# Introducing Variables to Data Objects in BPMN

Maximilian König, Tom Lichtenstein, Anjo Seidel, and Mathias Weske

Hasso Plattner Institute, University of Potsdam, Potsdam, Germany
{firstname.lastname}@hpi.de

**Abstract.** The management of data is crucial in today's organizations, making it necessary to specify exactly how data is created, accessed, and manipulated during business process enactment. Given the importance of data, it comes as a surprise that approaches like BPMN only provide limited support for modeling data and how it is read and written. In particular, they cannot represent multiple data objects of the same type, and they lack concise semantics for multi-instance data objects. Behind this background, this paper proposes an extension to BPMN process models by introducing variable identifiers to distinguish individual data objects of the same class in a given process. The behavior is detailed using translational semantics to Colored Petri nets, and a set of verification mechanisms is presented that allow for a more precise analysis of data objects in business processes.

**Keywords:** BPMN · Data in Processes · Translational Semantics · Colored Petri Nets · Variables.

## 1 Introduction

Helping organizations to maintain an overview of the complex processes driving their value creation is an important aspect of business process management. For that purpose, a variety of methodologies is provided to support the entire lifecycle of business processes, from design and analysis to configuration, enactment, and evaluation [33]. While control flow has been the main focus of process modeling languages, recent endeavors emphasize data objects that are manipulated through process activities. This can also be seen in object-centricity as a novel paradigm [1,3,15], in which business processes are considered from the perspective of data objects rather than process instances.

In industry and academia, BPMN process diagrams [24] are a widely used activity-centric modeling language [12]. However, its support for data is limited [23]. While version 2.0 introduced concepts to approach that deficiency, capturing the processing of multiple objects of the same class in a single process is not well-supported. The current specification also does not allow for the unique identification of two objects of the same class in the same process. For example, one might want to single out the best paper and the runner-up from the list of accepted papers at a conference. Unfortunately, BPMN does not allow us to independently refer to two data objects of class 'paper' in one process

instance. Additionally, concise semantics that allow for verification and precise enactment only have been introduced for control flow [8,10] and simple types of data interactions [5,27,28]. Complex constructs involving lists of data objects and the unique identification of different objects of the same class, on the other hand, have not been addressed sufficiently.

To approach these issues, we propose a simple, but very relevant extension to BPMN to include variable identifiers for data objects. Therewith, different objects of the same class can be defined and individually accessed in a given process instance. The extension is underpinned with a concise execution semantics, and verification properties to detect potentially erroneous behavior are discussed.

This paper is structured as follows: Section 2 introduces foundational knowledge on BPMN and Colored Petri nets, based on which Section 3 motivates the paper's contribution. In Section 4 we then informally describe the proposed extension to BPMN before Section 5 formally specifies the behavior using Colored Petri nets as formalism. Afterward, we show how the formalism can be used for verification and compliance checking in Section 6. Section 7 provides an overview of other works in the field, followed by Section 8 discussing the results of this work and Section 9 outlining future research opportunities and concluding the paper.

## 2 Foundations

This section presents the key concepts our approach utilizes. We provide an overview of BPMN's data representation capabilities and introduce Colored Petri nets.

### 2.1 Data in BPMN

BPMN provides a widely used standardized modeling language for business processes with an emphasis on control flow [24]. Activities, i.e., units of work performed in the context of the process, and events, i.e., instantaneous, process-relevant occurrences, can be ordered using control flow structures such as gateways which allow the representation of decisions and concurrency. To address the increasing significance of data in processes, version 2.0 of the standard introduced concepts to describe relevant data and its interaction with the control flow. *Data object nodes* (document shapes in Fig. 1) visualize the interaction of activities and events with certain types of data. Specifically, each node specifies a data class and a state denoted in square brackets. Data classes define the structure of the objects belonging to them, while data states induce conditions on the expected data an object contains. BPMN does not provide a notation to define either data classes or data states in more detail.

The availability of data objects can be a precondition for activity instances' enablement which is indicated by a read operation, i.e., a data object node having an arc toward the activity. Otherwise, if the arc points toward the data object node, a write operation is performed. Two kinds of write operations must

be distinguished. If an object is written without being read, this constitutes the creation of a new object. If an object of the same class is also read by the activity, that object is updated to the state specified in the outgoing data object node. For example, activity 'Review paper' in a BPMN process diagram in Fig. 1 requires a 'Paper' in state 'submitted' for enablement. After terminating, the activity writes that object in state 'reviewed'.

According to the BPMN specification, a data object node always refers to the same data object per process instance [24, p. 206]. Therewith, *blind writes* as known from database terminology may occur. Given an activity creating a new object. If an object of the same class has previously been created, the existing object's content will be blindly overwritten because we cannot distinguish the objects on a model level. To reference a list of data objects of the same class in the same state, but not the objects within that list individually, a data object node may be annotated with the multi-instance marker ≡. In that case, all objects in the specified state are accessed. If such an object list is read by an activity, it must contain at least one object to enable the activity.

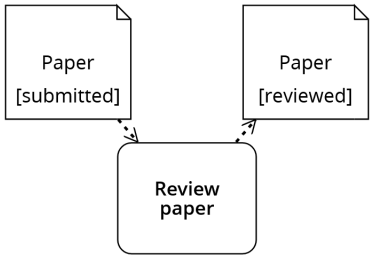


**Fig. 1.** Data object read and written by an activity

For BPMN activities, a set of markers exists to indicate that multiple instances will be executed sequentially (≡ or ↻) or concurrently (≡). For the loop marker, the number of instances can be specified using text annotations. For the others, a data object list node in the precondition specifies that the activity will be executed once for each element in that list. If there is no data precondition, the number of instances is undefined.

### 2.2 Colored Petri Nets

Petri nets [26] are a formal modeling language initially introduced to describe concurrent behavior. They are bipartite graphs consisting of transitions and places connected by arcs. The state of a net is represented by the distribution of tokens over all its places, called a marking. State changes occur upon the execution of a transition. A transition can fire, if all places in its preset, i.e., the set of places with an arc toward that transition, hold at least one token. Upon execution, a token is consumed from every place in its preset, and a token is produced in every place of its postset, i.e., the set of places with an arc from the transition toward them.

Colored Petri nets (CPNs) are an extension of traditional Petri nets introducing *colorsets*, i.e., data types, for tokens [18]. Therewith, tokens can be distinguished, enabling the representation of multiple different objects in the same net. In addition, tokens may hold concrete data values based on which the behavior of the net can be further specified.

Arc expressions bind token values to variables and specify the values of newly created tokens. Transition guards determine under which conditions a transition is enabled based on the values of tokens it would consume. For example, Fig. 2 shows a small example CPN. The place *counter* of type *INT* holds one token with value *5*. Before transition *Increment* can fire, the value of a token is bound to $c$ and the guard checks whether $c < 10$ holds. In case the guard evaluates to true, a token with value $c + 1$ is returned.

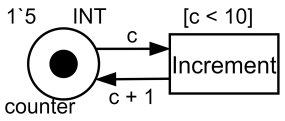


**Fig. 2.** Example of a counter implemented as Colored Petri net.

## 3 Problem Statement

Although BPMN provides basic data modeling capabilities, it has limited support for modeling multiple data objects of the same class within a process. Based on the semantics of the standard, we identified three main limitations in this regard. This section outlines and illustrates these limitations using the examples shown in Figure 3.

*Distinguishing Data Objects.* As described in Section 2.1, data object nodes of the same class always refer to the same data object upon their first assignment. Consequently, several data objects of the same class cannot be referenced separately within a process and therefore cannot be distinguished from one another. In the example shown in the Figure 3 (a), the activities 'SBP' and 'SRP' both write to data object nodes of the 'Paper' class with the intention of referencing the best and runner-up best paper separately for later use. However, according to the BPMN standard, both activities are writing to the same data object, resulting in the second activity overwriting the data written by the first activity. Given the current semantics, the intended behavior cannot be modeled for objects of the same class.

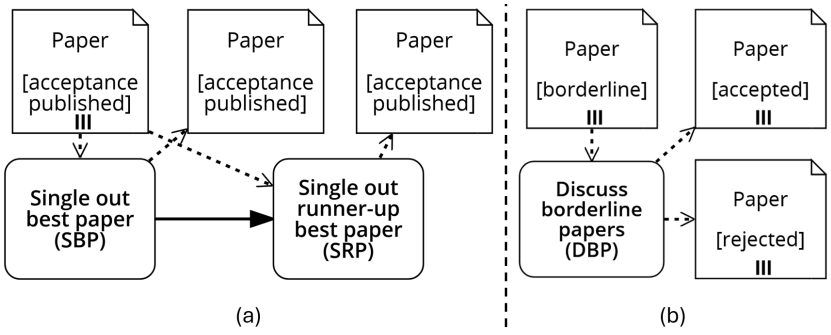


**Fig. 3.** BPMN process model excerpts visualizing deficiencies in data handling.

*Distinguishing Data Object Lists.* Similar to individual data objects, BPMN does not support distinguishing between lists of the same class. In addition, objects referenced by a list node must have the same state. These restrictions imply that lists cannot be split or merged during process execution, as discussed in [19]. In the example illustrated in Figure 3 (b), the 'DBP' activity writes to two lists of papers in different states. While the intended behavior is to split the list of 'borderline' papers into 'accepted' and 'rejected', the semantics require the activity to write to only one of the lists during execution. Similarly, merging multiple lists of the same class into a single list is not supported.

*Referencing Data Objects from Lists.* According to the standard, data object nodes and list nodes must not overlap. Since this also applies to lists of the same class, BPMN does not support creating a reference to an individual data object contained in a list. For example, considering Figure 3 (a), the 'SBP' activity aims to single out the best paper from a list of accepted papers. However, according to the standard, the data object written by the activity must not be included in the list.

In summary, the data semantics of BPMN restrict the handling of multiple data objects of the same class within processes. These limitations, as illustrated by the examples in Figure 3, can complicate the accurate modeling of data flow in business processes.

## 4 Handling Data Object Nodes with Variables

To address the limitations of the current data semantics of BPMN outlined in Section 3, we extend BPMN data object nodes with variables. A variable serves as an identifier denoted on the data object node that is assigned to a concrete object at runtime. If the variable is reused on another node in the model, the same object can be referenced again. Therewith, we can lift the assumption that every node of the same class refers to the same object, allowing for independent processing of multiple objects of the same class in one process.

In the following, we will informally describe the notation and intended behavior for create, read, and update operations on objects alongside the extended paper review process example depicted in Fig. 4, before Section 4 provides a formalization.

Variables are specified in the labels of data object nodes as prefixes to the data class, separated by a colon. For example, 'P:Paper' indicates that variable 'P' references a certain set of objects of the class 'Paper' at runtime. As a convention, variables starting with uppercase letters are used to identify data object lists, e.g., 'P:Paper', while those starting with lowercase letters refer to single objects, e.g., 'bp:Paper'. In the example, that allows us to identify the best paper 'bp:Paper' singled out from the list of accepted papers 'Pa:Paper' and reuse it later to prepare the award for the respective winner. At the same time, we can assign the runner-up best paper to 'rp:Paper' without overwriting the previous reference. Similarly, we can now split and merge lists. For example, deciding on the reviewed papers' acceptance ('DA') results in three lists of

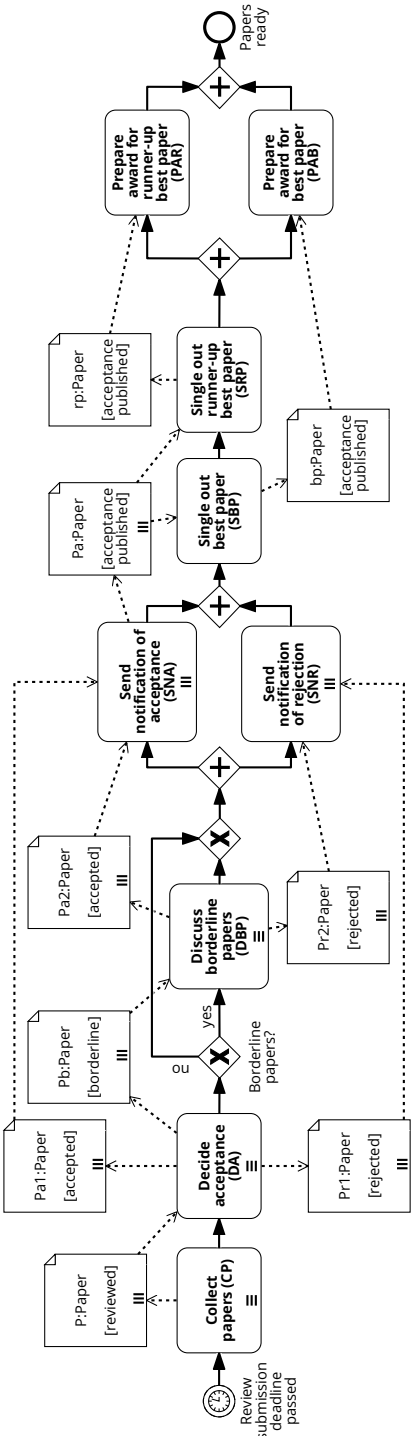


**Fig. 4.** BPMN process model of a paper reviewing process using data objects with variables.

accepted ('Pa1'), rejected ('Pr1'), and borderline ('Pb') papers, showing that variables can effectively address the shortcomings detailed in Section 3.

*Create.* The semantics of create operations in BPMN originally depended on the existence of an object of the same class, making it a blind write if an object exists already. With our approach, we modify the semantics in a way that create operations always create a new object. In addition, created objects are assigned to the variable denoted in the respective node for future reference. This may also include the reassignment of a variable, if it was previously assigned to another object. The same concept applies to lists, where all created objects are assigned to the same variable. In Fig. 4, this happens for the list 'P:Paper' created by activity 'Collect papers'.

*Read.* Reading a data object from a variable requires (1) that there is an object assigned to the variable through a previous write operation and (2) that the referenced object is in the state specified in the data object node in the model. If that is not the case, the reading activity is not enabled, i.e., cannot be executed.

Reading an object list assigned to a variable follows a similar pattern. Instead of one object, all referenced objects of the specified class in the required state are accessed. For activity enablement, at least one object adhering to these criteria must exist. If an activity reads multiple lists of the same class assigned to different variables, they are merged before activity execution. It is sufficient if the union of these lists contains at least one element for enablement. This is visualized in Fig. 4 for activity 'Send notification of acceptance', where lists 'Pa1:Paper' and 'Pa2:Paper' are both accessed.

*Update.* Updating a data object requires that the object is read and written by the same activity. In that case, the object is assigned to the variable specified in the outgoing data object node. If the target variable is the same as the source variable, the assignment remains the same. However, an object might also be assigned to a new variable. Therewith, multiple variables can reference the same object. That also holds if a state change occurs. For example, activity 'Send notification of acceptance' accesses two lists 'Pa1' and 'Pa2' and assigns their union to list 'Pa'. After that, 'Pa1' and 'Pa2' still refer to the same objects as before and could be reused later on in the model. By allowing different variables to reference the same objects, we address the third issue presented in Section 3. Single objects can now be selected from a list. For example, 'Single out best paper' now copies a reference to one of the incoming objects to the variable 'bp' for future use.

Another benefit of variables in the context of updating lists is that they can now be split and merged. For example, 'Discuss borderline papers' takes the list of reviewed papers 'P' and returns two lists of accepted and rejected papers, which constitutes the desired behavior described in Section 3. The decision on each individual object is made at runtime. As discussed in [19], this may result in empty lists. With this behavior, we extend our previous approach in [19], where a first semantics for splitting and merging lists is presented. However, the prior

mapping distinguishes list data objects only via disjoint states. Extending on that, the novel mapping also allows referencing the same object with different variables, i.e., from different process perspectives.

Next, Section 5 proposes a translation of BPMN process diagrams with variables to colored Petri nets, providing a concise execution semantics of the described behavior.

## 5 Formal Execution Semantics

The proposed notational extension allows for the specification of additional behavior in BPMN process models. To formally describe that behavior, this section introduces a formal semantics for the introduced concepts by translating them to CPNs.

*Assumptions.* To focus on the formalization of the new concepts and avoid unnecessary complexity, we make several assumptions: (1) Data object lists must contain at least one element to fulfill an activity's data precondition. An exception is multiple lists of the same class being read in one activity. In that case, their union is required to contain at least one element. With this assumption, we avoid multi-instance activities being executed zero times, which would lead to potentially inconsistent process states. (2) Every data object node in the BPMN refers to a variable. If none is specified in the model, they implicitly refer to a class-specific default variable.

*Colorsets.* For the translation to colored Petri nets, we first define the data types, i.e., colorsets, for our places and tokens. As primitive units, we will use *int* for integer values, *unit* for tokens without a specific value, and *string*. Colorsets consisting of sets of another colorset are denoted as $Set < colorset >$. Based thereon, we define $Object : ID \times State$ for data objects consisting of an $ID$ of type *int* and a *state* of type *string*. Any additional attributes of objects are abstracted from in the course of this paper. Control flow tokens will be of type $CF : unit$ since we do not need any specific data to be transported by them.

*Places.* The first step of the translational semantics is to create a set of places. For each data class in the data model, we create a single place of type $Set < Object >$ and an initial token of value [ ] (cf. Fig. 5 (a)). As a general rule, the set of all objects of one class is always represented by exactly one token in exactly one place, similar to a table in a database. That token can then be queried in transition guards to access specific objects. Every variable introduced for data objects is also mapped to a place. If the variable references a single object, that place is of type $ID$ with an initial token 0 (cf. Fig. 5 (b)), if it references a list of objects the type is $Set < ID >$ with [ ] as initial marking (cf. Fig. 5 (c)). To generate new object IDs, we will use a unique *counter* place of type *int* with an initial token of value 1 (cf. Fig. 5 (d)). Whenever a new object is created, that token's value serves as its ID and gets incremented by one. The initial value

ensures that uninitialized variables can be recognized by holding a token of value 0 or [ ].

In general, a control flow arc in BPMN corresponds to one place of type *CF* without an initial marking in the CPN (cf. Fig. 5 (e)). Additional rules considering gateways are discussed by Dijkman et al. [10]. Since the translation of the control flow is not the focus of this work, we will utilize their mapping rules for gateways.

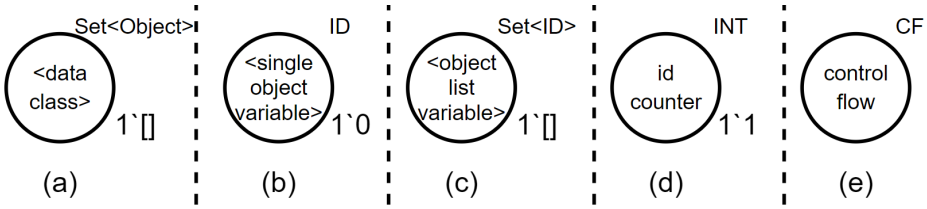


**Fig. 5.** Created places for the translational semantics to CPNs. The top left denotes a place's colorset, its name is in the center, and the initial marking is specified on the bottom right in the format $< \#tokens > \backslash < value >$. Mapped concepts are (a) all objects belonging to one data class, (b) variables referencing a single data object, (c) variables referencing a list of objects, (d) the counter to provide unique objects IDs, and (e) control flow arcs.

*Single-instance data access.* Following [10], we map single-instance tasks to a single transition each. The transition is connected to the places representing preceding and succeeding control flow arcs. Additionally, we connect the transition bidirectionally to the places of all data classes on which the respective task performs a read operation.

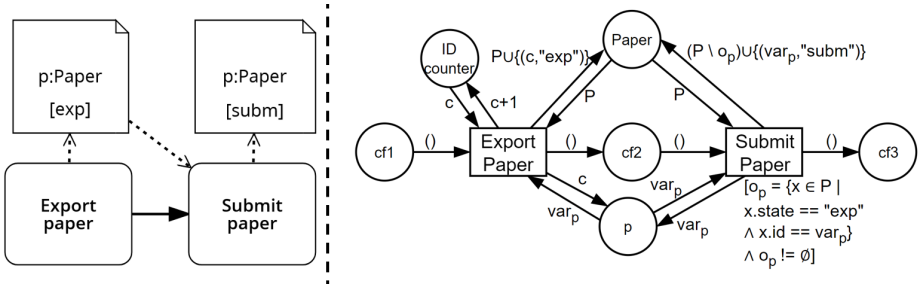


**Fig. 6.** CPN translation of basic data access operations. *Export Paper* creates a *Paper* in state *exp* by incrementing the *ID counter*, adding a new object to the place holding all *Papers*, and assigning the object's ID to the variable *p*. *Submit Paper* queries the list of all papers *P* for the object in state *exp* with the ID stored in *p*. If such an object exists, its state is updated to *subm* in *P*.

Creating an object in a specific state $s$ requires the token holding all objects of the respective class $O$ and the token of the id counter place $c$. In the arc expression returning $O$, a new object is added with id $c$ and state $s$: $O \cup \{(c, s)\}$. In addition, $c + 1$ is returned to the ID counter place. If a variable is explicitly specified in the BPMN model, the transition also overwrites the token in the variable's place to hold the id of the newly created object. Exemplarily, this is shown in Fig. 6 with activity 'Export Paper'.

Reading an object of class $C$ in a state $s$ is implemented through a guard expression querying the list of objects of the class stored in a token $O$. The query includes the required state as well as the id, as specified in the respective variable $v$. The result is assigned to an arc variable. If the arc variable is empty, the guard does not evaluate to true. Generally, the guard looks as follows: $[o_{C[s]} = \{x \in O \mid x.state == s \wedge x.id = var_v\} \wedge o_{C[s]} \neq \emptyset]$. In Fig. 6, this is visualized for activity 'Submit Paper'. In the example, variable $p$ is used to uniquely identify the paper object. After reading the paper object, 'Submit Paper' also performs a state transition. To capture that, we extend the arc expression that returns the token holding all papers $P$. We remove the outdated element stored in the variable $o_p$ and add the updated element consisting of the ID stored in the variable and the new state as specified in the model: $(P \setminus \{o_p\}) \cup \{(var_p, "subm")\}$.

*Multi-instance data access.* Working with sets of objects of the same class requires some adaptations to the previously introduced mappings, but the general concepts remain identical.

The creation of a list iterates the behavior for creating a single object. Hence, multiple transitions are required to represent that behavior, namely a starting transition, a terminating transition, and a transition repeatedly creating new objects. A *running* place holds the control flow token, and the creating transition adds elements to a temporary list of objects. If at least one object has been created, the terminating transition can fire, adding the temporary list to the token holding all objects of the respective class. The set of IDs of the created objects is added to the referencing variable's place. An example is shown in Fig. 7 for activity 'Collect papers' from the example process in Fig. 4.

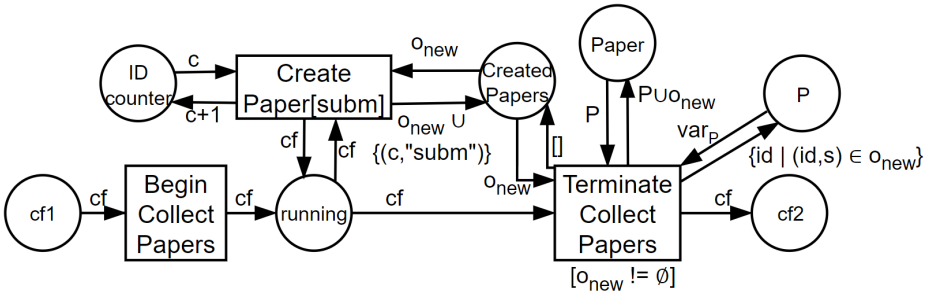


**Fig. 7.** CPN representation of creating a data object list.

Reading a data object list retrieves all objects of the specified class in the required state referenced by the assigned variable. Hence, instead of filtering for the object with a specific ID as shown in Fig. 6, we select all objects with an ID matching those stored in the variable's place. If there is no suitable object, the guard evaluates to false. An example can be found in Fig. 8, where transition 'Begin Discuss Borderline Papers' accesses all papers of list 'Pb' in state 'borderline'. If multiple lists are read, the guard comprises the conjunction of the expressions for each list. As per assumption (2), if multiple lists of the same class are read, the union of all lists must contain at least one element, rather than requiring each list to be non-empty.

State transitions for lists of objects build upon the mapping for reading lists. To transition read objects to a new state, an arc expression is added to the arc toward the place storing the objects. Essentially, all entries for read objects $O_{C[s]}$ of a class $C$ in state $s$ are replaced with entries for the same objects in the new state: $(C \setminus O_{C[s]}) \cup \{(id, "newState") \mid id \in var_X\}$ where $C$ represents all objects of class $C$ and $X$ refers to the read and written variable. If transitioned objects are assigned to a new variable, the transition writes the list of their IDs to that variable's place. If multiple output lists may be created from one list, multiple transitions are required. An initial transition reads the required objects and temporarily stores their IDs, while removing them from the place holding all objects of that class to avoid concurrent access. Afterward, transitions for each target state can update the state of one ID at a time. Finally, a terminating transition takes all temporary objects and assigns them to their variables, and returns the objects to the place of their class. An example can be found in Fig. 8, where the transition 'Discuss Borderline Papers' reads all borderline papers and transitions all objects to either accepted or rejected, effectively splitting the list and assigning the resulting sublists to new variables 'Pa2' and 'Pr2'.

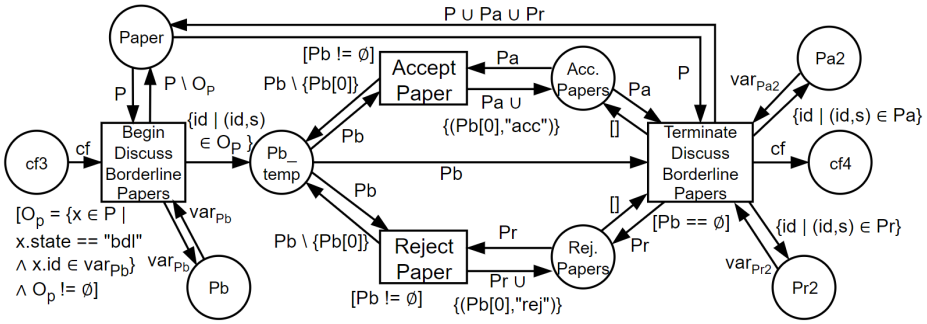


**Fig. 8.** CPN representation of splitting a list into two lists with different states.

With the presented translational semantics, we concisely define the intended behavior of variables for data objects, effectively extending BPMN's data modeling capabilities to handle multiple objects of the same class. The full application

of the mapping to the examples in Fig. 6 and Fig. 4 can be found on GitHub[1]. To view and execute the CPNs, an installation of CPN Tools[2] is required.

## 6 Analysis

In this section, the formal semantics of Section 5 is used for compliance checking and verification. For that purpose, we use BPMN-Q, a BPMN-based visual query language for business processes [4]. It provides an easy-to-understand approach to specify conditions for model verification and compliance checking. While the initial version of the language exclusively considered control flow constructs, an extension presented in [7] introduces data objects and their states to it.

An exemplary query can be found in Fig. 9. It represents the constraint that *there must be a paper whose acceptance has been published before the award for the best paper can be prepared.* '@A' represents a *variable activity*, indicating that there must be any activity fulfilling the required condition. The arrow with the // marker means that we look for a path from its source to its target, and the data condition implies that an object matching the node must be written by the respective activity. Next to *precedes* relations, BPMN-Q also supports *leads to* relations as shown in Fig. 10.

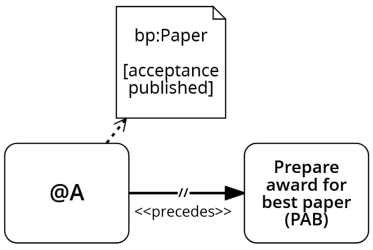


**Fig. 9.** BPMN-Q query using a data object node with variables.

As our approach integrates the declaration of variables into the labels of data object nodes, it can be seamlessly integrated with the BPMN-Q notation. Instead of requiring an unspecified object of a certain class in a state, queries may include objects assigned to certain variables. For example, the query visualized in Fig. 9 specifies that the paper in state 'acceptance published' must be assigned to the variable 'bp', i.e., the one being the result of 'Single out best paper' in Fig. 4.

Further, using BPMN-Q allows for the detection of erroneous behavior. For example, ensuring that a variable is assigned before being accessed (cf. Fig. 10 (a)) or that a list of rejected papers must be written by activity 'Decide acceptance' before the notification of rejection can be sent (cf. Fig. 10 (b)). Notably, the second query is not fulfilled in our example in Fig. 4, since all papers might be accepted or borderline, resulting in an empty list that does not fulfill this query. Therewith, we can identify situations where empty lists may result in deadlocks. As a solution, the parallel gateway could be replaced with an inclusive gateway and conditions requiring the lists of accepted and rejected papers to be non-empty before the respective path gets enabled.

To evaluate BPMN-Q queries, Past Linear Temporal Logic (PLTL) [21] statements are derived from each query [6]. For example, the PLTL query for Fig. 9 is: $G(ready(PAB) \rightarrow O(state(bp : Paper, acceptancepublished)))$ where *ready* is

[1] https://github.com/bptlab/bpmn-data-object-variables
[2] https://cpntools.org/

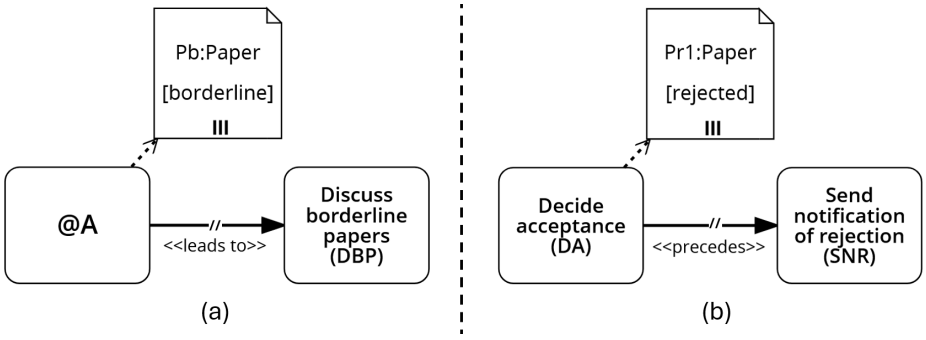


**Fig. 10.** BPMN-Q queries checking (a) whether borderline papers will always be discussed if list 'Pb' is written to state 'borderline' and (b) if activity 'Decide acceptance' writes a list of rejected papers 'Pr1' before notifications of rejection are sent.

the function returning whether activity 'PAB' is control flow enabled in a marking, *state* determines whether an object is in the specified state, and $G$ (always) and $O$ (once) are PLTL operators as introduced in [21]. To check the query for a given process model, the model is translated into a Petri net and its state space is generated [6]. In our case, this is done by applying the translational semantics from Section 5 and computing the state space in CPN Tools. Based on the state space, the PLTL queries can then be evaluated. The state space for Fig. 4 can be found in the respective CPN file in the GitHub repository.

As stated in [6], a finite state space is required for such evaluations. However, the translational semantics presented in this paper innately result in nets with infinite state spaces if an activity creates a data object list, exemplified in Fig. 7. To circumvent that, we introduce a guard to the creating transition ('Create Paper[subm]') in the example), checking that the list of newly created objects ('$o_{new}$') has at most as many elements as there are variables defined for that data class in the BPMN model. If all variables defined for this data class depend on the created list, this measure ensures that the state space includes a state where each of them is assigned to at least one object.

## 7 Related Work

A number of BPMN extensions to improve data representation exist in related work. Meyer et al. extend BPMN with foreign key relationships between objects and a mapping to SQL queries for read and write operations on data objects [23]. A similar approach is proposed by Combi et al., assigning an SQL statement specifying the data abstracted from by a data object [9]. Haarmann et al. address the issue of data objects shared by multiple processes [16], including a translational semantics to colored Petri nets. However, neither approach considers data object lists. Ghilardi et al. present delta-BPMN, combining an SQL-based data specification language with BPMN instead of a visual representation through data nodes [14]. That greatly increases the modeling complexity

and required domain knowledge, which is why we stick to the abstract representation of BPMN with data object nodes and data states. In a previous work, we introduced an approach to cover list creation, splitting, and merging [19], which we extend with this work. There, only data object lists were considered, with the state serving as an additional identifier besides an object's class. That comes with a number of deficiencies that were addressed in this paper: On the one hand, referencing single objects from lists is not defined with this approach. On the other hand, multiple lists in the same state, as shown in Fig. 4, are not supported. Hence, an object can always be referenced by one data object node.

To address the intersection of process control flow and data, object- and data-centric process modeling approaches have been introduced. An overview of existing approaches along with a framework to compare them is presented by Steinau et al. [29]. Instead of focusing on control flow, many of these approaches, e.g., PhilharmonicFlows [20], fragment-based case management [17] or object-centric behavioral constraints [2], employ a data-first approach, focusing on the object lifecycles more than control flow dependencies between activities. However, these approaches are not yet adopted in organizations. BPMN, on the other hand, is an already established language in industry in academia [12], which is why we focused on extending it regarding its data modeling capabilities.

Formalizing BPMN execution semantics is not a novel topic. Target formalisms include, but are not limited to, process algebras (e.g., CSP, $\pi$-calculus) [8,34], graph rewrite rules [11], WS-BPEL [24,25], and Petri net-based languages [5,10,19,22,27,28]. However, most of these approaches do not consider the data dimension at all. Stackelberg et al. include data objects in their translational semantics to Petri nets, but disregard data states and explicitly enforce the single instance assumption introduced by the standard [28]. Similarly, Awad et al. also implement that assumption while considering data states, but exclude data object lists from their mapping [5]. Choosing CPNs as target language, Ramadan et al. present another formalization including complex control flow constructs such as subprocesses and boundary events [27]. However, they do not go into detail regarding data object lists.

Our approach builds on BPMN-Q for data flow analysis in processes. In the context of compliance checking on processes with data, Voglhofer et al. provide an overview of contemporary literature [32]. A language-independent categorization of data anomalies has been presented by Sun et al. [30], which was adapted to BPMN by Stackelberg et al. [28]. Other approaches extend Petri nets with data operations and define data flow error detection mechanisms for them [31,35]. Neither of these approaches, however, supports our extension out-of-the-box.

## 8 Discussion

The proposed approach does not consider all data modeling capabilities BPMN provides. For example, input and output sets as well as *input output specifications* describing the relations between them are not covered. Their inclusion would further increase the expressiveness of the extension, for example, by allowing to

model explicitly that the resulting lists might be empty after splitting. Further, the semantics of BPMN multi-instance activities interacting with multiple data object lists remain underspecified. If several object lists of different data classes are read by a multi-instance activity, the activity could be executed once for each element of each list. At the same time, objects could be correlated, meaning that one activity instance processes one or multiple related objects of either list. For example, the latter would be desirable if the decision on a paper's acceptance in Fig. 4 also depended on the reviews for each paper.

Besides colored Petri nets, other formalisms were considered to define the semantics of the presented extension. While traditional Petri nets lack token differentiation, recent works propose new Petri net-based languages tailored to object-centric processes, namely object-centric Petri nets (OCPNs) [3], object-centric Petri nets with identifiers (OPIDs) [15], and synchronous proclets [13]. All of these approaches include the capability to model single objects and object lists. However, only OPIDs explicitly define identifiers for objects, and only synchronous proclets define the use of labels as variables that can be reused for synchronizing objects. In comparison, our approach allows for variables in the model and explicit object identifiers in model instances. Further, OCPNs and OPIDs cannot ensure that all objects with certain properties must be processed by a transition, which is required for BPMN semantics as discussed in Section 2.

BPMN-Q cannot evaluate queries on infinite state spaces [6]. The introduction of variables to BPMN data objects introduces additional constructs that may result in infinite behavior due to the added object identities. For example, the cyclic reassignment of variables to newly created objects leads to unboundedly many different states. Even though this might be desired behavior, it currently cannot be analyzed by our approach.

## 9 Conclusion

In this paper, we describe an approach to extend BPMN to capture complex data behavior involving different objects of the same class. For that purpose, we introduce the concept of variables to BPMN data object nodes to differentiate individual objects and object lists within one process instance. The described behavior is underpinned with a translational semantics to colored Petri nets. To analyze models incorporating variables on data object nodes, we propose to build on the visual query language BPMN-Q for verification and compliance checks. The BPMN-Q queries can be applied to a formal representation of the process model derived from the translational semantics.

The presented approach currently requires a manual translation of process models to CPNs, which is tedious and error-prone. Hence, tool support is desirable and will be approached in future work. Additional research regarding the incorporation of additional BPMN data concepts such as input and output sets should be conducted. At the same time, to improve the usability of the approach in general, a set of guidelines would help to draw attention to, for example, the explicit handling of potentially empty lists.

## References


1. van der Aalst, W.M.P.: Object-centric process mining: Dealing with divergence and convergence in event data. In: Ölveczky, P.C., Salaün, G. (eds.) SEFM 2019. LNCS, vol. 11724, pp. 3–25. Springer (2019). https://doi.org/10.1007/978-3-030-30446-1_1
2. van der Aalst, W.M.P., Artale, A., Montali, M., Tritini, S.: Object-centric behavioral constraints: Integrating data and declarative process modelling. In: Artale, A., Glimm, B., Kontchakov, R. (eds.) Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18-21, 2017. CEUR Workshop Proceedings, vol. 1879. CEUR-WS.org (2017), https://ceur-ws.org/Vol-1879/paper51.pdf
3. van der Aalst, W.M.P., Berti, A.: Discovering object-centric petri nets. Fundam. Informaticae **175**(1-4), 1–40 (2020)
4. Awad, A.: BPMN-Q: A language to query business processes. In: Reichert, M., Strecker, S., Turowski, K. (eds.) EMISA 2007. LNI, vol. P-119, pp. 115–128. GI (2007), https://dl.gi.de/handle/20.500.12116/22195
5. Awad, A., Decker, G., Lohmann, N.: Diagnosing and Repairing Data Anomalies in Process Models. In: Rinderle-Ma, S., Sadiq, S.W., Leymann, F. (eds.) BPM Workshops 2009. LNBIP, vol. 43, pp. 5–16. Springer (2009). https://doi.org/10.1007/978-3-642-12186-9_2
6. Awad, A., Decker, G., Weske, M.: Efficient compliance checking using BPMN-Q and temporal logic. In: Dumas, M., Reichert, M., Shan, M. (eds.) BPM 2008. LNCS, vol. 5240, pp. 326–341. Springer (2008). https://doi.org/10.1007/978-3-540-85758-7_24
7. Awad, A., Weidlich, M., Weske, M.: Specification, verification and explanation of violation for data aware compliance rules. In: Baresi, L., Chi, C., Suzuki, J. (eds.) ICSOC 2009. Lecture Notes in Computer Science, vol. 5900, pp. 500–515 (2009). https://doi.org/10.1007/978-3-642-10383-4_37
8. Boussetoua, R., Bennoui, H., Chaoui, A., Khalfaoui, K., Kerkouche, E.: An automatic approach to transform BPMN models to Pi-Calculus. In: AICCSA 2015. pp. 1–8. IEEE Computer Society (2015). https://doi.org/10.1109/AICCSA.2015.7507176
9. Combi, C., Oliboni, B., Weske, M., Zerbato, F.: Conceptual modeling of interdependencies between processes and data. In: SAC 2018. pp. 110–119. ACM (2018). https://doi.org/10.1145/3167132.3167141
10. Dijkman, R.M., Dumas, M., Ouyang, C.: Semantics and Analysis of Business Process Models in BPMN. Inf. Softw. Technol. **50**(12), 1281–1294 (2008). https://doi.org/10.1016/j.infsof.2008.02.006
11. Dijkman, R.M., Gorp, P.V.: BPMN 2.0 Execution Semantics Formalized as Graph Rewrite Rules. In: Mendling, J., Weidlich, M., Weske, M. (eds.) Business Process Modeling Notation - Second International Workshop, BPMN 2010, Potsdam, Germany, October 13-14, 2010. Proceedings. LNBIP, vol. 67, pp. 16–30. Springer (2010). https://doi.org/10.1007/978-3-642-16298-5_4
12. Dumas, M., Pfahl, D.: Modeling software processes using BPMN: When and when not? In: Managing Software Process Evolution. Springer (2016)
13. Fahland, D.: Describing behavior of processes with many-to-many interactions. In: Application and Theory of Petri Nets and Concurrency - 40th International Conference, PETRI NETS 2019, Aachen, Germany, June 23-28, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11522. Springer (2019). https://doi.org/10.1007/978-3-030-21571-2_1, https://doi.org/10.1007/978-3-030-21571-2_1

14. Ghilardi, S., Gianola, A., Montali, M., Rivkin, A.: Delta-bpmn: A concrete language and verifier for data-aware BPMN. In: BPM 2021. LNCS, vol. 12875, pp. 179–196. Springer (2021)
15. Gianola, A., Montali, M., Winkler, S.: Object-centric conformance alignments with synchronization. In: CAiSE 2024. LNCS, vol. 14663, pp. 3–19. Springer (2024)
16. Haarmann, S., Weske, M.: Cross-case data objects in business processes: Semantics and analysis. In: Fahland, D., Ghidini, C., Becker, J., Dumas, M. (eds.) BPM Forum 2020. LNBIP, vol. 392, pp. 3–17. Springer (2020). https://doi.org/10.1007/978-3-030-58638-6_1
17. Hewelt, M., Weske, M.: A Hybrid Approach for Flexible Case Modeling and Execution. In: Rosa, M.L., Loos, P., Pastor, O. (eds.) BPM Forum 2016. LNBIP, vol. 260, pp. 38–54. Springer (2016). https://doi.org/10.1007/978-3-319-45468-9_3
18. Jensen, K., Kristensen, L.M., Wells, L.: Coloured petri nets and CPN tools for modelling and validation of concurrent systems. Int. J. Softw. Tools Technol. Transf. **9**(3-4), 213–254 (2007). https://doi.org/10.1007/s10009-007-0038-x
19. König, M., Weske, M.: Multi-instance data behavior in BPMN. In: Fonseca, C.M., et al. (eds.) ER Forum. CEUR Workshop Proceedings, vol. 3618. CEUR-WS.org (2023), https://ceur-ws.org/Vol-3618/forum_paper_4.pdf
20. Künzle, V., Reichert, M.: PHILharmonicFlows: towards a framework for object-aware process management. J. Softw. Maintenance Res. Pract. **23**(4), 205–244 (2011). https://doi.org/10.1002/smr.524, https://doi.org/10.1002/smr.524
21. Laroussinie, F., Schnoebelen, P.: A hierarchy of temporal logics with past. Theor. Comput. Sci. **148**(2), 303–324 (1995). https://doi.org/10.1016/0304-3975(95)00035-U
22. Meghzili, S., Chaoui, A., Strecker, M., Kerkouche, E.: An Approach for the Transformation and Verification of BPMN Models to Colored Petri Nets Models. Int. J. Softw. Innov. **8**(1), 17–49 (2020). https://doi.org/10.4018/IJSI.2020010102
23. Meyer, A., Pufahl, L., Fahland, D., Weske, M.: Modeling and enacting complex data dependencies in business processes. In: BPM 2013. LNCS, vol. 8094, pp. 171–186. Springer (2013)
24. OMG: Business Process Model and Notation (BPMN), Version 2.0.2. Tech. rep., Object Management Group (2014), https://www.omg.org/spec/BPMN/2.0.2
25. Ouyang, C., Dumas, M., ter Hofstede, A.H.M., van der Aalst, W.M.P.: From BPMN Process Models to BPEL Web Services. In: (ICWS 2006). pp. 285–292. IEEE Computer Society (2006). https://doi.org/10.1109/ICWS.2006.67
26. Petri, C.A.: Kommunikation mit Automaten. PhD Thesis, Institut für instrumentelle Mathematik, Bonn (1962)
27. Ramadan, M., Elmongui, H.G., Hassan, R.: BPMN formalisation using coloured petri nets. In: Proceedings of the 2nd GSTF annual international conference on software engineering & applications (SEA 2011). pp. 83–90 (2011)
28. von Stackelberg, S., Putze, S., Mülle, J., Böhm, K.: Detecting Data-Flow Errors in BPMN 2.0. Open Journal of Information Systems (OJIS) **1**(2), 1–19 (2014)
29. Steinau, S., Marrella, A., Andrews, K., Leotta, F., Mecella, M., Reichert, M.: DALEC: a framework for the systematic evaluation of data-centric approaches to process management software. Softw. Syst. Model. **18**(4) (2019)
30. Sun, S.X., Zhao, J.L., Jr., J.F.N., Sheng, O.R.L.: Formulating the Data-Flow Perspective for Business Process Management. Inf. Syst. Res. **17**(4), 374–391 (2006). https://doi.org/10.1287/isre.1060.0105
31. Trcka, N., van der Aalst, W.M.P., Sidorova, N.: Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows. In: van Eck, P., Gordijn, J., Wieringa,

R.J. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 425–439. Springer (2009). https://doi.org/10.1007/978-3-642-02144-2_34
32. Voglhofer, T., Rinderle-Ma, S.: Collection and Elicitation of Business Process Compliance Patterns with Focus on Data Aspects. Bus. Inf. Syst. Eng. **62**(4), 361–377 (2020). https://doi.org/10.1007/s12599-019-00594-3
33. Weske, M.: Business Process Management - Concepts, Languages, Architectures, Third Edition. Springer (2019). https://doi.org/10.1007/978-3-662-59432-2
34. Wong, P.Y.H., Gibbons, J.: Formalisations and applications of BPMN. Sci. Comput. Program. **76**(8), 633–650 (2011). https://doi.org/10.1016/j.scico.2009.09.010
35. Xiang, D., Liu, G., Yan, C., Jiang, C.: Detecting data-flow errors based on petri nets with data operations. IEEE CAA J. Autom. Sinica **5**(1), 251–260 (2018). https://doi.org/10.1109/JAS.2017.7510766