Verification of Human Driven Data-Centric Dynamic Systems

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

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

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

1 Introduction

Business Process Management (BPM) refers to a systematic, holistic approach for overseeing how work is performed in an organization, and aligning it with the business objectives of the organization and the needs of its customers. The core artifact in BPM is the notion of *business process*, an explicit conceptual model that accounts for the way business activities can be executed over time (control-flow perspective), who is in charge of executing them (resource perspective), and how the activity execution impact on data (data perspective).

A huge amount of research has been devoted to the development of languages, methodologies, and techniques so as to support business stakeholders throughout the entire business process lifecycle (van der Aalst 2012): from design to execution, a-posteriori analysis, mining and re-engineering. However, as pointed out in (Harrison-Broninski 2005): *"despite advances in business automation over the past fifty years, the heart and soul of e very organization is still its people [. . .]. Yet there is presently no complete way to manage the complex, continually changing work processes carried out by humans"*.

Arguably, part of Harrison-Broninski's criticism is due to the fact that the majority of BPM approaches tend to mainly focus on the (internal) control-flow dimension of business processes, while neglecting the connection with data and humans. Two main lines of research emerged, in the last decade, to overcome these limitations:

- Data- and artifact-centric approaches have been proposed to explicitly and conceptually tackle the interplay between processes and data (Nigam and Caswell 2003; Hull 2008; Calvanese, De Giacomo, and Montali 2013). In this light, a business process is seen as a set of dynamic constraints that determine the allowed evolutions of business-relevant entities (like a purchase order or a loan). This does not only contribute to better highlight how data are manipulated by the process, but also to raise the level of abstraction in process modelling and understandability (Bhattacharya et al. 2007).
- Declarative approaches have been studied, so as to put minimal constraints on the way activities can be executed, and give more freedom of choice to the involved humans (see, e.g., (Montali et al. 2010; Slaats et al. 2013)). They are suitable to support knowledge-intensive domains, where it is not possible to fully automatize in advance which are the allowed courses of execution, but it is instead needed to let humans dynamically decide how to proceed depending on the given constraints, but also the context of execution and their own background knowledge.

These two kinds of approaches ultimately converged in concrete languages and execution environments such as the Guard-Stage-Milestone model by IBM (Damaggio, Hull, and Vaculín 2013) and the OMG standard for Case Management Model and Notation[1].

In this work, we tackle the foundations of such processes, by leveraging on the recently proposed framework of Data-Centric Dynamic Systems (DCDSs) (Bagheri Hariri et al. 2013). Specifically, we extend DCDSs to better model human driven processes, and we recast the verification results provided in (Bagheri Hariri et al. 2013) to such a revised framework. It is worth noting that, in the rich setting where both processes and data are simultaneously taken into account, verification is highly undecidable even for simple propositional temporal properties (such as reachability) (Deutsch et al. 2009; Bagheri Hariri et al. 2013). Therefore, suitable decidable fragments need to be isolated.

DCDSs already support a declarative style of modelling, tailored to human driven processes. In fact, the process control-flow is declaratively specified by means of condition-action rules that query the current data and determine which actions can be executed, and with which parameters. The action to be effectively executed is then freely chosen by the responsible persons among the available alternatives. The data manipulation induced by an action execution is specified by means of forward rules, each of which queries the current data and determines which facts will be asserted in the next state. Such facts are instantiated with values extracted by such queries, or with new values. To reflect human interaction with the process, we study here three main possible ways for inputting new values during an action execution:

- *Deterministic input* - when the user is asked to provide for the first time an input starting from a given combination of values, she freely choses the value to return; in the future, whenever the same kind of input is asked starting from the same combination of values, the returned value does not change. An example is the amount of money on the user's

[1] http://www.omg.org/spec/CMMN/1.0/Beta1/

bank account on a given date.

• *Nondeterministic input* - whenever the user is asked to provide an input, she freely choses the value to return, independently from the previously returned values. An example is the actual amount of money on the user's bank account.

• *Selection input* - the user is asked to choose an input value from a predefined set of alternatives. An example is the 1-to-5 star rating for an item.

These three input modalities can be used to formalize the typical web-based interaction with users, such as text fields, optional choices, and combo boxes. Notice that the first two ways have already been introduced in the original DCDS proposal (Bagheri Hariri et al. 2013), in the form of deterministic vs nondeterministic service calls, but were never mixed in the same specification. The third modality is instead studied here for the first time.

With this extended framework, we reconstruct the decidability results studied in (Bagheri Hariri et al. 2013) for the classes of *run-bounded* and *state-bounded* DCDSs. Furthermore, we show how the sufficient, syntactic checks introduced in (Bagheri Hariri et al. 2013) for determining whether a DCDS with (non)deterministic service calls is run- or state-bounded, have to be reconsidered when all the aforementioned types of user input are mixed in the same specification.

For the detailed proofs refer to (Bagheri Hariri 2013) (Sections 5 and 6.5)[2].

2 Human Driven DCDSs

Data-centric dynamic systems (DCDSs), originally introduced in (Bagheri Hariri et al. 2013), are systems in which both the process controlling the system dynamics and the manipulated data are equally central. DCDSs are a pristine version of several proposals in the literature (e.g., (Berardi et al. 2005; Deutsch et al. 2009)), and are in particular equally expressive to the most sophisticated business artifact models, such as (Damaggio, Hull, and Vaculín 2013). To better account for human interaction, in this paper we introduce *human driven data-centric dynamic systems* as an extension of the original model. In this extension, the system can simultaneously support different types of user inputs: deterministic, nondeterministic, finite-range, and infinite-range (see below). In the rest of this paper, we maintain the acronym of "DCDS" to refer to human driven DCDSs.

A DCDS is constituted by *(i)* a *data layer*, which holds the relevant information to be manipulated by the system, and is technically a full-fledged *relational* database; *(ii)* a *process layer*, formed by invokable *(atomic) actions*, and declarative process rules that determine which actions are executable.

Executing an action has effects on the data manipulated by the system, on the process state, and on the information exchanged with the external users. Specifically, an action may request users to input new data into the system. [3] As a result, the transition system that accounts for the DCDS execution semantics is in general infinite-state.

[2] http://www.inf.unibz.it/~bbagheri/thesis.pdf

[3] In (Bagheri Hariri et al. 2013), the term "external service call" is used to denote a user request. Here we stick on the second acception.

2.1 Syntax

Formally, 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 it. Intuitively, the data layer keeps all the data of interest, while the process layer modifies and evolves such data.

Data Layer The *data layer* $\mathcal{D}$ is a tuple $\langle \mathcal{C}, \mathcal{R}, \mathcal{E}, \mathcal{I}_0 \rangle$ where: $\mathcal{C}$ is a countably infinite set of constants/values[4], $\mathcal{R} = \{R_1, \ldots, R_n\}$ 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, and $\mathcal{I}_0$ is a database instance that represents the initial state of the data layer. $\mathcal{I}_0$ must conform to the schema $\mathcal{R}$ and *satisfy* all constraints in $\mathcal{E}$. 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 FOL query over $\mathcal{R}$ using constants from $\textsc{adom}(\mathcal{I}_0)$ whose free variables are $\vec{x}$, and z_{ij} and y_{ij} are either variables in $\vec{x}$ or constants in $\textsc{adom}(\mathcal{I}_0)$, which is the set of constants explicitly appearing in $\mathcal{I}_0$.

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 requesting user input to introduce new data into the system. Specifically, the process layer is constituted by: *(i) actions*, which are the atomic update steps on the data layer; *(ii) user inputs*, which can be requested during the execution of actions; and *(iii)* a *process* specification, which is essentially a nondeterministic program that use actions as atomic instructions. The process is specified in a declarative way by means of condition-action rules. Such rules determine, at any time point, the set of currently executable action, letting the human users decide which action has to be executed among the available alternatives. This execution pattern is known in BPM as *deferred choice*.

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 template of a *user request*), $\mathcal{A}$ is a finite set of *actions* (whose execution updates the data layer, and may involve user requests), and ϱ is a finite set of *condition-action rules* that form the specification of the overall *process* (telling 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 requested user inputs, so as to instantiate the data layer in the new state. To specify the action effects, we resort to rules that resemble tuple-generating dependencies (TGDs) (Abiteboul, Hull, and Vianu 1995), except that we allow for negation when querying the database and we replace user requests with actual values instead of considering them as labeled nulls. Note that negation is key to capturing "if-then-else" style business rules.

Formally, an *action* $\alpha \in \mathcal{A}$ is an expression $\alpha(p_1, \ldots, p_n) : \{e_1, \ldots, e_m\}$ where: $\alpha(p_1, \ldots, p_n)$ is the action *signature*, constituted by a name α and a sequence

[4] Constants are interpreted as themselves, blurring their distinction with *values*. We will use the two terms interchangeably.

$p_1, \ldots, p_n$ of *parameters*, to be substituted with values when the action is invoked, and $\{e_1, \ldots, e_m\}$, also denoted as $\text{EFFECT}(\alpha)$, is a set of *specifications of effects*, which are assumed to take place simultaneously. Each effect specification 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 (UCQ), and Q_i^- is an arbitrary FOL formula whose free variables are among those of q_i^+. Intuitively, q_i^+ selects the tuples to instantiate the effect, and Q_i^- filters away some of them. E_i is the *effect*, i.e., a set of facts for $\mathcal{R}$, which includes as terms: terms in $\text{ADOM}(\mathcal{I}_0)$, free variables of 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 so as to return a value chosen by an external user/environment when executing the action.

Each *condition-action rule* in the *process* ϱ has the form $Q \mapsto \alpha$, where α is an action in $\mathcal{A}$ and Q is a FOL query over $\mathcal{R}$, whose free variables are exactly the parameters of α, and whose other terms can be either quantified variables or constants in $\text{ADOM}(\mathcal{I}_0)$.

2.2 User Requests

We detail now the different user requests that can be used in a DCDS. First of all, each user request is bound to *range*, i.e., a set of values $\mathcal{D} \subseteq \mathcal{C}$ of constants among which the user can choose. Given a user request template $f \in \mathcal{F}$, we use $f^{\mathcal{D}}$ to show the range of f. If $\mathcal{D} = \mathcal{C}$, then we omit the range. We say that a user request template $f^{\mathcal{D}}$ is *finite-range (resp. infinite-range)*, if its range $\mathcal{D}$ is a finite (resp. infinite) set. When $\mathcal{D} = \mathcal{C}$, then the input that has to be provided by the user is completely unconstrained.

Example 1. *Finite-range requests are in fact typically used to model the user's option given a set of pre-defined alternatives. For example,* $rate^{\{\text{“1”},\text{“2”},\text{“3”},\text{“4”},\text{“5”}\}}(item)$ *models the 1-to-5 star rating for a given item. Consider now the unconstrained user request template* $password(username)$*. Each corresponding (ground) request asks the user to choose a password for the provided username. E.g.,* $password(\text{“}helen1981\text{”})$ *requests to enter a password for* “$helen1981$”.

We now differentiate between deterministic and nondeterministic requests. *Deterministic requests* are those for which the returned input value is independent from the moment in which the input is provided: whenever, along a run of the system, a request is issued twice with the same parameters, the user will return the same value. In contrast, *nondeterministic requests*, do not obey to this assumption: the user can freely return a value from the range of the request, even if the same request is issued many times with the same parameters. To disambiguate deterministic and nondeterministic request templates, we respectively put n and d as subscript of the corresponding functions.

Example 2. *Deterministic requests can be typically used to model human interaction regarding historical data. An example is the request template* $balance_d(account, date)$*, used to ask the balance present in the given account at the given date. Nondeterministic requests are instead used to account for the possibly continuously changing behavior of humans. E.g., both requests templates mentioned in Example 1 are, in fact, nondeterministic.*

2.3 Execution Semantics

The semantics of a DCDS is defined in terms of a possibly infinite transition system whose states are labeled by database instances. The resulting transition system represents all possible computations that the process layer can do on the data layer. This means that the human behaviors related to *(i)* selecting an action (among the available alternatives), *(ii)* providing an input for a deterministic request issued for the first time, *(iii)* and providing an input a nondeterministic request, have to be considered in a purely nondeterministic way, exploring all the possible alternatives (which, in the case of infinite-range user requests, could be infinitely many).

Each state of the transition system maintains the current database instance $\mathcal{I}$, which is made up of constants in $\mathcal{C}$, conforms to the schema $\mathcal{R}$ and satisfies the equality constraints in $\mathcal{E}$. In addition, it also needs to remember all user inputs provided for the deterministic requests so far, so as to make sure that deterministic requests are substituted by values in a consistent way. More precisely, we define the set of (terms representing) user requests as $\mathbb{SC} = \{f^{\mathcal{D}}(v_1, \ldots, v_n) \mid f/n \in \mathcal{F} \text{ and } \{v_1, \ldots, v_n\} \subseteq \mathcal{C}\}$, where $f^{\mathcal{D}}/n$ stands for a function $f^{\mathcal{D}}$ of arity n. Then we introduce a *user request map*, which is an assignment $\mathcal{M} : \mathbb{SC} \to \mathcal{C}$ such that $\mathcal{M}(f^{\mathcal{D}}(v_1, \ldots, v_n))$ is either undefined or belongs to $\mathcal{D}$. For convenience, we extend the map to constants by requiring that for each $c \in \text{ADOM}(\mathcal{I}_0)$, $\mathcal{M}(c) = c$. When convenient, we also consider $\mathcal{M}$ as a (functional) binary relation. With this notion at hand, we can define a state as a pair $\langle \mathcal{I}, \mathcal{M} \rangle$.

We consider now the semantics of executing an action. Let α be an action of the form $\alpha(p_1, \ldots, p_n) : \{e_1, \ldots, e_m\}$ with $e_i = q_i^+ \wedge Q_i^- \rightsquigarrow E_i$. The parameters for α are guarded by a condition-action rule $Q \mapsto \alpha$ in ϱ. Let σ be a substitution for the input parameters $p_1, \ldots, p_n$ with values taken from $\mathcal{C}$. We say that σ is *legal* for α in state $\langle \mathcal{I}, \mathcal{M} \rangle$ if $\langle p_1, \ldots, p_m \rangle \sigma \in ans(Q, \mathcal{I})$. This attests that the user can choose to apply α with paramater assignment σ (denoted $\alpha\sigma$) to the current database instance $\mathcal{I}$. To capture the semantics of this execution, we introduce the function $\text{DO}()$. The result of $\text{DO}(\mathcal{I}, \alpha\sigma)$ is a database instance obtained from the union of the results of applying the effects specifications $\text{EFFECT}(\alpha)$ (partially grounded using σ). 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 $\mathcal{I}$. Formally: $\text{DO}(\mathcal{I}, \alpha\sigma) = \bigcup_{q_i^+ \wedge Q_i^- \rightsquigarrow E_i \in \text{EFFECT}(\alpha)} \bigcup_{\theta \in ans((q_i^+ \wedge Q_i^-)\sigma, \mathcal{I})} E_i\sigma\theta$.

Notice that the set of facts returned by $\text{DO}(\mathcal{I}, \alpha\sigma)$ does not constitute a database instance. In general, such facts contain some ground user requests that need to be substituted with actual values, capturing the interaction with the user. As discussed before, the application of this substitution depends, for a given request, on whether the request is deterministic or not.

As for nondeterministic requests, we introduce the function $\text{USER-INPUT}_n(\mathcal{I}, \alpha\sigma)$, which returns a (temporary) user request map that puts in correspondence all the nondeterministic user requests contained in $\text{DO}(\mathcal{I}, \alpha\sigma)$ with corre-

sponding input values. This map is temporary because, being the requests nondeterministic, there is no need for keeping their correspondence with values in the resulting state. Technically, a *nondeterministic user input* ϑ for $\mathcal{I}$ and $\alpha\sigma$ is an assignment to each user request in $\text{DO}(\mathcal{I}, \alpha\sigma)$ of a value in the corresponding domain. Formally:

$$\vartheta = \{f_n^{\mathcal{D}}(\vec{v}) \mapsto d \mid f_n^{\mathcal{D}}(\vec{v}) \text{ occurs in } \text{DO}(\mathcal{I}, \alpha\sigma) \text{ and } d \in \mathcal{D}\}$$

We denote with $\text{USER-INPUT}_n(\mathcal{I}, \alpha\sigma)$ the set of all nondeterministic user inputs for $\mathcal{I}$ and $\alpha\sigma$.

The handling of deterministic requests differs from that of nondeterministic ones in two respects. First of all, the current user request map $\mathcal{M}$ must be taken into account, so as to apply determinism for those requests that were already issued in the past. Second, the current request $\mathcal{M}$ needs to be suitably extended when constructing the next state: for new deterministic requests (which behave nondeterministically), the corresponding results need to be stored into the resulting user request map, in addition to those already present in $\mathcal{M}$. Technically, we introduce the set $\text{USER-INPUT}_d(\mathcal{I}, \alpha\sigma, \mathcal{M})$ of *deterministic user inputs*, each of which is defined as:

$$\mathcal{M} \cup \{f_d^{\mathcal{D}}(\vec{v}) \mapsto d \mid d \in \mathcal{D}, f_d^{\mathcal{D}}(\vec{v}) \text{ occurs in } \text{DO}(\mathcal{I}, \alpha\sigma), \text{ and } \mathcal{M}(f_d^{\mathcal{D}}(\vec{v})) \text{ is undefined}\}$$

Putting everything together, we now define the notion of "possible execution step". Given two states $\langle \mathcal{I}, \mathcal{M} \rangle$ and $\langle \mathcal{I}', \mathcal{M}' \rangle$, and an action $\alpha \in \mathcal{A}$ with parameter assignment σ, we call $\langle \langle \mathcal{I}, \mathcal{M} \rangle, \alpha\sigma, \langle \mathcal{I}', \mathcal{M}' \rangle \rangle \in \text{EXEC}_{\mathcal{S}}$ *a possible execution step in the DCDS* $\mathcal{S}$, if:

1. both $\mathcal{I}$ and $\mathcal{I}'$ satisfy $\mathcal{E}$;[5]
2. σ is a legal parameter assignment for α in state $\langle \mathcal{I}, \mathcal{M} \rangle$;
3. $\mathcal{M}' \in \text{USER-INPUT}_d(\mathcal{I}, \alpha\sigma, \mathcal{M})$;
4. given $\mathcal{M}_{temp} = \mathcal{M} \cup \vartheta$, where $\vartheta \in \text{USER-INPUT}_n(\mathcal{I}, \alpha\sigma)$, we have $\mathcal{I}' = \mathcal{M}_{temp}(\text{DO}(\mathcal{I}, \alpha\sigma))$.

The execution step relation $\text{EXEC}_{\mathcal{S}}$ is the core notion for constructing the transition system that defines the execution semantics for DCDS $\mathcal{S}$. Intuitively, it is constructed as follows. We start from the initial state $s_0 = \langle \mathcal{I}_0, \emptyset \rangle$. For each condition-action rule $Q \mapsto \alpha$ in $\mathcal{P}$, we evaluate Q over $\mathcal{I}_0$, and for each resulting assignment σ, we consider the application of $\alpha\sigma$ as a possible execution step. Specifically, we compute all states s such that $\langle s_0, \alpha\sigma, s \rangle \in \text{EXEC}_{\mathcal{S}}$. We then repeat this procedure starting from each such state s. Observe that the resulting transition system is in general infinite-branching, due to the presence of infinite-range user requests, and contains infinite runs, in which each state is labeled by a different database instance.

3 Verification Formalisms

To specify dynamic properties over a DCDS, we rely on the μ-calculus (Emerson 1996), one of the most powerful temporal logics for which model checking has been investigated in the finite-state setting. Indeed, μ-calculus subsumes linear time logics such as LTL, and branching time logics such as CTL and CTL*. From a technical viewpoint, μ-calculus separates local properties, i.e., properties asserted on the current state or its immediate successors, and properties that talk about states that are arbitrarily far away from the current one. The latter are expressed using fixpoints.

Since a DCDS manipulates a relational database, it is important to have the ability of querying it inside the temporal properties. This is why we focus on first-order variants of the μ-calculus, and in particular on the two logics $\mu\mathcal{L}_A$ and $\mu\mathcal{L}_P$ introduced in (Bagheri Hariri et al. 2013). Such logics differ in the way they can combine first-order quantification with temporal operators, so as to compare values across different states of the system. Details on semantics and properties of these logics can be found in (Bagheri Hariri et al. 2013).

The first logic, $\mu\mathcal{L}_A$, requires first-order quantification to range over the active domain of the state in which it is evaluated. This is syntactically enforced by using the special predicate $\text{LIVE}(x)$, which states that x belongs to the current active domain.[6] Note that $\mu\mathcal{L}_A$ is able to implicitly "remember" values encountered in the past, even if they disappeared in intermediate states. Specifically, $\mu\mathcal{L}_A$ is defined as:

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

where Q is a possibly open FOL query, and Z is a second order predicate variable (of arity 0).

The second logic we consider, $\mu\mathcal{L}_P$, limits the active domain quantification of $\mu\mathcal{L}_A$: it is able only to quantify over values that continuously persist in the active domain along the system evolution. As soon as a value disappears from the active domain, a $\mu\mathcal{L}_P$ formula predicating over such a value trivially evaluates to false or true. This makes it impossible, for $\mu\mathcal{L}_P$, to distinguish new values from values encountered in the past, but that disappeared in intermediate states. Specifically, $\mu\mathcal{L}_P$ is defined as:

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

where Q is a possibly open FOL query, Z is a second order predicate variable, and the following assumption holds: in $\langle - \rangle(\text{LIVE}(\vec{x}) \wedge \Phi)$ and $[-](\text{LIVE}(\vec{x}) \wedge \Phi)$, variables $\vec{x}$ are exactly the free variables of Φ, with the proviso that we substitute to each bounded predicate variable Z in Φ its bounding formula $\mu Z.\Phi'$. This mechanism ensures that the formula trivializes either to false or true (depending on the use of negation) for values that disappear from the active domain.

4 Restrictions on DCDSs

The expressive power of DCDSs makes verification undecidable even for simple propositional temporal properties (such as reachability) (Bagheri Hariri et al. 2013).

In (Bagheri Hariri et al. 2013), the two interesting classes of run-bounded and state-bounded DCDS have been studied, providing decidability and undecidability results for the verification of $\mu\mathcal{L}_A$ and $\mu\mathcal{L}_P$ properties. We now reassess these results by considering the extended notion of DCDS with (mixed) user requests. In particular, we show that:

- the same decidability results apply in the extended setting;
- the sufficient syntactic conditions proposed in (Bagheri Hariri et al. 2013) to check run- and state-boundedness must be refined.

[5] This means that it is forbidden to execute an action that produces a database instance violating some constraint in $\mathcal{E}$.

[6] We also use $\text{LIVE}(x_1, \ldots, x_n) = \bigwedge_{i \in \{1,\ldots,n\}} \text{LIVE}(x_i)$.

4.1 Semantic Restrictions

We recall the restrictions of run- and state-boundedness for DCDS. Notice that these are "semantic" restrictions, which are undecidable to check (Bagheri Hariri et al. 2013).

Run-Bounded DCDSs Consider the transition system Υ for a DCDS $\mathcal{S}$. A run $\tau = s_0 s_1 s_2 \cdots$ in Υ is *bounded* if the number of values mentioned inside the databases instances associated to such states is bounded. We say that $\mathcal{S}$ is *run-bounded* if there exists a bound b such that every run in Υ is bounded by b. Notice that run boundedness does not impose any restriction on the branching of Υ. Intuitively, an unbounded run represents an execution of $\mathcal{S}$ in which infinitely many distinct values are injected by the user. We show that this restriction guarantees decidability for $\mu\mathcal{L}_A$ verification of run-bounded DCDSs with deterministic user requests.

The next theorem shows that verification of $\mu\mathcal{L}_A$ properties over run-bounded DCDSs is decidable. Since $\mu\mathcal{L}_A$ is more expressive than $\mu\mathcal{L}_P$, this result applies to $\mu\mathcal{L}_P$ as well.

Theorem 1. *Verification of $\mu\mathcal{L}_A$ properties on run-bounded DCDSs is decidable, and can be reduced to model checking of propositional μ-calculus over a finite transition system.*

The presence of both nondeterministic and deterministic user requests in the same specification was not foreseen in (Bagheri Hariri et al. 2013), and hence Theorem 1 extends the decidability result in (Bagheri Hariri et al. 2013).

State-Bounded DCDSs Given a DCDS $\mathcal{S}$ and its transition system Υ, we say that $\mathcal{S}$ is *state-bounded* if there is a finite bound b such that for each database instance $\mathcal{I}$ appearing in a state of Υ, $|\textsc{adom}(\mathcal{I})| < b$. Notice that, in contrast to run-boundedness, state-boundedness allows runs in which infinitely many distinct values are encountered. The restriction, however, forbids the possibility of accumulating such infinitely many values *within* a single state.

Interestingly, verification of $\mu\mathcal{L}_A$ properties over state-bounded DCDSs is undecidable (Bagheri Hariri et al. 2013). The situation changes for $\mu\mathcal{L}_P$, as attested by the following theorem, which extends the one in (Bagheri Hariri et al. 2013) by considering DCDSs where both deterministic and nondeterministic user requests are considered.

Theorem 2. *Verification of $\mu\mathcal{L}_P$ properties on state-bounded DCDSs is decidable, and can be reduced to model checking of propositional μ-calculus over a finite transition system.*

4.2 Syntactic Restrictions

To mitigate the fact that state- and run-boundedness are undecidable to check, in (Bagheri Hariri et al. 2013) sufficient, checkable syntactic conditions have been introduced: weak acyclicity to test run-boundedness, and GR-acyclicity to test state-boundedness. Such conditions are tested over a dependency graph constructed by considering the effect specifications contained in the actions of the DCDS. If they are met, then the DCDS is guaranteed to be run-/state-bounded, otherwise nothing can be said. We discuss how these syntactic conditions can be applied in the extended framework considered in this paper.

Weakly Acyclic DCDSs Weak acyclicity is a polynomially checkable syntactic condition borrowed from data exchange, where the notion of *weakly acyclic* TGDs (Fagin et al. 2005) is introduced to guarantee chase termination.

In (Bagheri Hariri et al. 2013), weakly acyclic DCDSs are guaranteed to be run-bounded when deterministic service calls (i.e., deterministic user requests in our extended framework) are considered. However, weak acyclicity does not work anymore when nondeterministic, infinite-range user requests come into play:

Theorem 3. *There exists a weakly acyclic (human driven) DCDS which is not run-bounded.*

On the positive side, weak acyclicity can be again used to check whether a human driven DCDS is run-bounded, when no user request is allowed to be both nondeterministic and infinite-range:

Theorem 4. *Every weakly acyclic DCDS whose nondeterministic user request templates are finite-range is run-bounded.*

GR-Acyclic DCDSs GR-acyclicity is proposed in (Bagheri Hariri et al. 2013) to ensure that only a bounded number of (new) values accumulate in the same state of the system. In essence, GR-acyclicity forbids that the actions of the DCDS realize a combination of a "generate cycle" (which potentially produces infinitely many values due to infinitely many user requests) that feeds a corresponding "recall cycle" (which accumulates all such values). The following theorem shows that this result continues to be true also in the extended framework considered here:

Theorem 5. *Every GR-acyclic DCDS is state-bounded.*

5 Example: Travel Reimbursement System

We model part of the process of reimbursing travel expenses in a university, which manages the submission of reimbursement requests by an employee, and the preliminary inspection and approval steps. A more detailed model of the complete reimbursement system is provided in (Bagheri Hariri 2013).

To keep the example simple we model a travel reimbursement request as being associated to the name of the requester, and the travel information. After a request is submitted, a monitor will check the request and will decide whether to accept or reject it. If a request is rejected, the employee needs to modify the travel information. Then, the monitor will again check 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 generated, and the system is ready to process the next travel request.

The system schema includes *(i)* $\mathsf{Status} = \langle \mathsf{status} \rangle$, which is a unary relation that keeps in a single tuple the state of the system, and can take three different values: *'readyForRequest'*, *'readyToVerify'*, and *'readyToUpdate'*, *(ii)* $\mathsf{Travel} = \langle \mathsf{eName} \rangle$, holding the name of the employee, and *(iii)* $\mathsf{TravelInfo} = \langle \mathsf{tInfo} \rangle$, holding the travel information.

The process layer includes the user requests *(i)* $\textsc{inEName}_n()$, for the employee's name; *(ii)* $\textsc{inTravelInfo}_n()$, for the travel information, and *(iii)* $\textsc{decide}_n^{\mathcal{D}}()$, with the specified range $\mathcal{D} = \{$*'accepted'*,

'*readyToUpdate*'}, which models the decision of the human monitor, returning *'accepted'* if the request is accepted, and '*readyToUpdate*' if the request needs to be updated.

The actions are as follows:

- *InitiateRequest*. When a request is initiated *(i)* a travel event is generated and the employee fills in his name, *(ii)* the employee fills in the travel information, and *(iii)* the system changes state "ready for verification":

$$\begin{aligned} &\mathsf{true} \rightsquigarrow \mathsf{Travel}(\textsc{inEName}_n()) \\ &\mathsf{true} \rightsquigarrow \mathsf{TravelInfo}(\textsc{inTravelInfo}_n()) \\ &\mathsf{true} \rightsquigarrow \mathsf{Status}(\textit{'readyToVerify'}) \end{aligned}$$

- *VerifyRequest* models the preliminary check by the monitor. Travel information are copied unchanged to the next state, which is necessary to preserve such information according to our semantics. Instead, the system status is set based on the input of a human monitor, modeled by the finite-range user request $\textsc{decide}_n^{\mathcal{D}}()$. If the monitor accepts, the next state is set to *'accepted'*, triggering action *AcceptRequest*, otherwise it is set to *'readyToUpdate'*, triggering action *UpdateRequest*:

$$\begin{aligned} \mathsf{Travel}(x) &\rightsquigarrow \mathsf{Travel}(x) \\ \mathsf{TravelInfo}(x) &\rightsquigarrow \mathsf{TravelInfo}(x) \\ \mathsf{true} &\rightsquigarrow \mathsf{Status}(\textsc{decide}_n^{\mathcal{D}}()) \end{aligned}$$

- *UpdateRequest* collects once again the travel information from the employee, moving the status to *'readyToVerify'*:

$$\begin{aligned} \mathsf{Travel}(x) &\rightsquigarrow \mathsf{Travel}(x) \\ \mathsf{true} &\rightsquigarrow \mathsf{TravelInfo}(\textsc{inTravelInfo}_n()) \\ \mathsf{true} &\rightsquigarrow \mathsf{Status}(\textit{'readyToVerify'}) \end{aligned}$$

- *AcceptRequest*:

$$\mathsf{true} \rightsquigarrow \mathsf{Status}(\textit{'readyForRequest'})$$

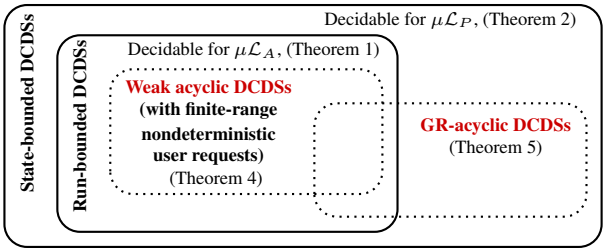
The following condition-action rules guard the actions by the current system's state:

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

The request system is not run-bounded. This is because there is no a priory bound on the number of different requests that can be submitted to the system. Moreover, it violates the finite-range requirement of nondeterministic user requests in the weak acyclicity definition. As a result, the system is also not weakly acyclic. On the other hand, it is easy to show that the system is GR-acyclic, and consequently is state-bounded. State-boundedness is also an immediate consequence of the fact that in each moment there is only one active request.

6 Conclusion

The following schema summarizes our results:



References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison Wesley Publ. Co.

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

Bagheri Hariri, B. 2013. *Borders of Decidability in Verification of Data-Centric Dynamic Systems*. Ph.D. Dissertation, Free University of Bozen - Bolzano.

Berardi, D.; Calvanese, D.; De Giacomo, G.; Hull, R.; and Mecella, M. 2005. Automatic composition of transition-based Semantic Web services with messaging. In *Proc. of VLDB*, 613–624.

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

Calvanese, D.; De Giacomo, G.; and Montali, M. 2013. Foundations of data-aware process analysis: A database theory perspective. In *Proc. of PODS*, 1–12. ACM Press.

Damaggio, E.; Hull, R.; and Vaculín, R. 2013. On the equivalence of incremental and fixpoint semantics for business artifacts with Guard-Stage-Milestone lifecycles. *Information Systems*.

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

Emerson, E. A. 1996. Automated temporal reasoning about reactive systems. In Moller, F., and Birtwistle, G., eds., *Logics for Concurrency: Structure versus Automata*, volume 1043 of *LNCS*. Springer. 41–101.

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

Harrison-Broninski, K. 2005. *Human Interactions: The Heart And Soul Of Business Process Management*. Meghan Kiffer.

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

Montali, M.; Pesic, M.; van der Aalst, W. M. P.; Chesani, F.; Mello, P.; and Storari, S. 2010. Declarative specification and verification of service choreographiess. *ACM Trans. on the Web*.

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

Slaats, T.; Mukkamala, R. R.; Hildebrandt, T. T.; and Marquard, M. 2013. Exformatics declarative case management workflows as dcr graphs. In *Proc. of BPM*.

van der Aalst, W. M. P. 2012. A decade of business process management conferences: Personal reflections on a developing discipline. In *Proc. of BPM*.