, V. 2009. Automatic verification of data-centric business processes. In *Proc. of the 12th Int. Conf. on Database Theory (ICDT)*, 252–267.

Deutsch, A.; Nash, A.; and Remmel, J. B. 2008. The chase revisited. In *Proc. of the 27th Symp. on Principles of Database Systems (PODS)*.

Dufourd, C.; Finkel, A.; and Schnoebelen, P. 1998. Reset nets between decidability and undecidability. In *Proc. of the 25th Coll. on Automata, Languages and Programming (ICALP)*, volume 1443, 103–115.

Dufourd, C.; Jancar, P.; and Schnoebelen, P. 1999. Boundedness of reset p/t nets. In *Proc. of the 26th Coll. on Automata, Languages and Programming (ICALP)*, volume 1644 of *Lecture Notes in Computer Science*, 301–310. Springer.

Dumas, M. 2011. On the convergence of data and process engineering. In *Proc. of ADBIS*, volume 6909 of *Lecture Notes in Computer Science*, 19–26. Springer.

Esparza, J., and Nielsen, M. 1994. Decidability issues for petri nets - a survey. *Bulletin of the EATCS* 52:244–262.

Fagin, R.; Kolaitis, P. G.; Miller, R. J.; and Popa, L. 2005. Data exchange: Semantics and query answering. *Theoretical Computer Science* 336(1):89–124.

Hull, R. 2008. Artifact-centric business process models: Brief survey of research results and challenges. In *Proc. of the On the Move Confederated Int. Conf. (OTM 2008)*, volume 5332 of *Lecture Notes in Computer Science*, 1152–1163. Springer.

Mayr, R. 2003. Undecidable problems in unreliable computations. *Theoretical Computer Science* 297(1-3):337–354.

Meier, M.; Schmidt, M.; and Lausen, G. 2009. On chase termination beyond stratification. *PVLDB* 2(1).

Rogers, H. 1967. *Theory of recursive functions and effective computability*. McGraw-Hill.

Solomakhin, D.; Montali, M.; Tessaris, S.; and De Masellis, R. 2013. Verification of artifact-centric systems: Decidability and modeling issues. In *Proc. of the 11th Int. Conf. on Service Oriented Computing (ICSOC)*.

Vianu, V. 2009. Automatic verification of database-driven systems: a new frontier. In *Proc. of the 12th Int. Conf. on Database Theory (ICDT)*, 1–13.