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

(Extended Version)

Diego Calvanese[1] , Silvio Ghilardi[2], Alessandro Gianola[1],
Marco Montali[1], Andrey Rivkin[1]

[1]Faculty of Computer Science, Free University of Bozen-Bolzano (Italy)
[2]Dipartimento di Matematica, Università degli Studi di Milano (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 [25]. Conventional finite-state verification techniques only work in this setting if data are abstractly represented, e.g., as finite sate machines [24] or process annotations [28]. 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 [2]. 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 [2,12]. Recent results in this stream of research [14,5] 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), 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 main advantage of these techniques is that they tame the infinity of the state space to be verified with a symbolic approach, in turn paving the way for the development of feasible implementations [21,13], or for the exploitation of state-of-the-art symbolic model checkers for infinite-state systems [5,19].

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 conceptual modeling and enactment [22,9,7], 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 locally carried by tokens [27,20], to case data with different data types [10], and/or persistent relational data manipulated with the full power of FOL/SQL [11,23]. While these formalisms qualify well to directly capture data-aware extensions of BPMN (e.g., [22,9]), 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 do not lend themselves to be symbolically verified and 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 [5] and the verification of safety properties (i.e., properties that must hold in every state of the analyzed system). Specifically, 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 [16], which relies on UML and OCL instead of BPMN as we do here. While [16] 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 [5]. Thanks to this encoding, we can effectively verify safety properties of DABs using the well-established MCMT (*Model Checker Modulo Theories*) model checker [17,18]. MCMT implements a symbolic backward reachability procedure that relies on state-of-the-art Satisfiability Modulo Theories (SMT) solvers, and that has been widely employed 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 [21,5]. 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.

This paper is the extended version of [3]. Full proofs of our technical results and the files of the experiments with MCMT can be found in [4].

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 [29].[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. Given the aim of our approach, recall that if something is not supported in the language, it is because it would hamper soundness and completeness of SMT-based (parameterized) verification.

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 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 or irrelevant. We compactly indicate a possibly empty tuple $\langle x_1, \ldots, x_n \rangle$ of variables as $\vec{x}$, and with slight abuse of notation, we write $\vec{x} \subseteq \vec{y}$ if all variables in $\vec{x}$ also appear in $\vec{y}$.

[1] The interplay among multiple cases is also crucial. The technical report [4] already contains an extension of the framework presented here, in which multiple cases are modeled and verified.

2.1 The Data Schema

Consistently with the BPMN standard, 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., we set $R.\mathsf{name} = R$). We also use the predicate notation $R(A)$ to represent a relation schema $\langle R, A \rangle$. An example of a relation schema is given by $User(Uid{:}\mathsf{Int}, Name{:}\mathsf{String})$, where the first component represents the id-number of a user, whereas the second component is the string formed by 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. Such a storage could contain, for example, the catalog of product types and the set of registered customers and their addresses in an order-to-cash scenario.

Definition 2. *A* catalog *Cat is a set of relation schemas satisfying the following requirements:*

(single-column primary key) *Every relation schema R is such that the first attribute in $R.\mathsf{attrs}$ has type in $\mathcal{S}_{id}$, and denotes the* primary key *of the relation; we refer to such attribute using the dot notation $R.\mathsf{id}$.*

(non-ambiguity of primary keys) *for every pair R_1 and R_2 of* distinct *relation schemas in Cat, we have that the types of $R_1.\mathsf{id}$ and $R_2.\mathsf{id}$ are different.*

(foreign keys) *for every relation schema $R \in Cat$ and non-id attribute $a \in R.\mathsf{attrs} \setminus R.\mathsf{id}$ with type $S \in \mathcal{S}_{id}$, there exists a relation schema $R_2 \in \mathcal{R}$ such that the type of $R_2.\mathsf{id}$ is S; a is hence a* foreign key *referring to R_2.* ◁

Example 1. Consider a simplified example of a job hiring process in a company. To represent information related to the process we make use of the *Cat* consisting of the following relation schemas:

- $JobCategory(Jcid{:}\mathsf{jobcatID})$ contains the different job categories available in the company (e.g., programmer, analyst, and the like) - we just store here the identifiers of such categories;
- $User(Uid{:}\mathsf{userID}, Name{:}\mathsf{StringName}, Age{:}\mathsf{NumAge})$ stores data about users registered to the company website, and who are potentially interested in job positions offered by the company.

[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.

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

We now define the data schema of a BPMN process, which 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*, such that:*

- *for every relation schema* $R \in Repo$ *and every attribute* $a \in R.\mathsf{attrs}$ *whose type is* $S \in \mathcal{S}_{id}$*, there exists* $R \in Cat$ *such that the type of* $R.\mathsf{id}$ *is* S*;*
- *for every case variable* $\mathbf{x} \in X$ *whose type is* $S \in \mathcal{S}_{id}$*, there exists* $R \in Cat$ *such that the type of* $R.\mathsf{id}$ *is* S*.* ◁

We use bold-face to distinguish a case variable **x** from a "normal" variable x. It is worth noting that 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. *This is essential towards soundness and completeness of SMT-based verification of DABs*. It will be clear how tuples can be inserted and removed from the repository once we will 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:

$$Application(Jcid{:}\mathsf{JobcatID}, Uid{:}\mathsf{UserID}, Eligible{:}\mathsf{Bool})$$

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 con-

sidering boolean *conditions* over (case) variables. These conditions will be attached to choice points 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* φ *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 $JobCategory(jt) \wedge jt \neq \mathtt{HR}$ 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* $\mathcal{D}.\mathsf{repo}$*.* ◁

Notice that *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.

As anticipated before, this language can be seen as a standard query language to retrieve data from a snapshot, but also as a mechanism to constrain the combinations of data objects that can be injected into the process. E.g., a simple 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 interpreted as a (constrained) user input where the user decides the values for y and z.

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 α *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` η_1 `ELSE` η_2*, where: (i)* $\vec{u}$ *is a tuple containing variables in* $\vec{x}$ *or constant objects from* $\mathbb{D}$*; (ii)* ψ *is a repo-free guard (called* filter*); (iii)* R *is a relation schema from* $\mathcal{D}.\mathsf{repo}$*; (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)* η_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 α is executable in a given data snapshot if there is at least one answer to the precondition $\alpha.\mathsf{pre}$ in that snapshot. If this is the case, then the process executor(s) can nondeterministically decide which answer to pick so as to *bind* the answer variables of $\alpha.\mathsf{pre}$ to corresponding data objects in $\mathbb{D}$. This confirms the interpretation discussed in Section 2.1 for which the answer variables of $\alpha.\mathsf{pre}$ can be seen as *constrained user inputs* in case multiple bindings are available.

Once a specific binding for the answer variables is selected, the corresponding effect rule $\alpha.\mathsf{eff}$, instantiated using that binding, is 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 repository 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 repository relations do not have an explicit primary key, two possible semantics can be attached to the insertion of a tuple $\vec{u}$ in the instance of a repository relation R:

(multiset insertion) Upon insertion, $\vec{u}$ gets an implicit, fresh primary key. The insertion then always results in the genuine addition of the tuple to the current instance of R, even in the case where the tuple already exists there.

(set insertion) In this case, R comes not only with its implicit primary key, but also with an additional, genuine key constraint defined over a subset $K \subseteq R.\mathsf{attrs}$ of its attributes. Upon insertion, if there already exists a tuple in the current instance of R that agrees with $\vec{u}$ on K, then that tuple is *updated* according to $\vec{u}$. If no such tuple exists, then as in the previous case $\vec{u}$ gets implicitly assigned to a fresh primary key, and inserted into the current instance of R. By default, if no explicit key is defined over R, then the entire set of attributes $R.\mathsf{attrs}$ is considered as a key, consequently enforcing a *set semantics* for insertion.

Example 3. We continue the job hiring example, by considering two update specifications of type insert&set. When a new case is created, the first update is about indicating

what is the category of job associated to the case. This is done through the update specification `InsJobCat`, where `InsJobCat.pre` selects a job category from the corresponding catalog relation, while `InsJobCat.eff` assigns the selected job category to the case variable **jcid**:

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

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

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

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:

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

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 consequently subject to a more fine-grained evaluation of her application, resulting in a score that is then registered in the repository (together with the applicant data). This is done via the `EvalApp` specification:

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

Here, the insertion indicates an `undef` eligibility, since it will be assessed in a consequent step of the process.

Notice that, by adopting the *multiset insertion semantics*, the same user may apply multiple times for the same job (resulting multiple times as applicant). With a *set insertion semantics*, we could enforce the uniqueness of the application by declaring the second component (i.e., the user id) of *Application* as a key. ◁

If α.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. For each tuple in R, if it passes the filter associated to the rule, then the tuple is updated according to the `THEN` part, whereas if the filter evaluates to false, the tuple is updated 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:

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

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:

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

Deleting the tuple is useful in the situation where the selected winner may refuse the job, and consequently should not be considered again if a new winner selection is carried out. 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 interleave with other update specifications potentially operating over the same portion of the repository. This is why in our approach 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 from the perspective of an external event trigger), and the specification effect how reacting to a triggered event impacts on the underlying data snapshot. More concretely, 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

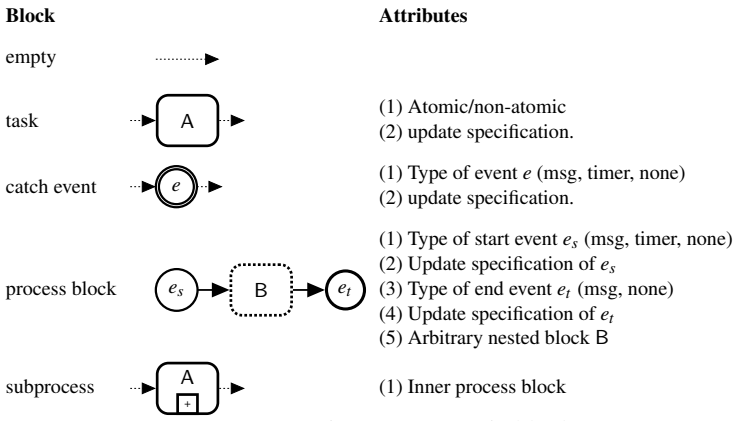


Fig. 1: DAB Basic blocks

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 of all, 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 effect operates, in this context, 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 basic (cf. Figure 1), flow (cf. Figure 2), and exception handling (cf. Figure 3) patterns. Figure 4 gives an idea about what is covered by our approach. With these blocks at hand, we finally obtain the full definition of a DAB.

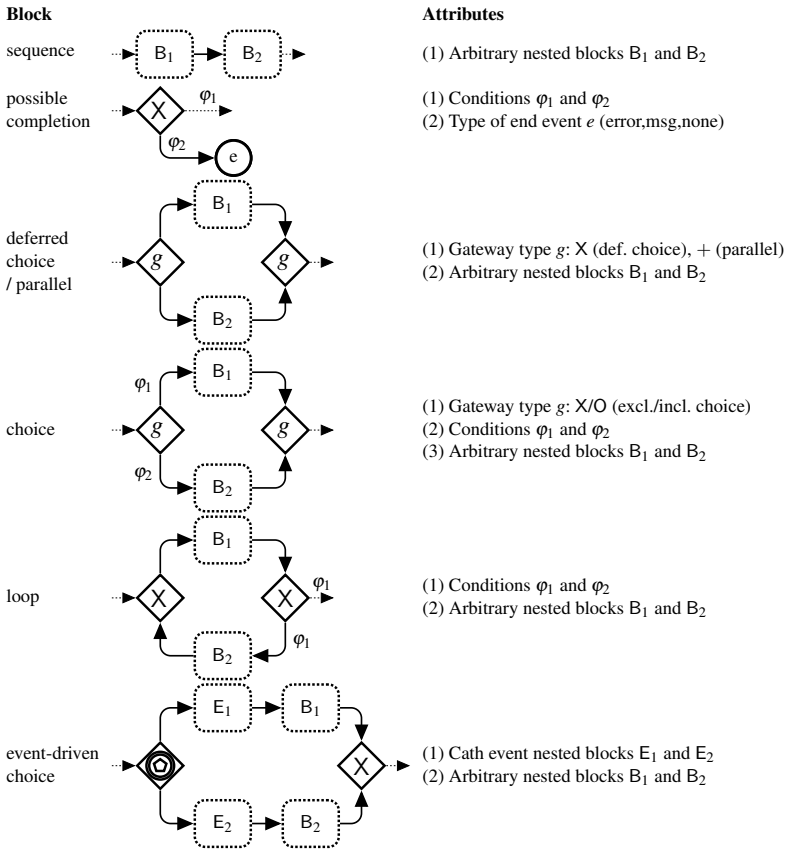


Fig. 2: Flow DAB blocks; for simplicity, we consider only two nested blocks, but multiple nested blocks can be seamlessly handled.

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 Figure 4, 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,

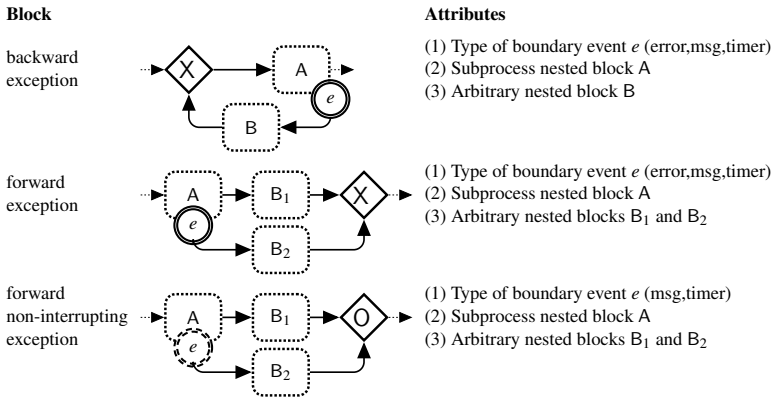


Fig. 3: DAB exception handling blocks; for simplicity, we show a single boundary event, but multiple boundary events and their corresponding handlers can be seamlessly handled.

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 Section 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 that indicates the current state of the block, and how the state may evolve depending on the specific semantics of the block, and the evolution of its inner blocks. In Section 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. For example, the lifecycle of a sequence block S with nested blocks B_1 and B_2 can be described as follows (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`, simultaneously inducing a transition of B_1 from `idle` to `enabled`; *(ii)* if B_1 is `compl`, then it becomes `idle`, simultaneously inducing a transition of B_2 from `idle` to `enabled`; *(iii)* if B_2 is `compl`, then it becomes `idle`, simultaneously inducing S to move from `active` to `compl`. The lifecycle of other block types can be defined analogously.

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*, in turn constituted by a data snapshot

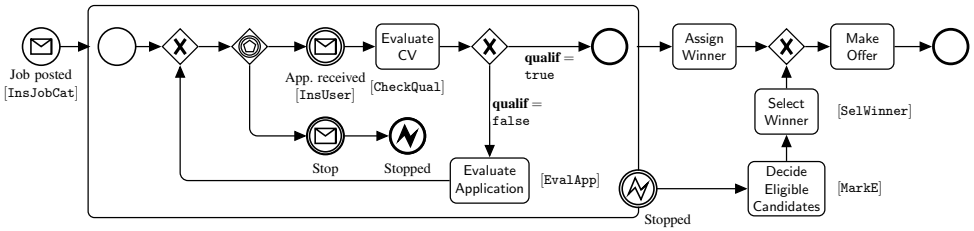
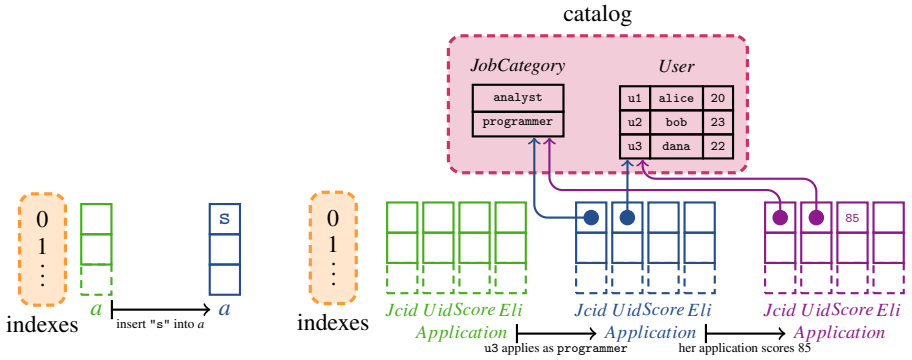


Fig. 4: The job hiring process. Elements in squared brackets attach the update specifications in Examples 3 and 4 to corresponding tasks/events.



(a) Insertion of value `"s"` into an empty string array

(b) Array-based representation of the job hiring repository of Example 2, and manipulation of a job application with a fixed catalog.

Fig. 5: Graphical intuition showing the evolution of different array-based systems. The current state of the array is represented in green, whereas consequent states resulting from updates are shown in blue and violet. Empty cells implicitly hold the `undef` value of their corresponding type.

of $\mathcal{D}$ (cf. Section 2.1), and a further assignment mapping each block in $\mathcal{P}$ to its current lifecycle state.

Initially, the data snapshot fixes the immutable content of the catalog $\mathcal{D}.\mathtt{cat}$, 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 then obtained by following Section 2.2.

3 Parameterized Verification of Safety Properties

We now focus on parameterized verification of DABs using SMT-based techniques grounded in the theory of arrays.

3.1 Array-Based Artifact Systems and Safety Checking

We recall the key notions behind array-based systems, and the array-based artifact systems recently studied in [5] to bridge the gap between SMT-based model checking of array-based systems [17,18], and verification of data- and artifact-centric processes [12,14].

In general terms, an array-based system logically describes the evolution of array data structures of unbounded size. Figure 5a intuitively shows a simple array-based system consisting of a single array storing strings. The logical representation of an array relies on a theory with two types of sorts, one accounting for the array indexes, and the other for the elements stored in the array cells. Since the content of an array changes over time, it is referred to using a *function* variable, called *array state variable*. The interpretation of such a variable in a state is that of a total function mapping indexes to elements: for each index, it returns the element stored by the array in that index. In the initial green state of Figure 5a, the array a is interpreted as a total function mapping every index to the undefined string.

Starting from an initial configuration, the interpretation changes when moving from one state to another, reflecting the intended manipulation on the array. Hence, the definition of an array-based system with array state variable a always requires *(i)* a state formula $I(a)$ describing the *initial configuration(s)* of the array a; *(ii)* a formula $\tau(a,a')$ describing the *transitions* that transform the content of the array from a to a'. By suitably using logical operators, τ can express in a single formula a repertoire of different updates over a.

In such a setting, one of the most fundamental, and studied, verification problem is that of checking whether the evolution induced by τ over a starting from a configuration in $I(a)$ eventually *reaches* one of the *unsafe* configurations described by a state formula $K(a)$. This, in turn, can be tackled by showing that the formula $I(a_0) \wedge \tau(a_0,a_1) \wedge \cdots \wedge \tau(a_{n-1},a_n) \wedge K(a_n)$ is satisfiable for some n. If no such n exists, then no finite run of the system can reach the undesired configurations, and hence the system is safe. Several mature model checkers exist to ascertain (un)safety of these type of systems, such as MCMT [19] and CUBICLE [8]. Specifically, MCMT handles this verification problem through a symbolic *backward reachability* procedure. This is a goal-directed procedure that 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 τ, directly or indirectly reach configurations satisfying $K(a)$. Two checks are then applied, so as to determine whether the procedure has to stop or must continue the iteration. The first one, called *fixpoint check*, tests if the newly computed preimages all coincide with already computed states: if no new state can be produced, the procedure stops by emitting *safe*. Otherwise, a second test, called *fixpoint check*, is applied to determine if one of such iterated preimages satisfies $I(a)$: if so, the procedure stops by emitting *unsafe* as a verdict; if not, new iterated preimages are computed and the procedure is repeated. MCMT generates the proof obligations arising from safety and fixpoint checks, and passes them to a state-of-the-art SMT solver (currently, YICES [15] is employed).

In [5], we have extended array-based systems towards an array-based version of the artifact-centric approach, considering in particular the sophisticated model in [21]. 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, it operates over a set of relations possibly containing unboundedly many updatable entries (cf. our DAB repository). Figure 5b gives an intuitive idea of how this type of system looks like, using the catalog and repository relations from Example 2. Contrast this with the simple array system of Figure 5a. On the one hand, 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. On the other hand, each repository relation is treated as a set of arrays, in which each array accounts for one component of the corresponding repository relation. A tuple in the relation is reconstructed by accessing all such arrays with the same index. In [5], we focus on parameterized (un)safety of RAS, verifying whether there exists an instance of the read-only database such that the artifact system can reach an unsafe configuration. Since the cells of the arrays may point to identifiers in the catalog, in turn related to other catalog relations via foreign keys, the standard backward reachability procedure needs to be suitably revised [5]. In fact, when computing preimage formulae over RAS, existentially quantified "data" variables may be introduced, breaking the format of state formulae. To restore the key property that the preimage of a state is again represented symbolically as a state formula, such additional quantified variables must be eliminated. Suitable quantifier elimination techniques have been studied in [5,6] and implemented in the latest version 2.8 of MCMT, which can now natively handle the verification of RAS. In addition, while the unsafety verification is in general undecidable for RAS, several subclasses with decidable unsafety have been singled out. One of such classes corresponds to RAS operating over arrays whose maximum size is bounded a-priori. A RAS of this type is called SAS (for *simple* artifact systems). All in all, the RAS 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). We use $F_{\mathcal{P}}$ to denote the set of all these additional case *control* variables.

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 $y_1 \in \mathcal{D}.\mathsf{cvars}$.* $\triangleleft$

Example 6. By naming *HP* the root process block of Figure 4, 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** = $\texttt{completed} \wedge \mathit{Application}(j,u,s,\texttt{true}) \wedge s > 100$ 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 (cfr. the **5th** safe property from Section 3.3). ◁

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 [4]. 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 Figure 5b: 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 transition in place, we define BackReach as the backward reachability procedure that: (1) takes as input *(i)* a DAB $\mathcal{M}$, *(ii)* a property φ 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 $\widehat{\mathcal{M}}$, and φ into a corresponding property $\widehat{\varphi}$ over $\widehat{\mathcal{M}}$ (Definition 10 ensures that φ' is indeed a RAS state formula); (3) returns the result produced by the MCMT backward reachability procedure (cf. Section 3.1) on $\widehat{\mathcal{M}}$ and $\widehat{\varphi}$.

3.3 Verification Results

By exploiting the DAB-to-RAS translation and the formal results in [5], we are now ready to provide our main technical contributions. First and foremost: 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 for which BackReach is guaranteed to terminate, then becoming a full decision procedure for unsafety. The first, unavoidable condition is on the constraints used in the catalog: its foreign keys cannot form referential cycles (where a table directly or indirectly refers to itself). This is in line with [21,5]. To define acyclicity, we associate to a catalog *Cat* a characteristic graph

$G(Cat)$ that captures the dependencies between relation schema components induced by primary and foreign keys. Specifically, $G(Cat)$ is a directed graph such that:

- for every $R \in Cat$ and every attribute $a \in R.\mathsf{attrs}$, the pair $\langle R, a\rangle$ is a node of $G(Cat)$ (and nothing else is a node);
- $\langle R_1, a_1\rangle \to \langle R_2, a_2\rangle$ if and only if one of the two following cases apply: *(i)* $R_1 = R_2$, $a_1 \neq a_2$, and $a_1 = R_1.\mathsf{id}$; *(ii)* $a_2 = R_2.\mathsf{id}$ and a_1 is a foreign key referring R_2.

Definition 11. *A DAB is* acyclic *if the characteristic graph of its catalog is so.* ◁

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 required conditions 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 repository relations cannot be used. To avoid indirect comparisons/joins, queries cannot mix case variables and repository 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 12. *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* χ*,* $R(\vec{y})$*, and* ξ *optional), where: (i)* χ *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)* ξ *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.* ◁

Intuitively, a separated guard consists of two isolated parts: one part χ inspecting the content of case variables and their relationship with the catalog, and another part $R(\vec{y}) \wedge \xi$ retrieving a single tuple $\vec{y}$ in some repository relation R, possibly filtering it through inspection of the catalog via ξ.

Example 7. Consider the refinement $\mathtt{EvalApp}.\mathsf{pre} \triangleq GetScore(s : \mathsf{NumScore}) \leftarrow \xi \wedge \chi$ of the guard $\mathtt{EvalApp}.\mathsf{pre}$ 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 χ and ξ 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* u.eff *is a* conditional update *rule, then* u.pre *is* repo-free *and* boolean *(i.e., it returns either* `false` *or the empty tuple), so that* u.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* $\mathcal{M}$. ◁

Notably, the conditions of Theorem 3 represent a concrete, BPMN-like counterpart of the abstract conditions used in [21] and [5] towards decidability.

Specifically, Theorem 3 uses two conditions: *(i)* repo-freedom, or *(ii)* the combination of separation with the manipulation of *all* case variables at once. We now intuitively explain how these conditions substantiate the aforementioned 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, single repository tuple retrieved by the separated precondition.

Example 8. By considering the data and process schema of the hiring process DAB, one can directly show that it obeys to all conditions in Theorem 3, in turn guaranteeing termination of BackReach. For example, rule `EvalApp` in Example 3 matches point 1 since `EvalApp`.pre is repo-free. `SelWinner` from the same example matches instead point 3, since `SelWinner`.pre is trivially separated and *all* case variables appear in the `SET` part of `SelWinner`.eff. ◁

First Experiments with MCMT. We have encoded the job hiring DAB described in the paper into MCMT, systematically following the translation rules recalled in Section 3.2, and fully spelled out in [4] when proving the main theorems of Section 3.3. Running MCMT Version 2.8 (`http://users.mat.unimi.it/users/ghilardi/mcmt/`), we have checked the encoding of the job hiring DAB for process termination (which took 0.43sec), and against five safe and five unsafe properties. For example, the **1**st **unsafe** prop-

	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

erty 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: **EvalApplifecycle** = $\texttt{completed} \wedge Application(j,u,score,e) \wedge score > 0$. The **4**th **safe** property represents instead the situation in which a winner has been selected after the deadline (i.e., SelWin is completed), but the case variable **result** witnesses that the winner is not an eligible candidate. Formally: **SelWinlifecycle** = $\texttt{completed} \wedge$ **result** = $\texttt{false}$. MCMT returns SAFE, witnessing that this configuration is not reachable from the initial states. Additional properties (taken from the table on the right) are described in [4].

The table on the right summarizes the obtained, encouraging results, reporting the MCMT running time in seconds. The MCMT specifications containing all the properties to check (together with their intuitive interpretation) are available in [4], and all tests are directly reproducible. Experiments were performed on a machine with Ubuntu 16.04, 2.6 GHz Intel Core i7 and 16 GB RAM.

4 Conclusion and Discussion

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 correctly tackled by off-the-shelf array-based SMT techniques, and in particular by the backward reachability procedure implemented in the MCMT model checker. Differently from conventional process-centric verification, the verification language proposed in this paper supports properties that address both process and data aspects.

We have then identified classes of DABs suitably controlling the data components and the way the process manipulates it, guaranteeing termination of backward reachability. We have finally shown that a realistic example of DAB can be actually verified by MCMT with a very promising performance.

There are plenty of avenues for future work. We enumerate the most important ones, considering methodological, foundational, and experimental aspects.

From the methodological point of view, the conditions we have introduced to guarantee termination can be seen as modeling principles for data-aware process designers who aim at making their processes verifiable. The applicability of such principles to real-life processes is an open question, calling for genuine, further research on empirical validation on real-world scenarios, as well as on the definition of guidelines helping modeling and refactoring of arbitrary DABs into fully verifiable ones. Frameworks for the empirical validation of data-aware process models have been recently brought forward [26], and can be in fact extended also considering the verifiability factor.

From the foundational perspective we are interested in equipping DABs with datatypes and corresponding rigid predicates, including arithmetic operators, as done in [14] for artifact systems. This is promising especially considering that there are plenty of state-of-the-art SMT techniques to handle arithmetics. At the same time, we want to attack the main limitation of our approach, namely that guards and conditions are actually existential formulae, and the only (restricted) form of universal quantification available in the update language is that of conditional updates. Universal guards in tran-

sition formulae could be very useful in specifications: for example, they would allow us to specify a branch in a job hiring process that is followed only if *no* applicant satisfies a certain condition. The question has been debated since longtime in the literature and the most effective solution so far is the introduction of suitable "monotonic abstractions" (see [1] for a survey). Notably, this is already implemented in MCMT. Monotonic abstractions could introduce spurious unsafe traces, and in fact MCMT warns the user about this (in practice, not so frequent) possibility. An orthogonal, challenging question is how, and to what extent, some of the most recent techniques developed for temporal model checking of artifact-centric systems [14] can be incorporated in our approach, allowing us to prove more sophisticated properties beyond safety.

From the experimental point of view, while a systematic evaluation is out of scope of this paper, the initial experiments carried out in this paper and [5] indicate that the approach is promising. We intend to fully automate the translation from DABs to array-based systems, and to set up a benchmark to evaluate the performance of verifiers for data-aware processes, starting from the examples collected in [21]: they are inspired by reference BPMN processes, and consequently should be easily encoded as DABs.

References

1. F. Alberti, S. Ghilardi, and N. Sharygina. Monotonic abstraction techniques: from parametric to software model checking. In *Proc. MOD**, EPTCS, pages 1–11, 2014.
2. D. Calvanese, G. De Giacomo, and M. Montali. Foundations of data aware process analysis: A database theory perspective. In *Proc. PODS*, pages 1–12, 2013.
3. D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, and A. Rivkin. Formal modeling and SMT-based parameterized verification of data-aware BPMN. In *Proc. of BPM*, LNCS. Springer, 2019.
4. D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, and A. Rivkin. Formal modeling and SMT-based parameterized verification of multi-case data-aware BPMN. Technical Report arXiv:1905.12991, arXiv.org, 2019.
5. D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, and A. Rivkin. From model completeness to verification of data aware processes. In *Description Logic, Theory Combination, and All That*, LNCS. Springer, 2019.
6. D. Calvanese, S. Ghilardi, A. Gianola, M. Montali, and A. Rivkin. Model completeness, covers and superposition. In *Proc. of CADE*, 2019.
7. C. Combi, B. Oliboni, M. Weske, and F. Zerbato. Conceptual modeling of processes and data. In *Proc. ER*, volume 11157 of *LNCS*, pages 236–250. Springer, 2018.
8. S. Conchon, A. Goel, S. Krstic, A. Mebsout, and F. Zaïdi. Cubicle: A parallel SMT-based model checker for parameterized systems - Tool paper. In *Proc. CAV*, pages 718–724, 2012.
9. G. De Giacomo, X. Oriol, M. Estañol, and E. Teniente. Linking data and BPMN processes to achieve executable models. In *Proc. CAISE*, 2017.
10. M. de Leoni, P. Felli, and M. Montali. A holistic approach for soundness verification of decision-aware process models. In *Proc. ER*, LNCS, pages 219–235. Springer, 2018.
11. R. De Masellis, C. Di Francescomarino, C. Ghidini, M. Montali, and S. Tessaris. Add data into business process verification: Bridging the gap between theory and practice. In *Proc. AAAI*, pages 1091–1099. AAAI Press, 2017.
12. A. Deutsch, R. Hull, Y. Li, and V. Vianu. Automatic verification of database-centric systems. *SIGLOG News*, 5(2):37–56, 2018.

13. A. Deutsch, R. Hull, and V. Vianu. Automatic verification of database-centric systems. *SIGMOD Record*, 43(3):5–17, 2014.
14. A. Deutsch, Y. Li, and V. Vianu. Verification of hierarchical artifact systems. In *Proc. PODS*, pages 179–194, 2016.
15. B. Dutertre and L. De Moura. The YICES SMT solver. Technical report, SRI International, 2006.
16. M. Estañol, M. Sancho, and E. Teniente. Verification and validation of UML artifact-centric business process models. In *Proc. CAISE*, LNCS, pages 434–449. Springer, 2015.
17. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT model checking of array-based systems. In *Proc. IJCAR*, pages 67–82, 2008.
18. S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6(4), 2010.
19. S. Ghilardi and S. Ranise. MCMT: A model checker modulo theories. In *Proc. IJCAR*, 2010.
20. S. Lasota. Decidability border for petri nets with data: WQO dichotomy conjecture. In *Proc. PETRI NETS*, volume 9698 of *LNCS*, pages 20–36. Springer, 2016.
21. Y. Li, A. Deutsch, and V. Vianu. VERIFAS: A practical verifier for artifact systems. *PVLDB*, 11(3):283–296, 2017.
22. A. Meyer, L. Pufahl, D. Fahland, and M. Weske. Modeling and enacting complex data dependencies in business processes. In *Proc. BPM*, volume 8094 of *LNCS*, pages 171–186. Springer, 2013.
23. M. Montali and A. Rivkin. DB-Nets: on the marriage of colored Petri Nets and relational databases. *ToPNoC*, 28(4), 2017.
24. D. Müller, M. Reichert, and J. Herbst. Data-driven modeling and coordination of large process structures. In *Proc. OTM*, volume 4803 of *LNCS*, pages 131–149. Springer, 2007.
25. M. Reichert. Process and data: Two sides of the same coin? In *Proc. OTM*, volume 7565 of *LNCS*. Springer, 2012.
26. H. A. Reijers, I. T. P. Vanderfeesten, M. G. A. Plomp, P. Van Gorp, D. Fahland, W. L. M. van der Crommert, and H. D. D. Garcia. Evaluating data-centric process approaches: Does the human factor factor in? *Software and System Modeling*, 16(3):649–662, 2017.
27. F. Rosa-Velardo and D. de Frutos-Escrig. Decidability and complexity of petri nets with unordered data. *Theor. Comput. Sci.*, 412(34):4439–4451, 2011.
28. N. Sidorova, C. Stahl, and N. Trcka. Soundness verification for conceptual workflow nets with data: Early detection of errors with the most precision possible. *Inf. Syst.*, 36(7):1026–1043, 2011.
29. W. M. P. van der Aalst. Verification of workflow nets. In *Proc. ICATPN*, 1997.