Verification of Relational Data-Centric Dynamic Systems with External Services*

Babak Bagheri Hariri
Diego Calvanese
Marco Montali
Free Univ. of Bozen/Bolzano
lastname@inf.unibz.it

Giuseppe De Giacomo
Sapienza Università di Roma
degiacomo@dis.uniroma1.it

Alin Deutsch
UC San Diego
deutsch@cs.ucsd.edu

ABSTRACT

Data-centric dynamic systems are systems where both the process controlling the dynamics and the manipulation of data are equally central. We study verification of (first-order) μ-calculus variants over *relational data-centric dynamic systems*, where data are maintained in a relational database, and the process is described in terms of atomic actions that evolve the database. Action execution may involve calls to external services, thus inserting fresh data into the system. As a result such systems are infinite-state. We show that verification is undecidable in general, and we isolate notable cases where decidability is achieved. Specifically we start by considering service calls that return values deterministically (depending only on passed parameters). We show that in a μ-calculus variant that preserves knowledge of objects appeared along a run we get decidability under the assumption that the fresh data introduced along a run are bounded, though they might not be bounded in the overall system. In fact we tie such a result to a notion related to weak acyclicity studied in data exchange. Then, we move to nondeterministic services and we investigate decidability under the assumption that knowledge of objects is preserved only if they are continuously present. We show that if infinitely many values occur in a run but do not accumulate in the same state, then we get again decidability. We give syntactic conditions to avoid this accumulation through the novel notion of "generate-recall acyclicity", which ensures that every service call activation generates new values that cannot be accumulated indefinitely.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification
TERMS(Verification)

Keywords

Business artifacts, data-centric processes, first-order temporal logics.

*This research has been partially supported by the EU under the ICT Collaborative Project ACSI (Artifact-Centric Service Interoperation), grant agreement n. FP7-257593, and by the NSF under grant IIS-0916515.

PODS'13, June 22–27, 2013, New York, New York, USA.

1. INTRODUCTION

Business process management is central to the operation of organizations in various domains, ranging from business to governmental, scientific, and beyond. Business process specification frameworks have recently evolved from the traditional process-centric approach towards data-awareness. Process-centric formalisms focus on control flow while under-specifying the underlying data and its manipulations by the process tasks, often abstracting them away completely. In contrast, data-aware formalisms treat data as first-class citizens [37, 30, 19, 16, 40, 1]. The holistic view of data and processes together promises to avoid the notorious discrepancy between data modeling and process modeling of more traditional approaches that consider these two aspects separately [9]. In particular, this separation precludes the development of data-aware automatic tools for formal verification, i.e., static analysis and run-time monitoring. Such tools are desperately needed given the complexity of modern business processes, much of which is due to subtle interactions between business process tasks and data.

A notable exponent of the data-aware class of specification frameworks is the *artifact-centric model* pioneered in [37], deployed by IBM in commercial products and consulting services, and further studied in a line of follow-up works [8, 28, 29, 9, 32, 23, 20, 21]. Business artifacts (or simply "artifacts") model key business-relevant entities, updated by a set of business process tasks (actions). This modeling approach has been successfully deployed in practice, yielding proven savings when performing business process transformations [8] to expand and/or streamline the process.

Data-aware processes deeply challenge formal verification by requiring simultaneous attention to both data and process: on the one hand they deal with full-fledged processes and require analysis in terms of sophisticated temporal properties [18]; on the other hand, the presence of possibly unbounded data makes the usual analysis based on finite-state model checking impossible in general, since, when data is taken into account, the system becomes infinite-state.

In this work we focus on data-aware static verification, selecting the artifact-centric model as a natural vehicle for our investigation due to its practical relevance. Given the family of variations on this model found in the literature, for the sake of a uniform terminology we introduce our own pristine formalization, which captures the artifact-centric dialects in [7, 24, 25, 23, 21]. We call our business process formalism "Data-Centric Dynamic Systems" (DCDSs). The correspondence between DCDSs and the family of artifact models is discussed in Sections 6 and 7. DCDSs comprise *(i)* a *data layer*, which holds the relevant information to be manipulated by the system and technically can be seen as a *relational* database, and *(ii)* a *process layer* formed by invokable *(atomic) actions* and a process based on them. Such a process characterizes the dynamic behavior of the system. Executing an action has effects on the data manip-

ulated by the system, on the process state, and on the information exchanged with the external world.

Our setting is in line with recent industrial artifact model proposals [19] and research papers [7, 24, 25, 23] and it subsumes particular case variations in which the artifact is a record, modeled as a single-tuple database [20]. The execution of actions may involve calls to external services, providing fresh data inserted into the system. As a result such systems are infinite-state in general.

As verification formalism, we adopt a FO variant of μ-calculus [34, 38, 26, 14]. μ-calculus is well known to be more expressive than virtually all temporal logics used in verification, including widely adopted logics such as CTL, LTL, and CTL*. Our variant of μ-calculus is based on first-order queries over data in the states of the DCDS, and allows for first-order quantification across states (within and across runs), though in a controlled way. No limitations whatsoever are instead put on the fixpoint formulae, which are the key element of the μ-calculus.

In particular we consider two FO variations of μ-calculus. The first, called $\mu\mathcal{L}_A$, requires that first-order quantification across states is always bounded to the active domain of the state where the quantification is evaluated. This quantification mechanism indirectly preserves, at any point, knowledge of objects that appeared in the history so far, even if they disappeared in the meantime. The second, called $\mu\mathcal{L}_P$, further restricts the first-order quantification in $\mu\mathcal{L}_A$ by stating that only quantified objects that are still present in the current domain remain of interest as we move from one state to the next. That is, knowledge of objects is preserved only if they are continuously present. We define novel notions of bisimulation that characterize the forms of quantification used in the two logics.

Verification of data-aware processes, including artifact systems and DCDSs, is undecidable even for very simple system specifications and propositional CTL/LTL properties [7, 25, 23, 17]. However, we isolate two notable sufficient conditions over DCDSs, which respectively guarantee decidability of full $\mu\mathcal{L}_A$ and $\mu\mathcal{L}_P$ verification under specific assumptions about the external services.

Specifically we start by considering service calls that return values deterministically (depending only on passed parameters). We show that verification of $\mu\mathcal{L}_A$ properties is decidable under the assumption that the cardinality of fresh data introduced along each run is bounded (*run-bounded* DCDSs), though it need not be bounded across runs. Decidability is not obvious, since the logic permits quantification over values occurring across (potentially infinitely many) branching run continuations. Run-boundedness is an undecidable semantic property for which we propose a sufficient syntactic condition related to the notion of weak acyclicity studied in data exchange [27]. Then, we move to nondeterministic services where same-argument calls possibly return different values at different time moments. To exploit the results on run-bounded DCDSs in this case we would have to limit the number of service calls that can be invoked during the execution, which would be a too restrictive condition on the form of DCDSs. We show that if infinitely many values occur in a run but do not accumulate in the same state (our system is then called *state-bounded*) then $\mu\mathcal{L}_P$ verification is decidable (while $\mu\mathcal{L}_A$ is not). This is remarkable, since when compared to run-boundedness, state-boundedness permits an additional kind of data unboundedness (*within* the run, as opposed to only *across* runs). State-boundedness is also an undecidable semantic property, for which we provide a novel sufficient syntactic condition called "generate-recall acyclicity".

The decidability results come with an EXPTIME upper bound on the size of the initial database of the DCDS data layer. This is in line with previously known complexity bounds on systems that can be seen as special cases of our framework [7, 24, 25, 23, 17]. While at a first sight this seems an obstacle for practical verification, we observe that when DCDSs represent artifacts, which is our main use case, verification is affected only by the specific data needed to progress the artifacts along their lifecycle (process). The size of such data is in practice small when compared to the size of the entire data layer. Verification of data aware processes according to algorithms exponential in the initial state has already been successfully implemented in systems such as [24]. This is a strong indication of feasibility potential even in the more general setting presented here.

The rest of the paper is organized as follows. Section 2 introduces DCDSs. Section 3 introduces verification of DCDSs and the two variants of μ-calculus that we consider. Section 4 focusses the analysis of DCDSs under the assumption that external service calls behave deterministically. Section 5 considers the case in which external service calls behave nondeterministically. Section 6 discusses the various notions introduced. Section 7 reports on related work. Finally, Section 8 concludes the paper.

An extended version of this paper with full proofs is available [4].

2. DATA-CENTRIC DYNAMIC SYSTEMS

We base our investigation on a model called *(relational) data-centric dynamic system*, or simply DCDS, which can be seen as a pristine version of several proposals in the literature [7, 24, 25, 23], and is in particular equivalent in expressive power to the most expressive artifact model variations, such as [21] (see Section 6).

A DCDS is a pair $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ formed by two interacting layers: a *data layer* $\mathcal{D}$ and a *process layer* $\mathcal{P}$ over $\mathcal{D}$. Intuitively, the data layer keeps all the data of interest, while the process layer modifies and evolves such data. We keep the structure of both layers to the minimum, in particular we do not distinguish between various possible components providing the data, nor those providing the subprocesses running concurrently.

Data Layer. The data layer represents the information of interest in our application. It is constituted by a relational schema $\mathcal{R}$ equipped with equality constraints[1] $\mathcal{E}$, e.g., to state keys of relations, and an initial database instance $\mathcal{I}_0$, which conforms to the relational schema and the equality constraints. The values stored in this database belong to a countably infinite domain $\mathcal{C}$. The elements of this domain are treated as constants, interpreted as themselves, blurring the distinction between constants and values. We will use the two terms interchangeably.

Given a database instance $\mathcal{I}$, its active domain $\text{ADOM}(\mathcal{I})$ is the subset of $\mathcal{C}$ such that $c \in \text{ADOM}(\mathcal{I})$ if and only if c occurs in $\mathcal{I}$.

Formally, a *data layer* is a tuple $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ where:

- $\mathcal{R} = \{R_1, \ldots, R_n\}$ is a database schema, constituted by a finite set of relation schemas;
- $\mathcal{E}$ is a finite set $\{\mathcal{E}_1, \ldots, \mathcal{E}_m\}$ of equality constraints. Each $\mathcal{E}_i$ has the form $Q_i \rightarrow \bigwedge_{j=1,\ldots,k} z_{ij} = y_{ij}$, where Q_i is a domain independent FO query over $\mathcal{R}$, possibly using constants from $\text{ADOM}(\mathcal{I}_0)$, whose free variables are $\vec{x}$, and z_{ij} and y_{ij} are either variables in $\vec{x}$ or constants in $\text{ADOM}(\mathcal{I}_0)$.[2]
- $\mathcal{I}_0$ is a database instance that represents the initial state of the data layer, which conforms to the schema $\mathcal{R}$ and *satisfies* the constraints $\mathcal{E}$: namely, for each constraint $Q_i \rightarrow \bigwedge_{j=1,\ldots,k} z_{ij}{=}y_{ij}$ and for each tuple (i.e., substitution for the free variables) $\theta \in ans(Q_i, \mathcal{I})$, it holds that $z_{ij}\theta = y_{ij}\theta$.[3]

[1] Other kinds of constraints can also be included without affecting the results reported here (cf. Section 6).

[2] For convenience, and without loss of generality, we assume that all constants used inside formulae appear in $\mathcal{I}_0$.

[3] We use the notation $t\theta$ (resp., $\varphi\theta$) to denote the term (resp., the formula) obtained by applying the substitution θ to t (resp., φ).

Process Layer. The process layer constitutes the progression mechanism for the DCDS. We assume that at every time the current instance of the data layer can be arbitrarily queried, and can be updated through action executions, possibly involving external service calls to get new values from the environment. Hence, the process layer is composed of three main notions: *actions*, which are the atomic update steps on the data layer; *external services*, which can be called during the execution of actions; and *processes*, which are essentially nondeterministic programs that use actions as atomic instructions. While we require the execution of actions to be sequential, we do not impose any such constraints on processes, which in principle can be formed by several concurrent branches, including fork, join, and so on. Concurrency is to be interpreted by interleaving and hence reduced to nondeterminism, as often done in formal verification [5, 26]. There can be many ways to provide the control flow specification for processes. Here we adopt a simple rule-based mechanism, but our results can be immediately generalized to processes whose control flow is finite-state. Observe that this does not imply that the transition system associated to a process over the data layer is finite-state as well, since the data manipulated in the data layer may grow over time in an unbounded way.

Formally, a process layer $\mathcal{P}$ over a data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$, is a tuple $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$ where:

- $\mathcal{F}$ is a finite set of *functions*, each representing the interface to an *external service*. Such services can be called, and as a result the function is activated and the answer is produced. How the result is actually computed is *unknown* to the DCDS since the services are external.
- $\mathcal{A}$ is a finite set of *actions*, whose execution updates the data layer, and may involve external service calls.
- 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.

The crucial aspect of actions is how they affect the data layer. Actions query the current state of the data layer and use the results of such queries, together with the data returned from the external service calls, to instantiate the data layer in the new state. To specify the action effects, we resort to rules that resemble tuple generating dependencies (TGDs) [2], except that we allow for negation when querying the database and we use results of service calls instead of labeled nulls. Note that negation is key to capturing "if-then-else" style business rules, while service calls are used for modeling the input of new data from the external environment.

Formally, an *action* $\alpha \in \mathcal{A}$ 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 α 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 $\text{ADOM}(\mathcal{I}_0)$, 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[4].
- E_i is the effect, i.e., a set of facts for $\mathcal{R}$, which includes as terms: terms in $\text{ADOM}(\mathcal{I}_0)$, 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 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 $\mathcal{A}$ 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 in $\text{ADOM}(\mathcal{I}_0)$.

Semantics via Transition System. The semantics of a DCDS is defined in terms of a possibly infinite transition system whose states are labeled by databases. Such a transition system represents all possible computations that the process layer can do on the data layer. A transition system Υ is a tuple $\langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$, where:

- Δ is a countably infinite set of values;
- $\mathcal{R}$ is a database schema;
- Σ is a set of states;
- $s_0 \in \Sigma$ is the initial state;
- db is a function that, given a state $s \in \Sigma$, returns the database of s, which is made up of values in Δ and conforms to $\mathcal{R}$;
- $\Rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation over states.

In order to precisely build the transition system associated to a DCDS, we need to better characterize the behavior of the external services, which are called in the effects of actions. This is done in Sections 4 and 5.

EXAMPLE 2.1. (Travel Reimbursement DCDS) We model the process of reimbursing travel expenses in a university, and the corresponding audit system, in two different subsystems. In particular, the first subsystem, called the *request system* manages the submission of reimbursement requests by an employee, and preliminary inspection and approval of the request by a *monitor* working in the accounting department. The log of accepted requests will be submitted to the second subsystem, the *audit system*, in which requests can be accumulated, and they can be checked for accuracy by calling external web services (for instance to obtain the exchange rate from foreign currency to USD on a past date, or to check that the employee actually was on the declared flight). Here we model selected parts of the request system (the full-fledged example, including the whole audit system, is developed in [4]).

A request is associated with the name of the employee and comprises information related to the corresponding flight and hotel costs. The monitor decides to accept or reject the request. In case of rejection, the employee needs to modify the information regarding hotel and flight. After the update by the employee, the monitor checks again the request, and the reject-check loop continues until the monitor accepts the request. After a request is accepted a log of the request is sent to the audit system, and the request system is ready to process the next travel request.

We model the request system by a DCDS $\mathcal{S}_R = \langle \mathcal{D}, \mathcal{P} \rangle$, where $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$, $\mathcal{I}_0$ contains the fact $\mathsf{Status}(\textit{'readyForRequest'})$ as well as the initial state of the $\mathsf{ApprHotel}$ relation, and $\mathcal{R}$ is a database schema including

- $\mathsf{Status} = \langle \mathsf{status} \rangle$, a unary relation that keeps the state of the request subsystem, and can take three different values: *'readyForRequest'*, *'readyToVerify'*, and *'readyToUpdate'*,
- $\mathsf{Travel} = \langle \mathsf{eName} \rangle$, holding the name of the employee;

Furthermore, given a FO query Q and a database instance $\mathcal{I}$, the *answer* $ans(Q, \mathcal{I})$ to Q over $\mathcal{I}$ is the set of assignments θ from the free variables of Q to $\text{ADOM}(\mathcal{I})$, such that $\mathcal{I} \models Q\theta$. We treat $Q\theta$ as a boolean query, and with some abuse of notation, we say $ans(Q\theta, \mathcal{I}) \equiv \mathsf{true}$ if and only if $\mathcal{I} \models Q\theta$.

[4]Note that while in principle we could replace $q_i^+ \wedge Q_i^-$ with any domain independent FO query, distinguishing between q_i^+ and Q_i^- gives us leverage (under the control of the designer) for singling out interesting syntactic conditions for decidability (see Sections 4.3 and 5.3).

- $\mathsf{Hotel} = \langle \mathsf{hName}, \mathsf{date}, \mathsf{price}, \mathsf{currency}, \mathsf{priceInUSD} \rangle$, holding the hotel cost information of the employee's travel, which might have been paid in some other currency than USD,
- $\mathsf{Flight} = \langle \mathsf{date}, \mathsf{fNum}, \mathsf{price}, \mathsf{currency}, \mathsf{priceInUSD} \rangle$, holding the flight cost information,
- $\mathsf{ApprHotel} = \langle \mathsf{hName} \rangle$, holding a list of approved hotels.

The process layer is defined as $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$, where $\mathcal{F}$ is a set of service calls, modeling an input of an external value by the employee for each attribute of the request. For instance, $\mathcal{F}$ includes INHNAME() for the hotel name, INHDATE() for the arrival date, etc. DECIDE() models the decision of the human monitor, returning *'accepted'* if the request is accepted, and *'readyToUpdate'* if the request needs to be updated by the employee.

The set $\mathcal{A}$ of actions includes *InitiateRequest*, *VerifyRequest*, *UpdateRequest*, and *AcceptRequest*. When a request is initiated (modeled by the action *InitiateRequest*), *(i)* the system changes state to "waiting for verification", *(ii)* a travel event is generated and the employee fills in his name, and *(iii)* the employee fills in hotel and flight information. Action *VerifyRequest* models the preliminary check by the monitor. Travel event, hotel, and flight information are copied unchanged to the next state, and this is necessary to preserve such information according to our semantics (see Sections 4 and 5). The system status is set as follows: if the hotel is on the approved list, then the request is automatically accepted. Otherwise, the request is handled by a human monitor, modeled by the non-deterministic service call DECIDE(). If the monitor rejects, she sets the next state to *'readyToUpdate'*, triggering the action *UpdateRequest*, which collects once again the hotel and flight information from the employee, moving the status to *'readyToVerify'*. Finally, action *AcceptRequest* returns the system in the state *'readyForRequest'*, in which it is ready to accept a new request. The condition-action rules in the set ϱ below guard the actions by the current system's state:

$$\begin{aligned} \mathsf{Status}(\textit{'readyForRequest'}) &\mapsto \textit{InitiateRequest} \\ \mathsf{Status}(\textit{'readyToVerify'}) &\mapsto \textit{VerifyRequest} \\ \mathsf{Status}(\textit{'readyToUpdate'}) &\mapsto \textit{UpdateRequest} \\ \mathsf{Status}(\textit{'accepted'}) &\mapsto \textit{AcceptRequest} \end{aligned}$$

We conclude the example by detailing *VerifyRequest*:

$$\begin{aligned} \mathsf{Hotel}(x_1,\ldots,x_5) \wedge \mathsf{ApprHotel}(x_1) &\rightsquigarrow \mathsf{Status}(\textit{'accepted'}) \\ \mathsf{Hotel}(x_1,\ldots,x_5) \wedge \neg\mathsf{ApprHotel}(x_1) &\rightsquigarrow \mathsf{Status}(\text{DECIDE}()) \\ \mathsf{Travel}(n) &\rightsquigarrow \mathsf{Travel}(n) \\ \mathsf{Hotel}(x_1,\ldots,x_5) &\rightsquigarrow \mathsf{Hotel}(x_1,\ldots,x_5) \\ \mathsf{Flight}(x_1,\ldots,x_5) &\rightsquigarrow \mathsf{Flight}(x_1,\ldots,x_5) \\ \mathsf{ApprHotel}(x) &\rightsquigarrow \mathsf{ApprHotel}(x) \end{aligned}$$ ■

3. VERIFICATION

To specify dynamic properties over a DCDS, we use μ-calculus [26, 39, 14], one of the most powerful temporal logics for which model checking has been investigated in the finite-state setting. Indeed, such a logic is able to express both linear time logics such as LTL and PSL, and branching time logics such as CTL and CTL* [18]. The main characteristic of μ-calculus is the ability of expressing directly least and greatest fixpoints of (predicate-transformer) operators formed using formulae relating the current state to the next one. By using such fixpoint constructs one can easily express sophisticated properties defined by induction or co-induction. This is the reason why virtually all logics used in verification are essentially fragments of μ-calculus. From a technical viewpoint, μ-calculus separates local properties, i.e., properties asserted on the current state or on states that are immediate successors of the current one, and properties that talk about states that are arbitrarily far away from the current one [14]. The latter are expressed using fixpoints.

In this work, we use a first-order extension of the μ-calculus [38], called $\mu\mathcal{L}$ and defined as follows:

$$\Phi ::= Q \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\Phi \mid \langle - \rangle \Phi \mid Z \mid \mu Z.\Phi$$

where Q is a possibly open FO query, and Z is a second order predicate variable (of arity 0). We make use of the following abbreviations: $\forall x.\Phi = \neg(\exists x.\neg\Phi)$, $\Phi_1 \vee \Phi_2 = \neg(\neg\Phi_1 \wedge \neg\Phi_2)$, $[-]\Phi = \neg\langle - \rangle\neg\Phi$, and $\nu Z.\Phi = \neg\mu Z.\neg\Phi[Z/\neg Z]$.

As usual in μ-calculus, formulae of the form $\mu Z.\Phi$ (and $\nu Z.\Phi$) must obey to the *syntactic monotonicity* of Φ w.r.t. Z, which states that every occurrence of the variable Z in Φ must be within the scope of an even number of negation symbols. This ensures that both $\mu Z.\Phi$ and $\nu Z.\Phi$ always exist.

Since $\mu\mathcal{L}$ also contains formulae with both individual and predicate free variables, given a transition system Υ, we introduce an individual variable valuation v, i.e., a mapping from individual variables x to Δ, and a predicate variable valuation V, i.e., a mapping from predicate variables Z to subsets of Σ. With these three notions in place, we assign meaning to formulae by associating to Υ, v, and V an *extension function* $(\cdot)^{\Upsilon}_{v,V}$, which maps formulae to subsets of Σ. Formally, the extension function $(\cdot)^{\Upsilon}_{v,V}$ is defined inductively as shown in Figure 1. When Φ is a closed formula, $(\Phi)^{\Upsilon}_{v,V}$ depends neither on v nor on V, and we denote the extension of Φ simply by $(\Phi)^{\Upsilon}$. We say that a closed formula Φ holds in a state $s \in \Sigma$ if $s \in (\Phi)^{\Upsilon}$. In this case, we write $\Upsilon, s \models \Phi$. We say that a closed formula Φ holds in Υ, denoted by $\Upsilon \models \Phi$, if $\Upsilon, s_0 \models \Phi$, where s_0 is the initial state of Υ. We call *model checking* verifying whether $\Upsilon \models \Phi$ holds. In particular, we are interested in formally verifying properties of a DCDS. Given the transition system $\Upsilon_{\mathcal{S}}$ of a DCDS $\mathcal{S}$ and a $\mu\mathcal{L}$ dynamic property Φ,[5] we say that $\mathcal{S}$ *verifies* Φ if $\Upsilon_{\mathcal{S}} \models \Phi$.

EXAMPLE 3.1. It is easy to write $\mu\mathcal{L}$ formulae that express typical temporal properties such as:

- *liveness (on a run)*: there exists a run such that α eventually holds, i.e. $\mu Z.\alpha \vee \langle - \rangle Z$;
- *liveness (on all runs)*: eventually in the future α will hold, i.e., $\mu Z.\alpha \vee [-]Z$;
- *safety (on all runs)*: for all (future) situations α holds, i.e., $\nu Z.\alpha \wedge [-]Z$;
- *response (on all runs)*: always when α then eventually β, i.e., $\nu Z_1.(\alpha \rightarrow \mu Z_2.\beta \vee [-]Z_2) \wedge [-]Z_1$;
- *strong fairness (on a run)*: there exists a run where α is true infinitely often, i.e., $\nu X.\mu Y.(\alpha \wedge \langle - \rangle X) \vee \langle - \rangle Y$. ■

EXAMPLE 3.2. Consider the $\mu\mathcal{L}$ formula:

$$\exists x_1,\ldots,x_n.\bigwedge_{i\neq j} x_i \neq x_j \wedge \bigwedge_{i\in\{1,\ldots,n\}} \mu Z.(\mathit{Stud}(x_i) \vee \langle - \rangle Z)$$

It asserts that there are at least n distinct objects/values, each of which eventually denotes a student along some execution path. The formula does not imply that all of these students will be in the same state, nor that they will all occur in a single run. It only says that in the entire transition system there are (at least) n distinct students. ■

The challenging point is that $\Upsilon_{\mathcal{S}}$ is in general infinite-state, so we would like to devise a finite-state transition system to model check, which is a faithful abstraction of $\Upsilon_{\mathcal{S}}$ in the sense that it

[5] We remind the reader that, without loss of generality, we assume that all constants used inside formulae Φ appear in the initial database instance of the DCDS.

$$
\begin{aligned}
(Q)^{\Upsilon}_{v,V} &= \{s \in \Sigma \mid ans(Qv, db(s))\} \\
(\neg\Phi)^{\Upsilon}_{v,V} &= \Sigma - (\Phi)^{\Upsilon}_{v,V} \\
(\Phi_1 \wedge \Phi_2)^{\Upsilon}_{v,V} &= (\Phi_1)^{\Upsilon}_{v,V} \cap (\Phi_2)^{\Upsilon}_{v,V} \\
(\exists x.\Phi)^{\Upsilon}_{v,V} &= \{s \in \Sigma \mid \exists t.t \in \Delta \text{ and } s \in (\Phi)^{\Upsilon}_{v[x/t],V}\} \\
(\langle - \rangle \Phi)^{\Upsilon}_{v,V} &= \{s \in \Sigma \mid \exists s'.s \Rightarrow s' \text{ and } s' \in (\Phi)^{\Upsilon}_{v,V}\} \\
(Z)^{\Upsilon}_{v,V} &= V(Z) \\
(\mu Z.\Phi)^{\Upsilon}_{v,V} &= \bigcap\{\mathcal{S} \subseteq \Sigma \mid (\Phi)^{\Upsilon}_{v,V[Z/\mathcal{S}]} \subseteq \mathcal{S}\}
\end{aligned}
$$

Figure 1: Semantics of $\mu\mathcal{L}$.

preserves the truth value of all $\mu\mathcal{L}$ formulae. Unfortunately, this program is doomed if we insist on using full $\mu\mathcal{L}$ as the verification formalism. Indeed, there are $\mu\mathcal{L}$ formulae, such as the one shown in Example 3.2, that defeat any kind of finite-state abstraction (in the precise sense of Theorem 4.5). So next we introduce two interesting sublogics of $\mu\mathcal{L}$ that better serve our objective.

3.1 History-Preserving Mu-Calculus

The first fragment of $\mu\mathcal{L}$ that we consider is $\mu\mathcal{L}_A$, which is characterized by the assumption that quantification over objects is restricted to objects that are present in the current database. To enforce such a restriction, we introduce a special predicate $\text{LIVE}(x)$, which states that x belongs to the current active domain. The logic $\mu\mathcal{L}_A$ is defined as follows:

$$\Phi ::= Q \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\text{LIVE}(x) \wedge \Phi \mid \langle - \rangle \Phi \mid Z \mid \mu Z.\Phi$$

We make use of the usual abbreviations, including $\forall x.\text{LIVE}(x) \rightarrow \Phi = \neg(\exists x.\text{LIVE}(x) \wedge \neg\Phi)$. Formally, the extension function $(\cdot)^{\Upsilon}_{v,V}$ is defined inductively as in Figure 1, with the new special predicate $\text{LIVE}(x)$ interpreted as follows:

$$(\text{LIVE}(x))^{\Upsilon}_{v,V} = \{s \in \Sigma \mid x/d \in v \text{ implies } d \in \text{ADOM}(db(s))\}$$

EXAMPLE 3.3. $\mu\mathcal{L}_A$ requires the bindings of quantified variables to be live in the step when the quantification is evaluated. This can be done by using LIVE or simply by using any relation, such as $Stud$ and $Grad$ in the following formula:

$$\nu X.(\forall x.Stud(x) \rightarrow \mu Y.(\exists y.Grad(x,y) \vee \langle - \rangle Y) \wedge [-]X)$$

The formula states that, along every path, it is always true, for each student x, that there exists an evolution that eventually leads to the graduation of x (with some final mark y). ■

We are going to show that under suitable conditions we can get a faithful finite abstraction for a DCDS that preserves all formulae of $\mu\mathcal{L}_A$, and hence enables us to use standard model checking techniques. Towards this goal, we introduce a notion of bisimulation between two transition systems, which is suitable for the kind of transition systems we consider here. In particular, we have to take into account that the two transition systems are over different data domains, and hence we have to consider the correspondence between the data in the two transition systems and how such data evolve over time. To do so, we introduce the following notions.

Given two domains Δ_1 and Δ_2, a *partial bijection* h between Δ_1 and Δ_2 is a bijection between a subset of Δ_1 and Δ_2. Given a partial function $f : S \rightarrow S'$, we denote with $\text{DOM}(f)$ the domain of f, i.e., the set of elements in S on which f is defined, and with $\text{IM}(f)$ the image of f, i.e., the set of elements s' in S' such that $s' = f(s)$ for some $s \in S$. A partial bijection h' *extends* h if $\text{DOM}(h) \subseteq \text{DOM}(h')$ (or equivalently $\text{IM}(h) \subseteq \text{IM}(h')$) and $h'(x) = h(x)$ for all $x \in \text{DOM}(h)$ (or equivalently $h'^{-1}(y) = h^{-1}(y)$ for all $y \in \text{IM}(h)$). Let db_1 and db_2 be two databases over two domains Δ_1 and Δ_2 respectively, both conforming to the same schema $\mathcal{R}$. We say that a partial bijection h *induces an isomorphism* between db_1 and db_2 if $\text{ADOM}(db_1) \subseteq \text{DOM}(h)$, $\text{ADOM}(db_2) \subseteq \text{IM}(h)$, and h projected on $\text{ADOM}(db_1)$ is an isomorphism between db_1 and db_2.

Let $\Upsilon_1 = \langle \Delta_1, \mathcal{R}, \Sigma_1, s_{01}, db_1, \Rightarrow_1 \rangle$ and $\Upsilon_2 = \langle \Delta_2, \mathcal{R}, \Sigma_2, s_{02}, db_2, \Rightarrow_2 \rangle$ be transition systems and H the set of partial bijections between Δ_1 and Δ_2 that are the identity between $\text{ADOM}(db_1(s_{01}))$ and $\text{ADOM}(db_2(s_{02}))$. A *history preserving bisimulation* between Υ_1 and Υ_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times H \times \Sigma_2$ such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$ implies that:

1. h is a partial bijection between Δ_1 and Δ_2 that induces an isomorphism between $db_1(s_1)$ and $db_2(s_2)$;
2. for each s_1', if $s_1 \Rightarrow_1 s_1'$ then there is an s_2' with $s_2 \Rightarrow_2 s_2'$ and a bijection h' that extends h, such that $\langle s_1', h', s_2' \rangle \in \mathcal{B}$.
3. for each s_2', if $s_2 \Rightarrow_2 s_2'$ then there is an s_1' with $s_1 \Rightarrow_1 s_1'$ and a bijection h' that extends h, such that $\langle s_1', h', s_2' \rangle \in \mathcal{B}$.

A state $s_1 \in \Sigma_1$ is *history-preserving bisimilar* to $s_2 \in \Sigma_2$ *w.r.t. a partial bijection* h, written $s_1 \approx_h s_2$, if there exists a history-preserving bisimulation $\mathcal{B}$ between Υ_1 and Υ_2 such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$. A state $s_1 \in \Sigma_1$ is *history-preserving bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \approx s_2$, if there exists a partial bijection h and a history-preserving bisimulation $\mathcal{B}$ between Υ_1 and Υ_2 such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$. A transition system Υ_1 is *history-preserving bisimilar* to Υ_2, written $\Upsilon_1 \approx \Upsilon_2$, if $s_{01} \approx s_{02}$. The next theorem gives us the classical invariance result of μ-calculus w.r.t. bisimulation, in our setting.

THEOREM 3.1. *Consider two transition systems Υ_1 and Υ_2 such that $\Upsilon_1 \approx \Upsilon_2$. For every $\mu\mathcal{L}_A$ closed formula Φ, we have: $\Upsilon_1 \models \Phi$ if and only if $\Upsilon_2 \models \Phi$.*

3.2 Persistence-Preserving Mu-Calculus

The second fragment of $\mu\mathcal{L}$ that we consider is $\mu\mathcal{L}_P$, which further restricts $\mu\mathcal{L}_A$ by requiring that individuals over which we quantify must continuously persist along the system evolution for the quantification to take effect. In the following, we use $\text{LIVE}(x_1, \ldots, x_n)$ as an abbreviation for $\bigwedge_{i \in \{1,\ldots,n\}} \text{LIVE}(x_i)$.

The logic $\mu\mathcal{L}_P$ is defined as follows:

$$\begin{aligned}\Phi ::= Q \mid \neg\Phi \mid \Phi_1 \wedge \Phi_2 \mid \exists x.\text{LIVE}(x) \wedge \Phi \mid \text{LIVE}(\vec{x}) \wedge \langle - \rangle \Phi \mid \\ \text{LIVE}(\vec{x}) \wedge [-]\Phi \mid Z \mid \mu Z.\Phi\end{aligned}$$

where the following assumption holds: in $\text{LIVE}(\vec{x}) \wedge \langle - \rangle \Phi$ and $\text{LIVE}(\vec{x}) \wedge [-]\Phi$, the variables $\vec{x}$ are exactly the free variables of Φ, once we substitute to each bounded predicate variable Z in Φ its bounding formula $\mu Z.\Phi'$. We use the usual abbreviations, including: $\text{LIVE}(\vec{x}) \rightarrow \langle - \rangle \Phi = \neg(\text{LIVE}(\vec{x}) \wedge [-]\neg\Phi)$ and $\text{LIVE}(\vec{x}) \rightarrow [-]\Phi = \neg(\text{LIVE}(\vec{x}) \wedge \langle - \rangle \neg\Phi)$. Intuitively, the use of $\text{LIVE}(\cdot)$ in $\mu\mathcal{L}_P$ ensures that individuals are only considered if they persist along the system evolution, while the evaluation of a formula with individuals that are not present in the current database trivially leads to false or true.

EXAMPLE 3.4. Consider the $\mu\mathcal{L}_A$ formula in Example 3.3. $\mu\mathcal{L}_P$ can express two variations of such a formula. The first one,

$$\begin{aligned}\nu X.(\forall x.Stud(x) \rightarrow \\ \mu Y.(\exists y.Grad(x,y) \vee (\text{LIVE}(x) \wedge \langle - \rangle Y)) \wedge [-]X)\end{aligned}$$

strengthens the original formula stating that, along every path, it is always true, for each student x, that there exists an evolution in which x persists in the database until she eventually graduates (with some final mark y). The second variation,

$$\begin{aligned}\nu X.(\forall x.Stud(x) \rightarrow \\ \mu Y.(\exists y.Grad(x,y) \vee (\text{LIVE}(x) \rightarrow \langle - \rangle Y)) \wedge [-]X)\end{aligned}$$

weakens the original formula stating that, along every path, it is always true, for each student x, that there exists an evolution in which if x persists, she eventually graduates (with final mark y). ■

The bisimulation relation that captures $\mu\mathcal{L}_P$ is as follows. Let $\Upsilon_1 = \langle \Delta_1, \mathcal{R}, \Sigma_1, s_{01}, db_1, \Rightarrow_1 \rangle$ and $\Upsilon_2 = \langle \Delta_2, \mathcal{R}, \Sigma_2, s_{02}, db_2, \Rightarrow_2 \rangle$ be transition systems, and H the set of partial bijections between Δ_1 and Δ_2, which are the identity between $\text{ADOM}(db_1(s_{01}))$ and $\text{ADOM}(db_2(s_{02}))$. A *persistence-preserving bisimulation* between Υ_1 and Υ_2 is a relation $\mathcal{B} \subseteq \Sigma_1 \times H \times \Sigma_2$ such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$ implies that:

1. h is an isomorphism between $db_1(s_1)$ and $db_2(s_2)$;[6]
2. for each s_1', if $s_1 \Rightarrow_1 s_1'$ then there exists an s_2' with $s_2 \Rightarrow_2 s_2'$ and a bijection h' that extends $h|_{\text{ADOM}(db_1(s_1)) \cap \text{ADOM}(db_1(s_1'))}$, such that $\langle s_1', h', s_2' \rangle \in \mathcal{B}$;[7]
3. for each s_2', if $s_2 \Rightarrow_2 s_2'$ then there exists an s_1' with $s_1 \Rightarrow_1 s_1'$ and a bijection h' that extends $h|_{\text{ADOM}(db_1(s_1)) \cap \text{ADOM}(db_1(s_1'))}$, such that $\langle s_1', h', s_2' \rangle \in \mathcal{B}$.

We say that a state $s_1 \in \Sigma_1$ is *persistence-preserving bisimilar* to $s_2 \in \Sigma_2$ *w.r.t. a partial bijection* h, written $s_1 \sim_h s_2$, if there exists a persistence-preserving bisimulation $\mathcal{B}$ between Υ_1 and Υ_2 such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$. A state $s_1 \in \Sigma_1$ is *persistence-preserving bisimilar* to $s_2 \in \Sigma_2$, written $s_1 \sim s_2$, if there exists a partial bijection h and a persistence-preserving bisimulation $\mathcal{B}$ between Υ_1 and Υ_2 such that $\langle s_1, h, s_2 \rangle \in \mathcal{B}$. A transition system Υ_1 is *persistence-preserving bisimilar* to Υ_2, written $\Upsilon_1 \sim \Upsilon_2$, if $s_{01} \sim s_{02}$. The next theorem shows that $\mu\mathcal{L}_P$ enjoys invariance under this notion of bisimulation.

THEOREM 3.2. *Consider two transition systems Υ_1 and Υ_2 such that $\Upsilon_1 \sim \Upsilon_2$. Then for every $\mu\mathcal{L}_P$ closed formula Φ, we have that $\Upsilon_1 \models \Phi$ if and only if $\Upsilon_2 \models \Phi$.*

EXAMPLE 3.5. We illustrate some $\mu\mathcal{L}_P$ properties pertaining to the proper operation of the request system of Example 2.1. Recall that $\mu\mathcal{L}_P$ requires the bindings of quantified variables to be continuously live between the step when the quantification was evaluated and the step when the variable is used. This can be done by using LIVE or by using any relation, in our example Travel.

A property of interest is that once initiated, a request will eventually be decided by the monitor, and the decision can only be *'readyToUpdate'* or *'accepted'* (a liveness property):

$$\nu X.(\forall n.\mathsf{Travel}(n) \rightarrow \mu Y.(\mathsf{Status}(\textit{'readyToUpdate'}) \vee \mathsf{Status}(\textit{'accepted'}) \vee \mathsf{Travel}(n) \wedge [-]Y)) \wedge [-]X$$

Another property of interest is that if the flight cost is not specified, then the request is not accepted (a safety property):

$$\nu X.(\neg(\exists x_1, \ldots, x_4.\mathsf{Status}(\textit{'accepted'}) \wedge \mathsf{Flight}(x_1, x_2, \bot, x_3, x_4))) \wedge [-]X$$

where the special constant $\bot$ denotes a "non-specified" value (this need not be treated specially in the semantics, any value of the domain can be reserved for this purpose). ■

4. DETERMINISTIC SERVICES

We revisit the semantics of DCDSs, and analyze them under the assumption that external services behave deterministically. This means that the evaluation of functions $f \in \mathcal{F}$, representing the services in the process layer, is independent from the moment in which the function is called: if an external service is called twice with the same parameters, it must return the same value. So, for example, if the function invocation $f(a)$ returned b at a certain time, then in all successive moments the call $f(a)$ will return b again. In particular, *stateless* services can be modeled with deterministic service calls. In Example 2.1 (and its extended version in [4]) all web services invoked by the audit system (e.g., to determine the monetary conversion rate on a given date, or to check whether the employee took a specific flight), are inherently deterministic.

4.1 Semantics

We now define the transition system of a DCDS under the assumption of deterministic services. We call such a transition system "concrete" to avoid confusion with an "abstract" transition system that we are going to introduce for our verification technique.

Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$.

First we focus on what is needed to characterize the states of the concrete transition system. One such state obviously needs to maintain the current instance of the data layer. This instance is a database made up of values in $\mathcal{C}$, which conforms to the schema $\mathcal{R}$ and satisfies the equality constraints in $\mathcal{E}$. Together with the current instance, however, we also need to remember all answers we had so far when calling the external services.

To meet the requirement that service calls behave deterministically, the states of the transition system keep track of all results of the service calls made so far, in the form of equalities between Skolem terms (involving functions in $\mathcal{F}$ and arguments in $\mathcal{C}$) and returned values in $\mathcal{C}$.[8] More precisely, we define the set of (Skolem terms representing) service calls as $\mathbb{SC} = \{f(v_1, \ldots, v_n) \mid f/n \in \mathcal{F} \text{ and } \{v_1, \ldots, v_n\} \subseteq \mathcal{C}\}$, where f/n stands for a function f of arity n. Then we introduce a *service call map*, which is a partial function $\mathcal{M} : \mathbb{SC} \rightarrow \mathcal{C}$. Now we are ready to formally define states of the concrete transition system. A *(concrete) state* is a pair $\langle \mathcal{I}, \mathcal{M} \rangle$, where $\mathcal{I}$ is a relational instance of $\mathcal{R}$ over $\mathcal{C}$ satisfying each equality constraint in $\mathcal{E}$, and $\mathcal{M}$ is a service call map. The *initial concrete state* is $\langle \mathcal{I}_0, \emptyset \rangle$.

Next we look at the result of executing an action in a state. Let α be an action in $\mathcal{A}$ with parameters $p_1, \ldots, p_m$, $\alpha(p_1, \ldots, p_m) : \{e_1, \ldots, e_m\}$ where $e_i = q_i^+ \wedge Q_i^- \rightsquigarrow E_i$. Let σ be a substitution for $p_1, \ldots, p_m$ with values taken from $\mathcal{C}$. We say that σ is *legal* for α in state $\langle \mathcal{I}, \mathcal{M} \rangle$ if there exists a condition-action rule $Q \mapsto \alpha$ in ϱ such that $\langle p_1, \ldots, p_m \rangle \sigma \in ans(Q, \mathcal{I})$.

We denote with $\text{DO}(\mathcal{I}, \alpha, \sigma)$ the instance obtained by evaluating the effects of action α with parameters σ on instance $\mathcal{I}$, i.e.:

$$\text{DO}(\mathcal{I}, \alpha, \sigma) = \bigcup_{q_i^+ \wedge Q_i^- \rightsquigarrow E_i \in \text{EFFECT}(\alpha)} \bigcup_{\theta \in ans((q_i^+ \wedge Q_i^-)\sigma, \mathcal{I})} E_i \sigma \theta$$

Intuitively, the returned instance is the union of the results of applying the effects specifications $\text{EFFECT}(\alpha)$, where the result of each effect specification $q_i^+ \wedge Q_i^- \rightsquigarrow E_i$ is, in turn, the set of facts $E_i\sigma\theta$ obtained from $E_i\sigma$ grounded on all the assignments θ that satisfy the query $q_i^+ \wedge Q_i^-$ over $\mathcal{I}$.

[6] Notice that this implies $\text{DOM}(h) = \text{ADOM}(db_1(s_1))$ and $\text{IM}(h) = \text{ADOM}(db_2(s_2))$.

[7] Given a set D, we denote by $f|_D$ the *restriction* of f to D, i.e., $\text{DOM}(f|_D) = \text{DOM}(f) \cap D$, and $f|_D(x) = f(x)$ for every $x \in \text{DOM}(f) \cap D$.

[8] Notice that we assume no knowledge of the specific functions adopted by the external services, and we simply assume that such functions return some value from $\mathcal{C}$. Services returning values from an enumerated subset of $\mathcal{C}$, such as DECIDE() in Ex. 2.1, are syntactic sugar that we can simulate in our model. We are going to have different executions of the system corresponding to each way to assign values to the Skolem terms representing the service calls.

DO() generates an instance over values from $\mathcal{C}$ and over Skolem terms, which model service calls. For any such instance $\bar{\mathcal{I}}$, we denote with $\text{CALLS}(\bar{\mathcal{I}})$ the set of calls it contains. For a given set $D \subseteq \mathcal{C}$, we denote with $\text{EVALS}_D(\mathcal{I}, \alpha, \sigma)$ the set of substitutions that replace all service calls in $\text{DO}(\mathcal{I}, \alpha, \sigma)$ with values in D,

$$\text{EVALS}_D(\mathcal{I}, \alpha, \sigma) = \{\theta \mid \theta \text{ is a total function } \theta : \text{CALLS}(\text{DO}(\mathcal{I}, \alpha, \sigma)) \to D\}.$$

Each substitution in $\text{EVALS}_D(\mathcal{I}, \alpha, \sigma)$ models the simultaneous evaluation of all service calls, which replaces the calls with results selected arbitrarily from D. In the following, we refer to these substitutions as *evaluations*.

Concrete action execution. To capture what happens when α is executed in a state using a substitution σ for its parameters, we introduce a transition relation $\text{D-EXEC}_\mathcal{S}$ between states, called *concrete execution* of $\alpha\sigma$, such that $\langle\langle \mathcal{I}, \mathcal{M}\rangle, \alpha\sigma, \langle \mathcal{I}', \mathcal{M}'\rangle\rangle \in \text{D-EXEC}_\mathcal{S}$ if the following holds:

1. σ is a legal parameter assignment for α in state $\langle \mathcal{I}, \mathcal{M}\rangle$,
2. there exists $\theta \in \text{EVALS}_\mathcal{C}(\mathcal{I}, \alpha, \sigma)$ that is compatible with $\mathcal{M}$ (i.e., θ and $\mathcal{M}$ agree on the common values in their domains),
3. $\mathcal{I}' = \text{DO}(\mathcal{I}, \alpha, \sigma)\theta$, and $\mathcal{I}'$ satisfies $\mathcal{E}$,
4. $\mathcal{M}' = \mathcal{M} \cup \theta$.

In the above definition, the purpose of $\mathcal{M}$ is to record the history of service calls and their results, while θ contains the service calls invoked in the current transition, with their results (arbitrary values from $\mathcal{C}$). The compatibility of $\mathcal{M}$ and θ in condition (2) forces the current invocation of a call to return the same result as its past invocations, thus realizing the intended deterministic semantics.

Concrete transition system. The *concrete transition system* $\Upsilon_\mathcal{S}$ for $\mathcal{S}$ is a (possibly infinite-state) transition system $\langle \mathcal{C}, \mathcal{R}, \Sigma, s_0, db, \Rightarrow\rangle$ where: $s_0 = \langle \mathcal{I}_0, \emptyset\rangle$; db is such that $db(\langle \mathcal{I}, \mathcal{M}\rangle) = \mathcal{I}$; Σ and $\Rightarrow$ are defined by simultaneous induction as the smallest sets satisfying the following properties: *(i)* $s_0 \in \Sigma$; *(ii)* if $\langle \mathcal{I}, \mathcal{M}\rangle \in \Sigma$, then for all substitutions σ for the parameters of α and for all $\langle \mathcal{I}', \mathcal{M}'\rangle$ such that $\langle\langle \mathcal{I}, \mathcal{M}\rangle, \alpha\sigma, \langle \mathcal{I}', \mathcal{M}'\rangle\rangle \in \text{D-EXEC}_\mathcal{S}$, we have $\langle \mathcal{I}', \mathcal{M}'\rangle \in \Sigma$ and $\langle \mathcal{I}, \mathcal{M}\rangle \Rightarrow \langle \mathcal{I}', \mathcal{M}'\rangle$.

Intuitively, to define the concrete transition system of the DCDS $\mathcal{S}$ we start from the initial state $s_0 = \langle \mathcal{I}_0, \emptyset\rangle$, and calculate all states s such that $\langle s_0, \alpha\sigma, s\rangle \in \text{D-EXEC}_\mathcal{S}$. Then we repeat the same steps considering each s, and so on. The computation of successor states is done by: getting all legal substitutions of the action parameters according to the condition-action rule in the process; executing the instantiated actions; picking all the possible combinations of resulting values for the newly introduced service calls; and filtering away those successors that violate the equality constraints. It is worth noting that the number of successors can be countably infinite because the service call results come from $\mathcal{C}$.

4.2 Run-Bounded Systems

We now study the verification of DCDSs with deterministic services. In particular, we are interested in the following problem: given a DCDS $\mathcal{S}$ and a temporal property Φ, check whether $\Upsilon_\mathcal{S} \models \Phi$. Not surprisingly, given the expressive power of DCDS as a computation model, the verification problem is undecidable for all the μ-calculus variants introduced in Section 3. In fact, we can show an even stronger undecidability result, namely for safety properties expressible in both propositional LTL and CTL [5].

THEOREM 4.1. *There exists a DCDS $\mathcal{S}$ with deterministic services, and a propositional safety property Φ expressible in LTL $\cap$ CTL, such that checking $\Upsilon_\mathcal{S} \models \Phi$ is undecidable.*

Next, we isolate a notable class of DCDS for which verification of $\mu\mathcal{L}_A$ is not only decidable, but can also be reduced to standard finite-state model checking. Consider a transition system $\Upsilon = \langle \Delta, \mathcal{R}, \Sigma, s_0, db, \Rightarrow\rangle$. A *run* τ in Υ is a (finite or infinite) sequence of states $s_0 s_1 s_2 \cdots$ rooted at s_0, where $s_i \Rightarrow s_{i+1}$. We use $\tau(i)$ to denote s_i and $\tau[i]$ to represent the finite prefix $s_0 \cdots s_i$ of τ. A run $\tau = s_0 s_1 s_2 \cdots$ is *(data) bounded* if the total number of values occurring in its databases is bounded, i.e., there exists a finite bound $b > |\bigcup_{s \text{ state of } \tau} \text{ADOM}(db(s))|$. This is equivalent to saying that, for every finite prefix $\tau[i]$ of τ, $|\bigcup_{j\in\{0,\ldots,i\}} \text{ADOM}(db(s_j))| < b$. We say that Υ is *run-bounded* if there exists a bound b such that every run in Υ is (data) bounded by b. A DCDS $\mathcal{S}$ is run-bounded if its concrete transition system $\Upsilon_\mathcal{S}$ is run-bounded.

Intuitively, a (data) unbounded run represents an execution of the DCDS in which infinitely many distinct values occur because infinitely many different service calls are issued. Since we model deterministic services whose number is finite, this can only happen if some service is repeatedly called with arguments that are the result of previous service calls. This means that the values of the run indirectly depend on arbitrarily many states in the past.

Notice that run boundedness does not impose any restriction about the branching of the transition system; in particular, $\Upsilon_\mathcal{S}$ is typically infinite-branching because new service calls may return any possible value. We show that this restriction guarantees decidability of $\mu\mathcal{L}_A$ verification for run-bounded DCDSs with deterministic services.

THEOREM 4.2. *Verification of $\mu\mathcal{L}_A$ properties on run-bounded DCDSs with deterministic services is decidable.*

We get this result by showing that for run-bounded DCDSs we can always construct, without knowing the bound beforehand, an abstract finite-state transition system that is history-preserving bisimilar to the concrete one, and hence satisfies the same $\mu\mathcal{L}_A$ formulae as the concrete transition system.

PROPOSITION 4.3. *Let $\mathcal{S}$ be a run-bounded DCDS with deterministic services and $\Upsilon_\mathcal{S}$ its concrete transition system. Then there exists an (abstract)* finite-state *transition system $\Theta_\mathcal{S}$ such that $\Theta_\mathcal{S}$ is history-preserving bisimilar to $\Upsilon_\mathcal{S}$, i.e., $\Theta_\mathcal{S} \approx \Upsilon_\mathcal{S}$.*

Let Σ be the set of states of $\Theta_\mathcal{S}$ and $\text{ADOM}(\Theta_\mathcal{S}) = \bigcup_{s_i\in\Sigma} \text{ADOM}(db(s_i))$. If $\Theta_\mathcal{S}$ is finite-state, then there exists a bound b such that $|\text{ADOM}(\Theta_\mathcal{S})| < b$. Consequently, it is possible to transform a $\mu\mathcal{L}_A$ property Φ into an equivalent *finite* propositional μ-calculus formula $\text{PROP}(\Phi)$, where $\text{PROP}(\Phi)$ is inductively defined by recurring over the structure of Φ and substituting $\text{PROP}(\exists x.\text{LIVE}(x) \wedge \Psi(x))$ with $\bigvee_{t_i\in\text{ADOM}(\Theta_\mathcal{S})} \text{LIVE}(t_i) \wedge \text{PROP}(\Psi(t_i))$. Clearly, $\Theta_\mathcal{S} \models \Phi$ if and only if $\Theta_\mathcal{S} \models \text{PROP}(\Phi)$.

THEOREM 4.4. *Verification of $\mu\mathcal{L}_A$ properties for run-bounded DCDSs with deterministic services can be reduced to model checking of propositional μ-calculus over a finite transition system.*

By the above theorem, and recalling that model checking of propositional μ-calculus formulae over finite transition systems is decidable [26], we get Theorem 4.2.

We conclude the section by observing that the approach presented above for $\mu\mathcal{L}_A$ does not extend to full $\mu\mathcal{L}$.

THEOREM 4.5. *There exists a run-bounded DCDS $\mathcal{S}$ for which it is impossible to find a faithful finite-state abstraction that satisfies the same $\mu\mathcal{L}$ properties as $\mathcal{S}$.*

Theorem 4.5 is proved by exhibiting, for every n, a $\mu\mathcal{L}$ property that requires the existence of at least n objects in the transition system, such as the one in Example 3.2. While this result does not imply undecidability of model checking $\mu\mathcal{L}$ properties over run-bounded DCDSs, it dashes any hope of reducing this problem to standard, finite-state model checking.

4.3 Weakly Acyclic DCDSs

The decidability results in Section 4.2 rely on the hypothesis that the DCDS is run-bounded, which is a semantic condition. A natural question is whether it is possible to check run-boundedness of a DCDS. We provide a negative answer to this question.

THEOREM 4.6. *Checking run-boundedness of DCDSs with deterministic services is undecidable.*

To mitigate this issue, we investigate a sufficient syntactic condition that can be effectively tested over the process layer of the DCDS: if the condition is met, then the DCDS is guaranteed to be run-bounded, otherwise nothing can be said. To this end, we recast the approach of [17] and [3] in the more expressive framework here presented. To derive a sufficient condition for $\mathcal{S}$ to be run-bounded, we exploit a correspondence between (a carefully constructed approximation of) the execution of an action and a step in the chase of a set of TGDs in data exchange [2, 27]. In particular, we resort to a well-known result in data exchange, namely chase termination for *weakly acyclic* TGDs [27]. In our setting, the weak acyclicity of a process layer $\mathcal{P}$ is a property over a dataflow graph constructed from $\mathcal{P}$, where we consider only the contribution of the union of conjunctive queries q^+ in each action effect. We omit the definition of weak acyclicity and provide an intuition here (see [4] for details). The problem of non run-bounded DCDSs comes from services that are repeatedly called, every time using fresh values that are directly or indirectly obtained by manipulating previous results produced by the same service. This self-dependency can potentially lead to incorporating unboundedly many results of these service calls into the run. Weak acyclicity rules out such self dependencies, being in fact a sufficient, polynomially-checkable condition for run-boundedness.[9]

THEOREM 4.7. *Every weakly acyclic DCDS with deterministic services is run-bounded.*

This, together with Theorem 4.4, gives us an effective way to verify DCDSs by reduction to conventional model checking.

THEOREM 4.8. *Verification of $\mu\mathcal{L}_A$ properties for weakly acyclic DCDSs with deterministic services is decidable.*

5. NONDETERMINISTIC SERVICES

We now consider DCDSs under the assumption that services behave nondeterministically, i.e., two calls of a service with the same arguments may return distinct results during the same run. This case captures both services that model a truly nondeterministic process (e.g., human operators, random processes), and services that model stateful servers. In Example 2.1, all the reimbursement system services (e.g., DECIDE() and INHNAME()) are inherently nondeterministic as they model human input. In the remainder of this section, services are implicitly assumed nondeterministic.

5.1 Semantics

As in the case of deterministic services, we define the semantics of a DCDS $\mathcal{S}$ in terms of a (possibly infinite) transition system $\Upsilon_{\mathcal{S}}$.

Let $\mathcal{S} = \langle \mathcal{D}, \mathcal{P} \rangle$ be a DCDS with data layer $\mathcal{D} = \langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ and process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho \rangle$. A *state* is simply a relational instance of $\mathcal{R}$ over $\mathcal{C}$ satisfying each constraint in $\mathcal{E}$. We denote the *initial state* with $\mathcal{I}_0$.

Next, we define the semantics of action application. Let α be an action in $\mathcal{A}$ with parameters $p_1, \ldots, p_m$. Let σ be a substitution for $p_1, \ldots, p_m$ with values taken from $\mathcal{C}$, that is legal according to the process ϱ. We recall the definitions of DO() and EVALS$_D$() from Section 4.1. DO$(\mathcal{I}, \alpha, \sigma)$ denotes the instance obtained by evaluating the effects of action α with parameters σ on instance $\mathcal{I}$. For a given set $D \subseteq \mathcal{C}$, EVALS$_D(\mathcal{I}, \alpha, \sigma)$ is the set of substitutions that replace all service calls in DO$(\mathcal{I}, \alpha, \sigma)$ with values in D.

Concrete action execution. We introduce a transition relation N-EXEC$_{\mathcal{S}}$ between states, called *concrete execution* of $\alpha\sigma$, such that $\langle \mathcal{I}, \alpha\sigma, \mathcal{I}' \rangle \in$ N-EXEC$_{\mathcal{S}}$ if we have:

1. σ is a legal parameter assignment for α in state $\mathcal{I}$,
2. there exists $\theta \in$ EVALS$_{\mathcal{C}}(\mathcal{I}, \alpha, \sigma)$,
3. $\mathcal{I}' =$ DO$(\mathcal{I}, \alpha, \sigma)\theta$, and $\mathcal{I}'$ satisfies the constraints $\mathcal{E}$.

In contrast to the deterministic services case, the choice of evaluation θ is not subject to the requirement that it evaluates a service call to the same result *across* concrete execution steps (indeed, we no longer accumulate the successive choices of θ in the service call map $\mathcal{M}$). However, notice that *within* a concrete execution step, all occurrences of the same service call evaluate to the same result (modeling the fact that a call with given arguments is invoked only once per transition, and the returned result is copied as needed).

Concrete transition system. The *concrete transition system* $\Upsilon_{\mathcal{S}}$ for $\mathcal{S}$ is a transition system whose states are labeled by databases. More precisely, $\Upsilon_{\mathcal{S}} = \langle \mathcal{C}, \mathcal{R}, \Sigma, s_0, db, \Rightarrow \rangle$ where $s_0 = \mathcal{I}_0$ and db is such that $db(\mathcal{I}) = \mathcal{I}$. Σ and $\Rightarrow$ are defined by simultaneous induction as the smallest sets satisfying the following properties: *(i)* $\mathcal{I}_0 \in \Sigma$; *(ii)* if $\mathcal{I} \in \Sigma$, then for all α, σ and $\mathcal{I}'$ such that $\langle \mathcal{I}, \alpha\sigma, \mathcal{I}' \rangle \in$ N-EXEC$_{\mathcal{S}}$, we have that $\mathcal{I}' \in \Sigma$, and $\mathcal{I} \Rightarrow \mathcal{I}'$.

5.2 State-Bounded Systems

We consider the verification problem for DCDS with nondeterministic services. As in the deterministic case, an analogous undecidability result to Theorem 4.1 holds.

THEOREM 5.1. *There exists a DCDS $\mathcal{S}$ with nondeterministic services, and a propositional safety property Φ expressible in LTL $\cap$ CTL, such that checking $\Upsilon_{\mathcal{S}} \models \Phi$ is undecidable.*

Towards devising interesting decidable cases we start by observing that, with nondeterministic services, the run-boundedness restriction of Section 4.2 is very limiting on the form of the DCDS, as it boils down to imposing a bound on how many times each service may be called with the same arguments. Contrast this with deterministic services, where unlimited same-argument calls are allowed, as they all return the same result. We propose a less restrictive alternative. We say that DCDS $\mathcal{S}$ is *state-bounded* if there is a finite bound b such that for each state $\mathcal{I}$ of $\Upsilon_{\mathcal{S}}$, $|\text{ADOM}(\mathcal{I})| < b$. In contrast to run-boundedness, state-boundedness allows for runs in which infinitely many distinct values occur because infinitely many service calls are issued. These call results are distributed *across* states of the run, but may not accumulate *within* a single state. For example, the request system in Example 2.1 is not run-bounded, since a user can update the request information with an unbounded number of new values during a run. However, it is state-bounded, since each state contains exactly one request. $\mu\mathcal{L}_A$ can record past values in the run through quantification even if they are not present in the system anymore. The following result shows that this leads to undecidability.

THEOREM 5.2. *Verification of $\mu\mathcal{L}_A$ properties on state-bounded DCDSs with nondeterministic services is undecidable.*

We therefore focus on to the logic $\mu\mathcal{L}_P$ (presented in Section 3.2), in which this recording through quantification is disallowed.

THEOREM 5.3. *Verification of $\mu\mathcal{L}_P$ properties by state-bounded DCDS with nondeterministic services is decidable.*

[9] Notice that we can also use other variants of weak acyclicity [35].

We give the main ideas behind the proof of this theorem. Given a DCDS $\mathcal{S}$, we show that if concrete transition system $\Upsilon_{\mathcal{S}}$ is state-bounded, then there is a finite-state abstract transition system $\Theta_{\mathcal{S}}$ that is persistence-preserving bisimilar to $\Upsilon_{\mathcal{S}}$ (and hence satisfies the same $\mu\mathcal{L}_P$ properties, by Theorem 3.2). Since $\Theta_{\mathcal{S}}$ is finite-state, the verification of $\mu\mathcal{L}_P$ properties on $\Upsilon_{\mathcal{S}}$ reduces to finite-state model checking on $\Theta_{\mathcal{S}}$, and hence is decidable.

THEOREM 5.4. *Verification of $\mu\mathcal{L}_P$ properties for state-bounded DCDSs with nondeterministic services can be reduced to model checking of propositional μ-calculus over a finite transition system.*

The existence of $\Theta_{\mathcal{S}}$ follows from the key fact that if two states of $\Upsilon_{\mathcal{S}}$ are isomorphic, then they are persistence-preserving bisimilar. This implies that one can construct a finitely-branching transition system $\Theta_{\mathcal{S}}$ (i.e. with finite number of successors per state), such that $\Theta_{\mathcal{S}}$ is persistence-preserving bisimilar to $\Upsilon_{\mathcal{S}}$, by dropping sibling states from $\Upsilon_{\mathcal{S}}$ as follows: instead of listing among the successors of s one state for each possible instantiation of the service call results, just keep a *representative* state for each isomorphism type. Since the number of service calls made in each state is finite, the number of distinct isomorphism types is finite, so the finite branching follows. We call a transition system $\Theta_{\mathcal{S}}$ obtained as above a *pruning of* $\Upsilon_{\mathcal{S}}$.

Despite being finitely-branching, any pruning $\Theta_{\mathcal{S}}$ can still have infinitely many states, as it may contain infinitely long simple runs τ (a *simple* run is one in which no state appears more than once), along which the service calls return in each state "fresh" values, i.e., values distinct from all values appearing in the predecessors of this state on τ. This problem is solved by judiciously selecting which representatives to keep in $\Theta_{\mathcal{S}}$ for the successors of a state s. Namely, whenever the representatives of a given isomorphism type $\mathcal{T}$ include states generated exclusively by service calls that "recycle" values, select only such states (finitely many thereof, of course). By recycled values we mean values appearing on a path leading into s.

If $\Upsilon_{\mathcal{S}}$ is state-bounded, then the number of service calls per state is bounded, and due to the construction's preference for recycling, it follows that all simple runs in $\Theta_{\mathcal{S}}$ must have finite length. Together with the finite branching, this implies finiteness of $\Theta_{\mathcal{S}}$.

Notice that proving the existence of $\Theta_{\mathcal{S}}$ does not suffice for decidability, as the proof is non-constructive. We therefore provide an algorithm for constructing $\Theta_{\mathcal{S}}$ (cf. algorithm RCYCL in [4]). One of the technical problems we need to overcome in developing the algorithm is that we cannot start from the infinite-state concrete transition system, and instead need to explore a portion thereof. This means that it is not obvious how to decide whether the successors of a state are generated by recycling service calls, since these calls may recycle from paths that the algorithm has not explored yet. Therefore, the algorithm may sometimes select non-recycling service calls even when a recycling alternative exists. However, we can prove that it constructs what we call an *eventually recycling pruning*, which in essence means it may fail to detect recycling service calls, but only a bounded number of times. We note that the algorithm does not need to know a priori the bound on the state size; its mere existence guarantees that the construction terminates. This, together with Theorem 3.2, directly implies Theorem 5.3.

5.3 GR-Acyclic DCDSs

As with run-boundedness in the deterministic services case, for nondeterministic services state-boundedness is a semantic property, which in general is undecidable.

THEOREM 5.5. *Checking state-boundedness of DCDSs is undecidable.*

Consequently we propose a sufficient syntactic condition. Intuitively, for a run to have unbounded states, it must issue unboundedly many service calls. Since there are only a bounded number of effects in the process layer specification, there must exist some service-calling effect that "cyclically generates" fresh values (i.e., is invoked infinitely many times during the run). Notice that unbounded generation of fresh values does not break state-boundedness per se: these values must also accumulate in the states to do so. But by definition of the DCDS semantics, a transition drops ("forgets") all values that are not explicitly copied ("recalled") into the successor. Therefore, to accumulate, a value must be "cyclically recalled" throughout the run (copied infinitely many times).

GR-acyclicity is stated in terms of a dataflow graph constructed by analyzing the process layer. The graph identifies how service calls and value recalls can chain. In essence, GR-acyclicity requires the absence of a "generate cycle" that feeds into a "recall cycle".

GR-acyclicity. We call *dataflow graph* of a set $\mathcal{A}$ of actions the directed edge-labeled graph $\langle N, G\rangle$ whose set N of nodes is the set of relation names occurring in $\mathcal{A}$, and in which each edge in G is a 4-tuple (R_1, id, R_2, b), where R_1 and R_2 are two nodes in N, id is a (unique) edge identifier, and b is a boolean flag used to mark *special* edges. Formally, G is the minimal set satisfying the following condition: for each effect e in $\mathcal{A}$ of the form $q^+ \wedge Q^- \rightsquigarrow E$, each R in q^+, each $Q(t_1, \ldots, t_m)$ in E, and each $i \in \{1, \ldots, m\}$:

- if t_i is either an element of $\text{ADOM}(\mathcal{I}_0)$ or a free variable, then $(R, id, Q, \mathsf{false}) \in G$, where id is a fresh edge identifier.
- if t_i is a service call, then $(R, id, Q, \mathsf{true}) \in G$, where id is a fresh edge identifier.

We say that $\mathcal{A}$ is *GR-acyclic* if there is no path $\pi = \pi_1\pi_2\pi_3$ in the dataflow graph of $\mathcal{A}$, such that π_1, π_3 are simple cycles and π_2 is a path containing a special edge, and having at least some edge disjoint from the edges of π_1: $edges(\pi_2) \setminus edges(\pi_1) \neq \emptyset$. We say that a process layer $\mathcal{P} = \langle \mathcal{F}, \mathcal{A}, \varrho\rangle$ is GR-acyclic, if $\mathcal{A}$ is GR-acyclic. A DCDS GR-acyclic if its process layer is GR-acyclic.

Note that GR-acyclicity is a purely syntactic notion. It can be checked in coNP since the dataflow graph is polynomial in the size of the process layer specification, and since, if there is a violation of GR-acyclicity, then there is a polynomial-sized one as well.

THEOREM 5.6. *Any GR-acyclic DCDS is state-bounded.*

We give the main ideas behind the proof of this theorem, noting that the dataflow analysis is significantly more subtle than suggested above. First, note that ordinary edges correspond to an effect copying a value from a relation of the current state to a relation of the successor state. Special edges correspond to feeding a value of the current state to a service call and storing the result in a relation of the successor state. Note that the cycles π_1 and π_3 allow both kinds of edges, reflecting the insight that the size of the state is affected in the same way regardless of whether a value is *copied* to the successor, or it is *replaced* with a service call result. π_1, π_3 are both "recall cycles": the number of values moving around them does not decrease (this is of course a conservative statement; reality depends on the semantics of queries in the effects, which is abstracted away). Note that π_2 contains a special edge G, which means that the values moving around π_1 are cyclically fed into the service call f of G. The key insight here is that, even if the set of values moving around π_1 does not change (no special edges in π_1 replace them), and thus the service call f sees the same bounded set of distinct arguments over time, it can still generate an unbounded number of fresh values because f is nondeterministic. $\pi_1\pi_2$ constitute the "generate cycle" we mention above. The generated values are stored in the recall cycle π_3, where they accumulate and force the size of the relations of π_3 to grow unboundedly.

We observe that GR-acyclicity is not related to weak acyclicity. In particular, a DCDS may be GR-acyclic but not weakly acyclic.

GR$^+$-acyclicity. GR-acyclicity can be relaxed based on the insight that, for a cycle Υ_S in the dataflow graph to truly preserve the number of values moving in it, Υ_S's edges must not all be simultaneously inactive. We say that an edge is *active* in a step of the run when the action corresponding to it executes. By the DCDS semantics, if all edges of Υ_S are simultaneously inactive, then none of the corresponding copy/call operations are executed and all relations involved in Υ_S forget their value in the next state. Υ_S is effectively flushed. GR$^+$-acyclicity is the relaxation that does allow a path $\pi = \pi_1\pi_2\pi_3$ as in the definition of GR-acyclicity, provided that π_2 contains an edge e that cannot be active at the same time as any of the subsequent edges e' in $\pi_2\pi_3$. Formally, we associate with every edge ϵ in the dataflow graph the action $action(\epsilon)$ it corresponds to (computed via simple inspection of the process layer). Then for every $e \in \pi_2, e' \in \pi_2\pi_3$ as above, we require that $action(e) \neq action(e')$. Semantically this ensures that in order for the generate cycle $\pi_1\pi_2$ to push fresh values toward recall cycle π_3, the action corresponding to e must execute, and in the meantime all actions maintaining the values in cycle π_3 are disabled, thus flushing π_3. π_3 thus receives an unbounded number of waves of fresh values from $\pi_1\pi_2$, but it forgets each wave before the next arrives. GR$^+$-acyclicity can also be checked in coNP. Theorem 5.6 extends to GR$^+$-acyclicity.

THEOREM 5.7. *Every GR$^+$-acyclic DCDS is state-bounded.*

This, together with Theorem 5.3, implies:

THEOREM 5.8. *Verification of $\mu\mathcal{L}_P$ properties for GR$^+$-acyclic DCDS with nondeterministic services is decidable.*

EXAMPLE 5.1. The dataflow graph for the request system of Example 2.1 is depicted in Figure 2, where special edges are starred.

Notice that there can be multiple normal/special edges between the same two nodes (these are distinguished by unique edge ids; to avoid clutter in the figure we omit the ids, and show edge multiplicities in square brackets; missing brackets denote multiplicity 1). For example, the red simple edge from Hotel to Status is due to the first effect of action *VerifyRequest*, while the special edge in parallel with it is due to the DECIDE() call in the second effect. The red self-loops on Flight, Hotel, Travel and ApprHotel reflect the remaining effects of *VerifyRequest*. The five black special edges from the true node (with the obvious meaning) to the Hotel node correspond to the employee filling in the hotel information in action *UpdateRequest*, modeled by calls to such services as INHNAME(). We refer to [4] for a complete treatment of the example. An inspection of this dataflow graph reveals that the request system is not GR-acyclic, since it contains several instances of two simple cycles connected by a path that includes a special edge: for instance, the path π comprised of the normal self-loops around Hotel and Status and the special edge between them. However, the request system is GR$^+$-acyclic, confirming that the system is indeed state-bounded: all edges of cycles downstream of special edges e are due to actions (colors) disjoint from e's. ■

6. DISCUSSION

Summary of results. We summarize our results in Table 1 where, for completeness, we add additional results that can be proven analogously to the ones in the body of the paper (see [4]). Arrows denote implications between results. We note that exhibiting a finite faithful abstraction of a concrete transition system is more

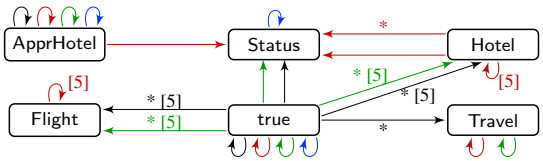

Figure 2: Dataflow graph for Example 5.1: colors black, red, green, blue correspond respectively to actions *InitiateRequest*, *VerifyRequest*, *UpdateRequest*, and *AcceptRequest*.

than a means towards showing decidability, being a desirable goal in its own right as the most promising avenue towards practical implementation. We list as open the verification of $\mu\mathcal{L}$ properties on run-bounded DCDSs, but recall from Section 4.2 that in this case there exists no faithful finite-state abstract transition system.

Complexity. Both in the case of weakly acyclic DCDSs with deterministic services and of GR$^+$-acyclic DCDSs with nondeterministic services, our construction generates a finite transition system whose number of states is exponential in the size of the initial database. Let Φ be a $\mu\mathcal{L}_A$ or $\mu\mathcal{L}_P$ formula of size ℓ with k alternating nested fixpoints. The proofs of Theorems 4.7 and 5.7 guarantee that the total number of values appearing in the abstract transition system is bounded by a polynomial $P(n)$ of the size n of the DCDS initial database. Then, considering the complexity of propositional μ-calculus model checking on finite transition systems [26], the complexity of verification of Φ is $O(2^{P(n)^a} \cdot P(n)^\ell)^k$, where a is the maximum arity of the database schema. Hence, verification is in EXPTIME in the size n of the initial database.

Mixed semantics. It is possible to integrate into a unique system both deterministic and nondeterministic services, and to extend our verification results to this case. The request and audit modules of the running example are part of such a mixed system (see [4]).

Support for arbitrary integrity constraints. Note that the definition of DCDS semantics is independent of the type of constraints used, as it simply requires their satisfaction by each state of the concrete transition system. Our decidability results hold even in the presence of integrity constraints on the database expressed as arbitrary FO sentences under the active domain semantics (see [4] for how we can model them without changes to the DCDS model).

Connection with the artifact model. In terms of expressive capabilities, our DCDS model is equivalent to the artifact-centric model [37, 21] (see Section 7). The reductions between the DCDS and artifact models are sketched in [4].

7. RELATED WORK

As mentioned in Section 6, the unrestricted artifact-centric and DCDS models have equivalent expressive capabilities. Our work is therefore most closely related to prior work on verification of

	DETERMINISTIC			NONDETERMINISTIC		
	$\mu\mathcal{L}$	$\mu\mathcal{L}_A$	$\mu\mathcal{L}_P$	$\mu\mathcal{L}$	$\mu\mathcal{L}_A$	$\mu\mathcal{L}_P$
unrestricted	**U**	← **U**	← **U**[1](Th.4.1)	**U**	← **U**	←**U**[1](Th.5.1)
	↑	↑		↑	↑	
state-bounded	**U**	← **U** [4]	**D** [4]	**U**	←**U** (Th.5.2)	**D**[3](Th.5.3)
			↓			↓
run-bounded	**?**[2](Th.4.5)	**D**[3](Th.4.2)→	**D**	**?**[2](Th.4.5)	**D**[3] [4] →	**D**

[1]Holds even for propositional LTL/CTL.
[2]Decidability cannot rely on faithful finite-state abstraction.
[3]Via reduction to finite-state model checking.

Table 1: Summary of our (un)decidability results

artifact-centric business processes. The difference lies in how each work trades off between restricting the class of business processes versus the class of properties to verify.

Artifact-centric processes with no database. Work on formal analysis of artifact-based business processes in restricted contexts has been reported in [28, 29, 9]. Properties investigated include reachability [28, 29], general temporal constraints [29], and the existence of complete execution or dead end [9]. For each considered variant, verification is generally undecidable; decidability results were obtained only under rather severe restrictions, e.g., restricting all pre-conditions to be "true" [28], restricting to bounded domains [29, 9], or restricting the pre- and post-conditions to be propositional, and thus not referring to data values [29]. [16] adopts an artifact model variation with arithmetic operations but no database. It proposes a criterion for comparing the expressiveness of specifications using the notion of *dominance*, based on the input/output pairs of business processes. Decidability relies on restricting runs to bounded length. [41] addresses the problem of the existence of a run that satisfies a temporal property, for a restricted case with no database and only propositional LTL properties. All of these works model no underlying database (and hence no integrity constraints).

Artifact-centric processes with underlying database. More recently, two lines of work have considered artifact-centric processes that also model an underlying relational database. One considers branching time, one only linear time.

Branching time. Our approach for deterministic services stems from a line of research that started with [17] and continued with [3] in the context of artifact-centric processes. These works, however, considered a limited form of deterministic services, and use as verification formalism, first-order μ-calculus without first-order quantification across states. The connection between evolution of data-centric dynamic systems and data exchange that we exploit in this paper was first devised in [17]. There the dynamic system transition relation itself is described in terms of TGDs mapping the current state to the next, and the evolution of the system is essentially a form of chase. Under suitable weak acyclicity conditions such a chase terminates, thus making the DCDS transition system finite. Notice the role of getting new objects/values from the external environment, played here by service calls, is played there by nulls. These ideas where further developed in [3], where TGDs were replaced by action rules with the same syntax as here. Semantically however the dynamic system formalism there is deeply different: what we call here service calls are treated there as uninterpreted Skolem terms. This results in an ad-hoc interpretation of equality which sees every Skolem term as equal only to itself (as in the case of nulls [17]). A form of weak acyclicity gives a sufficient condition for getting finite-state transition systems and decidability.

Differently from [3], in our framework we do interpret service calls. This decision is motivated by our goal of modeling real-life external services, for which two distinct service calls may very well return equal results, even under the deterministic semantics (for instance if the same service is called with different arguments, or if distinct services are invoked). Interpreting service calls raises a major challenge: even under the run-bounded restriction, the concrete transition system is infinite, because it is infinitely branching (a service call can be interpreted with any of the values from the infinite domain). In contrast to [3], what we show in this case is not that the concrete transition system is finite (it never is), but that it is *bisimilar* to a finite abstract transition system. This leads to a proof technique that is interesting in its own right, being based on novel notions of bisimilarity for the considered μ-calculus variants. The reason standard bisimilarity is insufficient is that our logics $\mu\mathcal{L}_P$ and $\mu\mathcal{L}_A$ allow first-order quantification across states, so bisimilarity must respect the connection between values appearing both in the current and successor state. Our decision to include first-order quantification across states was motivated by the need to express liveness properties that refer to the same data at various points in time (e.g. "if student x is enrolled now and continues to be enrolled in the future, then x will eventually graduate").

For our treatment of non-deterministic services, the most related work is [6]. Inspired by [3], [6] builds a similar framework where actions are specified via pre- and post-conditions given as FO formulae interpreted over active domains which include new values passed through action parameters. The verification logic considered is an FO variant of CTL, which precludes expressing certain desirable properties such as fairness. [6] shows that if each state has a bounded active domain, one can construct an abstract finite transition system that can be checked instead of the original concrete transition system, which is infinite-state in general. [6] develops independently an approach that is similar to ours for nondeterministic services. Indeed, the results we present here on state-boundedness apply to the setting of [6] as well. In contrast to [6], we investigate non-trivial, sufficient syntactic conditions for state-boundedness. Moreover, as opposed to [6], the bound on the state size need not be a priori known to our abstraction building algorithm. The bound's mere existence guarantees the algorithm's convergence. On the other hand, [6] presents an interesting result that applies to our setting: verifying formulas corresponding to CTL is decidable.

Linear time. [23] considers an artifact model that has the same expressive capabilities as an unrestricted class of DCDS in which the infinite domain is equipped with a dense linear order, which can be mentioned in pre-, post-conditions, and properties. Runs can receive unbounded external input from an infinite domain, and this input corresponds to nondeterministic services in a DCDS. Verification is decidable even if the input accumulates in states, and runs are neither run-bounded, nor state-bounded. However, this expressive power requires restrictions that render the result incomparable to ours. First, the property language is a first-order extension of LTL, and it is shown that extension to branching time (CTL*) leads to undecidability. Second, pre-, post-conditions and properties access read-only and read-write database relations differently, querying the latter only in limited fashion. In essence, data can arbitrarily accumulate in read-write relations, but these can be queried only by checking that they contain a given tuple of constants. It is shown that this restriction is tight, as even the ability to check emptiness of a read-write relation leads to undecidability. In addition, no integrity constraints are supported as it is shown that allowing a single functional dependency leads to undecidability. [20] disallows read-write relations entirely (only the artifact variables are writable), but this allows the extension of the decidability result to integrity constraints expressed as embedded dependencies with terminating chase, and to any decidable arithmetic. Again the result is incomparable to ours, as our modeling needs to include read-write relations and their unrestricted querying.

Infinite-state systems. DCDSs are a particular case of infinite-state systems. Research on verification of infinite-state systems has also focused on extending classical model checking techniques (e.g., see [15] for a survey). However, in much of this work the emphasis is on studying recursive control rather than data, which is either ignored or finitely abstracted. More recent work has focused specifically on data as a source of infinity. This includes augmenting recursive procedures with integer parameters [12], rewriting systems with data [11], Petri nets with data associated to tokens [33], automata and logics over infinite alphabets [36, 10, 11], and temporal logics manipulating data [13, 22, 31]. However, the restricted use of data

and the particular properties verified have limited applicability to the business process setting we target with the DCDS model.

8. CONCLUSIONS

We believe that DCDSs are a natural and expressive model for business processes powered by an underlying database, and thus are an ideal vehicle for foundational research with potential to transfer to alternative models. The design space for FO extensions of propositional μ-calculus is broad, and notoriously contains bounded-state settings for which satisfiability of even modest extensions of propositional LTL is highly undecidable (e.g. LTL with freeze quantifier over infinite data words [22]). In light of this, our decidability results come as a pleasant surprise, and the two $\mu\mathcal{L}$ variants studied here, paired with the respective DCDS classes, strike a fortuitous balance between expressivity and verification feasibility.

9. REFERENCES

[1] S. Abiteboul, P. Bourhis, A. Galland, and B. Marinoiu. The AXML artifact model. In *TIME*, 2009.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison Wesley, 1995.

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

[4] B. Bagheri Hariri, D. Calvanese, G. De Giacomo, A. Deutsch, and M. Montali. Verification of relational data-centric dynamic systems with external services. Corr technical report, arXiv.org e-Print archive, 2012. Available at `http://arxiv.org/abs/1203.0024`.

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

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

[7] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella. Automatic composition of transition-based semantic web services with messaging. In *VLDB*, 2005.

[8] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, and F. Y. Wu. Artifact-centered operational modeling: Lessons from customer engagements. *IBM Systems Journal*, 46(4):703–721, 2007.

[9] K. Bhattacharya, C. E. Gerede, R. Hull, R. Liu, and J. Su. Towards formal analysis of artifact-centric business process models. In *BPM*, 2007.

[10] M. Bojanczyk, A. Muscholl, T. Schwentick, L. Segoufin, and C. David. Two-variable logic on words with data. In *LICS*, 2006.

[11] A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. Rewriting systems with data. In *FCT*, 2007.

[12] A. Bouajjani, P. Habermehl, and R. Mayr. Automatic verification of recursive procedures with one integer parameter. *TCS*, 295, 2003.

[13] P. Bouyer, A. Petit, and D. Thérien. An algebraic approach to data languages and timed languages. *Information and Computation*, 182(2), 2003.

[14] J. Bradfield and C. Stirling. Modal mu-calculi. In *Handbook of Modal Logic*, volume 3. Elsevier, 2007.

[15] O. Burkart, D. Caucal, F. Moller, and B. Steffen. Verification of infinite structures. In *Handbook of Process Algebra*. Elsevier Science, 2001.

[16] D. Calvanese, G. De Giacomo, R. Hull, and J. Su. Artifact-centric workflow dominance. In *ICSOC*, 2009.

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

[18] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. The MIT Press, 1999.

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

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

[21] 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.

[22] S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM TOCL*, 10(3), 2009.

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

[24] A. Deutsch, M. Marcus, L. Sui, V. Vianu, and D. Zhou. A verifier for interactive, data-driven web applications. In *SIGMOD*, 2005.

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

[26] E. A. Emerson. Model checking and the mu-calculus. In *Descriptive Complexity and Finite Models*, 1996.

[27] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data exchange: semantics and query answering. *TCS*, 336(1), 2005.

[28] C. E. Gerede, K. Bhattacharya, and J. Su. Static analysis of business artifact-centric operational models. In *IEEE Int. Conf. on Service-Oriented Computing and Applications*, 2007.

[29] C. E. Gerede and J. Su. Specification and verification of artifact behaviors in business process models. In *ICSOC*, 2007.

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

[31] M. Jurdzinski and R. Lazić. Alternation-free modal mu-calculus for data trees. In *LICS*, 2007.

[32] J. Küster, K. Ryndina, and H. Gall. Generation of BPM for object life cycle compliance. In *BPM*, 2007.

[33] R. Lazić, T. Newcomb, J. Ouaknine, A. Roscoe, and J. Worrell. Nets with tokens which carry data. In *ICATPN*, 2007.

[34] D. C. Luckham, D. M. R. Park, and M. Paterson. On formalised computer programs. *JCSS*, 4(3), 1970.

[35] M. Meier, M. Schmidt, F. Wei, and G. Lausen. Semantic query optimization in the presence of types. In *PODS*, 2010.

[36] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM TOCL*, 5(3), 2004.

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

[38] D. M. R. Park. Finiteness is mu-ineffable. *TCS*, 3(2), 1976.

[39] C. Stirling. *Modal and Temporal Properties of Processes*. Springer, 2001.

[40] W. M. P. van der Aalst, P. Barthelmess, C. A. Ellis, and J. Wainer. Proclets: A framework for lightweight interacting workflow processes. *Int. J. of Cooperative Information Systems*, 10(4), 2001.

[41] X. Zhao, J. Su, H. Yang, and Z. Qiu. Enforcing constraints on life cycles of business artifacts. In *TASE*, 2009.