# Formal Modeling and SMT-Based Parameterized Verification of Data-Aware BPMN

Diego Calvanese[1], Silvio Ghilardi[2], Alessandro Gianola[1(✉)], Marco Montali[1], and Andrey Rivkin[1]

[1] Faculty of Computer Science, Free University of Bozen-Bolzano, Bolzano, Italy
gianola@inf.unibz.it

[2] Dipartimento di Matematica, Università degli Studi di Milano, Milan, Italy

**Abstract.** We propose DAB – a data-aware extension of BPMN where the process operates over case and persistent data (partitioned into a read-only database called catalog and a read-write database called repository). The model trades off between expressiveness and the possibility of supporting parameterized verification of safety properties on top of it. Specifically, taking inspiration from the literature on verification of artifact systems, we study verification problems where safety properties are checked irrespectively of the content of the read-only catalog, and accepting the potential presence of unboundedly many tuples in the catalog and repository. We tackle such problems using an array-based backward reachability procedure fully implemented in MCMT – a state-of-the-art array-based SMT model checker. Notably, we prove that the procedure is sound and complete for checking safety of DABs, and single out additional conditions that guarantee its termination and, in turn, show decidability of checking safety.

## 1 Introduction

In recent years, increasing attention has been given to multi-perspective models of business processes that strive to capture the interplay between the process and data dimensions [21]. Conventional finite-state verification techniques only work in this setting if data are abstractly represented, e.g., as finite sate machines [20] or process annotations [23]. If data are instead tackled in their full generality, verifying whether a process meets desired temporal properties (e.g., is safe) becomes highly undecidable, and cannot be directly attacked using conventional finite-state model checking techniques [1]. This triggered a flourishing research on the formalization and the boundaries of verifiability of data-aware processes, focusing mainly on data- and artifact-centric models [1,10]. Recent results in this stream of research [4,11] come with two strong advantages. First, they consider the relevant setting where the running process evolves a set of relations (henceforth called a data *repository*) containing data objects that may have been injected from the external environment (e.g., due to user interaction),


T. Hildebrandt et al. (Eds.): BPM 2019, LNCS 11675, pp. 157–175, 2019.
https://doi.org/10.1007/978-3-030-26619-6_12

or borrowed from a read-only relational database with constraints (henceforth called *catalog*). The repository acts as a working memory and a log for the process. Notably, it may accumulate unboundedly many tuples resulting from complex constructs in the process, such as while loops whose repeated activities insert new tuples in the repository (e.g., the applications sent by candidates in response to a job offer). The catalog stores background, contextual facts that do not change during the process execution, such as the catalog of product types, the usernames and passwords of registered customers in an order-to-cash process. In this setting, verification is studied parametrically to the catalog, so as to ensure that the process works as desired irrespectively of the specific read-only data stored therein. This is crucial to verify the process under robust conditions, also considering that actual data may not yet be available at modeling time. The second advantage of these techniques is that they tame the infinity of the state space to be verified with a symbolic approach, paving the way for the development of feasible implementations [17] or the usage of mature symbolic model checkers for infinite-state systems [4,15].

In a parallel research line more conventional, activity-centric approaches, such as the de-facto standard BPMN, have been extended towards data support, mainly focusing on modeling and enactment [6,7,18], but not on verification. At the same time, several formalisms have been brought forward to capture multi-perspective processes based on Petri nets enriched with various forms of data: from data items carried by tokens [16,22], to case data with different datatypes [8], and persistent relational data manipulated with the full power of FOL/SQL [9,19]. While these formalisms qualify well to directly capture data-aware extensions of BPMN (e.g., [7,18]), they suffer of two main limitations. On the foundational side, they require to specify the data present in the read-only storage, and only allow boundedly many tuples (with an a-priori known bound) to be stored in the read-write ones. On the applied side, they have not yet led to the development of actual verifiers.

This leads us to the main question tackled by this paper: *how to extend BPMN towards data support, guaranteeing the applicability of the existing parameterized verification techniques and the corresponding actual verifiers, so far studied only in the artifact-centric setting?* We answer this question by considering the framework of [4] and the verification of safety properties (i.e., properties that must hold in every state of the analyzed system). Our *first contribution* is a data-aware extension of BPMN called DAB, which supports case data, as well as persistent relational data partitioned into a read-only catalog and a read-write repository. Case and persistent data are used to express conditions in the process as well as task preconditions; tasks, in turn, change the values of the case variables and insert/update/delete tuples into/from the repository. The resulting framework is similar, in spirit, to the BAUML approach [12], which relies on UML and OCL instead of BPMN as we do here. While [12] approaches verification via a translation to first-order logic with time, we follow a different route, by encoding DABs into the array-based artifact system framework from [4]. Thanks to this encoding, we can effectively verify safety properties of

DABs using the MCMT (*Model Checker Modulo Theories*) model checker [13,14]. MCMT implements a backward reachability procedure that relies on state-of-the-art Satisfiability Modulo Theories (SMT) solvers, and that has been widely used to verify infinite-state *array-based systems*.

Using the encoding above, we provide our *second contribution*: we show that this backward reachability procedure is sound and complete when it comes to checking safety of DABs. In this context, soundness means that whenever the procedure terminates the returned answer is correct, whereas completeness means that if the process is unsafe then the procedure will always discover it.

The fact that the procedure is sound and complete does not guarantee that it will always terminate. This brings us to the *third and last contribution* of this paper: we introduce further conditions that, by carefully controlling the interplay between the process and data components, guarantee the termination of the procedure. Such conditions are expressed as syntactic restrictions over the DAB under study, thus providing a concrete, BPMN-grounded counterpart of the conditions imposed in [4,17]. By exploiting the encoding from DABs to array-based artifact systems, and the soundness and completeness of backward reachability, we derive that checking safety for the class of DABs satisfying these conditions is decidable.

To show that our approach goes end-to-end from theory to actual verification, we finally report some preliminary experiments demonstrating how MCMT checks safety of DABs. An extended version of this paper is available in [2]. Full proofs of our technical results and the files of the experiments with MCMT can be found in [3].

## 2 Data-Aware BPMN

We start by describing our formal model of data-aware BPMN processes (DABs). We focus here on private, single-pool processes, analyzed considering a single case, similarly to soudness analysis in workflow nets [24].[1] Incoming messages are therefore handled as pure nondeterministic events. The model combines a wide range of (block-structured) BPMN control-flow constructs with task, event-reaction, and condition logic that inspect and modify persistent as well as case data.

First, some preliminary notation. We consider a set $\mathcal{S} = \mathcal{S}_v \uplus \mathcal{S}_{id}$ of (semantic) *types*, consisting of *primitive types* $\mathcal{S}_v$ accounting for data objects, and *id types* $\mathcal{S}_{id}$ accounting for identifiers. We assume that each type $S \in \mathcal{S}$ comes with a (possibly infinite) domain $\mathbb{D}_S$, a special constant $\texttt{undef}_S \in \mathbb{D}_S$ to denote an undefined value in that domain, and a type-wise equality operator $=_S$. We omit the type and simply write `undef` and $=$ when clear from the context. We do not consider here additional type-specific predicates (such as comparison and arithmetic operators for numerical primitive types); these will be added in future

[1] The interplay among multiple cases is also crucial. The technical report [3] already contains an extension of the framework presented here, in which multiple cases are modeled and verified.

work. In the following, we simply use *typed* as a shortcut for $\mathcal{S}$*-typed*. We also denote by $\mathbb{D}$ the overall domain of objects and identifiers (i.e., the union of all domains in $\mathcal{S}$). We consider a countably infinite set $\mathcal{V}$ of typed variables. Given a variable or object $x$, we may explicitly indicate that $x$ has type $S$ by writing $x : S$. We omit types whenever clear. We indicate a possibly empty tuple $\langle x_1, \ldots, x_n \rangle$ of variables as $\vec{x}$, and write $\vec{x} \subseteq \vec{y}$ if all variables in $\vec{x}$ also appear in $\vec{y}$.

### 2.1 The Data Schema

Consistently with BPMN, we consider two main forms of data: *case data*[2], instantiated and manipulated on a per-case basis; *persistent data* (cf. *data store references* in BPMN), accounting for global data that are accessed by all cases. For simplicity, case data are defined at the whole process level, and are directly visible by all tasks and subprocesses (without requiring the specification of input-output bindings and the like).

To account for persistent data, we consider relational databases. We describe relation schemas by using the *named perspective*, i.e., by assigning a dedicated typed attribute to each component (i.e., column) of a relation schema. Also for an attribute, we use the notation $a : S$ to explicitly indicate its type.

**Definition 1.** *A* relation schema *is a pair* $R = \langle N, A \rangle$*, where: (i)* $N = R.\mathsf{name}$ *is the relation* name*; (ii)* $A = R.\mathsf{attrs}$ *is a nonempty tuple of attributes.* ◁

We call $|A|$ the *arity* of $R$. We assume that distinct relation schemas use distinct names, blurring the distinction between the two notions (i.e., $R.\mathsf{name} = R$). We also use the predicate notation $R(A)$ to represent a relation schema $\langle R, A \rangle$. A sample relation schema is $\mathit{User}(\mathit{Uid}{:}\mathsf{Int}, \mathit{Name}{:}\mathsf{String})$, where the first component represents the id-number of a user, and the second component is the string of her name.

**Data Schema.** First of all, we define the *catalog*, i.e., a read-only, persistent storage of data that is not modified during the execution of the process. Examples of the cat-relations are product types and registered customers in an order-to-cash scenario.

**Definition 2.** *A* catalog *Cat is a set of relation schemas equipped with* single-column primary key *and* foreign key *constraints. We assume that the primary key of relation schema R is always its first attribute, and denote it by* $R.\mathsf{id}$*. The type of* $R.\mathsf{id}$ *is a dedicated id type from* $\mathcal{S}_{id}$ *(i.e., no two relation schemas from Cat have the same id type). If another attribute a of R has as type an id-type* $S \in \mathcal{S}_{id}$*, then a is a foreign key referencing the relation schema whose primary key has type S.* ◁

**Example 1.** Consider a simplified example of a job hiring process. To store background information related to the process we use the catalog with relation schemas:

[2] These are called *data objects* in BPMN, but we prefer to use the term *case data* to avoid name clashes with the formal notions.

- *JobCategory*(*Jcid*:jobcatID) - storing the (ids of the) job categories available in the company (e.g., programmer, analyst);
- *User*(*Uid*:userID, *Name*:StringName, *Age*:NumAge) - storing data about users registered to the company website, who may apply to positions offered by the company.

Each case of the process is about a job. Jobs are identified by the type jobcatID. ◁

The full data schema of a BPMN process combines a catalog with: (i) a persistent data *repository*, consisting of updatable relation schemas possibly referring to the catalog; (ii) a set of *case variables*, constituting local data carried by each process case.

**Definition 3.** *A* data schema $\mathcal{D}$ *is a tuple* $\langle Cat, Repo, X \rangle$*, where (i)* $Cat = \mathcal{D}.\mathsf{cat}$ *is a* catalog*, (ii)* $Repo = \mathcal{D}.\mathsf{repo}$ *is a set of relation schemas called* repository*, and (iii)* $X = \mathcal{D}.\mathsf{cvars} \subset \mathcal{V}$ *is a finite set of typed variables called* case variables*.*

We use bold-face to distinguish case and normal variables. We call repo-relation (resp., cat-relation) a relation whose schema is in the repository (resp., catalog).

Relation schemas in the repository are not equipped with an explicit primary key, and thus they cannot reference each other, but may contain foreign keys pointing to the catalog or the case identifiers. In particular, similarly to foreign keys in the catalog, every attribute in *Repo* and case variable in $X$ whose type is an id-type $S \in \mathcal{S}_{id}$ references a corresponding cat-relation whose primary key has type $S$. It will be clear how tuples can be inserted and removed from the repository once we introduce updates.

**Example 2.** To manage key information about the applications submitted for the job hiring, the company employs a repository that consists of one relation schema:

$$Application(Jcid{:}\mathsf{jobcatID}, Uid{:}\mathsf{userID}, Score{:}\mathsf{NumScore}, Eligible{:}\mathsf{Bool})$$

NumScore is a finite-domain type containing 100 scores in the range $[1, 100]$. For readability, we use the usual comparison predicates for variables of type NumScore: this is syntactic sugar and does not require to introduce datatype predicates in our framework. Since each posted job is created using a dedicated portal, its corresponding data do not have to be stored persistently and thus can be maintained just for a given case. At the same time, some specific values have to be moved from a specific case to the repository and vice-versa. This is done by resorting to the following case variables $\mathcal{D}.\mathsf{cvars}$: *(i)* **jcid** : jobcatID references a job type from the catalog, matching the type of job associated to the case; *(ii)* **uid** : userID references the identifier of a user who is applying for the job associated to the case; *(iii)* **result** : Bool indicates whether the user identified by **uid** is eligible for winning the position or not; *(iv)* **qualif** : Bool indicates whether the user identified by **uid** qualifies for directly getting the job

(without the need of carrying out a comparative evaluation of all applicants); *(v)* **winner** : userID contains the identifier of the applicant winning the position. ◁

At runtime, a *data snapshot* of a data schema consists of three components:

- An immutable *catalog instance*, i.e., a fixed set of tuples for each relation schema contained therein, so that the primary and foreign keys are satisfied.
- An assignment mapping case variables to corresponding data objects.
- A *repository instance*, i.e., a set of tuples forming a relation for each schema contained therein, so that the foreign key constraints pointing to the catalog are satisfied. Each tuple is associated to a distinct primary key that is not explicitly accessible.

**Querying the Data Schema.** To inspect the data contained in a snapshot, we need suitable query languages operating over the data schema of that snapshot. We start by considering boolean *conditions* over (case) variables, to express choices in the process.

**Definition 4.** *A* condition *is a formula of the form* $\varphi ::= (x = y) \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2$*, where* $x$ *and* $y$ *are variables from* $\mathcal{V}$ *or constant objects from* $\mathbb{D}$*.* ◁

We make use of the standard abbreviation $\varphi_1 \vee \varphi_2 = \neg(\neg\varphi_1 \wedge \neg\varphi_2)$.

We now extend conditions to also access the data stored in the catalog and repository, and to ask for data objects subject to constraints. We consider the well-known language of unions of conjunctive queries with atomic negation, which correspond to unions of select-project-join SQL queries with table filters.

**Definition 5.** *A* conjunctive query with filters *over a data component* $\mathcal{D}$ *is a formula of the form* $Q ::= \varphi \mid R(x_1, \ldots, x_n) \mid \neg R(x_1, \ldots, x_n) \mid Q_1 \wedge Q_2$*, where* $\varphi$ *is a condition with only* atomic *negation,* $R \in \mathcal{D}.\mathsf{cat} \cup \mathcal{D}.\mathsf{repo}$ *is a relation schema of arity* $n$*, and* $x_1, \ldots, x_n$ *are variables from* $\mathcal{V}$ *(including* $\mathcal{D}.\mathsf{cvars}$*) or constant objects from* $\mathbb{D}$*. We denote by* $free(Q)$ *the set of variables occurring in* $Q$ *that are* not *case variables in* $\mathcal{D}.\mathsf{cvars}$*.* ◁

For example, a conjunctive query $JobCategory(c) \wedge c \neq \texttt{HR}$ lists all the job categories available in the company, apart from HR.

**Definition 6.** *A* guard $G$ *over a data component* $\mathcal{D}$ *is an expression of the form* $q(\vec{x}) \leftarrow \bigvee_{i=1}^{n} Q_i$*, where: (i)* $q(\vec{x})$ *is the* head *of the guard with* answer variables $\vec{x}$*; (ii) each* $Q_i$ *is a conjunctive query with filters over* $\mathcal{D}$*; (iii) for some* $i \in \{1, \ldots, n\}$*,* $\vec{x} \subseteq free(Q_i)$*. We denote by* $casevars(G) \subseteq \mathcal{D}.\mathsf{cvars}$ *the set of case variables used in* $G$*, and by* $normvars(G) = \bigcup_{i \in \{1,\ldots,n\}} free(Q_i)$ *the other variables used in* $G$*.* ◁

To distinguish guard heads from relations, we write the former in camel case, while the latter shall always begin with capital letters.

**Definition 7.** *A* guard $G$ *over a data component* $\mathcal{D}$ *is* repo-free *if none of its atoms queries a relation schema from* $\mathcal{D}.\mathsf{repo}$*.* ◁

As anticipated before, this language can be seen as a query language to retrieve data from a snapshot, or as a mechanism to constrain the combinations of data objects that can be injected into the process. E.g., guard $input(y{:}\mathsf{string}, z{:}\mathsf{string}) \rightarrow y \neq z$ returns all pairs of strings that are different from each other. Picking an answer in this (infinite) set of pairs can be seen as a (constrained) user input for $y$ and $z$.

Going beyond this guard query language (e.g., by introducing universal quantification) would hamper the soundness and completeness of SMT-based verification over the resulting DABs. We will come back to this important aspect in the conclusion.

### 2.2 Tasks, Events, and Impact on Data

We now formalize how the process can access and update the data component when executing a task or reacting to the trigger of an external event.

**The Update Logic.** We start by discussing how data maintained in a snapshot can be subject to change while executing the process.

**Definition 8.** *Given a data schema $\mathcal{D}$, an* update specification $\alpha$ *is a pair $\langle G, E\rangle$, where: (i) $G = \alpha.\mathsf{pre}$ is a guard over $\mathcal{D}$ of the form $q(\vec{x}) \leftarrow Q$, called* precondition*; (ii) $E = \alpha.\mathsf{eff}$ is an* effect rule *that changes the snapshot of $\mathcal{D}$, as described next. Each effect rule has one of the following forms:*

- **(Insert&Set)** *`INSERT` $\vec{u}$ `INTO` $R$ `AND SET` $\mathbf{x}_1 = v_1, \ldots, \mathbf{x}_n = v_n$, where: (i) $\vec{u}, \vec{v}$ are variables in $\vec{x}$ or constant objects from $\mathbb{D}$; (ii) $\vec{\mathbf{x}} \in \mathcal{D}.\mathsf{cvars}$ are distinct case variables; (iii) $R$ is a relation schema from $\mathcal{D}.\mathsf{repo}$ whose arity (and types) match $\vec{u}$. Either the `INSERT` or `SET` parts may be omitted, obtaining a pure* **Insert rule** *or* **Set rule**.
- **(Delete&Set)** *`DEL` $\vec{u}$ `FROM` $R$ `AND SET` $\mathbf{x}_1 = v_1, \ldots, \mathbf{x}_n = v_n$, where: (i) $\vec{u}, \vec{v}$ are variables in $\vec{x}$ or constant objects from $\mathbb{D}$; (ii) $\vec{\mathbf{x}} \in \mathcal{D}.\mathsf{cvars}$; (iii) $R$ is a relation schema from $\mathcal{D}.\mathsf{repo}$ whose arity (and types) match $\vec{u}$. As in the previous rule type, the `AND SET` part may be omitted, obtaining a pure (repository)* **Delete rule**.
- **(Conditional update)** *`UPDATE` $R(\vec{v})$ `IF` $\psi(\vec{u}, \vec{v})$ `THEN` $\eta_1$ `ELSE` $\eta_2$, where: (i) $\vec{u}$ is a tuple containing variables in $\vec{x}$ or constant objects from $\mathbb{D}$; (ii) $\psi$ is a repo-free guard (called* filter*); (iii) $R$ is a repo-relation schema; (iv) $\vec{v}$ is a tuple of new variables, i.e., such that $\vec{v} \cap (\vec{u} \cup \mathcal{D}.\mathsf{cvars}) = \emptyset$; (v) $\eta_i$ is either an atomic formula of the form $R(\vec{u}')$ with $\vec{u}'$ a tuple of elements from $\vec{x} \cup \mathbb{D} \cup \vec{v}$, or a nested `IF`... `THEN`... `ELSE`.* ◁

We now comment on the semantics of update specifications. An update specification $\alpha$ is executable in a given data snapshot if there is at least one answer to $\alpha.\mathsf{pre}$ in that snapshot. If so, the process executor(s) nondeterministically pick an answer, *binding* the answer variables of $\alpha.\mathsf{pre}$ to corresponding data objects in $\mathbb{D}$. This confirms the interpretation of $\alpha.\mathsf{pre}$ as a *constrained user input* when multiple bindings are available. Once a binding for the answer variables is picked,

the effect rule $\alpha.\mathsf{eff}$ is instantiated with that binding and issued. How this affects the current data snapshot depends on which effect rule is adopted.

If $\alpha.\mathsf{eff}$ is an insert&set rule, the binding is used to *simultaneously* insert a tuple in one of the repo-relations, and update some of the case variables – with the implicit assumption that those not explicitly mentioned in the `SET` part maintain their current values. Since repo-relations do not have an explicit primary key, upon insertion of a tuple $\vec{u}$ in the instance of a repo-relation $R$, a fresh primary key is generated and attached to $\vec{u}$. Two insertion semantics can then be used to characterize the insertion. Under the first semantics of *multiset insertion*, the update always succeeds in inserting $\vec{u}$ into $R$. Under the *set insertion* semantics, instead, $R$ comes not only with its implicit primary key, but also with a key constraint defined over a subset $K \subseteq R.\mathsf{attrs}$ of its attributes (by default, coinciding with $R.\mathsf{attrs}$ itself); the update is then committed only if such a key constraint is satisfied. In the case of $K = R.\mathsf{attrs}$, this implies that the update succeeds only if $\vec{u}$ is not already present in $R$, thus treating $R$ as a proper set.

**Example 3.** We continue the job hiring example, by considering update specifications of type insert&set. When a new case is created, the update specification `InsJobCat` indicates what is the job category associated to the case. Specifically, `InsJobCat.pre` selects a job category from the corresponding cat-relation, while `InsJobCat.eff` assigns the selected job category to the case variable **jcid**:

$$\texttt{InsJobCat.pre} \triangleq \mathit{getJobType}(c) \leftarrow \mathit{JobCategory}(c) \quad \texttt{InsJobCat.eff} \triangleq \texttt{SET}\ \mathbf{jcid} = c$$

When the case receives an application, the user id is picked from the corresponding *User* via the update specification `InsUser`, where:

$$\texttt{InsUser.pre} \triangleq \mathit{getUser}(u) \leftarrow \mathit{User}(u, n, a) \qquad \texttt{InsUser.eff} \triangleq \texttt{SET}\ \mathbf{uid} = u$$

A different usage of precondition, resembling a pure external choice, is the update specification `CheckQual` to handle a quick evaluation of the candidate and check whether she has such a high profile qualifying her to directly get an offer:

$$\texttt{CheckQual.pre} \triangleq \mathit{isQualified}(q : \mathsf{Bool}) \leftarrow \texttt{true} \qquad \texttt{CheckQual.eff} \triangleq \texttt{SET}\ \mathbf{qualif} = q$$

As an example of insertion rule, we consider the situation where the candidate whose id is currently stored in the case variable **uid** has not been directly judged as qualified. She is then subject to a more fine-grained evaluation via the `EvalApp` specification, resulting in a score that is then registered in the repository.

$$\begin{aligned} \texttt{EvalApp.pre} &\triangleq \mathit{getScore}(s : \mathsf{NumScore}) \leftarrow 1 \leq s \wedge s \leq 100 \\ \texttt{EvalApp.eff} &\triangleq \texttt{INSERT}\ \langle \mathbf{jcid}, \mathbf{uid}, s, \texttt{undef} \rangle\ \texttt{INTO}\ \mathit{Application} \end{aligned}$$

Here, the insertion indicates an `undef` eligibility, since it will be assessed in a consequent step of the process. Notice that with the *multiset insertion semantics*, the same user may apply multiple times for the same job (resulting multiple times as applicant). With the *set insertion semantics*, we could instead enforce the uniqueness of the application by declaring the second component (i.e., the user id) of *Application* as a key. ◁

If $\alpha$.eff is a delete&set rule, then the executability of the update is subject to the fact that the tuple $\vec{u}$ selected by the binding and to be removed from $R$, is actually present in the current instance of $R$. If so, the binding is used to *simultaneously* delete $\vec{u}$ from $R$ and update some of the case variables – with the implicit assumption that those not explicitly mentioned in the SET part maintain their current values.

Finally, a conditional update rule applies, tuple by tuple, a bulk operation over the content of $R$. Each tuple in $R$ is reviewed. If the tuple passes the filter associated to the rule, it is updated according to the THEN part, otherwise according to the ELSE part.

**Example 4.** Continuing with our running example, we now consider the update specification MarkE handling the situation where no candidate has been directly considered as qualified, and so the eligibility of all received (and evaluated) applications has to be assessed. Here we consider that each application is eligible if and only if its evaluation resulted in a score greater than 80. Technically, MarkE.pre is a true precondition, and:

$$\begin{aligned}\texttt{MarkE.eff} \triangleq\ & \texttt{UPDATE}\ \mathit{Application}(c,u,s,e)\\ & \texttt{IF}\ s > 80\ \texttt{THEN}\ \mathit{Application}(c,u,s,\texttt{true})\ \texttt{ELSE}\ \mathit{Application}(c,u,s,\texttt{false})\end{aligned}$$

If there is at least one eligible candidate, she can be selected as a winner using the SelWinner update specification, which deletes the selected winner tuple from *Application*, and transfers its content to the corresponding case variables (also ensuring that the **winner** case variable is set to the applicant id). Technically:

$$\begin{aligned}\texttt{SelWinner.pre} &\triangleq \mathit{getWinner}(c,u,s,e) \leftarrow \mathit{Application}(c,u,s,e) \wedge e = \texttt{true}\\ \texttt{SelWinner.eff} &\triangleq \texttt{DEL}\ \langle c,u,s,e\rangle\ \texttt{FROM}\ \mathit{Application}\\ &\quad\ \texttt{AND SET}\ \mathbf{jcid} = c, \mathbf{uid} = u, \mathbf{winner} = u, \mathbf{result} = e, \mathbf{qualif} = \texttt{false}\end{aligned}$$

Deleting the tuple is useful when the selected winner may refuse the job, and hence should not be considered again if a new winner selection is done. To keep such tuple in the repository, one would just need to remove the DEL part from SelWinner.eff. ◁

**The Task/Event Logic.** We now substantiate how the update logic is used to specify the task/event logic within a DAB process. The first important observation, not related to our specific approach, but inherently present whenever the process control flow is enriched with relational data, is that update effects manipulating the repository must be executed in an atomic, non-interruptible way. This is essential to ensure that insertions/deletions into/from the repository are applied on the same data snapshot where the precondition is checked. Breaking simultaneity would lead to nondeterministic interleaving with other update specifications potentially operating over the same portion of the repository. This is why we consider two types of task: *atomic* and *nonatomic*.

Each atomic task/catching event is associated to a corresponding update specification. In the case of tasks, the specification precondition indicates under which circumstances the task can be enacted, and the specification effect how enacting the task impacts on the underlying data snapshot. In the case of events,

the specification precondition constrains the data payload that comes with the event (possibly depending on the data snapshot, which is global and therefore accessible also by an external event trigger). The effect dictates how reacting to a triggered event impacts on the data snapshot.

This is realized according to the following lifecycle. The task/event is initially `idle`, i.e., quiescent. When the progression of a case reaches an `idle` task/event, such a task/event becomes `enabled`. An `enabled` task/event may nondeterministically fire depending on the choice of the process executor(s). Upon firing, a binding satisfying the precondition of the update specification associated to the task/event is selected, consequently grounding and applying the corresponding effect. At the same time, the lifecycle moves from `enabled` to `compl`. Finally, a `compl` task/event triggers the progression of its case depending on the process-control flow, simultaneously bringing the task/event back to the `idle` state (which would then make it possible for the task to be executed again later, if the process control-flow dictates so).

The lifecycle of a nonatomic task diverges in two crucial respects. First, upon firing it moves from `enabled` to `active`, and later on nondeterministically from `active` to `compl` (thus having a duration). The precondition of its update specification is checked and bound to one of the available answers when the task becomes `active`, while the corresponding effect is applied when the task becomes `compl`. Since these two transitions occur asynchronously, to avoid the aforementioned transactional issues we assume that the task effect operates only on case variables (and not on the repository).

### 2.3 Process Schema

A process schema consists of a block-structured BPMN diagram, enriched with conditions and update effects expressed over a given data schema, according to what described in the previous sections. As for the control flow, we consider a wide range of block-structured patterns compliant with the standard. We focus on private BPMN processes, thereby handling incoming messages in a pure nondeterministic way. So we do for timer events, nondeterministically accounting for their expiration without entering into their metric temporal semantics. Focusing on block-structured components helps us in obtaining a direct, execution semantics, and a consequent modular and clean translation of various BPMN constructs (including boundary events and exception handling). However, it is important to stress that our approach would seamlessly work also for non-structured processes where each case introduces boundedly many tokens.

As usual, blocks are recursively decomposed into sub-blocks, the leaves being task or empty blocks. Depending on its type, a block may come with one or more nested blocks, and be associated with other elements, such as conditions, types of the involved events, and the like. We consider a wide range of blocks, covering the basic, flow, and exception handling patterns in [2]. Figure 1 gives an idea about what is covered by our approach. With these blocks at hand, we finally obtain the full definition of a DAB.

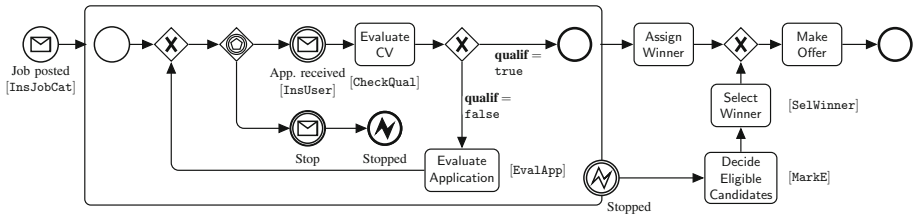


**Fig. 1.** The job hiring process. Elements in squared brackets attach the update specifications in Examples 3 and 4 to corresponding tasks/events.

**Definition 9.** *A DAB $\mathcal{M}$ is a pair $\langle \mathcal{D}, \mathcal{P} \rangle$ where $\mathcal{D}$ is a data schema, and $\mathcal{P}$ is a root* process block *such that all conditions and update effects attached to $\mathcal{P}$ and its descendant blocks are expressed over $\mathcal{D}$.* ◁

**Example 5.** The full hiring job process is shown in Fig. 1, using the update effects described in Examples 3 and 4. Intuitively, the process works as follows. A case is created when a job is posted, and enters into a looping subprocess where it expects candidates to apply. Specifically, the case waits for an incoming application, or for an external message signalling that the hiring has to be stopped (e.g., because too much time has passed from the posting). Whenever an application is received, the CV of the candidate is evaluated, with two possible outcomes. The first outcome indicates that the candidate directly qualifies for the position, hence no further applications should be considered. In this case, the process continues by declaring the candidate as winner, and making an offer to her. The second outcome of the CV evaluation is instead that the candidate does not directly qualify. A more detailed evaluation is then carried out, assigning a score to the application and storing the outcome into the process repository, then waiting for additional applications to come. When the application management subprocess is stopped (which we model through an error so as to test various types of blocks in the experiments reported in Sect. 3.3), the applications present in the repository are all processed in parallel, declaring which candidates are eligible and which not depending on their scores. Among the eligible ones, a winner is then selected, making an offer to her. We implicitly assume here that at least one applicant is eligible, but we can easily extend the DAB to account also for the case where no application is eligible. ◁

As customary, each block has a lifecycle indicating its current state, and how the state may evolve depending on the specific semantics of the block, and the evolution of its inner blocks. In Sect. 2.2 we have already characterized the lifecycle of tasks and catch events. For the other blocks, we continue to use the standard states `idle`, `enabled`, `active`, and `compl`. We use the very same rules of execution described in the BPMN standard to regulate the progression of blocks through such states, taking advantage from the fact that, being the process block-structured, only one instance of a block can be enabled/active at a

given time for a given case. As an example, we describe the lifecycle of a sequence block S with nested blocks $B_1$ and $B_2$ (considering that the transitions of S from `idle` to `enabled` and from `compl` back to `idle` are inductively regulated by its parent block): *(i)* if S is `enabled`, then it becomes `active`, inducing a transition of $B_1$ from `idle` to `enabled`; *(ii)* if $B_1$ is `compl`, then it becomes `idle`, inducing a transition of $B_2$ from `idle` to `enabled`; *(iii)* if $B_2$ is `compl`, then it becomes `idle`, inducing S to move from `active` to `compl`. Analogously for other block types.

### 2.4 Execution Semantics

We intuitively describe the execution semantics of a case over DAB $\mathcal{M} = \langle \mathcal{D}, \mathcal{P} \rangle$, using the update/task logic and progression rules of blocks as a basis. Upon execution, each state of $\mathcal{M}$ is characterized by an $\mathcal{M}$-*snapshot*, which consists of a data snapshot of $\mathcal{D}$ (cf. Sect. 2.1) and an assignment mapping each block in $\mathcal{P}$ to its current lifecycle state. Initially, the data snapshot fixes the immutable content of the catalog $\mathcal{D}$.`cat`, while the repository instance is empty, the case assignment is initialized to all `undef`, and the control assignment assigns to all blocks in $\mathcal{P}$ the `idle` state, with the exception of $\mathcal{P}$ itself, which is `enabled`. At each moment in time, the $\mathcal{M}$-snapshot is then evolved by nondeterministically evolving the case through one of the executable steps in the process, depending on the current $\mathcal{M}$-snapshot. If the execution step is about the progression of the case inside the process control-flow, then the control assignment is updated. If instead the execution step is about the application of some update effect, the new $\mathcal{M}$-snapshot is obtained according to Sect. 2.2.

## 3 Parameterized Verification of Safety Properties

We now focus on parameterized verification of DABs using the framework of array-based artifact systems of [4], which bridges the gap between SMT-based model checking of array-based systems [13,14], and data- and artifact-centric processes [10,11].

### 3.1 Array-Based Artifact Systems and Safety Checking

In general terms, an array-based system describes the evolution of array data structures of unbounded size. The logical representation of an array relies on a theory with two types of sorts: one for the array indexes, and the other for the elements stored in the array cells. Since the content of an array changes

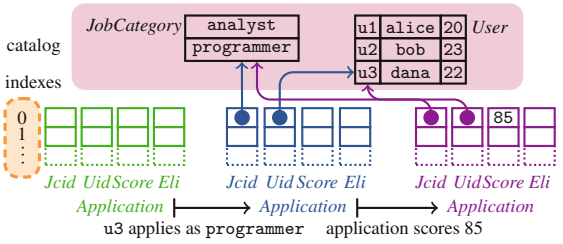


**Fig. 2.** Array-based representation of the *Application* repo-relation of Example 2, and manipulation of a job application. Empty cells contain `undef`.

over time, it is represented by a *function* variable, called *array state variable*, which defines for each index what is the value stored in the corresponding cell. Its interpretation changes when moving from one state to another, reflecting the intended manipulation of the array. Hence, an array-based system working over array $a$ is defined through: *(i)* a state formula $I(a)$ describing the *initial configuration(s)* of $a$; *(ii)* a formula $\tau(a, a')$ describing the *transitions* that transforms the content of the array from $a$ to $a'$. By suitably using logical operators, $\tau$ can express in a single formula a set of different updates over $a$. One of the most studied verification problems is that of *unsafety verification*: it checks whether the evolution induced by $\tau$ over $a$ starting from a configuration in $I(a)$ eventually *reaches* an *unsafe* configuration described by a state formula $K(a)$. Several mature model checkers exist to ascertain (un)safety of these type of systems. In our work, we rely on MCMT [15].

In [4], we have extended array-based systems towards an array-based version of the artifact-centric approach, considering in particular the sophisticated model in [17]. In the resulting formalism, called RAS, a *relational* artifact system accesses a read-only database with keys and foreign keys (cf. our DAB catalog). In addition, the RAS operates over a set of relations possibly containing unboundedly many updatable entries (cf. our DAB repository). Figure 2 gives an intuitive idea of how this type of system looks like, using the catalog and repository from Example 2. The catalog is treated as a rich, background theory, which can be considered as a more sophisticated version of the element sort in basic array systems. Each repo-relation is treated as a set of arrays, where each array accounts for one component of the corresponding repo-relation. A tuple in the relation is reconstructed by accessing all such arrays with the same index.

Verification of RAS is also tamed in [4], checking whether there exists an instance of the read-only database so that the RAS can reach an unsafe configuration. This is approached by extending the original symbolic *backward reachability* procedure for unsafety verification of array-based systems to the case of RAS. The procedure starts from the undesired states captured by $K(a)$, and iteratively computes so-called *preimages*, i.e., logical formulae symbolically describing those states that, through consecutive applications of $\tau$, directly or indirectly reach configurations satisfying $K(a)$. A preimage formula may contain existentially quantified variables referring to data objects in the catalog. MCMT employs novel quantifier elimination techniques [4,5] to suitably remove such variables, and obtain a state formula that describes the predecessor states. A fixpoint check is delegated to a state-of-the-art SMT solver, so as to check whether the computed predecessors all coincide with already iteratively computed states. If no new state has been produced, the procedure stops by emitting *safe*. Otherwise, the preimage formula is conjoined with $I(a)$ and sent to the SMT solver to check for satisfiability: if so, then the computed preimage states intersect the initial ones, and the procedure stops by emitting *unsafe* as a verdict; if not, new preimages are iteratively computed and the steps above are repeated. This procedure is shown to be sound and complete for checking unsafety of RAS in [4], where we also single out subclasses of RAS with decidable unsafety, and for which MCMT

is ensured to terminate. One class is that of RAS operating over arrays whose maximum size is bounded a-priori. This type of RAS is called SAS (for *simple* artifact system). All in all, this framework provides a natural foundational and practical basis to formally analyze DABs, which we tackle next.

### 3.2 Verification Problems for DABs

First, we need a language to express unsafety properties over a DAB $\mathcal{M} = \langle \mathcal{D}, \mathcal{P} \rangle$. Properties are expressed in a fragment of the *guard* language of Definition 6 that queries repo-relations and case variables as well as the cat-relations that tuples from repo-relations or case variables refer to. Properties also query the control state of $\mathcal{P}$. This is done by implicitly extending $\mathcal{D}$ with additional, special case *control variables* that refer to the lifecycle states of the blocks in $\mathcal{P}$ (where a block named $B$ gets variable **Blifecycle**). Given a snapshot, each such variable is assigned to the lifecycle state of the corresponding block (i.e., `idle`, `enabled`, and the like).

**Definition 10.** *A* property *over* $\mathcal{M} = \langle \mathcal{D}, \mathcal{P} \rangle$ *is a guard* $G$ *over* $\mathcal{D}$ *and the control variables of* $\mathcal{P}$*, such that every non-case variable in* $G$ *also appears in a relational atom* $R(y_1, \ldots, y_n)$*, where either* $R$ *is a repo-relation, or* $R$ *is a cat-relation and* $y_1 \in \mathcal{D}.\mathsf{cvars}$.

**Example 6.** By naming *HP* the root process block of Fig. 1, the property (**HPlifecycle** = `completed`) checks whether some case of the process can terminate. This property is *unsafe* for our hiring process, since there is at least one way to evolve the process from the start to the end. Since DAB processes are block-structured, this is enough to ascertain that the hiring process is *sound*. Property **EvalApplifecycle** = `completed` $\wedge$ *Application*$(j, u, s,$ `true`$) \wedge s > 100$ (the **5**th **safe** in Sect. 3.3) describes instead the undesired situation where, after the evaluation of an application, there exists an applicant with score greater than 100. The hiring process is *safe* w.r.t. this property. ◁

We study unsafety of these properties by considering the general case, and also the one where the repository can store only boundedly many tuples, with a fixed bound. In the latter case, we call the DAB *repo-bounded*.

**Translating DABs into Array-Based Artifact Systems.** Given an unsafety verification problem over a DAB $\mathcal{M} = \langle \mathcal{D}, \mathcal{P} \rangle$, we encode it as a corresponding unsafety verification problem over a RAS that reconstructs the execution semantics of $\mathcal{M}$. We only provide here the main intuitions behind the translation, which is fully addressed in [3]. In the translation, $\mathcal{D}.\mathsf{cat}$ and $\mathcal{D}.\mathsf{cvars}$ are mapped into their corresponding abstractions in RAS (namely, the RAS read-only database and artifact variables, respectively). $\mathcal{D}.\mathsf{repo}$ is instead encoded using the intuition of Fig. 2: for each $R \in \mathcal{D}.\mathsf{repo}$ and each attribute $a \in R.\mathsf{attrs}$, a dedicated array is introduced. Array indexes represent (implicit) identifiers of tuples in $R$, in line with our repository model. To retrieve a tuple from $R$, one just needs to access the arrays corresponding to the different attributes of $R$ with the same

index. Finally, case variables are represented using (bounded) arrays of size 1. On top of these data structures, $\mathcal{P}$ is translated into a RAS transition formula that exactly reconstructs the execution semantics of the blocks in $\mathcal{P}$.

With this translation in place, we define BackReach as the backward reachability procedure that: (1) takes as input *(i)* a DAB $\mathcal{M}$, *(ii)* a property $\varphi$ to be verified, *(iii)* a boolean indicating whether $\mathcal{M}$ is repo-bounded or not (in the first case, also providing the value of the bound), and *(iv)* a boolean indicating whether the semantics for insertion is set or multiset; (2) translates $\mathcal{M}$ into a corresponding RAS $\mathcal{M}'$, and $\varphi$ into a corresponding property $\varphi'$ over $\mathcal{M}'$ (Definition 10 ensures that $\varphi'$ is indeed a RAS state formula); (3) returns the result produced by the MCMT backward reachability procedure (cf. Sect. 3.1) on $\mathcal{M}'$ and $\varphi'$.

### 3.3 Verification Results

Using the DAB-to-RAS translation and the results in [4], we provide now our main technical contributions. First: DABs can be correctly verified using BackReach.

**Theorem 1.** BackReach *is sound and complete for checking unsafety of DABs that use the multiset or set insertion semantics.* ◁

Soundness tell us that when BackReach terminates, it produces a correct answer, while completeness guarantees that whenever a DAB is unsafe with respect to a property, then BackReach detects this. Hence, BackReach is a semi-decision procedure for unsafety.

We study additional conditions on the input DAB to guarantee termination of BackReach, then becoming a full decision procedure for unsafety. The first, unavoidable condition, in line with [4,17], is that the catalog must be *acyclic*: its foreign keys cannot form referential cycles (where a table directly or indirectly refers to itself).

**Theorem 2.** BackReach *terminates when verifying properties over repo-bounded and acyclic DABs using the multiset or set insertion semantics.* ◁

If the input DAB is not repo-bounded, acyclicity of the catalog is not enough: termination requires to carefully control the interplay between the different components of the DAB. While the conditions required by the technical proofs are quite difficult to grasp at the syntactic level, they can be intuitively understood using the following *locality principle*: whenever the progression of the DAB depends on the repository, it does so only via a single entry in one of its relations. Hence, direct/indirect comparisons and joins of distinct tuples within the same or different repo-relations cannot be used. To avoid indirect comparisons/joins, queries cannot mix case variables and repo-relations.

Thus, set insertions cannot be supported, since by definition they require to compare tuples in the same relation. The next definition is instrumental to enforce locality.

**Definition 11.** *A* guard $G \triangleq q(\vec{x}) \leftarrow \bigvee_{i=1}^{n} Q_i$ *over data component* $\mathcal{D}$ *is* separated *if* $normvars(Q_i) \cap normvars(Q_j) = \emptyset$ *for every* $i \neq j$*, and each* $Q_i$ *is of the form* $\chi \wedge R(\vec{y}) \wedge \xi$ *(with* $\chi$*,* $R(\vec{y})$*, and* $\xi$ *optional), where: (i)* $\chi$ *is a conjunctive query with filters only over* $\mathcal{D}.\mathsf{cat}$*, and that can employ case variables; (ii)* $R \in \mathcal{D}.\mathsf{repo}$ *is a repo-relation schema; (iii)* $\vec{y}$ *is a tuple of variables and/or constant objects in* $\mathbb{D}$*, such that* $\vec{y} \cap \mathcal{D}.\mathsf{cvars} = \emptyset$*, and* $normvars(\chi) \cap \vec{y} = \emptyset$*; (iv)* $\xi$ *is a conjunctive query with filters over* $\mathcal{D}.\mathsf{cat}$ *only, that possibly mentions variables in* $\vec{y}$ *but does* not *include any case variable, and such that* $normvars(\chi) \cap normvars(\xi) = \emptyset$*. A* property *is* separated *if it is so as a guard.* ◁

A separated guard is made of two isolated parts: a part $\chi$ inspecting case variables and their relationship with the catalog, and a part $R(\vec{y}) \wedge \xi$ retrieving a single tuple $\vec{y}$ from some repo-relation $R$, possibly filtering it through inspection of the catalog via $\xi$.

**Example 7.** Consider the refinement $\texttt{EvalApp.pre} \triangleq GetScore(s : \mathsf{Num}\ \mathsf{Score}) \leftarrow \xi \wedge \chi$ of the guard `EvalApp.pre` from Example 3, where $\chi := User(\mathbf{uid}, name, age)$ checks if the variables $\langle \mathbf{uid}, name, age \rangle$ form a tuple in $User$, and $\xi := 1 \leq s \wedge s \leq 100$. This guard is separated since $\chi$ and $\xi$ match the requirements of the previous definition. ◁

**Theorem 3.** *Let* $\mathcal{M}$ *be an acyclic DAB that uses the multiset insertion semantics, and is such that for each update specification* $\mathfrak{u}$ *of* $\mathcal{M}$*, the following holds: 1. If* $\mathfrak{u}.\mathsf{eff}$ *is an* insert&set *rule (with explicit* `INSERT` *part),* $\mathfrak{u}.\mathsf{pre}$ *is* repo-free*; 2. If* $\mathfrak{u}.\mathsf{eff}$ *is a* set *rule (with no* `INSERT` *part), then either (i)* $\mathfrak{u}.\mathsf{pre}$ *is* repo-free*, or (ii)* $\mathfrak{u}.\mathsf{pre}$ *is* separated *and* all *case variables appear in the* `SET` *part of* $\mathfrak{u}.\mathsf{eff}$*; 3. If* $\mathfrak{u}.\mathsf{eff}$ *is a* delete&set *rule, then* $\mathfrak{u}.\mathsf{pre}$ *is* separated *and* all *case variables appear in the* `SET` *part of* $\mathfrak{u}.\mathsf{eff}$*; 4. If* $\mathfrak{u}.\mathsf{eff}$ *is a* conditional update *rule, then* $\mathfrak{u}.\mathsf{pre}$ *is* repo-free *and* boolean*, so that* $\mathfrak{u}.\mathsf{eff}$ *only makes use of the new variables introduced in its* `UPDATE` *part (as well as constant objects in* $\mathbb{D}$*). Then,* BackReach *terminates when verifying separated properties over* $\mathcal{M}$*.*

The conditions of Theorem 3 represent a concrete, BPMN-like counterpart of the abstract ones used in [17] and [4] towards decidability.

Specifically, Theorem 3 employs: *(i)* repo-freedom, and *(ii)* separation with the manipulation of *all* case variables at once. We intuitively explain how these conditions substantiate the locality principle. Overall, the main difficulty is that case variables may be loaded with data objects extracted from the repository. Hence, the usage of a case variable may mask an underlying reference to a tuple component stored in some repo-relation. Given this, locality demands that no two case variables can simultaneously hold data objects coming from different tuples in the repository. At the beginning, this is trivially true, since all case variables are undefined. A safe snapshot guaranteeing this condition continues to stay so after an insertion of the form mentioned in point 1 of Theorem 3: a repo-free precondition ensures that the repository is not queried at all, and hence trivially preserves locality. Locality may be easily destroyed by arbitrary set or

delete&set rules whose precondition accesses the repository. Three aspects have to be considered to avoid this. First, we have to guarantee that the precondition does not mix case variables and repo-relations: Theorem 3 does so thanks to separation. Second, we have to avoid that when the precondition retrieves objects from the repository, it extracts them from different tuples therein: this is again guaranteed by separation, since only one tuple is extracted. A third, subtle situation that would destroy locality is the one in which the objects retrieved from (the same tuple in) the repository are only used to assign *a proper subset* of the case variables: the other case variables could in fact still hold objects previously retrieved from a *different* tuple in the repository. Theorem 3 guarantees that this never happens by imposing that, upon a set or delete&set operation, *all* case variables are involved in the assignment. Those case variables that get objects extracted from the repository are then guaranteed to all implicitly point to the same repository tuple retrieved by the separated precondition.

**Example 8.** The hiring DAB obeys to all conditions in Theorem 3, ensuring termination of BackReach. E.g., `EvalApp` in Example 3 matches point 1: its precondition is repo-free. `SelWinner` from the same example matches point 3: `SelWinner.pre` is trivially separated and *all* case variables appear in the `SET` part of `SelWinner.eff`. ◁

**First mcmt Experiments.** We have encoded the job hiring DAB in MCMT, systematically applying the translation rules recalled in Sect. 3.2, and fully spelled out in [3] when proving the main theorems of Sect. 3.3. We have then checked the DAB for process termination (which took 0.43 s), and against five safe and five unsafe properties. E.g., the **1**st **unsafe** property describes the desired situation in which, after having evaluated an application (i.e., EvalApp is completed), there exists at least an applicant with a score greater than 0. Formally: $\textbf{EvalApplifecycle} = \texttt{completed} \wedge \mathit{Application}(j, u, s, e) \wedge s > 0$. The **4**th **safe** property represents the situation where a winner has been selected after the deadline (i.e., SelWin is completed), but the case variable **result** indicates that the winner is not eligible. Formally: $\textbf{SelWinlifecycle} = \texttt{completed} \wedge \textbf{result} = \texttt{false}$. MCMT returns `SAFE`, witnessing that this configuration is not reachable from the initial one. The table on the right summarizes the obtained, encouraging results (time in seconds). The MCMT specifications with all the checked properties (and their intuitive interpretation) are available in [3]. All tests are directly reproducible.

| | *prop.* | *time(s)* |
|---|---|---|
| **safe** | **1** | 0.20 |
| | **2** | 5.85 |
| | **3** | 3.56 |
| | **4** | 0.03 |
| | **5** | 0.27 |
| **unsafe** | **1** | 0.18 |
| | **2** | 1.17 |
| | **3** | 4.45 |
| | **4** | 1.43 |
| | **5** | 1.14 |

## 4 Conclusion

We have introduced a data-aware extension of BPMN, called DAB, balancing between expressiveness and verifiability. We have shown that parameterized safety problems over DABs can be tackled by array-based SMT techniques, and

in particular the backward reachability procedure of the MCMT model checker. We have then identified classes of DABs with conditions that control how the process operates over data and ensure termination of backward reachability. Methodologically, such conditions can be seen as modeling principles for data-aware process designers who aim at making their processes verifiable. Whether these conditions apply to real-life processes is an open question that calls for novel research in their empirical validation on real-world scenarios, and in the definition of guidelines to refactor arbitrary DABs into verifiable ones.

From the foundational perspective we want to equip DABs with datatypes and arithmetic operators, widely supported by SMT solvers. We also want to attack the main limitation of our approach, namely that guards and conditions are existential formulae, and the only (restricted) form of universal quantification in the update language is that of conditional updates. From the experimental point of view, the initial results obtained in this paper and [4] indicate that the approach is promising. We intend to fully automate the translation from DABs to array-based systems, and benchmark the performance of verifiers for data-aware processes, starting from the examples in [17]: they are inspired by reference BPMN processes, and consequently should be easily encoded as DABs.

## References

1. Calvanese, D., De Giacomo, G., Montali, M.: Foundations of data aware process analysis: a database theory perspective. In: Proceedings of the PODS, pp. 1–12 (2013)
2. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Formal modeling and SMT-based parameterized verification of data-aware BPMN (extended version). Technical report arXiv:1906.07811 (2019)
3. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Formal modeling and SMT-based parameterized verification of multi-case data-aware BPMN. Technical report arXiv:1905.12991 (2019)
4. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: From model completeness to verification of data aware processes. In: Lutz, C., Sattler, U., Tinelli, C., Turhan, A.Y., Wolter, F. (eds.) Description Logic, Theory Combination, and All That. LNCS, vol. 11560, pp. 212–239. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-22102-7_10
5. Calvanese, D., Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Model completeness, covers and superposition. In: Automated Deduction - CADE 27, LNCS (LNAI), vol. 11716. Springer, Cham (2019)
6. Combi, C., Oliboni, B., Weske, M., Zerbato, F.: Conceptual modeling of processes and data: connecting different perspectives. In: Trujillo, J., et al. (eds.) ER 2018. LNCS, vol. 11157, pp. 236–250. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00847-5_18
7. De Giacomo, G., Oriol, X., Estañol, M., Teniente, E.: Linking data and BPMN processes to achieve executable models. In: Dubois, E., Pohl, K. (eds.) CAiSE 2017. LNCS, vol. 10253, pp. 612–628. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-59536-8_38

8. de Leoni, M., Felli, P., Montali, M.: A holistic approach for soundness verification of decision-aware process models. In: Trujillo, J., et al. (eds.) ER 2018. LNCS, vol. 11157, pp. 219–235. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-00847-5_17
9. De Masellis, R., Di Francescomarino, C., Ghidini, C., Montali, M., Tessaris, S.: Add data into business process verification: bridging the gap between theory and practice. In: Proceedings of AAAI, pp. 1091–1099. AAAI Press (2017)
10. Deutsch, A., Hull, R., Li, Y., Vianu, V.: Automatic verification of database-centric systems. SIGLOG News **5**(2), 37–56 (2018)
11. Deutsch, A., Li, Y., Vianu, V.: Verification of hierarchical artifact systems. In: Proceedings of the PODS, pp. 179–194 (2016)
12. Estañol, M., Sancho, M.-R., Teniente, E.: Verification and validation of UML artifact-centric business process models. In: Zdravkovic, J., Kirikova, M., Johannesson, P. (eds.) CAiSE 2015. LNCS, vol. 9097, pp. 434–449. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19069-3_27
13. Ghilardi, S., Nicolini, E., Ranise, S., Zucchelli, D.: Towards SMT model checking of array-based systems. In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 67–82. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-71070-7_6
14. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: termination and invariant synthesis. Log. Methods Comput. Sci. **6**(4), 1–48 (2010)
15. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 22–29. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_3
16. Lasota, S.: Decidability border for Petri nets with data: WQO dichotomy conjecture. In: Kordon, F., Moldt, D. (eds.) PETRI NETS 2016. LNCS, vol. 9698, pp. 20–36. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-39086-4_3
17. Li, Y., Deutsch, A., Vianu, V.: VERIFAS: a practical verifier for artifact systems. PVLDB **11**(3), 283–296 (2017)
18. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: Daniel, F., Wang, J., Weber, B. (eds.) BPM 2013. LNCS, vol. 8094, pp. 171–186. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40176-3_14
19. Montali, M., Rivkin, A.: DB-Nets: on the marriage of colored Petri Nets and relational databases. ToPNoC **28**(4), 91–118 (2017)
20. Müller, D., Reichert, M., Herbst, J.: Data-driven modeling and coordination of large process structures. In: Meersman, R., Tari, Z. (eds.) OTM 2007. LNCS, vol. 4803, pp. 131–149. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-76848-7_10
21. Reichert, M.: Process and data: two sides of the same coin? In: Meersman, R., et al. (eds.) OTM 2012. LNCS, vol. 7565, pp. 2–19. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33606-5_2
22. Rosa-Velardo, F., de Frutos-Escrig, D.: Decidability and complexity of Petri nets with unordered data. Theor. Comput. Sci. **412**(34), 4439–4451 (2011)
23. Sidorova, N., Stahl, C., Trcka, N.: Soundness verification for conceptual workflow nets with data: early detection of errors with the most precision possible. Inf. Syst. **36**(7), 1026–1043 (2011)
24. Aalst, W.M.P.: Verification of workflow nets. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 407–426. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63139-9_48