Fakultät für Ingenieurwissenschaften, Informatik und Psychologie
— Institut im Quellcode anpassen nicht vergessen! —

Transforming Object-Aware Processes into BPMN: Conceptual Design and Implementation

Abschlussarbeit an der Universität Ulm

Vorgelegt von:
Marko Pejic
marko.pejic@uni-ulm.de
1027682

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Marius Breitmayer

2023

Fassung December 7, 2022

Satz: PDF-LaTeX 2_ε

Contents

1 Introduction

1.1 Objective

1.2 Problem Statement

1.3 Structure of Thesis

2 Fundamentals

2.1 Business Process Management

2.2 Process Modeling

2.2.1 Activity-Centric

2.2.2 Data-Centric

2.3 Business Process Model and Notation

2.4 Object-Aware Process Management

3 Requirements

3.1 Functional Requirements

3.2 Non-Functional Requirements

4 Transforming Object-Aware Processes into BPMN

In this chapter, a conceptual transformation model is proposed that fulfills the defined requirements for the transformation of object-aware processes into BPMN (cf. Chap. 3). First, an overview of the necessary transformations and mapping rules for the mentioned conception is given (cf. Sect. 4.1). Next, the concrete mapping (cf. Sect. 4.2) and another component of the conception - Robotic Process Automation (RPA) (cf. Sect. 4.3) - are explained in more detail. Finally, an algorithm for transforming object-aware processes into BPMN can be developed as a result (cf. Sect. 4.4).

4.1 Transformation Compendium

The transformation model is divided into three different types: Object, Permission, and Relation, with each type providing different functionalities to fulfill a different concern in the transformation model. In addition, an external component completes the latter.

Type 1 (Object Transformation)**.** *Informell sagen was passiert.*

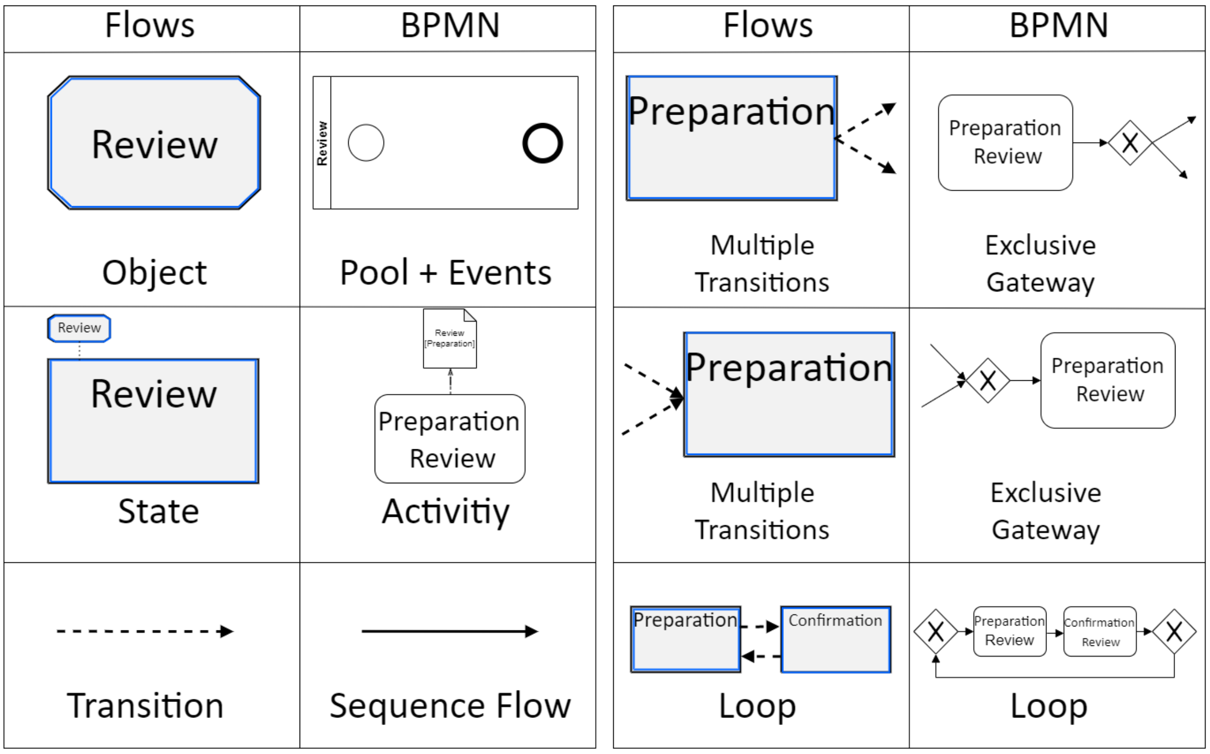

Figure 4.1: Initial example

Type 2 (Permission Integration)**.** *Informell sagen was passiert.*

Type 3 (Relation Integration)**.** *Informell sagen was passiert.*

4.2 Macro Process Model Transformation

The transformation of a macro process model in terms of BPMN requires the state-based view of the object's life cycle process. Based on this, an algorithm can be defined that uses specified transformation rules to generate a process model that meets the requirements of BPMN. Formally described, let $\omega = (\mathrm{n}, \Phi, \Theta)$ be an object. Given the state based view of the object's lifecycle process, i.e., Θ_{macro} = $(\omega, \Sigma, \mathrm{T}_{\Theta}^{ext}, \Psi, \sigma_{start}, \Sigma_{end}) \subset \Theta$, Algorithm TBD determines a pool $\rho = (\mathrm{n}, \delta)$, where δ contains further BPMN process modeling elements; i.e., events, activities, sequence flows, gateways and data objects. Note that in regard to a pool ρ, it is important to mention that the execution of an object's lifecycle process usually refers to one instance of the respective object. In most processes, however, there will be multiple instances of an object, each of which has its own lifecycle process to execute. In BPMN, the presence of multiple instances, or rather multiple participants, can be modeled by labeling the corresponding pool a *multi instance* pool. For example, the multiplicity of $\rho_{App.}$ indicates that multiple applications may exist simultaneously, each with its own independent process according to the lifecycle process of $\omega_{App.}$. The specification of Algorithm

1 is provided in pseudo code and, for illustration purposes, Fig. TBD provides an example of applying the algorithm to transform the macro process model of $\omega_{App.}$ in terms of BPMN.

Let $\Theta_{macro} = (\omega, \Sigma, \mathrm{T}_{\Theta}^{ext}, \Psi, \sigma_{start}, \Sigma_{end})$ be the state based view of an object's lifecycle process. First of all, every component that shall be determined is declared (line 1). Then, the pool ρ is initialized where the name of ρ corresponds to the name of ω (line 2). Thereupon, a start and end event are declared (line 3-4) where the start event is associated with a newly initialized data object (line 5-6) that corresponds to the respective object and shall represent the instantiating of the respective object with the start of its (lifecycle) process. Thereafter, every state $\sigma \in \Sigma$ is transformed into an activity α (line 8-11). If the latter is considered a *sub-process*, its internal BPMN process elements need to be determined by considering the transformation of the micro process model (cf. TBD). However, at this point, only the macro process model is considered, hence, for now, sub-processes are collapsed and their internal specification is considered as a black-box. Next, each transition $\tau_{\Theta} \in \mathrm{T}_{\Theta}^{ext}$ is transformed into a sequence flow (line 12-15). Furthermore, complementary sequence flows not covered by the transformation before, i.e., sequence flows including events, are added to the set of sequence flows (line 16-23). Consequently, activities and events may have multiple incoming or outgoing sequence flows. Such cases are complementary handled by an outsourced algorithm (cf. Algorithm 2) in line 24. Last, each backwards transition $\psi \in \Psi$ is transformed (line 25-27) by creating additional sequence flows and XOR gateways as specified in Algorithm 3. In this context, the current set of sequence flows and XOR gateways may be adapted as well. Finally, an output including ρ with the transformed BPMN elements stored in a 5-tuple δ ends the algorithm (line 28-30). Note that given a state type σ, the function GetBPMNElement(σ) provides the corresponding BPMNElement υ (i.e., an activity α). The remainder of this section describes the concrete transformation rules (TR) used in this algorithm.

Algorithm 1 Macro Process Model Transformation

```
Require: $\Theta_{macro} = (\omega, \Sigma, \mathrm{T}_{\Theta}^{ext}, \Psi, \sigma_{start}, \Sigma_{end})$
Ensure: $\rho$ = (n, $\delta$ = (
            E:{$\epsilon_{start}$: Event, $\epsilon_{end}$: Event}
            A:{$\alpha$: (n: String, $\iota$: ActivityType, $\delta$: 5-tuple, $\nu$: DataObject})
            $\mathrm{T}_{\delta}$:{$\tau_{\delta}$: ($\upsilon_{source}$: BPMNElement, $\upsilon_{target}$: BPMNElement, n: String})
            K:{$\kappa$}
            N:{$\nu$: (n: String, attributeType: AttributeType, access: {write, read})}
        ))
 1: $\rho, \mathrm{E}, \mathrm{A}, \mathrm{T}_{\delta}, \mathrm{K} \leftarrow$ new
 2: $\rho.\mathrm{n} \leftarrow \omega.\mathrm{n}$
 3: $\epsilon_{start}, \epsilon_{end} \leftarrow$ new
 4: $\mathrm{E} \leftarrow \mathrm{E} \cup \{\epsilon_{start}, \epsilon_{end}\}$
 5: $\widetilde{\nu} \leftarrow$ ($\omega$.n, Relation, write)
 6: $\epsilon_{start}.\nu \leftarrow \widetilde{\nu}$
 7: $\mathrm{N} \leftarrow \mathrm{N} \cup \{\nu\}$
 8: for each $\sigma \in \Sigma$ do
 9:     $\alpha \leftarrow$ CreateMacroActivity($\sigma$)
10:     $\mathrm{A} \leftarrow \mathrm{A} \cup \{\alpha\}$
11: end for
```

```
for each $\tau_\Theta \in \mathrm{T}_\Theta^{ext}$ do
    $\tau_\delta \leftarrow$ CreateMacroSequenceFlow($\tau_\omega$)
    $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
end for
$\alpha_{start} \leftarrow$ GetBPMNElement($\sigma_{start}$)
$\tau_\delta \leftarrow (\epsilon_{start}, \alpha_{start}, \perp)$
$\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
for each $\sigma_{end} \in \Sigma_{end}$ do
    $\alpha_{end} \leftarrow$ GetBPMNElement($\sigma_{end}$)
    $\tau_\delta \leftarrow (\alpha_{end}, \epsilon_{end}, \perp)$
    $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
end for
ResolveMultipleSequenceFlows($\mathrm{T}_\delta$, K)
for each $\psi \in \Psi$ do
    CreateLoop($\psi$, $\mathrm{T}_\delta$, K)
end for
$\delta \leftarrow (\mathrm{E}, \mathrm{A}, \mathrm{T}_\delta, \mathrm{K})$
$\rho.\delta \leftarrow \delta$
return $\rho$
```

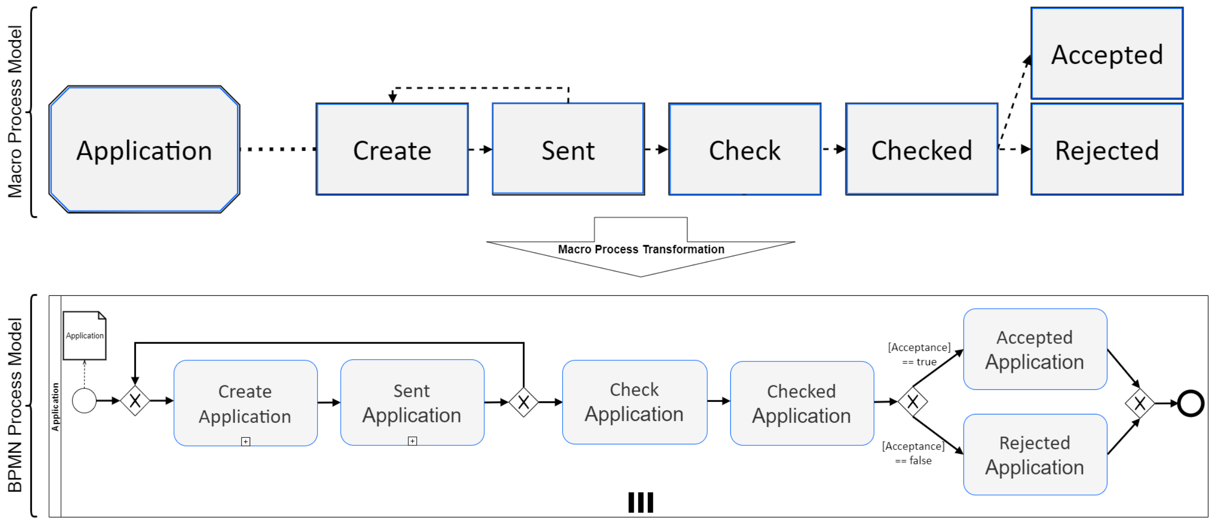

Figure 4.2: Example of applying Algorithm TBD to the transformation of an object's macro process model

4.2.1 State Type Transformation

Generally, data must be read or written for a state type to be executed within an object's lifecycle process by executing its micro step types; although neither may be necessary (e.g., if the state contains only an empty micro step type). For this purpose, forms are generated by state types, where the form fields must be filled with data (cf. TBD). Such a procedure is represented by form-based activities. In order to represent such activities in terms of BPMN, the latter provides *tasks* and *sub-processes*. Hence, a transformation rule to transform a state type into a semantically correct activity in terms of BPMN, i.e., task or sub-process, must be specified (cf. TR. 1). Note that, in the case of the latter, the

BPMN elements of the internal process are only declared, whereas its concrete initialization requires the transformation of the micro process model (cf. Sect. 4.3). At this point, however, it is sufficient to consider such internal processes as a black-box. Example 4.1 & 4.2 close this section with an application of TR. 1.

Transformation Rule 1 (State Type Transformation):
Let σ = $(\mathrm{n}, \Gamma_\sigma, \mathrm{T}_\sigma, \Psi_\sigma, \gamma_{start}, \Gamma_{end})$ be a state type. Then:

CreateMacroActivity: $\Sigma \mapsto \mathrm{A}$ determines a 4-tuple with:
CreateMacroActivity(σ) := $(\mathrm{n}, \iota, \delta, \nu)$ where

- n = $\sigma.\mathrm{n}$ + $\sigma.\omega.\mathrm{n}$; i.e., state type name followed by the object name the corresponding state type belongs to.
- ι = $\begin{cases} \text{Task} & \text{if } |\Gamma_\sigma| = 1 \land (\Gamma_\sigma.\gamma.\phi = \perp \lor |\Gamma_\sigma.\gamma.\mathrm{P}|^1 > 1), \\ \text{Sub-process} & \text{else.} \end{cases}$
- δ = $\begin{cases} (\mathrm{E}, \mathrm{A}, \mathrm{T}, \mathrm{K}, \mathrm{N})^2 & \text{if } \iota = \text{Sub-process}, \\ \perp & \text{else.} \end{cases}$
- ν = $\perp$; i.e., not initialized by default, but if σ contains a value-specific micro step type, ν is subsequently initialized in the course of the micro process model transformation (cf. Sect. 4.3).

$^1\mathrm{P} = \perp \iff |\mathrm{P}| = 0$
^{2}Determined via Algorithm 2

Example 4.1 (State Type Transformation):
Consider Fig. TBD. The *application* macro process model includes the state type σ_{Check}.
Then: σ_{Check} has only one empty micro step type; i.e., $|\Gamma_{Check}| = 1 \land \Gamma_{Check}.\gamma.\phi = \perp$.
Hence, CreateMacroActivity(σ_{Check}) = (Check Application, $task$, $\perp$, $\perp$).

Example 4.2 (State Type Transformation):
Consider Fig. TBD. The *application* macro process model includes the state σ_{Create}.
Then: σ_{Create} has more than one micro step type; i.e., $|\Gamma_{Create}| > 1$. Hence, CreateMacroActivity(σ_{Create}) = (Create Application, $sub-process$, δ, $\perp$).

4.2.2 Macro Transition Type Transformation

Generally, transitions of an object's lifecycle process correspond to sequence flows in BPMN. Nevertheless, the semantically correct transformation of a transition in terms of BPMN depends on the property, if the transition is external, as well as its source and target micro step type. In regard to a macro process model, all the transitions are external, hence, a specification for the semantically correct transformation of the latter must be provided. For this purpose, TR. 2 specifies the transformation of external transitions into a sequence flow. Note that TR. 2 uses the function GetBPMNElement(σ) that was already specified when describing Algorithm 1. In addition, given a micro step type γ, the function MicroStepType(γ) determines the kind of γ (cf. Def. 3). Example 4.3 & 4.4 provide an example of this transformation rule.

Transformation Rule 2 (Macro Transition Type Transformation):
Let τ_Θ = (γ_{source}, γ_{target}, true) be an external transition. Then:

CreateMacroSequenceFlow: $\mathrm{T}_\Theta^{ext} \mapsto \mathrm{T}_\delta$ determines a 3-tuple with:
CreateMacroSequenceFlow(τ_Θ) := (υ_{source}, υ_{source}, n) where

- υ_{source} = GetBPMNElement($\gamma_{source}.\sigma$)
- υ_{target} = GetBPMNElement($\gamma_{target}.\sigma$)
- $\mathrm{n} = \begin{cases} \gamma_{source}.\rho.\lambda & \text{if MicroStepType}(\gamma_{source})\text{=Predicate,} \\ \perp & \text{else.} \end{cases}$

Example 4.3 (Macro Transition Type Transformation):
Consider Fig. TBD. The *application* macro process model includes the transition τ_Θ = (γ_{CV}, $\gamma_{Sent\ Date}$, true). Then: τ_δ = CreateMacroSequenceFlow(τ_Θ) = ($\alpha_{Create\ App.}$, $\alpha_{Sent\ App.}$, $\perp$), since GetBPMNElement($\gamma_{CV}.\sigma_{Create}$) = $\alpha_{Create\ App.}$ and GetBPMNElement($\gamma_{Sent\ Date}.\sigma_{Sent}$) = $\alpha_{Sent\ App.}$.

Example 4.4 (Macro Transition Type Transformation):
Consider Fig. TBD. The *application* macro process model includes the transition τ_Θ = (γ_{PS1}, $\gamma_\perp$, true). Then: τ_δ = CreateMacroSequenceFlow(τ_Θ) = ($\alpha_{Checked\ App.}$, $\alpha_{Accepted\ App.}$, [Accepted] == true), since MicroStepType(γ_{PS1}) = Predicate, GetBPMNElement($\gamma_{PS1}.\sigma_{Checked}$) = $\alpha_{Checked\ App.}$ and GetBPMNElement($\gamma_\perp.\sigma_{Accepted}$) = $\alpha_{Accepted\ App.}$.

As a consequence of TR. 2, activities and events may have multiple incoming or outgoing sequence flows. To resolve this, for each such case, XOR gateways are subsequently created to group multiple incoming and outgoing sequence flows of the corresponding activity or event. For this purpose, the set of such sequence flows must first be specified (cf. Def. 1 & 2).

Definition 1 (Multiple Incoming Sequence Flows Set):
Let δ = $(\mathrm{E}, \mathrm{A}, \mathrm{T}_\delta, \mathrm{K}, \mathrm{N})$ be a 5-tuple of BPMN process elements. Further, let Υ = $\mathrm{A} \cup \{\mathrm{E}.\epsilon_{end}\}$. Then:

$\mathrm{T}_\delta^{\mathbf{mltplTrgt}}$:= $\bigcup_{\upsilon \in \Upsilon} \mathrm{T}_\upsilon^{mltplTrgt} \subset \mathrm{T}_\delta$ with:
$\mathrm{T}_\upsilon^{mltplTrgt}$:= $\{\tau_\delta | \tau_\delta \in \mathrm{T}_\delta \wedge \tau_\delta.\upsilon_{target} = \upsilon \wedge \exists i, j \in \mathbb{N} : i \neq j \wedge \tau_\delta^i.\upsilon_{target} = \tau_\delta^j.\upsilon_{target}\}$

describes a set, where each element represents another set of at least two sequence flows, all having the same target υ.

Definition 2 (Multiple Outgoing Sequence Flows Set):
Let δ = $(\mathrm{E}, \mathrm{A}, \mathrm{T}_\delta, \mathrm{K}, \mathrm{N})$ be a 5-tuple of BPMN process elements. Further, let Υ = $\mathrm{A} \cup \{\mathrm{E}.\epsilon_{start}\}$. Then:

$\mathrm{T}_\delta^{\mathbf{mltplSrc}}$:= $\bigcup_{\upsilon \in \Upsilon} \mathrm{T}_\upsilon^{mltplSrc} \subset \mathrm{T}_\delta$ with:
$\mathrm{T}_\upsilon^{mltplSrc}$:= $\{\tau_\delta | \tau_\delta \in \mathrm{T}_\delta \wedge \tau_\delta.\upsilon_{source} = \upsilon \wedge \exists i, j \in \mathbb{N} : i \neq j \wedge \tau_\delta^i.\upsilon_{source} = \tau_\delta^j.\upsilon_{source}\}$

describes a set, where each element represents another set of at least two sequence flows, all having the same source υ.

Based on Def. 1 & 2, a specification is provided to create XOR gateways as mentioned above, as well as the adaptation of the associated sequence flows. In this context, it should be noted that, a transformation rule in the form of a function is not sufficient to provide such a specification. Therefore, an algorithm is provided in pseudo code (cf. Algorithm 2). More precisely, the algorithm implicitly updates a given set of exclusive gateways K and T_δ from a given 5-tuple of BPMN process elements δ and therefore does not provide direct output. Also note that for a sequence flow τ_δ, the function SetTarget(υ) sets the target from τ_δ to υ. The function SetSource function is defined analogously. Finally, Example 4.5 & 4.6 illustrate resolving such sequence flows by applying Algorithm 2.

Algorithm 2 Resolve Multiple Sequence Flows

```
Require: $\mathrm{T}_\delta, \mathrm{K}$
1: for each $\mathrm{T}_\upsilon^{mltplTrgt} \in \mathrm{T}_\delta^{mltplTrgt}$ do
2:     $\kappa_{join} \leftarrow$ new
3:     $\tau_\delta \leftarrow (\kappa_{join}, \upsilon, \perp)$
4:     for each $\tau_\delta \in \mathrm{T}_\upsilon^{mltplTrgt}$ do
5:         $\tau_\delta$.SetTarget($\kappa_{join}$)
6:     end for
```

```
 7:     $\mathrm{K} \leftarrow \mathrm{K} \cup \kappa_{join}$
 8:     $\mathrm{T} \leftarrow \mathrm{T} \cup \tau_\delta$
 9: end for
10: for each $\mathrm{T}_\upsilon^{mltplSrc} \in \mathrm{T}_\delta^{mltplSrc}$ do
11:     $\kappa_{split} \leftarrow$ new
12:     $\tau_\delta \leftarrow (\upsilon, \kappa_{split}, \perp)$
13:     for each $\tau_\delta \in \mathrm{T}_\upsilon^{mltplSrc}$ do
14:         $\tau_\delta$.SetSource($\kappa_{split}$)
15:     end for
16:     $\mathrm{K} \leftarrow \mathrm{K} \cup \kappa_{split}$
17:     $\mathrm{T} \leftarrow \mathrm{T} \cup \tau_\delta$
18: end for
```

Example 4.5 (Resolving Multiple Incoming Sequence Flows):
Consider the set T_δ when executing Algorithm 2 onto Θ_{macro} from $\omega_{App.}$ until line 24. Then: τ_δ^1 = ($\alpha_{Accepted\ App.}$, ϵ_{end}, $\perp$), τ_δ^2 = ($\alpha_{Rejected\ App.}$, ϵ_{end}, $\perp$)$\in \mathrm{T}_\delta$. Thereupon: Algorithm TBD creates an exclusive gateway κ and adapts τ_δ^1 and τ_δ^2 as follows: τ_δ^1 = ($\alpha_{Accepted\ App.}$, κ, $\perp$) and τ_δ^2 = ($\alpha_{Rejected\ App.}$, κ, $\perp$). In addition, a new sequence flow τ_δ^3 = (κ, ϵ_{end}, $\perp$) is created.

Example 4.6 (Resolving Multiple Outgoing Sequence Flows):
Consider the set T_δ when executing Algorithm 2 onto Θ_{macro} from $\omega_{App.}$ until line 24. Then: τ_δ^1 = ($\alpha_{Checked\ App.}$, $\alpha_{Accepted\ App.}$, $\perp$), τ_δ^2 = ($\alpha_{Checked\ App.}$, $\alpha_{Rejected\ App.}$, $\perp$)$\in \mathrm{T}_\delta$. Thereupon: Algorithm TBD creates an exclusive gateway κ and adapts τ_δ^1 and τ_δ^2 as follows: τ_δ^1 = (κ, $\alpha_{Accepted\ App.}$, $\perp$) and τ_δ^2 = (κ, $\alpha_{Rejected\ App.}$, $\perp$). In addition, a new sequence flow τ_δ^3 = ($\alpha_{Checked\ App.}$, κ, $\perp$) is created.

4.2.3 Backwards Transition Transformation

Backwards transitions allow to reset the execution of a micro process by jumping back to a previous state. In turn, jumping back to a previous activity in an BPMN process model and resetting its execution cannot be represented by a single process modeling element. For example, jumping back to a previous activity can be achieved by a loop, but the execution of the activity is not necessarily reset. This, however, can be represented by the data objects associated to the corresponding activities within an internal process, i.e., within a sub-process. The reason being, that the repeated execution of a sub-process may lead to writing a new data object. Since an object can only be in one state at a time (cf. TBD), it appears as the execution of an activity has been reset. Altogether, a transformation rule in form of a function is not sufficient to completely integrate a transformed backwards transition into a BPMN process model. Thus, a specification is provided in form of an algorithm in pseudo code (cf. Algorithm 3). Note that besides a backwards transitions ψ as input, Algorithm 3 only implicitly updates a given set of exclusive gateways K and T_δ from a given

5-tuple of BPMN process elements δ, since both is needed when transforming ψ. Consequently, no direct output is provided by the algorithm. Further, the function Predecessor(α) determines the BPMNElement from which α is directly reachable, i.e., a BPMNElement υ for which it holds, that a sequence flow τ_δ = (υ, α, n) exists. In this regard, the function PredecessorSqncFlw(α) determines this corresponding sequence flow. The functions Successor and SuccessorSqncFlw are defined analogously. In addition, the functions SetTarget and SetSource as specified in Sect. 4.2.2 are used in the algorithm. 4.7 closes this section by applying Algorithm 3.

Algorithm 3 Create Loop

```
Require: $\psi$ = ($\sigma_{source}$, $\sigma_{target}$), $\mathrm{T}_\delta$, $\mathrm{K}$
1: $\alpha_{source} \leftarrow$ GetBPMNElement($\sigma_{source}$)
2: $\alpha_{target} \leftarrow$ GetBPMNElement($\sigma_{target}$)
3: $\upsilon_{pre} \leftarrow$ Predecessor($\alpha_{source}$)
4: $\upsilon_{suc} \leftarrow$ Successor($\alpha_{target}$)
5: $\tau_\delta^{pre} \leftarrow$ PredecessorSqncFlw($\alpha_{source}$)
6: $\tau_\delta^{suc} \leftarrow$ SuccessorSqncFlw($\alpha_{target}$)
7: if $\upsilon_{pre} \in \mathrm{K}$ then
8:     $\kappa_{source} \leftarrow \upsilon_{pre}$
9: else
10:    $\kappa_{source} \leftarrow$ new
11:    $\widetilde{\tau}_\delta \leftarrow (\kappa_{source}, \alpha_{source}, \perp)$
12:    $\tau_\delta^{pre}$.SetTarget($\kappa_{source}$)
13:    $\mathrm{K} \leftarrow \mathrm{K} \cup \kappa_{source}$
14:    $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \widetilde{\tau}_\delta$
15: end if
16: if $\upsilon_{suc} \in \mathrm{K}$ then
17:    $\kappa_{target} \leftarrow \upsilon_{suc}$
18: else
19:    $\kappa_{target} \leftarrow$ new
20:    $\widetilde{\tau}_\delta \leftarrow (\alpha_{target}, \kappa_{target}, \perp)$
21:    $\tau_\delta^{suc}$.SetSource($\kappa_{target}$)
22:    $\mathrm{K} \leftarrow \mathrm{K} \cup \kappa_{target}$
23:    $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \widetilde{\tau}_\delta$
24: end if
25: $\tau_\delta \leftarrow (\kappa_{target}, \kappa_{source}, \perp)$
26: $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \tau_\delta$
```

Example 4.7 (Backwards Transition Transformation): Consider Fig. TBD. The *application* macro process model includes the backwards transition ψ = (σ_{Create}, σ_{Sent}). Then: Algorithm 3 creates two XOR gateways κ_{source} and κ_{target}. Furthermore, the algorithm adapts the target element of τ_δ^{pre} and source element of τ_δ^{suc} to: τ_δ^{pre} = (κ_{source}, $\alpha_{Create\ App.}$, $\perp$) and τ_δ^{suc} = ($\alpha_{Sent\ App.}$, κ_{source}, $\perp$). In addition, τ_δ = (κ_{target}, κ_{source}, $\perp$) is created.

4.3 Micro Process Model Transformation

In contrast to the transformation of the macro process model, the transformation of the micro process model does not create a new BPMN process model. Rather, the latter extends an already created BPMN process model derived from a macro process model to make its specification more precise. More specifically, each sub-process such a BPMN process model is specified via the micro process model transformation, i.e., sub-processes are no longer considered as a black box, but as a white box. For this purpose, an algorithm must be specified that. Formally described, let $\omega = (\mathrm{n}, \Phi, \Theta)$ be an object. Given the micro process model of the object's lifecycle process, i.e., Θ_{micro} = $(\Sigma, \Gamma, \mathrm{T}^{!ext}_{\Theta}) \subset \Theta$, Algorithm 4 determines for each non-trivial state $\sigma \in \Sigma.\Theta_{micro}$ its corresponding BPMN process modeling elements; i.e., events, activities, sequence flows, gateways and data objects. The specification of Algorithm 4 is provided in pseudo code and, for illustration purposes, Fig. TBD2 provides an example of applying the algorithm to transform the micro process model of $\omega_{App.}$ in terms of BPMN.

Let Θ_{micro} = $(\Sigma, \Gamma, \mathrm{T}^{!ext}_{\Theta}) \subset \Theta$ be the micro process model of an object's lifecycle process. First of all, a start and end event are created (line 1). Next, every non-trivial state $\sigma \in \Sigma$ needs to be determined (line 2). For each such determined state σ, its corresponding sub-process created within the macro process model transformation is determined (line 3) and its internal, and already declared, components are fetched (line 4-8), in order to assign the created process modeling elements from this algorithm to the right sub-process. Furthermore, for each σ, each step $\gamma \in \Gamma_\sigma$ is transformed in terms of BPMN depending on its kind (line 10-29). After that, each transition $\tau_\sigma \in \mathrm{T}_\sigma$ is transformed into a sequence flow (line 30-35). Thereupon, complementary sequence flows not covered by the transformation before, i.e., sequence flows including events, are added to the set of sequence flows as well (line 36-43). Consequently, as in the macro process model transformation, BPMN elements may have multiple incoming or outgoing sequence flows. Such cases are complementary handled by the same auxiliary algorithm (cf. Algorithm 2) as in the macro process model transformation in line 44. Last, an output including the transformed elements stored in a 5-tuple δ is provided (line 46-47). Note that given a state type γ, the function GetBPMNElement(γ) provides the corresponding BPMNElement υ (i.e., an activity α or exclusive gateway κ). Further, given a micro step type γ, the function MicroStepType(γ) determines the kind of γ (cf. Def. 3). The remainder of this section describes the concrete transformation rules (TR) used in this algorithm.

Algorithm 4 Micro Process Model Transformation

```
Require: $\Theta_{micro}$ = $(\Sigma, \Gamma, \mathrm{T}^{!ext}_{\Theta}) \subset \Theta$
Ensure: $\delta$ = (
        $\mathrm{E}$:{$\epsilon_{start}$: Event, $\epsilon_{end}$: Event}
        $\mathrm{A}$:{$\alpha$: ($\mathrm{n}$: String, $\iota$: ActivityType, $\delta$: 5-tuple, $\nu$: DataObject})
        $\mathrm{T}_\delta$:{$\tau_\delta$: ($\upsilon_{source}$: BPMNElement, $\upsilon_{target}$: BPMNElement, $\mathrm{n}$: String})
```

```
            K:{$\kappa$}
            N:{$\nu$: (n: String, attributeType: AttributeType, access: {write, read})}
        )
 1: $\epsilon_{start}, \epsilon_{end} \leftarrow$ new
 2: for each $\sigma \in \{\sigma | \sigma \in \Sigma : !\ (|\Gamma_\sigma| = 1 \wedge (\Gamma_\sigma.\gamma.\phi = \perp \vee \Gamma_\sigma.\gamma.\mathrm{P} \neq \perp))\}$ do
 3:     $\widetilde{\alpha} \leftarrow$ GetBPMNElement($\sigma$)
 4:     $\mathrm{E} \leftarrow \widetilde{\alpha}.\delta.\mathrm{E}$
 5:     $\mathrm{A} \leftarrow \widetilde{\alpha}.\delta.\mathrm{A}$
 6:     $\mathrm{T}_\delta \leftarrow \widetilde{\alpha}.\delta.\mathrm{T}_\delta$
 7:     $\mathrm{K} \leftarrow \widetilde{\alpha}.\delta.\mathrm{K}$
 8:     $\mathrm{N} \leftarrow \widetilde{\alpha}.\delta.\mathrm{N}$
 9:     $\mathrm{E} \leftarrow \mathrm{E} \cup \{\epsilon_{start}, \epsilon_{end}\}$
10:     for each $\gamma \in \Gamma_\sigma$ do
11:         $stepType \leftarrow$ MicroStepType($\gamma$)
12:         if $stepType$ = value-specific micro step then
13:             if $|\gamma.\mathrm{P}| = 1$ then
14:                 $\alpha \leftarrow$ CreateMicroActivity($\gamma$)
15:                 $\mathrm{A} \leftarrow \mathrm{A} \cup \{\alpha\}$
16:                 $\mathrm{N} \leftarrow \mathrm{N} \cup \{\alpha.\nu\}$
17:             else if $|\gamma.\mathrm{P}| > 1 \wedge \gamma \notin \Gamma_{end}$ then
18:                 $\kappa \leftarrow$ new
19:                 $\mathrm{K} \leftarrow \mathrm{K} \cup \{\kappa\}$
20:                 $\nu \leftarrow$ ($\gamma.\phi.\mathrm{n}$, $\gamma.\phi$.attributeType, Write)
21:                 $\alpha \leftarrow$ GetBPMNElement($\sigma$)
22:                 $\alpha.\mathrm{N} \leftarrow \alpha.\mathrm{N} \cup \{\nu\}$
23:             end if
24:         else
25:             $\alpha \leftarrow$ CreateMicroActivity($\gamma$)
26:             $\mathrm{A} \leftarrow \mathrm{A} \cup \{\alpha\}$
27:             $\mathrm{N} \leftarrow \mathrm{N} \cup \{\alpha.\nu\}$
28:         end if
29:     end for
30:     for each $\tau_\Theta \in \mathrm{T}_\Sigma$ do
31:         $\tau_\delta \leftarrow$ CreateMicroSequenceFlow($\tau_\Theta$)
32:         if $\tau_\delta \neq \perp$ then
33:             $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
34:         end if
35:     end for
36:     $\upsilon_{start} \leftarrow$ GetBPMNElement($\gamma_{start}$)
37:     $\tau_\delta \leftarrow (\epsilon_{start}, \upsilon_{start}, \perp)$
38:     $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
39:     for each $\gamma_{end} \in \Gamma_{end}$ do
40:         $\upsilon_{end} \leftarrow$ GetBPMNElement($\gamma_{end}$)
41:         $\tau_\delta \leftarrow (\upsilon_{end}, \epsilon_{end}, \perp)$
42:         $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
43:     end for
44:     ResolveMultipleSequenceFlows($\mathrm{T}_\delta$, K)
45: end for
46: $\delta \leftarrow (\mathrm{E}, \mathrm{A}, \mathrm{T}_\delta, \mathrm{K}, \mathrm{N})$
47: return $\delta$
```

Finally, it should be mentioned that in regard to the transformation model, a sub-process within a BPMN process model can also be declared as *ad-hoc*. The reason for this is that the order in which form fields are filled in, i.e. the order of executing micro step types and providing data, can be arbitrary. Hence, if a sub-process is declared as ad-hoc, its internal activities can also be executed in arbitrary order, thus enabling the arbitrary provision of data. In regard to the BPMN process model, only the internal activities and their corresponding data objects within the corresponding sub-process are visible (cf. Fig. TBD).

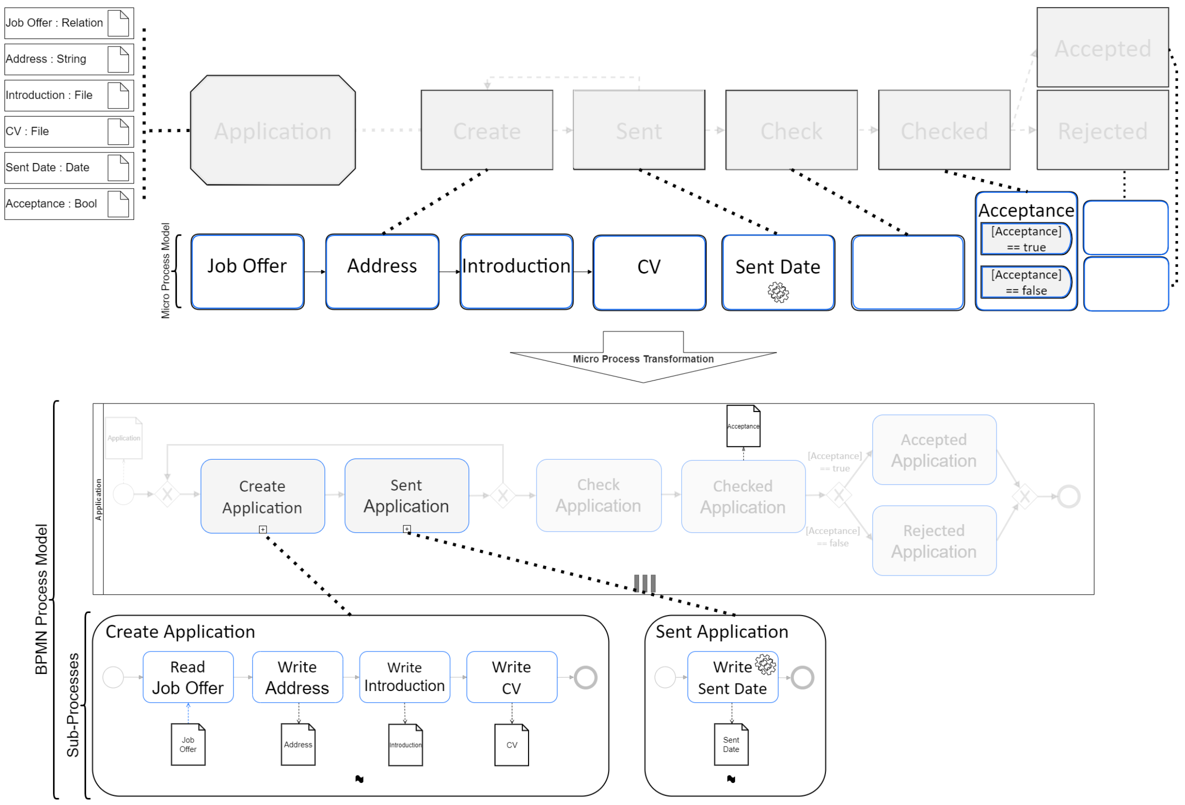

Figure 4.3: Example of applying Algorithm 4 to the transformation of an object's micro process model

4.3.1 Micro Step Type Transformation

Filling in form fields generated by a state type corresponds in the micro process to the execution of a micro step type. In other words, the execution of micro step type makes the process data-driven in the first place, as their execution ultimately drives the process flow. With regard to the transformation of a micro step type in terms of BPMN, this data-driven approach should also be apparent in the process model. For this purpose, data objects are created and associated with the corresponding activity derived from the micro step type. A Transformation rule (TR) for micro step types, as well as examples for these, are provided in the following of this chapter. Note that given a micro step type γ, the function MicroStepType(γ) determines the kind of γ. In the following, for the sake of brevity, micro step types will be referred to as steps.

Definition 3 (Determine Micro Step Type):
Let γ = (ϕ, σ, T_{in}, T_{out}, P, λ) be a micro step type. Then:

MicroStepType: $\Gamma \mapsto$ MicroStepTypes determines the kind of γ with:

$$\textbf{MicroStepType}(\gamma) := \begin{cases} \text{Empty} & \text{if } \gamma.\phi = \perp, \\ \text{Predicate} & \text{if GetValueSpecificStep}(\gamma) \neq \perp, \\ \text{Value-Specific} & \text{if } \gamma.\mathrm{P} \neq \perp, \\ \text{Computation} & \text{if } \gamma.\lambda \neq \perp, \\ \text{Atomic} & \text{else.} \end{cases}$$

A. Empty Micro Step Type

If γ is considered an *empty step* , it simultaneously holds, that γ is the only step of its corresponding state type $\gamma.\sigma$ (cf. TBD). Hence, the corresponding activity α that results from CreateMacroActivity($\gamma.\sigma$) is considered a *task*. In this context, γ does not require any specific transformation rule and is practically skipped when in the context of the step transformation. For instance, Example TBD illustrates the transformation of a state type inluding an empty step.

B. Predicate Micro Step Type

In the context of the micro step type transformation, a special case occurs when γ is considered a *predicate step*, since no direct and complete transformation of γ into the BPMN process model takes place. Rather, only the expressions provided by predicate steps are of interest and implicitly considered when generating sequence flows (cf. TBD). In addition, predicate steps are of interest, because their existence indicates a *value-specific step*. As mentioned, predicate steps are implicitly considered when generating sequence flows, hence, no explicit transformation rule is provided in this chapter.

C. Value-Specific Micro Step Type

As mentioned above, transforming a *value-specific step* $\widetilde{\gamma}$ requires the consideration of its predicate steps. On the one hand, if $\widetilde{\gamma}$ contains at least two predicate steps, it is transformed into an *exclusive gateway* in terms of BPMN (cf. Example TBD) to control the convergence and divergence of sequence flows in the BPMN process that arises from a value-specific micro step type. On the other hand, if $\widetilde{\gamma}$ only contains one predicate step, $\widetilde{\gamma}$ is transformed into a task (cf. Example TBD), according to TR. TBD. Algorithm TBD distincts these cases directly. Since only an exclusive gateway needs to be generated for the first case and an already defined transformation rule is used for the second case, the transformation of a value-specific step does not require the specification of a new transformation rule. However, XOR gateways in BPMN cannot be associated with data objects. As a consequence, the attribute associated with the respective value-specific step cannot be represented by a data object in the BPMN process mode. To prevent this, it should also be possible that activities derived from state types (cf. TBD) may also write data objects only in this special case. More simply, if an activity α derived from a state type contains a value-specific step γ, it shall write a data object corresponding to the attribute ϕ of γ. E.g., For the value-specific step $\gamma_{Acceptance} \in \Gamma_{Checked}$ it holds: $\alpha_{Checked\ App.}.\nu$ = (Acceptance, Bool, write). Such creation of a data object is considered subsequently in the micro process model transformation (cf. Algorithm TBD line TBD).

Example TBD (TBD):
Consider Fig. TBD. The *application* micro process type includes the state $\sigma_{Checked}$ where $\sigma_{Acceptance}$ includes the *value-specific step* $\gamma_{Acceptance}$, since $\gamma_{Acc.}.\mathrm{P} \neq \perp$. Furthermore, it holds $\gamma_{Acc.}.\mathrm{P} = \{\rho_1, \rho_2\}$ where ρ_1 = (γ, Acceptance == [true]) and ρ_2 = (γ, Acceptance == [false]). Then: An exclusive gateway κ is generated.

Example TBD (TBD):
Consider Fig. 1 in TBDCITEUIB201606. The *review* micro process type includes the state $\sigma_{Reject\ Proposed}$, where $\sigma_{Reject\ Proposed}$ includes the *value-specific step* $\gamma_{Finished}$, since $|\gamma_{Finished}.\mathrm{P}| > 1$. But, $\gamma_{Finished.}.\mathrm{P}$={($\gamma$, Finished == [true])}. Hence: CreateMicroActivity($\gamma_{Finished}$) = (Write Finished, $task$, $\perp$, $\nu_{Finished}$).

D. Computation Micro Step Type & Atomic Micro Step Type

If none of the above cases apply, γ is considered either a *atomic step* or a *computation step*. Since both cases are identical in their specification (except for their activity type), the same transformation rule TR TBD can be applied to both cases as follows. Example TBD illustrates the application of TR TBD onto an atomic step, whereas Example TBD illustrates its application onto a computation step.

Transformation Rule TBD (Micro Step Type Transformation):
Let γ = (ϕ, σ, T_{in}, T_{out}, P, λ) be a micro step type with $\phi \neq \perp \wedge \mathrm{P} = \perp$; i.e., γ neither represents an empty micro step nor a predicate step. Then:

CreateMicroActivity: $\Gamma \mapsto \mathrm{A}$ determines a 4-tuple with:
CreateMicroActivity(γ**)** = (n, ι, δ, ν) where

- $\mathrm{n} = \begin{cases} \text{Read } + \ \phi.\mathrm{n} & \text{if } \phi.\mathrm{attributeType} = \text{Relation}, \\ \text{Write } + \ \phi.\mathrm{n} & \text{else.} \end{cases}$
- $\iota = \begin{cases} \text{Service Task} & \lambda = \text{true}, \\ \text{Task} & \text{else.} \end{cases}$
- $\delta = \perp$; i.e., the respective activity does not contain an internal process to describe its activity.
- ν = ($\phi.\mathrm{n}, \phi.\mathrm{attributeType}$, access) with:
  access = $\begin{cases} \mathrm{read} & \text{if } \phi.\mathrm{attributeType} = \text{Relation}, \\ \mathrm{write} & \text{else.} \end{cases}$

Example TBD (Atomic Micro Step Type Transformation):
Consider Fig. TBD. The *application* micro process type includes the state σ_{Create}, which in turn, includes the atomic micro step type $\gamma_{Job\ Offer}$ = (Job Offer, σ_{Create}, T_{in}, T_{out}, $\perp$, false). Furthermore, it holds $\phi_{Job\ Offer}$ = Relation. Then: α = CreateMicroActivity($\gamma_{Job\ Offer}$) = (Read Job Offer, $task$, $\perp$, $\nu_{Job\ Offer}$) where $\nu_{Job\ Offer}$ = (Job Offer, Relation, read).

Example TBD (Computation Micro Step Type Transformation):
Consider Fig. TBD. The *application* micro process type includes the state σ_{Sent}, where σ_{Sent} includes the *computation micro step type* $\gamma_{Sent\ Date}$ = (Sent Date, σ_{Sent}, T_{in}, T_{out}, $\perp$, true), since λ = true. Furthermore, it holds $\phi_{Sent\ Date}$ = false. Then: CreateMicroActivity($\gamma_{Sent\ Date}$) = (Write Sent Date, $service\ task$, $\perp$, $\nu_{Sent\ Date}$) where $\nu_{Sent\ date}$ = (Sent Date, Date, write).

Note that naming a (service) task that is generated within the micro step type transformation, i.e., transforming an (computation) atomic step γ into a (service) task α (cf. Example TBD and TBD), differs from naming a task, that is generated directly from a state type. Thus, for the former, $\alpha.\mathrm{n}$ is determined by starting with either *Read* or *Write*, followed by the attribute instance ϕ demanded by γ. If latter relates to an object type within the process, i.e., $\phi.attributeType = Relation$, $\alpha.\mathrm{n}$ starts with *Read*, because an object instance of the attribute relating to γ must already exist at run-time. For example, when executing $\gamma_{Job\ Offer} \in \Gamma_{Created}$, at least one instance of $\omega_{Job\ Offer}$ must already exist, since $\phi_{Job\ Offer}.attributeType = Relation$. Hence, $\omega_{Job\ Offer}$ needs to be *read* within the process when executing $\gamma_{Job\ Offer}$ at run-time. Analogously, the value of access from the corresponding data object $\nu_{Job\ Offer}$ is set to read; i.e.; a reading association between $\alpha_{Read\ Job\ Offer}$ and $\nu_{Job\ Offer}$ is drawn. In turn, $\alpha.\mathrm{n}$ starts with *Write*, if an object instance of the attribute relating to γ must not necessarily exist at run-time, i.e., $\phi.attributeType \neq Relation$. As an example, when executing $\gamma_{Address} \in \Gamma_{Created}$, the attribute $\phi_{Address}$ can be provided at run-time, since it does not relate to an object type ($\phi_{Address}.attributeType = String \neq Relation$). Hence, $\phi_{Address}$ is *written* at run-time when executing $\gamma_{Address}$. Analogously, the value of access from the corresponding data object $\nu_{Address}$ is set to write; i.e.; a writing association between $\nu_{Address}$ and $\alpha_{Write\ Address}$ is drawn. In addition, the distinction between writing and reading an attribute in the BPMN process model implicitly shows whether the execution of an activity depends on the execution of another activity, potientially from another pool.

4.3.2 Micro Transition Type Transformation

As in the macro transitions type transformation (cf. TBD), any transition within an object's lifecycle process can generally be transformed into a sequence flow. However, in regard to a micro process, all transitions are non-external, hence, this chapter provides a specification to transform these kind of transitions in sequence flows. For this purpose, TR TBD specifies the transformation of a non-external transition into a sequence flow. Note that TR TBD uses the function GetBPMNElement(γ) that was already specified when describing Algorithm TBD. Further, the function MicroStepType(γ) is used as specified in Def. TBD.

Transformation Rule TBD (Micro Transition Type Transformation):
Let τ_Θ = (γ_{source}, γ_{target}, false) be a non-external transition. Then:

CreateMicroSequenceFlow: $\mathrm{T}_\Theta^{!ext} \mapsto \mathrm{T}_\delta$ with:

if StepType(γ_{target}) = Value-Specific $\wedge$ γ_{target} = $\gamma_{target} \in \gamma_{target}.\Sigma.\Gamma_{end}$ **then**

CreateMicroSequenceFlow(τ_Θ) := $\perp$

else

CreateMicroSequenceFlow(τ_Θ) := (υ_{source}, υ_{source}, n) with:

- υ_{source} = GetBPMNElement(γ_{source})
- υ_{target} = GetBPMNElement(γ_{target})
- n = $\begin{cases} \gamma_{source}.\rho.\lambda & \text{if MicroStepType}(\gamma_{source}) = Predicate, \\ \perp & \text{else.} \end{cases}$

Similarly to the macro transitions type transformation, as a consequence of applying TR. TBD, activities and events may have multiple incoming or outgoing sequence flows. To resolve this, Def. TBD, Def. TBD and Algorithm TBD as specified in chapter TBD can be used by the micro transitions type transformation aswell. Hence, no further transformation rules are provided.

4.4 Permission Integration

4.5 Coordination Process Transformation

4.6 Robotic Process Automation

4.7 Transformation Algorithm

Algorithm 5 Step One (S1)

```
Require: $\Omega$
1: $\mathrm{P} \leftarrow$ new
2: for $\omega \in \Omega$ do
3:     $\rho \leftarrow OTA(\omega)$
4:     $\mathrm{P}.add(\rho)$
5: end for
6: return $\mathrm{P}$
```

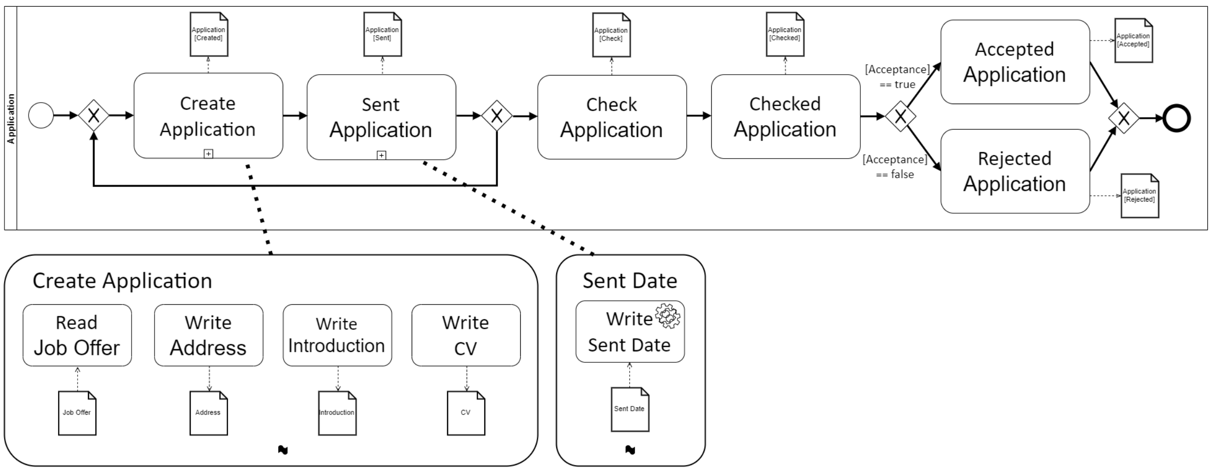

Figure 4.4: Initial example

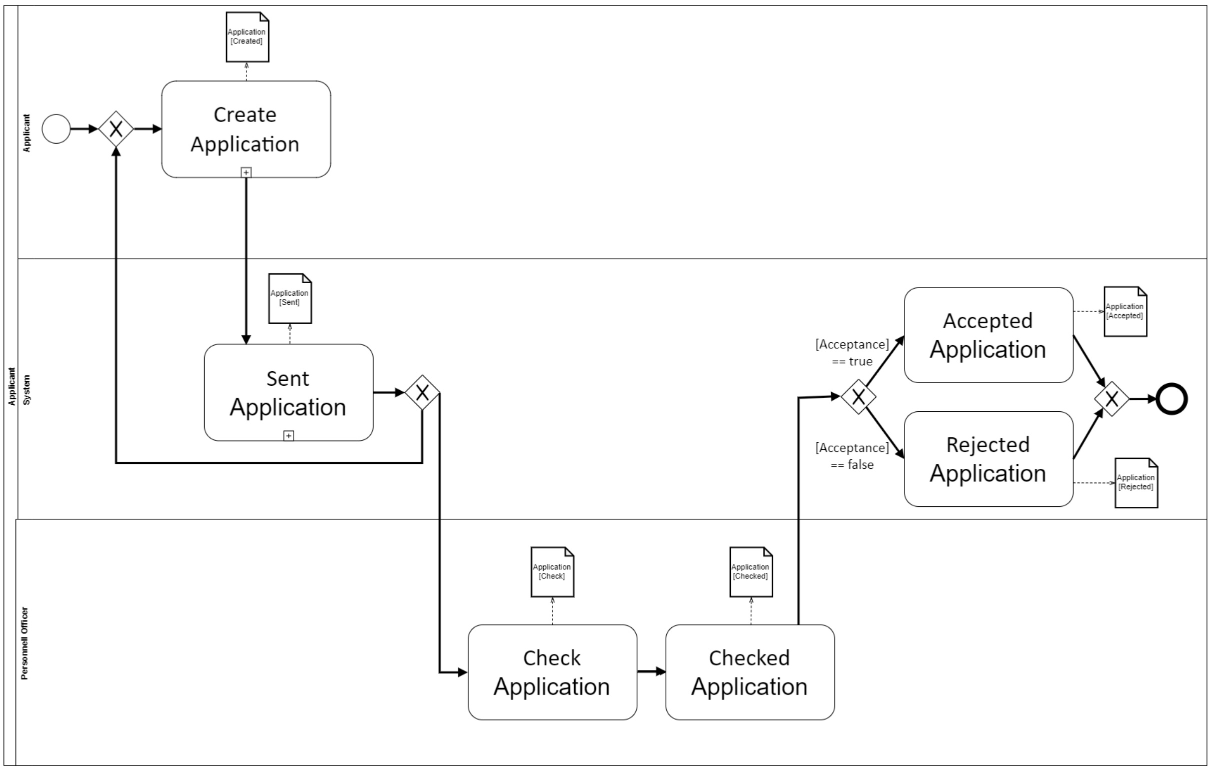

Figure 4.5: Initial example

5 Implementation

- Explain Algorithm in words and provide Pseudocode

6 Evaluation

6.1 Functional Requirements

6.2 Non-Functional Requirements

6.3 Limitations

7 Related Work

8 Conclusion

8.1 Contribution

8.2 Outlook

A Attachments

In diesem Anhang sind einige wichtige Quelltexte aufgeführt.

```java
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Bibliography

[1] Jörg Knappen. *Schnell ans Ziel mit LATEX 2e*. 3., überarb. Aufl. München: Oldenbourg, 2009.

[2] Frank Mittelbach, Michel Goossens, and Johannes Braams. *Der Latex-Begleiter*. 2., überarb. und erw. Aufl. ST - Scientific tools. München [u.a.]: Pearson Studium, 2005.

[3] Joachim Schlosser. *Wissenschaftliche Arbeiten schreiben mit LATEX : Leitfaden für Einsteiger*. 5., überarb. Aufl. Frechen: mitp, 2014.

[4] Thomas F. Sturm. *LATEX : Einführung in das Textsatzsystem*. 9., unveränd. Aufl. RRZN-Handbuch. Hannover [u.a.]: Regionales Rechenzentrum für Niedersachsen, RRZN, 2012.

[5] Herbert Voß. *LaTeX Referenz*. 2., überarb. u. erw. Aufl. Berlin: Lehmanns Media, 2010.

Name: Marko Pejic Matrikelnummer: 1027682

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den ....

Marko Pejic