Open Access

Open Journal of Information Systems (OJIS)
Volume 1, Issue 2, 2014

http://www.ronpub.com/journals/ojis
ISSN 2198-9281

Detecting Data-Flow Errors in BPMN 2.0

Silvia von Stackelberg, Susanne Putze, Jutta Mülle, Klemens Böhm

Institute for Program Structures and Data Organization, Karlsruhe Institute of Technology (KIT),
Am Fasanengarten 5, 76131 Karlsruhe, Germany,
{silvia.stackelberg, susanne.putze, jutta.muelle, klemens.boehm}@kit.edu

Abstract

Data-flow errors in BPMN 2.0 process models, such as missing or unused data, lead to undesired process executions. In particular, since BPMN 2.0 with a standardized execution semantics allows specifying alternatives for data as well as optional data, identifying missing or unused data systematically is difficult. In this paper, we propose an approach for detecting data-flow errors in BPMN 2.0 process models. We formalize BPMN process models by mapping them to Petri Nets and unfolding the execution semantics regarding data. We define a set of anti-patterns representing data-flow errors of BPMN 2.0 process models. By employing the anti-patterns, our tool performs model checking for the unfolded Petri Nets. The evaluation shows that it detects all data-flow errors identified by hand, and so improves process quality.

Type of Paper and Keywords

Regular research paper: *Business Processes, BPMN 2.0, Data-Flow Error, Anti-Patterns, Petri Nets, Model Checking*

1 Introduction

The language BPMN (Business Process Model and Notation), now an ISO standard, is widely accepted in research and practice. One important trait of BPMN 2.0 [5] is the possibility to specify executable processes. Further, the standard integrates the data aspect: With BPMN 2.0, data objects are first-class flow elements. This is not only an issue at a syntactic level, as is the case with, say, BPEL [16], but also addresses the semantics, i.e., how the data is used. Process designers can now model, for example, the data needs and data results of a task. The data needs describe the data elements a task requires for its execution, data results the ones available afterwards. This does not only hold for tasks, but - with restrictions - also for other flow elements, such as events or conditional sequence flows. Finally, one can specify alternatives for data as well as optional data.

Process designers model the *data flow*, i.e., how data traverses a process, by specifying the data needs and data results for individual flow elements. The problem studied here is to decide at design time whether the data-flow specifications in executable BPMN process models are correct. In line with other research [14, 18, 22], a data flow is correct if there are no anomalies regarding processed data. Such anomalies occur if data needed by flow elements is not available, if flow elements do not use data produced before, or if they use data inconsistently. Data-flow correctness is crucial. To illustrate, a *missing data* element (e.g., a non-initialized element) may lead to blocking due to starvation, or to incorrect gateway decisions [3, 18]. In executable BPMN models, specifications for optional data and alternatives for data can contain errors as well.

Example 1: *Think of a process withdrawing money from an ATM with two alternative authentication methods. In this process, a task* authentication *needs either the data elements cash card and PIN, or the elements*

cash card and fingerprint. If at least one of the data elements of an alternative authentication method may not be available at task execution time, it indicates a design fault, namely a missing data error *in one of the alternatives for data input of a task. This means that we define a* missing data error *in a strict manner: Not only an uninitialized data element which definitely leads to blocking is critical, but also an uninitialized one which might be compensated with alternatives at runtime.* □

In consequence, an approach for detecting data-flow errors in BPMN 2.0 process models, i.e., at design time, has to take alternatives for data and optional data into account.

It is advantageous to check the correctness of a data flow already at design time (*correctness at design time*). In particular, this holds for data-flow errors which one cannot detect at runtime. To illustrate, compensating missing data at runtime with an available alternative hides design errors, c.f. Example 1. Existing approaches for checking correctness, e.g., [14, 18, 22], are not able to detect such errors. Moreover, they cannot take the specifics of BPMN 2.0 into account, namely the execution semantics when using mandatory and optional data as well as alternatives for data.

Detecting data-flow errors in executable BPMN gives rise to the following challenges: (1) The BPMN 2.0 specification defines the execution semantics of flow elements with their data needs and data results only informally, in a textual representation. Hence, a formal verification requires a transformation of BPMN process models into a formal structure like Petri Nets. This transformation is complex, because it has to consider the execution semantics with respect to data elements. (2) Data-flow errors in executable BPMN may have to do with the fact that some data elements are optional, whereas others are mandatory. This leads to additional complexity, compared to the case where everything is mandatory. The definitions of data-flow errors must take all possible combinations into account. (3) A task may read or write data elements out of alternative sets. Detecting errors must take this into account as well. However, avoiding to explicitly handle the huge number of possible constellations for data within the process execution path that stem from such flexible alternatives is not trivial. As data elements may be optional at the same time, things become even more complex. (4) There is a lack of publicly available process models conforming to BPMN 2.0, which could be used in an evaluation. For instance, the repository provided by the BPM Academic Initiative [4] does not contain models suited for this purpose, as we will explain in Section 4.

This paper proposes an approach to detect data-flow errors in BPMN 2.0 process models. More specifically, we make the following contributions:

Definition of Anti-Patterns. Starting with classifications of anti-patterns from the literature [14, 21, 22], we define a set of data-flow anti-patterns enhanced to capture the specifics of data in BPMN 2.0. Our anti-patterns describe anomalies of the data flow. Their definitions consider the execution semantics for data needs and data results of BPMN 2.0 flow elements. In particular, they allow for combinations of alternative data needs and data results and distinguish between mandatory and optional data.

Transformation. We specify a new transformation of process models into unfolded Petri Nets. In particular, these Petri Nets formally express the execution semantics of the process and its flow elements using data. They do so by taking alternatives for data as well as optionality of data into account. They also avoid handling the huge number of data alternatives explicitly by coordinating data needs and data results separately.

Tool Support. We have implemented a tool to detect data-flow errors in BPMN 2.0 process models automatically at design time. It realizes the transformation just mentioned and finds data-flow errors in the Petri Net using a model checker. When a data-flow error is found, the tool points to the task where it occcurs.

Evaluation. We have asked a BPMN expert to develop a set of process models, which we then have used in our evaluation. Our tool systematically detects all data-flow errors generated by him as well as errors occurring in another user experiment.

Using this approach, process designers can now increase the quality of their models by analyzing the data flow of BPMN 2.0 processes at design time. This avoids costly process executions with errors. Our approach allows to detect data-flow errors such as *missing* and *redundant optional* data.

Section 2 analyzes and explains the data perspective and describes data-flow errors in BPMN 2.0 process models. Section 3 introduces our approach to check data-flow correctness in executable BPMN process models and its implementation. We describe the evaluation of our approach in Section 4. Section 5 discusses related work, and Section 6 concludes.

2 DATA IN BPMN

The BPMN 2.0 standard [5] distinguishes several representations for process models. They differ regarding expressiveness: The *executable subclass* contains the complete execution information. The graphical process diagrams in turn cover only a subset of this. Data elements play different roles in these representations: Data associations with flow elements in graphical process diagrams

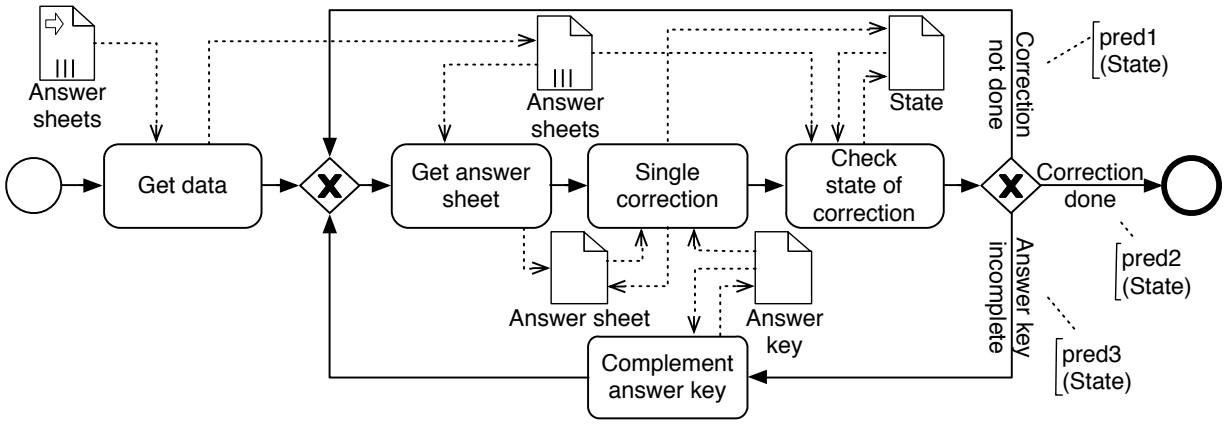


Figure 1: Exam Correction Process Diagram with Data-Flow Errors

mean potential data needs and potential data results of flow objects. The specifications for mandatory data, alternatives for data, and optional data in the executable sub-class give precise information on data needs and data results. So-called `InputSets` and `OutputSets`, containing `DataInputs` and `DataOutputs`, represent this information, which is not visible in the graphical process diagram.

In the following, we first describe concepts for specifying process diagrams with data graphically and give an example of a BPMN 2.0 process diagram with different data-flow errors. Then we analyze the execution semantics of process flow elements handling data. In the following, we will use the term BPMN instead of BPMN 2.0 if it is unambiguous.

2.1 Data in Graphical Process Diagrams

The most important concepts regarding data flow in process diagrams are `DataObjects` and their `DataAssociations` to the flow elements task and event. A representation of a `DataObject`, visualized as document, can be a single instance or a collection of data elements of the same type. Process designers can define potential data needs and data results by modeling `DataAssociations` to or from `DataObjects`, visualized as dotted lines. They represent potential reads or writes on the `DataObjects`. `DataObjects` are unique within a process, but one `DataObject` can be referenced (i.e., visualized) several times in a process model. All specifications for `DataObjects` together determine the graphically visible data flow within a process. BPMN describes `DataObjects` local to the process, so a `DataObject` does not require an explicit deletion. `DataStores`, `DataInput` and `DataOutput` of processes allow to specify data exchange between databases and processes, thus they do not affect the data flow within the process. While tasks can have `DataAssociations` to and from `DataObjects`, events have either `DataAssociations` to or `DataAssociations` from `DataObjects`, dependent on their type (catching or throwing).

Example 2: *Figure 1 displays a process diagram for correcting answer sheets of a written exam. At the beginning, the* `DataInput Collection` ***Answer sheets*** *of the process contains a set of answer sheets (i.e., filled out solutions of the exam). Task Get answer sheet selects one element of a local copy of this collection and writes it to the* `DataObject` ***Answer sheet****. A performer of task Single correction marks this* ***Answer sheet*** *using correction guidelines given in* `DataObject` ***Answer key****. Task Check state of correction reviews whether all* ***Answer sheets*** *have been corrected or not. The task writes its result in* `DataObject` ***State****. It records the status of the correction process. The performer of task Single correction may identify a solution in an* ***Answer sheet*** *which is not part of the correction guidelines in* ***Answer key*** *up to now. In this case, Single correction writes the status Answer key incomplete to* ***State****, and the process runs task Complement answer key later on. Because of this,* ***State*** *is optional output of this task and is written on demand. As the graphical representation does not allow modeling optionality of output, this characteristic is not visible in Figure 1. In the other cases, a performer corrects the next answer sheet (Correction not done & Answer key complete), or the process terminates (Correction done).* □

2.2 Data in Executable Processes

To get executable processes, BPMN requires to refine the potential data needs and data results of a task or event,

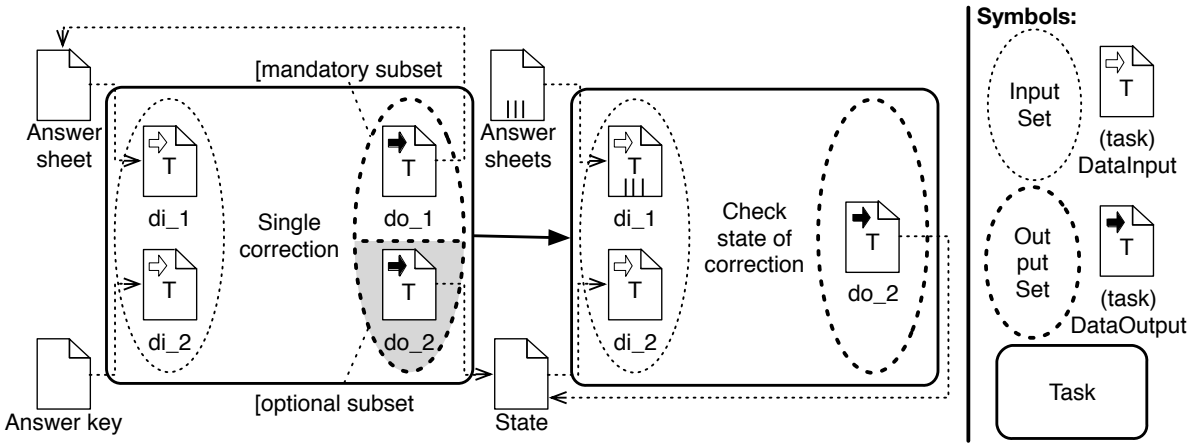


Figure 2: Data Specifications for two Executable Tasks

represented as `DataAssociations` in the process diagram, by a more concrete specification [5]. This is important because there exist several cases of ambiguity in graphical diagrams. For example, several input or output `DataAssociations` can mean that a task reads or writes all data elements, only one of them, or some combination. Further, process designers can combine specifications for alternatives and optional data.

Example 2 (cont.): *In Figure 1, it is unclear whether the data output* ***State*** *of task Single correction is optional or mandatory.* □

We now focus on the execution semantics for tasks. The data elements a task requires for its execution are its data needs. When a task is ready for execution, the process engine checks the *availability* of its data needs. The concepts `InputSet` and `OutputSet` describe the data needs and data results. Figure 2 shows an excerpt of our sample process of Figure 1. The boxes display two expanded tasks *Single correction* and *Check state of correction*, with their `InputSets` and `OutputSets`. Each `InputSet` consists of references to `DataInput`, which are associated with one or more `DataObjects`. In Figure 2, the ovals with dotted lines represent an `InputSet` and an `OutputSet` of a task (the first ones with small, the latter ones with larger dots), containing `DataInputs` and `DataOutputs`. For the visual representation of `DataInputs` and `DataOutputs` of a *task*, we have harnessed the visual notation of `DataInput` (empty arrow) and `DataOutput` (filled arrow) of processes and mark document symbols with "T".

An `InputSet` summarizes data needs of a task t, with a mandatory and an optional *subset*. In Figure 2, the parts of the ovals with grey background denote the *optional subset*s, and those with white background the *mandatory subset*s of `InputSets` or `OutputSets`. An `InputSet` $IS(t)$ is *available* if all `DataObjects` referred to in the mandatory subset of the `InputSet` $IS^M(t)$ are available. `DataObjects` referred to in the optional subset of the `InputSet` $IS^O(t)$ do not affect its availability. The converse concept to data needs is data results. They are the output of a task resulting from its execution. Analogously to the `InputSets`, `OutputSets` may have mandatory $OS^M(t)$ and optional subsets $OS^O(t)$. In Figure 2, *do_1* of task *Single correction* is mandatory (white), and *do_2* is optional (grey).

A task may have several `InputSets`, representing alternative data needs. If so, the process engine checks the availability of the `InputSets` in the order of their specification in the process model and takes the first one available to execute the task. In other words, at least one `InputSet` has to be available for execution. Analogously to the `InputSets`, a task may have several `OutputSets`. When a task terminates, the process uses one `OutputSet` for further execution. `InputSets` can optionally have references to the `OutputSets` that should be generated when the `InputSet` is used for the execution of the task (`outputSetRef`). This additional knowledge could be useful to limit the combination of `Input-/OutputSets` at a task to reduce the effort for model checking, see Subsection 3.4. However, the *execution semantics* in the standard does not take these relationships into account, i.e., they only may serve as guideline for implementing the task but not for detecting data-flow errors. Moreover, in the standard there does not exist any rule defining which `OutputSet` will be used during execution. The standard says: "The im-

plementation of the element where the `OutputSet` is defined determines the `OutputSet` that will be produced. So it is up to the Activity implementation or the Event to define which `OutputSet` will be produced." Consequently, for detecting data-flow errors we have to take the `OutputSet`s of a task without any ordering information of the `OutputSet`s or relationships to certain `InputSet`s.

2.3 Data-Flow Errors in BPMN 2.0

A common understanding is that a data flow of a process is correct if there are no anomalies regarding data processed [21]. To capture these anomalies, existing approaches [14, 21, 22] specify a set of data-flow errors for any data element of a process: *missing data*, *redundant data*, *lost data*, *inconsistent data*, and *wrongly or not destroyed data*. A *missing data* error occurs if a task needs a `DataObject`, but it is not available, because it has not been initialized yet. A *redundant data* error happens if a task writes a `DataObject` which is not read by any task in the subsequent process execution. A *lost data* error holds if a task writes a `DataObject`, and no upcoming task reads it until another task overwrites it. An *inconsistent data* error occurs if one task writes a `DataObject` and another task reads or writes it in parallel. BPMN does not foresee the possibility to delete `DataObject`s explicitly, so *wrongly or not destroyed data* is not relevant in our context. [22] further differentiate between weak and strong variants of redundant and lost data. In other words, it may be some or all execution paths containing the error. Regarding the data aspects of BPMN, these general classifications of errors are relevant as well. However, those publications do not cover the specifics of optionality of data as well as of alternatives for data in BPMN.

Optionality of Data: In BPMN a task reads or writes a `DataObject` mandatorily or optionally. This affects the data flow and its correctness. For example, optional `DataInput`s and optional `DataOutput`s can cause *lost data*, but with partly different effects as in the mandatory case. An *optionally lost data* error occurs if a task writes a `DataObject` optionally or mandatorily but it is not read by any task before it is optionally overwritten. This may be problematic, but does not have to be, in contrast to the mandatory case. Furthermore, optional outputs are not sufficient to avoid a *missing data* error.

Example 3: *Our example in Figure 1 contains four data-flow errors: (1 & 2) two* Missing Data *errors, (3)* Weakly Lost Data*, (4)* Strongly Redundant Data*. (1) In the first run of Single correction,* ***Answer key*** *is uninitialized. (2) Task Single correction initializes* ***State*** *only optionally, but Check state of correction needs it mandatorily. (3)* Weakly Lost Data *refers to* ***Answer sheet****. In two execution paths (Correction not done & answer key complete, Answer key incomplete) there is no task reading* ***Answer sheet*** *until task Get answer sheet writes it again. (4) There is no task using* ***Answer sheet*** *that has been updated by Single correction.* □

Alternatives for Data: The data flow of a BPMN process depends on the `InputSet`s and `OutputSet`s chosen during process execution. For example, to avoid compensations for uninitialized data with alternatives at runtime, this asks for analyzing data flow with respect to these alternatives for data at design time. A *missing data* error occurs in one alternative if at least one `DataObject` of this `InputSet` is uninitialized. In each `InputSet` the missing data error depends on the alternatives for `OutputSet`s chosen previously. To detect all potentially incorrect data alternatives, we have to consider all possible combinations of `InputSet`s and `OutputSet`s for each `DataObject` involved. By doing so, we do not have to take the order of `InputSet`s and `OutputSet`s into account.

Example 4: *Think of a slight modification of the example in Figure 1: Task Check state of correction requires* `DataObjects` ***Answer sheets*** *and* ***State*** *mandatorily. But now we assume that both* `DataObjects` *are alternative in use. I.e., there are two* `InputSets` *specified, each with one* `DataObject`*. In this case, the error* optionally missing data *occurs at task Check state of correction in one alternative for* `DataObject` ***State****.* □

Due to the specifics of BPMN with respect to data, detecting data-flow errors in BPMN asks for new anti-patterns and their specifications by using the classification of anti-patterns known from literature. This requires first to distinguish mandatory and optional data explicitly. Second, multiple `InputSet`s and `OutputSet`s must be taken into account. As both features are specified by the *executable subclass* of a process model, this need for new anti-patterns is not obvious having only a graphical process diagram at hand. Our anti-patterns for BPMN processes reflect alternatives for data and optionality, see Table 1.

3 Detecting Data-Flow Errors

To achieve data-flow correctness in BPMN 2.0 process models, we formalize the execution semantics regarding data-dependent flow elements in BPMN by using a transformation algorithm, see Section 3.1. In Section 3.2 we formalize generic data-flow anti-patterns for BPMN 2.0 and say how to generate process-specific anti-patterns for model checking. The final step is proving the possible

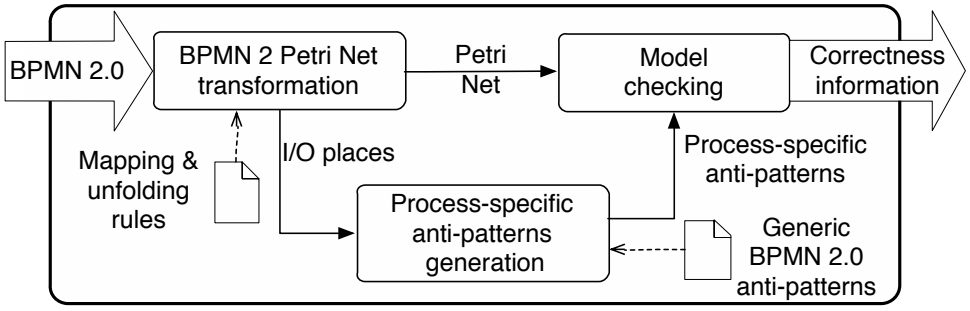


Figure 3: Overview of Our Approach on Detecting Data-Flow Errors

existence of data-flow errors in the process model with model checking. Figure 3 gives an overview of our approach.

3.1 The BPMN2PetriNets Transformation Algorithm

To analyze process models, many approaches employ Petri Nets to represent the models; see [24] or Appendix A for the definition. For example, [10] uses Petri Nets for representing BPMN 1.0 models and [11] for BPEL processes; [12] gives an overview of transformations of process models to Petri Nets.

We transform the *control flow* of BPMN models to Petri Nets by following the approach in [10]. However, [10] does not capture the data characteristics of BPMN 2.0. Existing work [2] transforms data objects but into BPMN 1.2. Their way to represent data with Petri Nets including the limited data perspective of BPMN 1.2 is not sufficient for our purposes, and their proprietary interpretation of the execution semantics of data objects and related states is not compatible with BPMN 2.0. In particular, supporting alternatives for data in [2] would inflate the Petri Nets because any possible combination of `InputSets` and `OutputSets` must be reflected. This asks for an appropriate transformation to represent alternative `InputSets` and `OutputSets`.

Alternatively, one could use Colored Petri Nets to represent data-dependent flow elements. This would slightly reduce the number of places of the Petri Nets, but would require complex firing rules for the representation of the execution semantics regarding data.

Due to the complexity of the required transformation, our BPMN2PetriNet algorithm transforms a BPMN process model into a Petri Net representation in two steps, see Algorithm 1: Step (1) *Mapping* the control flow to Petri Nets; Step (2) *Unfolding* the data needs and data results of mapped flow elements according to the execution semantics.

Mapping: The procedure `Map()` in Algorithm 1 starts with Step (1a) mapping the flow elements, including data elements, to a Petri Net using mapping rules. In Step (1b) it connects the mapped flow element with its successor and predecessor flow elements according to the sequence flow specified in the BPMN process model.

We distinguish two cases for the mapping.

Case (1): For the mapping of all data elements and data-dependent flow elements, i.e., tasks, events, and conditional sequence flows, we have developed new mapping rules to Petri Net representations. Our mapped Petri Nets depict data-dependent flow elements with interfaces for embedding data characteristics in the *unfolding step* later on. Further, we distinguish explicitly between a reading and a writing subnet, in the following called R/W subnet, of a mapped flow element. We represent multiple `InputSets` and `OutputSets` by combining the respective R/W subnets with an XOR split in the Petri Net. A place in the mapped Petri Net coordinates them. This gives rise to combining `InputSets` as well as `OutputSets` without the need for having to explicitly handle all possible combinatorial constellations of data uses.

Case (2): To map the sequence flow and data-independent flow elements, e.g., parallel gateways, we use an existing BPMN 1.0 to Petri Nets transformation [10], which requires some preconditions, e.g., for incoming and outgoing flows of split or join gateways, not affecting the generality of the proposed mapping [10]. This approach reflects the state-of-the art for transforming BPMN sequence flows.

We now describe our mapping rules for data elements and tasks, cf. Figure 4.

Data Elements. The mapping rules are applied to unique data elements, not to their references. The rules are straight-forward (see Figs. 4a and 4b): The two places $p.d\_id$ and $p.\neg d\_id$, with number of tokens $m(p.d\_id) + m(p.\neg d\_id) = 1$, represent the initialization status of a `DataObject` or a `DataInput` or - `DataOutput` of a process. We distinguish two cases: First, a `DataObject` or `DataOutput` of a process is uninitialized, i.e., $p.\neg d\_id$ has a token, see Fig. 4a. Sec-

Algorithm 1: The BPMN2PetriNet Transformation Algorithm

```
Algorithm BPMN2PetriNet()
    for each flow element f_i do
        Map(f_i)                                                    // Step (1)
        if f_i is a data-dependent flow element then
            if IS(f_i) ≠ ∅ ∨ OS(f_i) ≠ ∅ then
                Unfold(f_i)                                         // Step (2)
Procedure Map(Flow element f)
    Map f to Petri Net                                              // Step (1a)
    Connect f with predecessors & successors                        // Step (1b)
Procedure Unfold(Flow element f)
    if |IS(f)| ≥ 2 ∨ |OS(f)| ≥ 2 then
        Unfold InputSets/OutputSets of f                            // Step (2a)
    for each InputSet/OutputSet IOS_i(f) do
        for each data element d_j ∈ IOS_i(f) do
            Unfold input & output subsets of d_j in IOS_i(f)        // Step (2b)
            Record I/O places of d_j                                // Step (2c)
            if d_j ∈ O_M(IOS_i(f)) ∧ ∃ pred(d_j) then
                Generate predicate subnet of d_j                    // Step (2d)
```

ond, a DataInput of a process is already initialized, i.e., $p.d\_id$ has a token, see Fig. 4b. As a result, each unique DataObject is represented by two places in the Petri Net, even if the process model contains several references to this DataObject.

Tasks. The main idea behind the structure of a mapped task is to prepare the Petri Net for embedding complex data constellations, such as multiple InputSets or OutputSets as well as optional and mandatory data specifications. This asks for interfaces to expand a mapped task with input/output subnets for optional and mandatory data use as well as with combining places for data alternatives. Figure 4c shows the mapping rule for a task. Each mapped task has a reading and a writing (R/W) subnet with particular transitions. The abbreviations in the transition names mean a reading start (rs), a reading end (re), a writing start (ws), etc. The R/W subnets serve as interfaces for the input/output subnets generated in the unfolding step and represent the default setting, namely one 'empty' InputSet and one 'empty' OutputSet. But they do not contain their DataInput and DataOutput elements yet. The unfolding step will add them to the R/W subnets. Between the R/W subnet, there is place $p.t.t\_id$ to combine data alternatives. Additionally, the Petri Net representing a mapped task has two *connecting places* $p.x.t\_id$ and $p.t\_id.y$ (dashed lines) for connecting the mapped task with its predecessor flow element x and successor flow element y. Each reading subnet contains an input place $p.I\_i.t\_id$ and each writing subnet an output place $p.O\_i.t\_id$ (filled in grey). The input and output places, in short I/O places, in the R/W subnets are essential for checking the data-flow correctness.

Definition 1 (Input and output places): *An **input place** $p.I\_i.t\_id$ is a place of the reading subnet of the i-th InputSet of a task t_id with the following characteristics: If it has a token, all DataInputs of the i-th InputSet have been read successfully. The firing of transition $t.rs\_i.t\_id$ indicates the successful reading. An **output place** is the analogous place of a writing subnet.* □

The mapping rules for other data-dependent flow elements, i.e., events and conditional sequence flows, follow the same concept as described, see [25] for the details. In contrast to the mapping rules for tasks, the ones for events and conditional sequence flows are less complex. Mapped events have either one reading or one writing subnet, and mapped conditional sequence flows have a reading subnet only.

Unfolding: If a data-dependent flow element contains specifications for data, procedure Unfold() of Algorithm 1 adds Petri Net representations of the data needs and/or data results to an already mapped data-dependent flow element f. The added input/output subnets represent the execution semantics regarding data. In particular, our algorithm extends the default Petri Net repre-

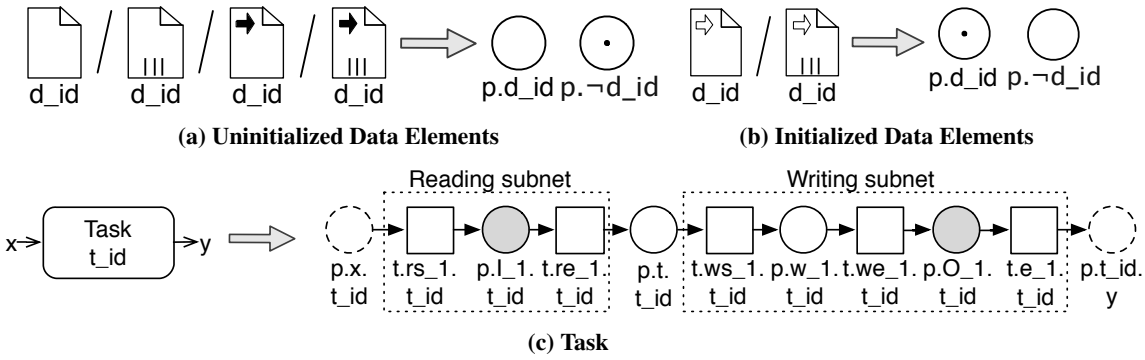


Figure 4: Mapping Rules

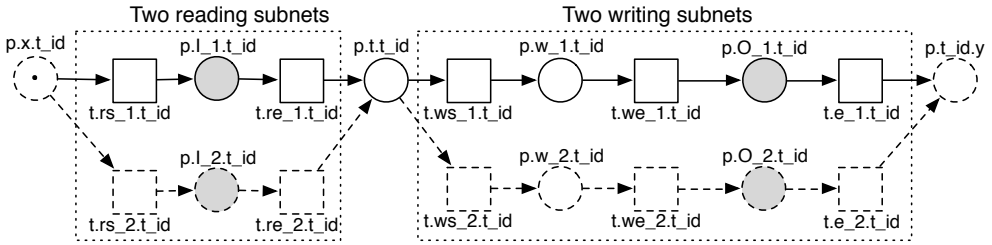


Figure 5: Task after Unfolding a Second `InputSet` and a Second `OutputSet` (Step (2a))

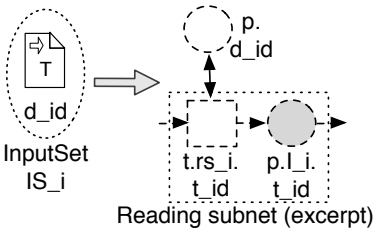
sentation by further `InputSets` and `OutputSets` if needed, and by representing the behavior for mandatory and optional data. It comprises the following four steps:

Step (2a): The mapped flow element contains one reading subnet or one writing subnet, representing one `InputSet` or one `OutputSet`. If f has several `InputSets` or `OutputSets`, we unfold each further `InputSet` $IS_i(f)$ by adding an additional reading subnet as alternative path to already existing reading subnets. For each further `OutputSet` $OS_i(f)$ we add an additional writing subnet as alternative path to already existing writing subnets. Figure 5 shows the resulting Petri Net of a task after unfolding a second `InputSet` and a second `OutputSet`. By doing so, we represent alternative `InputSets` and `OutputSets` of tasks and coordinate their paths at place $p.t.t\_id$, leaving aside their ordering.

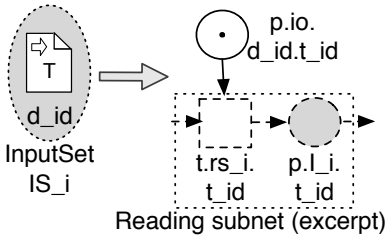
Step (2b): Each `InputSet/OutputSet` has a mandatory and an optional *subset* of `DataInput/DataOutput` elements. Figure 6 represents the subnets for these four different cases (input/output, mandatory/optional) for unfolding one element of an `InputSet/OutputSet` of a flow element t_id as well as their interfaces. The parts visualized with dashed lines in Fig. 6 represent the already mapped flow elements (c.f. procedure `Map()`) of the Petri Net. For each `DataObject` d_j in an `InputSet/OutputSet` we add the appropriate input/output subnet. Next, we connect each unfolded input/output subnet with the reading or writing subnet of the flow element mapped and with the two places $p.d\_id$ and $p.\neg d\_id$, which represent the data object d_j mapped.

This results in a representation of the execution semantics regarding data for `InputSets`/`OutputSets`, as well as for the process as a whole: The subnets connected to the reading or writing subnet of a mapped data-dependent flow element reflect the local execution semantics of a flow element regarding data. Further, the connections of the unfolded input/output subnets to the mapped places of `DataObjects` account for the global behavior of a process. The latter is because for each unique `DataObject` there exist exactly two places in the resulting Petri Net representing whether `DataObject` is initialized or not. Any subnet of the Petri Net representing the local execution semantics for a particular `DataObject` is connected to these two process-global places of the particular `DataObject`. These connections follow the structure as given in the unfolding rules (see Fig. 6). To illustrate, there are connections from transitions $t1$ and $t2$ to the two places for a subnet representing a mandatory output (see Fig. 6c). For a subnet representing an optional output (see Fig. 6d) transitions $t1$, $t2$, $t3$ and $t4$ are connected. We will describe the subnets in detail at the end of this subsection.

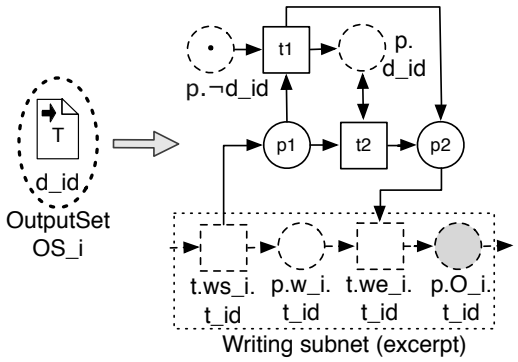
Step (2c): This step records all I/O places as well as information on the kind of usage (mandatory/optional, as



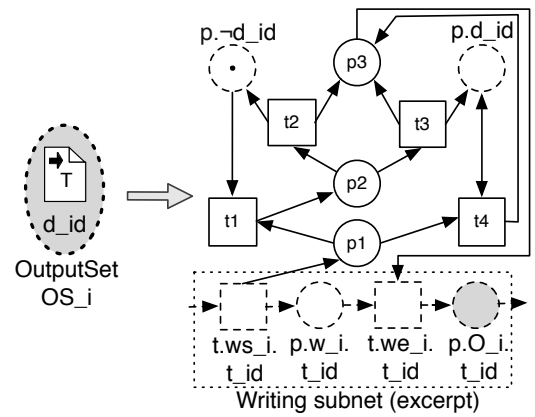
(a) Mandatory Input Subnet



(b) Optional Input Subnet



(c) Mandatory Output Subnet



(d) Optional Output Subnet

Figure 6: Unfolding Rules

alternatives) for `DataObject` d_j. We deploy this for the following process-specific anti-pattern formula generation.

Step (2d): If OS_i refers the `DataObject` mandatorily, and if any condition of a conditional sequence flow refers the `DataObject`, we generate a so-called predicate subnet [22]. [22] introduced 'unfolding' of predicate subnets to express that a `DataObject` may hold several values which influence the process execution by data-dependent gateways.

Input/Output Subnets. We now describe the unfolding rules, i.e., the input and output subnets, in more detail by focusing on the local execution semantics for data. Figure 6a displays the subnet for a *mandatory input*. As a task requires its mandatory data input for its execution, the subnet specifiying one mandatory `DataObject` is very simple: It consists of a bidirectional arc from the place $p.d\_id$, representing that the `DataObject` is initialized, to the reading transition of the mapped task. The bidirectional arc ensures that the `DataObject` stays initialized. An *optional data* input does not affect the execution semantics of a task. Thus, the subnet contains an optional input place $p.io.d\_id.t\_id$ with a token and a connection to the reading subnet of the unfolded task, see Fig. 6b. The subnet for a *mandatory output*, see Fig. 6c, reflects that a task t_id initializes a `DataObject` in any case, i.e., place $p.d\_id$ finally has a marking. Before the start of task t_id, either place $p.d\_id$ or place $p.\neg d\_id$ has a marking. Accordingly, either transition $t1$ or $t2$ fires. This subnet is connected to the writing part of the task mapped. The subnet in Fig. 6d covers all three execution scenarios of an *optional output* d_id of an `OutputSet` $OS(t\_id)$ of a flow element t_id: *(a)* transitions $t1$ and $t3$ fire if d_id has no value before start of t_id, and t_id initializes d_id, *(b)* transitions $t1$ and $t2$ fire if d_id has no value before start of t_id and t_id does not write to d_id, and *(c)* $t4$ fires if d_id is initialized before start of t_id, and t_id writes d_id again.

3.2 Formalization of BPMN 2.0 Data-Flow Anti-Patterns

In this section we formalize the data-flow errors as anti-patterns, which we have identified as relevant for a BPMN process model, see Section 2.3. The result is a set of so-called generic anti-patterns, reflecting the specifics of BPMN for several `Input-` and `OutputSets` and for optional data. These generic anti-patterns are independent of a concrete process model. Then we say how to generate process-specific anti-patterns.

Formalizing Generic Anti-Patterns: The ideas behind the formalization are as follows. BPMN allows specifying the data needs and data results of flow elements. This results in a data flow from the perspective of an in-

dividual DataObject. Thus, we examine data-flow errors for each DataObject d. To consider alternatives of data modelled as several Input- and OutputSets, we analyze the data flow regarding the combinations of these alternatives. Further, supporting mandatory or optional use of data gives rise to several data-flow errors we define anti-patterns for. We aim to achieve correctness at design time, however, the availability of data is only known during execution. This is why we have to analyze all possible execution variants that contain execution paths determined by the control flow as well as by the choice of alternative data needs and data results. These variants are contained in the state space of the unfolded Petri Net model which we use for error detection.

We illustrate how to formalize the data-flow errors with anti-patterns, using the example of a *Missing Data* flow error of a DataObject d. This error occurs if the process contains a flow element f which needs d mandatorily, and the process has no flow element f' which initializes d mandatorily before the execution of f. f and f' might have several alternative data needs in combination with several alternative data results. Hence, we consider all possible combinations of input and output alternatives where d is used to analyze its data flow. Note that the data flow of d considers its availability in the whole process (not only local to a certain task or event).

As a basis for our formalization, for each DataObject d we need information on where in the process d can be processed (as input or output, optionally or mandatorily). To this end, we now define these sets of BPMN flow elements for a certain DataObject. We will make use of these sets to specify the anti-patterns. For the definitions that follow we assume a process model containing a number of tasks $|tasks|$, a number of events $|events|$, and a number of conditional sequence flows $|conds|$. $set_f = tasks \cup events$ denotes the set of all tasks and events of the process model. A task or event f has a number of InputSets and OutputSets that we denote with $|IS(f)|$ and $|OS(f)|$. Note that these sets contain DataObjects as data needs and data results. Let $IS_i^O(f_m)$ be the subset containing the optional DataObjects of the i-th InputSet IS_i of the m-th task or event, and $OS_i^O(f_m)$ the subset containing the optional DataObjects of the i-th OutputSet OS_i of the m-th task or event; an event has at most one InputSet or one OutputSet depending on its type.

In Definition 2, we summarize all InputSets $Set\_I_O(d)$ resp. OutputSets $Set\_O_O(d)$ d is an optional element of. Using these sets, $I_O(d)$ of type Boolean specifies whether d has been read successfully in one of its InputSets; correspondingly, $O_O(d)$ of type Boolean specifies whether d has been written successfully in one of its OutputSets.

For the mandatory use of d, we define $I_M(d)$ and $O_M(d)$ analogously to Definition 2 In addition to tasks and events, a conditional sequence flow s can use a DataObject within a condition $cond(s)$ mandatorily as well. For Definition 3, $set_f = tasks \cup events \cup conds$ denotes the set of all tasks, events and conditional sequence flows of the process model. The definitions of $I_M(d)$ and $O_M(d)$ are analogous to the optional case. In the following definition, we summarize $Set\_I_M(d)$ is the set of all InputSets d is a mandatory element of. Analogously, $Set\_O_M(d)$ is the set of all such OutputSets. Using these sets, $I_M(d)$ specifies whether d has been read successfully in one of the InputSets of $Set\_I_M(d)$. Correspondingly, $O_M(d)$ specifies whether d has been written successfully in one of the OutputSets of $Set\_O_M(d)$. *read(x)* and *written(x)* as defined in Definitions 2 and 3 are formalized in the generation step of process-specific anti-patterns, see below.

We now formalize the anti-patterns for BPMN using a temporal logic formalism, namely CTL (**C**omputation **T**ree **L**ogic). See [8], Appendix A for a description of CTL. [8] introduces an effective algorithm to verify properties specified in CTL on Petri Net models. To achieve data-flow correctness, our approach aims to identify errors which occur during reading or writing of a DataObject in the context of a certain choice of alternatives. Next, the specification of data needs and results in BPMN allows for optional or mandatory use of data. This gives rise to a distinction between optional and mandatory errors, as described in Section 2.3. Further, we differentiate between *weak* and *strong* variants of redundant and lost data, see Section 2.3. The result is a set of *generic anti-patterns* for a DataObject d, in the following called *Data-Flow Anti-Patterns (DAP)*. Table 1 lists our generic anti-patterns tailored to the execution semantics of BPMN 2.0.

In the following, we explain our generic anti-patterns for BPMN 2.0. We provide their formal specification in CTL.

DAP 1: Missing Data occurs if a DataObject d is a mandatory input $I_M(d)$, but not a mandatory output $O_M(d)$ before, i.e., d is not initialized.

DAP 2: Missing Optional Data means that a DataObject d is optional input $I_O(d)$, and d is not initialized before by a mandatory output $O_M(d)$ or by an optional output $O_O(d)$.

DAP 3: Strongly Redundant Data is given if a DataObject d is a mandatory output $O_M(d)$, but there is no flow element using this DataObject as mandatory input $I_M(d)$ or as an optional input $I_O(d)$ in all following execution paths.

DAP 4: Weakly Redundant Data happens if there exists at least one execution path of the process where a

Definition 2 (Optional `InputSets/OutputSets` containing `DataObject` d):

$$Set\_I_O(d) = \bigcup_{m \in \{1..|set_f|\}} \{IS_i(f_m) \mid d \in IS_i^O(f_m) \ \wedge \ i \in \{1..|IS(f_m)|\}\}$$

$$Set\_O_O(d) = \bigcup_{m \in \{1..|set_f|\}} \{OS_i(f_m) \mid d \in OS_i^O(f_m) \ \wedge \ i \in \{1..|OS(f_m)|\}\}$$

$$read(x) = \textit{ all } \texttt{DataInputs} \textit{ in } \texttt{InputSet } x \textit{ have been read successfully } \% \textit{ type: boolean}$$

$$written(x) = \textit{ all } \texttt{DataOutputs} \textit{ in } \texttt{OutputSet } x \textit{ have been written successfully } \% \textit{ type: boolean}$$

$$I_O(d) = \bigvee_{x \in Set\_I_O(d)} \big(read(x)\big) \qquad O_O(d) = \bigvee_{x \in Set\_O_O(d)} \big(written(x)\big)$$

Definition 3 (Mandatory `InputSets/OutputSets` containing `DataObject` d):

$$Set\_I_M(d) = \{s_l \mid d \in cond(s_l) \ \wedge \ l \in \ \{1..|cond|\}\} \cup \Big(\bigcup_{m \in \{1..|set_f|\}} \{IS_i(f_m) \mid d \in IS_i^M(f_m) \ \wedge \ i \in \{1..|IS(f_m)|\}\}\Big)$$

$$Set\_O_M(d) = \bigcup_{m \in \{1..|set_f|\}} \{OS_i(f_m) \mid d \in OS_i^M(f_m) \ \wedge \ i \in \{1..|OS(f_m)|\}\}$$

$$read(x) = \textit{ all } \texttt{DataInputs} \textit{ in } \texttt{InputSet } x \textit{ have been read successfully } \% \textit{ type: boolean}$$

$$written(x) = \textit{ all } \texttt{DataOutputs} \textit{ in } \texttt{OutputSet } x \textit{ have been written successfully } \% \textit{ type: boolean}$$

$$I_M(d) = \bigvee_{x \in Set\_I_M(d)} \big(read(x)\big) \qquad O_M(d) = \bigvee_{x \in Set\_O_M(d)} \big(written(x)\big)$$

`DataObject` d is neither a mandatory input $I_M(d)$ nor an optional input $I_O(d)$, but `DataObject` d is a mandatory output $O_M(d)$ before.

DAP 5: Redundant Optional Data holds for a `DataObject` d if an optional output $O_O(d)$ is not used later on, i.e., `DataObject` d is no mandatory input $I_M(d)$ or optional input $I_O(d)$ in one of the succeeding execution paths.

DAP 6: Strongly Lost Data occurs if a `DataObject` d is mandatory output $O_M(d)$ several times, but before the later mandatory outputs happen, there is no mandatory input $I_M(d)$ in between. This situation holds for all execution paths and means that earlier data results (e.g., initializations, updates) for d are lost.

DAP 7: Weakly Lost Data means that a `DataObject` d is mandatory output $O_M(d)$, and there exists at least one execution path where it is mandatory output $O_M(d)$, but not mandatory input $I_M(d)$ before.

DAP 8: Optional `DataOutput` does not lead to an error in every case. *Lost optional data* are `DataObjects` d which are optional output ($O_O(d)$). Additionally, no upcoming flow elements exist that read d ($I_M(d) \vee I_O(d)$) until a flow element writes d mandatorily, i.e., U $O_M(d)$.

DAP 9: Optionally Lost Data includes all `DataObjects` d which are mandatory ($O_M(d)$) or optional output ($O_O(d)$), and, additionally, there is a flow element that optionally writes d subsequently without a flow element reading d in between (optionally or mandatorily).

DAP 10: Inconsistent Data Two elements are in conflict if one element f writes d, i.e., d belongs to its (optional or mandatory) data results ($OS(f)$), and another element f' reads ($d \in IS(f')$) or writes d optionally or mandatorily ($d \in OS(f')$). d is inconsistent if two elements in conflict regarding d can be executed in parallel.

Figure 7 shows examples of process models which illustrate the ten anti-patterns of Table 1. Some of these models include more than one data-flow error. For instance, the model in Figure 7f illustrates strongly lost data and also features a strongly redundant data-flow error. This is because the *Data Object* is not read by any task before the process terminates. Furthermore, each strong data-flow error also is a weak one.

Generating and Checking Process-Specific Anti-Patterns: Using our generic BPMN data-flow anti-patterns, the Step *Process-specific anti-patterns generation* of our approach in Figure 3 delivers specific formulas of the generic anti-patterns for each `DataObject`. To this end, we use the results of the transformation from

Anti-Patterns - DAP	Formalization
1 Missing Data	$\mathrm{E}\big(\neg O_M(d)\ \mathrm{U}\ I_M(d)\big)$
2 Missing Optional Data	$\mathrm{E}\big(\neg(O_O(d) \vee O_M(d))\ \mathrm{U}\ I_O(d))\big)$
3 Strongly Redundant Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{AX}(\mathrm{A}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ term])\big)$
4 Weakly Redundant Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{EX}(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ term])\big)$
5 Redundant Optional Data	$\mathrm{EF}\big(O_O(d) \wedge \mathrm{EX}(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ term])\big)$
6 Strongly Lost Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{AX}(\mathrm{A}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_M(d)])\big)$
7 Weakly Lost Data	$\mathrm{EF}\big(O_M(d) \wedge \mathrm{EX}\big(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_M(d)]\big)\big)$
8 Lost Optional Data	$\mathrm{EF}\big(O_O(d) \wedge \mathrm{EX}\big(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_M(d)]\big)\big)$
9 Optionally Lost Data	$\mathrm{EF}\big(\big(O_M(d) \vee O_O(d)\big) \wedge \big(\mathrm{EX}(\mathrm{E}[\neg(I_M(d) \vee I_O(d))\ \mathrm{U}\ O_O(d)])\big)\big)$
10 Inconsistent Data	$\bigvee_{f \in \{E \cup T\} \wedge d \in OS(f)\}} \mathrm{EF}\big[exec(f) \wedge \bigvee_{f' \neq f \wedge (d \in IS(f') \vee d \in OS(f'))} exec(f')\big]$

Legend (for CTL formulas):
A path operator (A or E) occurs together with a state operator (X, F, U).
A/E: the formula needs to hold in *all/at least one of the* succeeding execution paths.
X/F: the formula holds in the *next/at least in one succeeding* state.
$[\phi_1\ \mathrm{U}\ \phi_2]$: ϕ_1 holds until ϕ_2 is reached.
$exec(f)$ means that flow element f is ready to be executed, i.e., the respective transition is activated.
$term$ denotes the termination of the process.

Table 1: BPMN 2.0 Generic Anti-Patterns for a `DataObject` d

BPMN to Petri Nets with its *Input and Output Places*, see Definition 1. The information recorded in Step (2c) of Algorithm 1 comprises the I/O places as well as information on the kind of usage (mandatory/optional, in alternatives) of the `DataObject` d. The generation step of process-specific anti-patterns instantiates the generic BPMN 2.0 anti-patterns by means of the optional and mandatory usages of a `DataObject` d according to Definition 2. In order to be able to check whether d is an optional or mandatory data input, the model checker has to prove that an $IS \in Set\_I_O(d)/\in Set\_I_M(d)$ is available. To do so, it checks whether the input place of IS contains a token, see Definition 1. This means that we instantiate the anti-patterns by replacing $read(IS)$ (an `InputSet` of a flow element t_id) with m($p.I\_i.t\_id$) $= 1$ (respectively for a flow element e_id)). The picture is analogous for the `OutputSets`. $written(t\_id)$ is replaced using the *output place*, i.e., with $m(p.O\_i.t\_id) = 1$. The alternative use of d induces additional path expressions of the data flow in the process-specific anti-patterns.

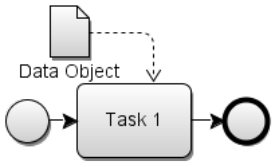
For the final step, we employ a representation of the possible states of the process model to find the states with errors. A CTL model checker, namely LoLA [19], analyzes the process-specific formulas of the anti-patterns for each `DataObject` on the unfolded Petri Net model in question. If a formula is satisfied, the data-flow error is detected.

3.3 Tool Support

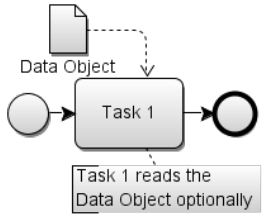
We have implemented a tool for automatically analyzing data-flow errors in process models specified with BPMN 2.0, see Figure 3. Input to the tool is a well-formed and sound BPMN process model, see also Subsection 3.4. For each data object of the model in question, the tool generates the process-specific anti-patterns. For each of these anti-patterns in turn, it runs the model checker LoLA to prove it. If a data-flow error has been found, our tool returns the data object and the task of the BPMN model where the error occurs. The naming of a BPMN task corresponds to the label of the respective place of the Petri Net. A special case is a *missing data* error of a data object d, because it causes a dead transition in the Petri Net. In this case the tool inserts an initialization of d into the Petri Net to repair the error before checking the other anti-patterns. Our evaluation makes use of the tool, see Section 4.

3.4 Discussion

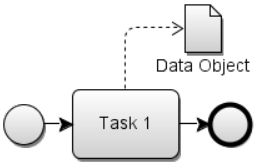
Features Supported: We support all standardized data features of BPMN 2.0. Our data-flow verification considers BPMN 2.0 process models according to the process execution conformance type of the BPMN 2.0 standard. We do not consider business process choreogra-



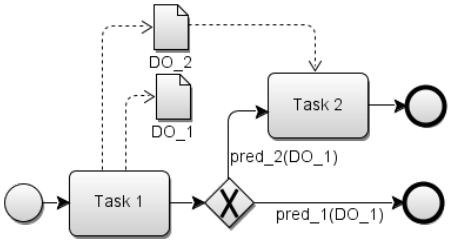
(a) DAP 1 - Missing Data



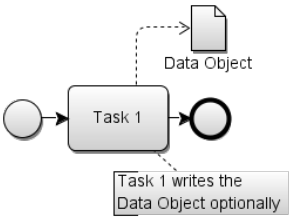
(b) DAP 2 - Missing Optional Data



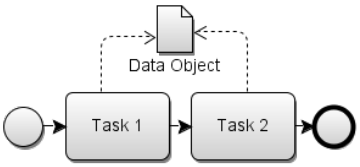
(c) DAP 3 - Strongly Redundant Data



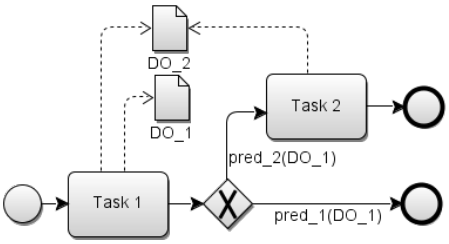
(d) DAP 4 - Weakly Redundant Data



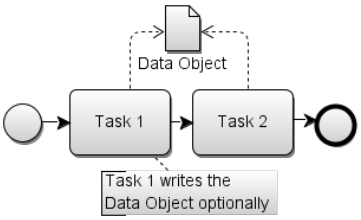
(e) DAP 5 - Redundant Optional Data



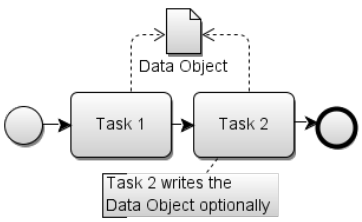
(f) DAP 6 - Strongly Lost Data



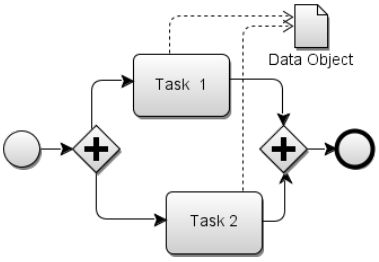
(g) DAP 7 - Weakly Lost Data



(h) DAP 8 - Lost Optional Data



(i) DAP 9 - Optionally Lost Data



(j) DAP 10 - Inconsistent Data

Figure 7: Examples of Process Models Illustrating all Anti-Patterns of Table 1

phies. Specifying more complex data dependencies in data-aware business process models, see e.g., [15], [27], [7] is not our current focus. Such proposals extend the standard, e.g., by annotations for newly created and deleted data objects, by introducing semantics on states of the data objects forming a life-cycle, or by semantic data constraints, e.g., in preconditions of tasks.

Incorrect Process Models: Our focus is on data-flow correctness. Here, we assume that the input process model is well-formed and sound. In other words, data-flow correctness and soundness are issues that can be dealt with separately from each other. This means that our tool can be integrated into a verification tool chain for BPMN process models, after the soundness check.

Complexity: The state space depends on the different cases enhancing the Petri Net: A) inserting the input/output subnets for each `DataInput/Ouput` of each `Input-/OutputSet` adds up to a locally restricted increase, i.e., with minor problems for verification, and B) inserting paths for several `Input-/Output- Sets` can result in a polynomial increase, accounting for state space explosion.

A) For each `DataInput` or `DataOutput` of a flow element, a predefined number of states is added to the state space. This is due to the additional input/output subnets of the unfolding step. $c = 4$ is the maximum number of states for these subnets, namely the number of states of subnet 6d. Let $n(f)$ be the number of data needs and data results of the flow element f. Then each state containing a marked place which represents a flow element f with data incurs the following number of additional states at most: $c^{n(f)}$.

B) The occurrence of a `DataObject` in several `Input/OutputSets` of tasks all over the process model can generate a growth of the state space in polynomial order. This is because alternatives for data generate parallel paths. This so-called state-space explosion is a well-known problem for model checking [9] and can hinder verification. Currently, alternatives for data in processes is a new concept in BPMN 2.0. We plan to investigate respective optimization methods as future work.

4 EVALUATION

The evaluation of our approach consists of several steps from designing a collection of process models with intensive use of data needs and data results, explicitly adding data-flow errors into some of the process models to a user experiment for modeling data in process models given. Step 1 is auxiliary, to prepare the evaluation of Steps 2 and 3.

To evaluate process models of realistic complexity, we first looked for publicly available process models which we could use as benchmarks. In particular, this includes the huge process repository of the BPM Academic Initiative [4] for BPMN 2.0 process models with intensive data usage. Unfortunately, only a small fraction of the process models contains `DataObjects`. Next, even fewer models of this share conform to the standard. For instance, some models have non-conformant annotations of message flows with `DataObjects`. In addition, the most important feature required for analyzing BPMN 2.0 process models with data is missing: None of the process models contains specifications for `DataInput`, `DataOutput`, `InputSet`, `OutputSet`, etc., being part of executable processes (see Section 2.2). We argue that in business processes modeling data as first-class objects is not ubiquitous yet and is very new in BPMN. Furthermore, we hypothesize that BPMN process models with specifications for optional data and alternatives are absent because the know-how regarding these new features currently is only in an early state. Without such specifications, the conventional detection of data flow errors (see for instance [14, 18, 22]) is possible, but not a detection of data-flow errors for optional data and alternatives of data in BPMN 2.0. Consequently, we have faced the problem that such a set of processes that we could use as benchmark has not yet been available publicly.

To deal with this problem, we have asked an expert to design process models for 11 scenarios. These scenarios use data intensively. Some examples are adapted from literature, others we have developed ourselves. The scenarios comprise, say, order handling (S1) and job interview (S7) and are available online, see http://dbis.ipd.kit.edu/2134.php. These process models do not have any data-flow errors according to our definition. We also checked these process models (namely the Processes Sx.1 in Table 2) with our data-flow analysis tool, which confirmed their error-freeness. Sx.n stands for the n-th variant of a process model of Scenario x.

In Step 2, our expert has added data-flow errors to some of the error-free process models. These process models cover all types of data-flow errors (see S3.2, S5.2, S8.2, S9.2 and S10.2 in Table 2). Our data-flow analysis tool has correctly detected all of them.

In Step 3, we have run a user experiment to understand the difficulties of modeling an error-free data flow, and also to obtain process models with data-flow errors for further evaluation of our tool. We have organized this experiment as an exercise of a lecture with seven students with knowledge in BPMN modeling, including the data aspect. These experienced individuals have started with two process models given, namely S1.1 and S2.1 (from Step 1 of our evaluation). We had removed the data elements from the models before. The task has been to enhance the process models with data needs and re-

Process	# Tasks	# Events	# XOR Splits	# conditions	# DataObjects	# DataAssoc.	# Places	# Transitions	# Miss. Data	# Redund. Data	# Lost Data	# Incon. Data
S1.1-8	6	4	1	2	3-8	5-12	49-72	40-50	0-4	0-4	0	0
S2.1-6	8	2	1	2	4-10	12-21	63-84	47-58	0-3	0-3	0-1	0
S3.1	8	2	3	6	5	17	85	74	0	0	0	0
S3.2	8	2	2	4	5	15	80	70	0	2	1	0
S4.1	14	2	0	0	12	24	107	82	0	0	0	0
S5.1	7	2	1	2	5	15	64	52	0	0	0	0
S5.2	7	2	1	2	6	16	64	52	0	0	2	0
S6.1	8	2	2	4	7	17	90	70	0	0	0	0
S7.1	6	2	1	2	5	10	58	33	0	0	0	0
S8.1	13	3	2	5	8	22	110	94	0	0	0	0
S8.2	14	3	2	5	9	16	121	104	1	5	5	2
S9.1	8	2	1	2	3	21	72	58	0	0	0	0
S9.2	8	2	1	2	5	24	79	62	0	2	2	0
S10.1	8	2	2	4	5	12	75	60	0	0	0	0
S10.2	8	2	2	5	6	13	91	81	0	1	0	0
S11.1	19	3	2	5	6	13	91	81	0	0	0	0

Table 2: Evaluation of the Correctness Tool

sults. Modifying them has also been allowed if necessary. We textually described the use of data, so that the participants were able to model the data perspective of the processes. We have obtained several process models for the two scenarios. They differ in the number of `DataObjects` and control flow elements, see Scenarios S1 & S2 in Table 2. Because the process models are not overly complex, we have been able to check manually if they contain data-flow errors. This serves as gold standard to evaluate our tool. Using it, we then have checked the models. Our tool has detected all data-flow errors contained in the models, 40 errors altogether.

Table 2 gives an overview of the results of our evaluation, i.e., detecting data-flow errors and confirming the correctness of process models without data-flow errors. Because of better readability we present the results for Scenarios 1 and 2 in an aggregated form. See [25] for the detailed results. We list the size of the BPMN process models analyzed, the size of the Petri Nets generated and the number of data-flow errors identifed. The number of the BPMN elements determines the size of the corresponding Petri Net, defined by the number of transitions and places. The input and output subnets in particular, added in the unfolding step of our transformation, increase the size of the Petri Net.

For one, the experiment of Step 3 (user experiment) shows that our tool has detected all data-flow errors in the process models created. Further, *Missing Data* is a frequent error in process modeling. The numbers of *Redundant Data* errors and of *Lost Data* errors reflect that we count both strong as well as weak variants of data-flow errors. *Inconsistent Data* errors occur when different tasks read or write a `DataObject` in parallel. This only happens with parallel execution paths. Only one of our scenarios (S8.2) has this characteristic. All in all, the evaluation shows that in process modeling all types of data-flow errors are relevant and can occur.

5 Related Work

Behavioural analysis of process models without the data aspect has been studied extensively, see [13] for an overview, but is not the focus of our work. In what follows, we concentrate on the correctness of the data flow. [18] is one of the first approaches illustrating the importance of data-flow correctness in process modeling. The authors have thoroughly analyzed problems which can occur with a data flow but do not provide a solution for error detection. [3] defines data-flow errors as patterns, but focuses on visually specifying compliance rules in order to explain the violations. There, key requirements are availability of data input and data output of an activity, and consistent flow of data between two activities. We in turn use the patterns to express the execution semantics of process models with data elements and thus analyze the correctness of the data flow not restricted to the availability perspective supported in [3]. [23] proposes using patterns for the analysis of general compliance violations. In particular, order and occurrence patterns support the users when specifying constraints on a

process model with data. However, they do not support BPMN but use BPEL with its specific data semantics for process modeling. This is different from BPMN and also does not handle optional data or alternatives for data.

[22] introduces a method based on CTL* that combines the detection of control-flow and data-flow errors. They use anti-patterns for missing data, inconsistent data, redundant data, and lost data for WorkflowNet process models. In contrast to our approach, they do not cover the BPMN 2.0 semantics of data during process execution, including specific more elaborate ways to use data, i.e., alternatives for data and mandatory as well as optional data. [21] regards data-flow analysis in processes based on UML activity diagrams. The authors provide separate correctness proofs directly implemented as procedures for each of the three basic types of data-flow errors, namely missing data, conflicting data, and redundant data. [14] extends the results of [21] for UML activity diagrams, by discussing additional data-flow errors such as inconsistent data. For error detection they also use separate checking procedures for each error type. To do so, they have to explicitly generate and store all possible paths of the process model. Due to XOR-nodes in particular, their number tends to be daunting. Another drawback is that they do not use a state-based approach to represent the dynamic behaviour of processes (e.g., with a state space of a Petri Net). Further approaches exist using UML data-flow analysis on UML activity models, e.g., [20], and [26], with another focus than ours. [17] deals with data-flow correctness of BPMN 2.0. They use the work of [21] adding optional reading and writing access. To this end, they add behavioral profiles consisting of information on conflicts between a pair of nodes of a process model. In other words they establish behavioral relationships each between a pair of tasks (nodes) of the process and use this list of relationships for the data-flow analysis. In contrast to our method, their modeling of errors with behavioral profiles handles data separately from the process model. Further, they do not take alternatives into account. Finally, they address only some of the data-flow errors, our approach can cope with. For instance, they define inconsistent data only for write-write conflicts, do not distinguish between redundant data with optionality, and do not consider lost data.

Summing up, our approach for detecting data-flow errors uses a process formalization by Petri Nets. To achieve this, we have provided new transformation rules, to represent execution semantics for data in particular. Our new data-flow anti-patterns distinguish mandatory and optional data as well as alternatives. By doing so, they cover any anomalies known from existing work. In particular, we are, to our knowledge, first to support the data-flow perspective of BPMN 2.0, as well as alternatives for data and mandatory and optional data.

Considering more intricate dependencies of data objects in data-aware process models is subject of further approaches. For instance, some deal with data dependencies like inclusion, referential dependencies [15], semantically defined constraints [6] or with information leaks [1]. These approaches focus on dependencies which are not part of the BPMN data perspective and would require to enhance its specification concepts. In other words, the problem is different from the one studied here.

6 Conclusions

In this paper, we have proposed a new method for detecting data-flows errors in BPMN 2.0 process models at design time. This approach takes alternatives for data as well as optional data into account. An automatic detection scheme requires a formal representation of the execution semantics of BPMN 2.0 flow elements with data associations. To achieve this, we have developed transformation rules and a set of anti-patterns representing data-flow errors in BPMN 2.0 process models. On this basis, we transform data-dependent flow elements of a process model into unfolded Petri Nets to detect data-flow errors by using an existing model checker. Experiments with users have shown that our tool identifies the data-flow errors present.

Acknowledgement

This research was supported in part by the European Social Fund and by the Ministry Of Science, Research and the Arts Baden-Württemberg.

We acknowledge support by Deutsche Forschungsgemeinschaft and Open Access Publishing Fund of Karlsruhe Institute of Technology.

References

[1] R. Accorsi and A. Lehmann, "Automatic Information Flow Analysis of Business Process Models", in *Proc. BPM*, 2012.

[2] A. Awad, G. Decker, and N. Lohmann, "Diagnosing and Repairing Data Anomalies in Process Models", in *Proc. BPM Workshops*, 2010.

[3] A. Awad, M. Weidlich, and M. Weske, "Visually Specifying Compliance Rules and Explaining Their Violations for Business Processes", *Visual Languages and Computing*, vol. 22, no. 1, 2011.

[4] Business Process Academic Initiative, "The BPM Academic Initiative Model Collection". bpmai.org/download, accessed 14th August 2014.

[5] Object Management Group, "Business Process Model and Notation, V2.0", *OMG Specification*, 2011.

[6] D. Borrego et al, "Diagnosing Correctness of Semantic Workflow Models", *Data & Knowledge Engineering*, vol. 87, Sept 2013.

[7] C. Cabanillas et al, "Automatic Generation of a Data-Centered View of Business Processes", in *Proc. CAiSE*, 2011.

[8] E. Clarke, E. Emerson, and A. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications", in *Proc. TOPLAS*, 1986.

[9] E. Clarke, O. Grumberg, and D. Peled, "Model checking", *MIT Press*, 1999.

[10] R. Dijkman, M. Dumas, and C. Ouyang, "Semantics and Analysis of Business Process Models in BPMN", *Information and Software Technology*, vol. 50, no. 12, 2008.

[11] S. Hinz, K. Schmidt, and C. Stahl, "Transforming BPEL to Petri Nets", in *Proc. BPM*, 2005.

[12] N. Lohmann, E. Verbeek, and R. Dijkman, "Petri Net Transformations for Business Processes – A Survey", in *Transactions on Petri Nets and Other Models of Concurrency II*. 2009.

[13] Z. Manna and A. Pnueli, "The Temporal Logic of Reactive and Concurrent Systems – Specification", *Springer*, 1992.

[14] H. S. Meda, A. K. Sen, and A. Bagchi, "On Detecting Data Flow Errors in Workflows", *Data and Information Quality*, vol. 2, no. 1, 2010.

[15] A. Meyer et al, "Modeling and Enacting Complex Data Dependencies in Business Processes", in *Proc. BPM*, 2013.

[16] OASIS, *Web Services Business Process Execution Language Version 2.0*, April 2007.

[17] A. Rogge-Solti et al, "Business Process Configuration Wizard and Consistency Checker for BPMN 2.0", in *Proc. BPMDS*, 2011.

[18] S. Sadiq et al., "Data Flow and Validation in Workflow Modelling", in *Proc. Australian Database Conference*, 2004.

[19] K. Schmidt, "LoLA a Low Level Analyser", in *Proc. Appl. and Theory of Petri Nets*, 2000.

[20] H. Störrle, "Semantics and Verification of Data Flow in UML 2.0 Activities", *Electronic Notes Theor. Comput. Sci.*, vol. 4, no. 127, 2005.

[21] S. X. Sun et al., "Formulating the Data-Flow Perspective for Business Process Management", *Information Systems Research*, vol. 17, no. 4, 2006.

[22] N. Trčka, W. v. d. Aalst, and N. Sidorova, "Data-Flow Anti-Patterns: Discovering Data-Flow Errors in Workflows", in *Proc. CAISE*, 2009.

[23] W.-J. van den Heuvel, A. Elgammal, and O. Turetken, "Using Patterns for the Analysis and Resolution of Compliance Violations", *Coop.Inf.Systems*, vol. 21, no. 1, 2012.

[24] W. M. P. van der Aalst, "The Application of Petri Nets to Workflow Management", *Circuits, Systems and Computers*, vol. 8, no. 1, 1998.

[25] S. von Stackelberg, S. Putze, J. Mülle, and K. Böhm, "Detecting Data-Flow Errors in BPMN 2.0", *Technical Report 2014-06*, Karlsruhe Institute of Technology, Department of Informatics, no. 06, 2014.

[26] T. Waheed, M. Iqbal, and Z. Malik, "Data Flow Analysis of UML Action Semantics for Executable Models", in *Proc. Europ. Conf. Model Driven Architecture*, 2008.

[27] I. Weber, J. Hoffmann, and J. Mendling, "Beyond soundness: on the verification of semantic business process models", *Distributed and Parallel Databases*, vol. 27, no. 3, 2010.

A BASIC CONCEPTS FOR PROCESS ANALYSIS

In this appendix, we review fundamental concepts for data-flow analysis in processes, namely Petri Nets with its state space formalism and the temporal logic CTL.

Petri Nets and State Space Formalism.

Petri Nets are a representative of formal graph-based process languages. The definition of a Petri Net used here is the one from [24]. A Petri Net is a directed bipartite graph with two types of nodes called places and transitions.

Definition 4 (Petri Net): *A Petri Net is a triple* (P, T, F) *with P a set of places, T a set of transitions* ($P \cap T = \emptyset$) *and* $F \subseteq (P \times T) \cup (T \times P)$ *a set of arcs (flow relation).*

$p \in P$ is an input place of $t \in T$ if $(p, t) \in F$ and an output place if $(t, p) \in F$. $\bullet t$ denotes the set of input places of t and $t\bullet$ the set of output places. A mapping $M : P \to \mathbb{N}_0$ maps every $p \in P$ to a positive number of tokens, i.e., at any time a place contains zero or more *tokens*. The distribution of tokens over places (M) represents a state of the Petri Net, often referred to as its *marking*. A transition $t \in T$ is activated in a state M if $\forall p \in \bullet t : M(p) \geq 1$. A transition $t \in T$ in M can fire, leading to a new state M' which reduces the value of $M(p)$ by 1 if $p \in \bullet t$, adds 1 to $M(p)$ if $p \in t\bullet$ and does not change otherwise. The set of reachable states from a start state M_0 of a Petri Net builds the *state space*. To check properties of a BPMN process, we need this state space, for which we use the Kripke structure [22] of the Petri Net corresponding to the original BPMN model.

CTL.

Computation **T**ree **L**ogic *CTL* is a temporal logic formalism often used to specify properties for model checking. In our case, those properties are data-flow anti-patterns. E.g., [8] describes CTL and an effective algorithm to verify properties specified in CTL.

The formal syntax of CTL is as follows:

Definition 5 (Computation Tree Logic): *Every atomic proposition* ap *is a CTL formula. If* ϕ_1 *and* ϕ_2 *are CTL formulas then* $\neg\phi_1$, $\phi_1 \vee \phi_2$, $\phi_1 \wedge \phi_2$, $\mathsf{AX}\phi_1$, $\mathsf{EX}\phi_1$, $\mathsf{AG}\phi_1$, $\mathsf{EG}\phi_1$, $\mathsf{AF}\phi_1$, $\mathsf{EF}\phi_1$, $\mathsf{A}[\phi_1\ \mathsf{U}\ \phi_2]$, $\mathsf{E}[\phi_1\ \mathsf{U}\ \phi_2]$ *are CTL formulas.*

In our context, ap is a state of a Petri Net which represents the status of a data element in the BPMN process.

The logical operators always occur in pairs: A path operator (A or E) together with a state operator (X,G, F or U). A means that the formula needs to hold in *all* succeeding execution paths. E means that at least one execution path *exists* where the formula holds. X means that the formula holds in the next state, G means the formula holds in all succeeding states, F means that the formula holds at least in one succeeding state, $[\phi_1\ \mathrm{U}\ \phi_2]$ means that ϕ_1 holds until ϕ_2 is reached.

AUTHOR BIOGRAPHIES

Dr. Silvia von Stackelberg is currently a postdoctoral researcher at the group Information Systems of the Institute for Program Structures and Data Organization (IPD) at Karlsruhe Institute of Technology (KIT), Germany. Her research and teaching at KIT focuses on business process management, in particular data and privacy, and crowd computing. She received her PhD in Computer Science from Technical University of Darmstadt and her diploma in Information Systems from University of Bamberg, Germany. She is a Margarete von Wrangell fellow.

M.Sc. Susanne Putze received her Diploma (M.Sc.) in Informatics from the Karlsruhe Institute of Technology (KIT), Germany in 2013. In her diploma thesis, she worked on data flow modeling and correctness in BPMN 2.0. Now, she is a research assistant in the information systems group of Klemens Böhm at IPD, KIT. Current research topics are crowd computing and process modeling.

M.Sc. Jutta Mülle received her Diploma (M.Sc.) in Informatics from the University of Karlsruhe, Germany. She is a senior research assistant in the information systems group of Klemens Böhm at IPD, Karlsruhe Institute of Technology (KIT). Current research topics are workflow management, process analysis, data and privacy in workflows. She has wide experience in project work also with industry.

Dr. Klemens Böhm is full professor (chair of databases and information systems) at Karlsruhe Institute of Technology (KIT), Germany, since 2004. Prior to that, he has been professor of applied informatics/data and knowledge engineering at University of Magdeburg, Germany, senior research assistant at ETH Zürich, Switzerland, and research assistant at GMD – Forschungszentrum Informationstechnik GmbH, Darmstadt, Germany. Current research topics at his chair are knowledge discovery and data mining, data privacy and workflow management. Klemens gives much attention to collaborations with other scientific disciplines and with industry.