universität
uulm

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 5. Dezember 2022

Satz: PDF-LaTeX 2$_\varepsilon$

Inhaltsverzeichnis

1 Introduction **1**
- 1.1 Objective . . . 1
- 1.2 Problem Statement . . . 1
- 1.3 Structure of Thesis . . . 1

2 Fundamentals **2**
- 2.1 Business Process Management . . . 2
- 2.2 Process Modeling . . . 2
  - 2.2.1 Activity-Centric . . . 2
  - 2.2.2 Data-Centric . . . 2
- 2.3 Business Process Model and Notation . . . 2
- 2.4 Object-Aware Process Management . . . 2

3 Requirements **3**
- 3.1 Functional Requirements . . . 3
- 3.2 Non-Functional Requirements . . . 3

4 Transforming Object-Aware Processes into BPMN **4**
- 4.1 Transformation Compendium . . . 5
- 4.2 Object Transformation . . . 5
  - 4.2.1 State Type Transformation . . . 8
  - 4.2.2 Micro Step Type Transformation . . . 10
  - 4.2.3 Transition Transformation . . . 12
  - 4.2.4 Backwards Transition Transformation . . . 15
- 4.3 User Integration . . . 16
- 4.4 Relation Integration . . . 16
- 4.5 Robotic Process Automation . . . 16
- 4.6 Transformation Algorithm . . . 16

5 Implementation **18**

6 Evaluation **19**
- 6.1 Functional Requirements . . . 19
- 6.2 Non-Functional Requirements . . . 19

6.3 Limitations 19

7 Related Work **20**

8 Conclusion **21**

8.1 Contribution 21

8.2 Outlook 21

A Attachments **22**

Literatur **23**

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

Transformation Rule 1 (Step Type)**.** Let γ = (ϕ, σ, T_{in}, T_{out}, P, λ) be a step. Then:

StepType: $\Gamma \mapsto$ ActivityStepTypes with:

$$\textbf{StepType}(\gamma) := \begin{cases} \text{Empty} & \text{if } \gamma.\phi = \bot, \\ \text{Computational} & \text{if } \gamma.\lambda \neq \bot, \\ \text{MacroPredicate} & \text{if } \gamma.\mathrm{P} \neq \bot, \\ \text{MicroPredicate} & \text{if IsPredicate}(\gamma) = \text{true}, \\ \text{Task Step} & \text{else.} \end{cases}$$

Transformation Rule 2 (Initialize Data Object)**.** Let α be an activity. Then:

DataObjectInit: $\Sigma \cup \Gamma \mapsto \mathrm{N}$ with:

- **DataObjectInit(σ)** := (n) where
  - $\mathrm{n} = \sigma.\omega.\mathrm{n} + \sigma.\mathrm{n}$
- **DataObjectInit(γ)** := (n) where
  - $\mathrm{n} = \gamma.\phi$

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.*

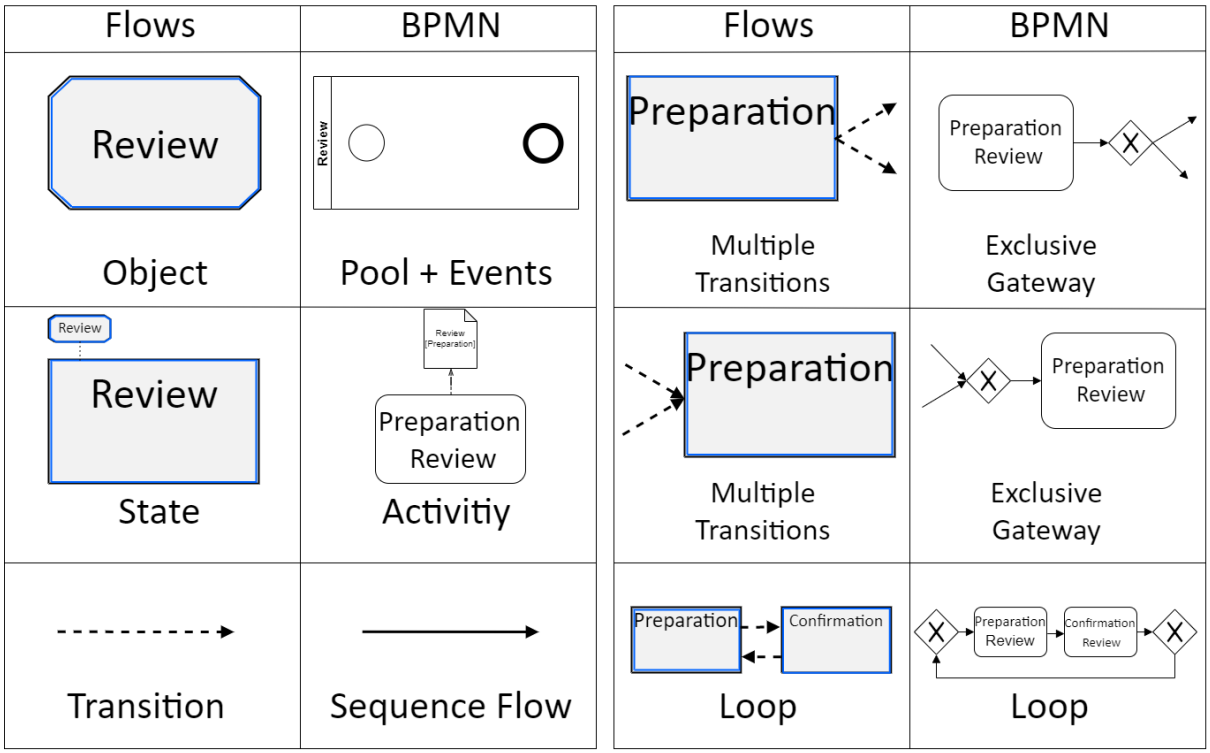

Abbildung 4.1: Initial example

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

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

4.2 Object Transformation

For the first step of the transformation model, any given object $\Omega \ni \omega = (\mathrm{n}, \Phi, \Theta)$ is transformed in terms of BPMN. More specifically, an object $\omega = (\mathrm{n}, \Phi, \Theta)$ is transformed into a pool $\rho = (\mathrm{n}, \delta)$. 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 cases, 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 calling the corresponding pool a *multi instance* pool by drawing three vertical lines in the bottom center of the 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 $\omega_{App.}$. Algorithm TBD specifies transformation from ω to ρ in pseudo code. To illustrate the algorithm, Fig. TBD provides an example of applying the algorithm to transform the object $\omega_{App.}$.

Let $\omega = (\mathrm{n}, \Phi, \Theta)$ be an object. First of all, a pool ρ is generated where the name of ρ corresponds to the name of ω (line 2). Thereupon, the pool is filled with the BPMN equivalent of each component within the respective object's lifecycle process $\Theta = (\Sigma, \Gamma, \mathrm{T}, \Psi, \sigma_{start}, \Sigma_{end})$; i.e., events, activities, sequence flows, gateways and data objects, which are stored in a 5-tuple δ. In this regard, first, a start and end event within the BPMN process are generated (line TBD), where the start event is associated with a data object (access : writing) that corresponds to the respective object (line 3-4). Thereupon, every state $\sigma \in \Sigma$ is transformed into an actvitiy α (line 7-9). If the latter is considered a *sub-process*, its internal BPMN process elements are determined with its determination (cf. Algorithm TBD). Besides that, a special case occurs if α is considered a *task* and its corresponding state contains only one *value-specific micro step*. In this case, an additional exclusive gateway needs to be generated (line 10-12). Next, each transition $\tau_\omega \in \mathrm{T}_\omega$ is transformed into a sequence flow and, eventually, exclusive gateways are generated (line 15-25). Last, each backwards transition $\psi \in \Psi$ is transformed (line 35-37) by generating additional sequence flows and exclusive gateways. In this context, the current set of sequence flows and exclusive gateways may be adapted aswell. To complete the algorithm, an output including ρ with the transformed elements stored in a 5-tuple δ is provided (line 38-40). Note that given a state type σ or micro step type γ, the function GetBPMNElement($\{\gamma, \sigma\}$) provides the corresponding BPMNElement υ (i.e., α or κ) that, in general, already has been determined when executing the function. Further, given a predicate step γ the function GetValueSpecificStep(γ) determines the value-specific micro step type to which γ belongs.

The remainder of this section describes the concrete transformation rules used in Algorithm TBD, where the functions GetBPMNElement($\{\gamma, \sigma\}$) and are needed repeatedly.

Algorithm 1 Object Transformation Algorithm (OTA)

Require: ω = $(\mathrm{n}, \Phi, \Theta)$
Ensure: ρ = $(\mathrm{n}, \delta$ = (
E:{ϵ_{start}: Event, ϵ_{end}: Event}
A:{α: (n: string, ι: ActivityType, δ: BPMNProcess, ν_{write}: DataObject, N_{read}: DataObjectSet})
T_δ:{τ_δ: (υ_{source}: BPMNElement, υ_{target}: BPMNElement, n: string})
K:{κ}
N:{ν: (n: string, attributeType: AttributeType, access: {write, read})}
))

1: $\rho, \mathrm{E}, \mathrm{A}, \mathrm{T}_\delta, \mathrm{K}, \mathrm{N} \leftarrow$ **new**
2: $\rho.\mathrm{n} \leftarrow \omega.\mathrm{n}$
3: $\mathrm{E} \leftarrow \mathrm{E} \cup \{\epsilon_{start}, \epsilon_{end}\}$
4: $\widetilde{\nu} \leftarrow$ (ω.n, Relation, write)
5: $\mathrm{N} \leftarrow \mathrm{N} \cup \{\widetilde{\nu}\}$

$\epsilon_{start}.\nu \leftarrow \widetilde{\nu}$
for each $\sigma \in \Theta.\Sigma$ **do**
  $\alpha \leftarrow$ CreateMacroActivity(σ)
  $\mathrm{A} \leftarrow \mathrm{A} \cup \{\alpha\}$
  if $\exists \gamma_{end} \in \sigma.\Gamma_{end}$: StepType($\sigma.\gamma_{end}$) = *value-specific micro step* **then**
    $\kappa \leftarrow$ **new**
    $\mathrm{K} \leftarrow \mathrm{K} \cup \{\kappa\}$
  end if
end for
for each $\tau_\omega \in \{\tau_\omega : \tau_\omega \in \Theta.\mathrm{T}_\omega \wedge \tau_\omega.ext = true\}$ **do**
  if $stepType(\tau_\omega.\gamma_{source}) = predicatestep$ **then**
    $\widetilde{\gamma} \leftarrow$ GetValueSpecificStep($\tau_\omega.\gamma_{source}$)
    $\widetilde{\nu} \leftarrow (\widetilde{\gamma}.\phi.\mathrm{n}, \widetilde{\gamma}.\phi.attributeType, write)$
    $\mathrm{N} \leftarrow \mathrm{N} \cup \{\widetilde{\nu}\}$
    $\alpha \leftarrow$ GetBPMNElement($\tau_\omega.\gamma_{target}.\sigma$)
    $\alpha.\nu \leftarrow \widetilde{\nu}$
  end if
  $\mathrm{T}_\delta \leftarrow$ CreateMacroSequenceFlows(τ_ω)
  $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \mathrm{T}_\delta$
end for
$\alpha_{start} \leftarrow$ GetBPMNElement($\Theta.\sigma_{start}$)
$\tau_\delta \leftarrow (\mathrm{E}.\epsilon_{start}, \alpha_{start}, \bot)$
$\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
for each $\sigma_{end} \in \Theta.\Sigma_{end}$ **do**
  $\alpha_{end} \leftarrow$ GetBPMNElement(σ_{end})
  $\tau_\delta \leftarrow (\alpha_{end}, \mathrm{E}.\epsilon_{end}, \bot)$
  $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
end for
ComplementSequenceFlows(T_δ, K)
for each $\psi \in \Psi$ **do**
  CreateLoop(ψ, T_δ, K)
end for
$\delta \leftarrow (\mathrm{E}, \mathrm{A}, \mathrm{T}_\delta, \mathrm{K}, \mathrm{N})$
$\rho.\delta \leftarrow \delta$
return ρ

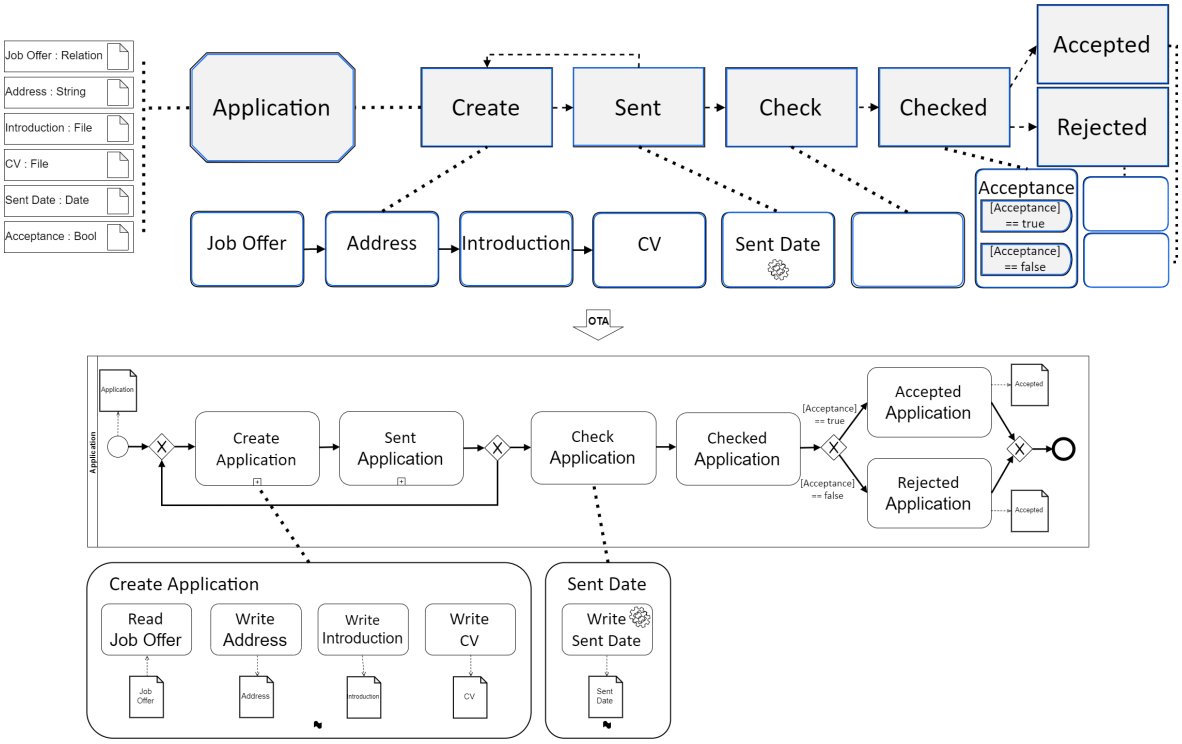

Abbildung 4.2: Initial example

4.2.1 State Type Transformation

In general, 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). In any case, however, the execution of a state type represents an activity from a user and ultimately drives the process flow. In order to represent such activities, BPMN provides *tasks* and *sub-processes*. Hence, a transformation rules to transform a state type into a semantically correct activity in terms of BPMN, i.e., task or sub-process, is specified (cf. TR TBD). If the latter occurs, an algorithm as specified in Algorithm TBD is used to determine its internal process. Example TBD and Exmaple TBD illustrate this tranformation rule, where Example TBD also illustrates Algorithm TBD.

Transformation Rule 3 (State Type Transformation)**.** Let σ = (n, ω, Γ_σ, T_σ, Ψ_σ, γ_{start}, Γ_{end}) be a state. Then:

CreateMacroActivity: $\Sigma \mapsto \mathrm{A}$ with:
CreateMacroActivity(σ) := (n, ι, δ, ν) where

- n = σ.n + σ.ω.n; i.e., state name followed by object name of the corresponding state.

- ι = $\begin{cases} \text{Task} & \text{if } |\Gamma_\sigma| = 1 \wedge (\Gamma_\sigma.\gamma.\phi = \bot \vee \Gamma_\sigma.\gamma.\mathrm{P} \neq \bot), \\ \text{Sub-process} & \text{else.} \end{cases}$
- δ = $\begin{cases} \text{CreateSubProcess}(\sigma) = (\mathrm{E}, \mathrm{A}, \mathrm{T}, \mathrm{K}, \mathrm{N}) & \text{if } \iota = \text{Sub-process}, \\ \bot & \text{else.} \end{cases}$
- ν = $\bot$; TBD explanation.

Algorithm 2 Create Sub-process

Require: $\sigma = (\mathrm{n}, \omega, \Gamma_\sigma, \mathrm{T}_\sigma, \Psi_\sigma, \gamma_{start}, \Gamma_{end})$
Ensure: δ = (
E:{ϵ_{start}: Event, ϵ_{end}: Event}
A:{α: (n: string, ι: ActivityType, δ: BPMNProcess, ν_{write}: DataObject, N_{read}: DataObjectSet})
T_δ:{τ_δ: (υ_{source}: BPMNElement, υ_{target}: BPMNElement, n: string})
K:{κ}
N:{ν: (n: string, attributeType: AttributeType, access: {write, read})}
)

1: $\mathrm{E}, \mathrm{A}, \mathrm{T}_\delta, \mathrm{K}, \mathrm{N} \leftarrow$ **new**
2: $\mathrm{E} \leftarrow \mathrm{E} \cup \{\epsilon_{start}, \epsilon_{end}\}$
3: **for each** $\gamma \in \Gamma_\sigma$ **do**
4: $stepType \leftarrow$ StepType(γ)
5: **if** $stepType$ = *value-specific micro step* $\wedge\ \gamma \notin \Gamma_{end}$ **then**
6: $\kappa \leftarrow$ **new**
7: $\mathrm{K} \leftarrow \mathrm{K} \cup \{\kappa\}$
8: **else if** $stepType$ = *computation* $\vee$ $stepType$ = *atomic micro step* **then**
9: $\alpha \leftarrow$ CreateMicroActivity(γ)
10: $\mathrm{A} \leftarrow \mathrm{A} \cup \{\alpha\}$
11: $\mathrm{N} \leftarrow \mathrm{N} \cup \{\alpha.\nu\}$
12: **end if**
13: **end for**
14: **for each** $\tau_\omega \in \mathrm{T}_\sigma$ **do**
15: $\tau_\delta \leftarrow$ CreateMicroSequenceFlow(τ_σ)
16: **if** $\tau_\delta \neq \bot$ **then**
17: $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
18: **end if**
19: **end for**
20: $\alpha_{start} \leftarrow$ GetBPMNElement(γ_{start})
21: $\tau_\delta \leftarrow (\epsilon_{start}, \alpha_{start}, \bot)$
22: $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
23: **for each** $\gamma_{end} \in \Gamma_{end}$ **do**
24: $\alpha_{end} \leftarrow$ GetBPMNElement(γ_{end})
25: $\tau_\delta \leftarrow (\alpha_{end}, \epsilon_{end}, \bot)$
26: $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \{\tau_\delta\}$
27: **end for**
28: ComplementSequenceFlows(T_δ, K)
29: $\delta \leftarrow (\mathrm{E}, \mathrm{A}, \mathrm{T}_\delta, \mathrm{K}, \mathrm{N})$
30: **return** δ

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

$\perp$. Hence, StateTransformation(σ_{Check}) = (Check Application, $task$, $\perp$, Application [Check], $\perp$).

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

4.2.2 Micro Step Type Transformation

As the transformation of state types suggest, micro step types need to be considered in the transformation model aswell. More specifically, depending on its kind (cf. Def. TBD), each micro step type within a state type is transformed into a semantically correct BPMN equivalent as specified in one of the transformation rules following this chapter. *A. Empty Micro Step Type*

If γ is considered an empty micro step type, it simultaneously holds, that γ is the only micro step type of its corresponding state type $\gamma.\sigma$. Hence, the corresponding activity α that results from StateTransformation($\gamma.\sigma$) is considered a *task*. In this context, γ does not require any specific transformation rule and is practically skipped when considering transforming micro step types. For instance, Example TBD can be applied onto the transformation of an empty micro step type aswell.

B. Predicate Step Type

In the context of the transformation model, 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, for the semantically correct transformation of a predicate step, the *value-specific micro step type* $\widetilde{\gamma}$ to which γ belongs is considered. In fact, predicate steps are only implicitly considered when a transformation to $\widetilde{\gamma}$ is performed.

C. Value-Specific Micro Step Type

As mentioned above, transforming a *value-specific micro step type* $\widetilde{\gamma}$ requires the consideration of its predicate step types. In general, $\widetilde{\gamma}$ is transformed into an *exclusive gateway* in terms of BPMN to control the process flow. More specifically, each predicate step within $\widetilde{\gamma}$ represents a different way the process flows. The transformation of value-specific micro step types is directly performed in Algorithm TBD.

Example TBD (Value-Specific Micro Step Type Transformation):
Consider Fig. TBD. The *application* micro process type includes the state $\sigma_{Checked}$,

where $\sigma_{Checked}$ includes the value-specific micro step type γ = (Acceptance, $\sigma_{Checked}$, T_{in}, T_{out}, P, false), since $\mathrm{P} = \{\rho_1, \rho_2\} \neq \bot$. Further, ρ_1 = (γ_1, $[Acceptance] == true$) and ρ_2 = (γ_2, $[Acceptance] == false$). Then: An exclusive gateway κ_{split} is generated.

D. Atomic Micro Step Type and Computation Micro Step Type

If none of the above cases apply, γ is considered either a *atomic micro step type*, that simply requires an attibute to be executed, or a *computation micro step type*. Both cases are smiliar in their specification, hence, for both cases TR TBD applies. Example TBD and Example TBD illustrate this transformation rule in both cases.

Transformation Rule 4 (Atomic and Computation Micro Step Type Transformation)**.** Let γ = (ϕ, σ, T_{in}, T_{out}, P, λ) be a lifecycle step with $\phi \neq \bot \wedge \mathrm{P} = \bot$; i.e., γ neither represents an empty micro step nor a predicate step. Then:

CreateMicroActivity: $\Gamma \mapsto \mathrm{A}$ with:
CreateMicroActivity(γ) = (n, ι, δ, ν) where

- n = $\begin{cases} \text{Read } + \ \phi.\mathrm{n} & \text{if } \phi.\mathrm{attributeType} = \text{Relation}, \\ \text{Write } + \ \phi.\mathrm{n} & \text{else.} \end{cases}$
- ι = $\begin{cases} \text{service task} & \lambda\text{= true}, \\ \text{task} & \text{else.} \end{cases}$
- $\delta = \bot$; 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 γ = (Job Offer, σ_{Create}, T_{in}, T_{out}, $\bot$, false). Furthermore, it holds IsRelationType(γ) = true. Then: α = TBD($\gamma_{Job\ Offer}$) = (Read Job Offer, $task$, $\bot$, $\bot$, {Job Offer}).

Example TBD (Computation Micro Step Transformation):
Consider Fig. TBD. The *application* micro process type includes the state σ_{Sent}, whe-

re σ_{Sent} includes the *computation micro step type* γ = (Sent Date, σ_{Sent}, T_{in}, T_{out}, $\perp$, true), since $\gamma.\lambda$ = true. Furthermore, it holds IsRelationType(γ) = false. Then: CreateServiceTask(γ) = (Write Sent Date, $service\ task$, $\perp$, Sent Date, $\perp$).

Note that naming a task that is generated within the micro step type transformation, i.e., transforming an (computation) atomic micro step type γ into a task α (cf. Example TBD), differs from naming a task, that is generated directly from a state within the state transformation. 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 refers to an object type within the process, i.e., IsRelationType(γ) = true, $\alpha.\mathrm{n}$ starts with *Read*, for the reason, that an instance of the attribute corresponding to γ, must already exist at run-time, e.g., when executing $\gamma_{Job\ Offer} \in \Gamma_{Created}$, at least one instance of $\omega_{Job\ Offer}$ must already exist. Hence, $\omega_{Job\ Offer}$ needs to be *read* within the process when executing $\gamma_{Job\ Offer}$ at run-time. In turn, $\alpha.\mathrm{n}$ starts with *Write*, if an instance of the attribute corresponding to γ must not necessairly exist at run-time, i.e., IsRelationType(γ) = false. For instance, 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 (IsRelationType($\gamma_{Address}$) = false). Hence, ϕ = Address is *written* at run-time when executing $\gamma_{Address}$. In addition, the distinction between writing and reading a data type in the BPMN process model enables to implicitly show whether the execution of an activity depends on the execution of another activity, potientially within another pool. Similiarly, the data objects associated with an activity derived from an (computation) atomic micro step type only include the name of the corrsponding attribute from the micro step type, for the reason, that the sole execution of one micro step type does not change the state of an object. Hence, regarding data objects and activites derived from micro step types, only writing and reading associations need to be specified.

4.2.3 Transition Transformation

In general, any transition within an object's lifecycle process can be transformed into a sequence flow. Nevertheless, the semantically correct transformation depends on the property, if the transition is external, as well as its source and target element. Especially, whether a transition is external or not, determines whether the corresponding sequence flow belongs to a BPMN process on first or second granularity. For this purpose, two separate algorithms in pseudo code are specified, where Algorithm TBD transforms every non external transition into a corresponding sequence flow and, in turn, Algorithm TBD transforms every external transition into a corresponding sequence flow. For the former, the resulting sequence flows belong to a BPMN process on first granularity, whereas for the latter, the resulting sequence flows belong to a BPMN process on second granularity. Note that Algorithm TBD and Algorithm TBD use the functions GetBPMNElement and GetValueSpe-

cificStep as specified in the description for Algorithm TBD.

Algorithm 3 Create Macro Sequence Flows

Require: τ_ω = $(\gamma_{source}, \gamma_{target}, true)$
Ensure: T_δ
1: $\mathrm{T}_\delta \leftarrow$ **new**
2: $\mathrm{n} \leftarrow \bot$
3: sourceStepType $\leftarrow$ StepType(γ_{source})
4: **if** $sourceStepType(\gamma_{source}) = predicatestep$ **then**
5: $\widetilde{\gamma} \leftarrow$ GetValueSpecificStep(γ_{source})
6: $\upsilon_{source} \leftarrow$ GetBPMNElement($\widetilde{\gamma}_{source}$)
7: $\mathrm{n} \leftarrow \gamma_{source}.\rho.\lambda$
8: **if** $\widetilde{\gamma} \in \widetilde{\gamma}.\sigma.\Gamma_{end}$ **then**
9: $\widetilde{\alpha} \leftarrow$ GetBPMNElement($\widetilde{\gamma}.\sigma$)
10: $\tau_\delta \leftarrow (\widetilde{\alpha}, \upsilon_{source}, \bot)$
11: $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \tau_\delta$
12: **end if**
13: **else**
14: $\upsilon_{source} \leftarrow$ GetBPMNElement($\gamma_{source}.\sigma$)
15: **end if**
16: $\upsilon_{target} \leftarrow$ GetBPMNElement($\gamma_{target}.\sigma$)
17: $\tau_\delta \leftarrow (\upsilon_{source}, \upsilon_{target}, \mathrm{n})$
18: $\mathrm{T}_\delta \leftarrow \mathrm{T}_\delta \cup \tau_\delta$
19: **return** T_δ

Algorithm 4 Create Micro Sequence Flow

Require: τ_ω = $(\gamma_{source}, \gamma_{target}, false)$
Ensure: τ_δ
1: $\tau_\delta \leftarrow$ **new**
2: $\mathrm{n} \leftarrow \bot$
3: sourceStepType $\leftarrow$ StepType(γ_{source})
4: **if** $sourceStepType(\gamma_{source}) = predicatestep$ **then**
5: $\widetilde{\gamma} \leftarrow$ GetValueSpecificStep(γ_{source})
6: $\upsilon_{source} \leftarrow$ GetBPMNElement($\widetilde{\gamma}$)
7: $\mathrm{n} \leftarrow \gamma_{source}.\rho.\lambda$
8: **else**
9: $\upsilon_{source} \leftarrow$ GetBPMNElement(γ_{source})
10: **end if**
11: **if** $sourceStepType(\gamma_{target}) = value - specificmicrostep \wedge \gamma_{target} = \gamma_{target}.\sigma.\gamma_{end}$ **then**
12: **end algorithm**
13: **else**
14: $\upsilon_{target} \leftarrow$ GetBPMNElement(γ_{target})
15: **end if**
16: $\tau_\delta \leftarrow (\upsilon_{source}, \upsilon_{target}, \mathrm{n})$
17: **return** τ_δ

In Fig. TBD (a) and (b), the applications of Algorithm TBD onto the given transitions are straightforward. However, as in (c) and (d), two special cases occur regarding external transitions including predicate steps. First, in (c), a sequence flow between *Activity 1* and *Gateway B* is missing in order to make the transformation sementically correct and complete. Such a case is considered by Algorithm TBD in line TBD-TBD. Second, the case in (d) occurs when the target of a transition is considered a value-specific step γ, but its corresponding exclusive gateway κ is on first granularity (line TBD), i.e., κ does not belong

to the corresponding sub-process derived from the state where γ belongs. In this case, the respective transition is not considered in the transformation and therefore, Algorithm TBD ends without outputting any sequence flow (line TBD).

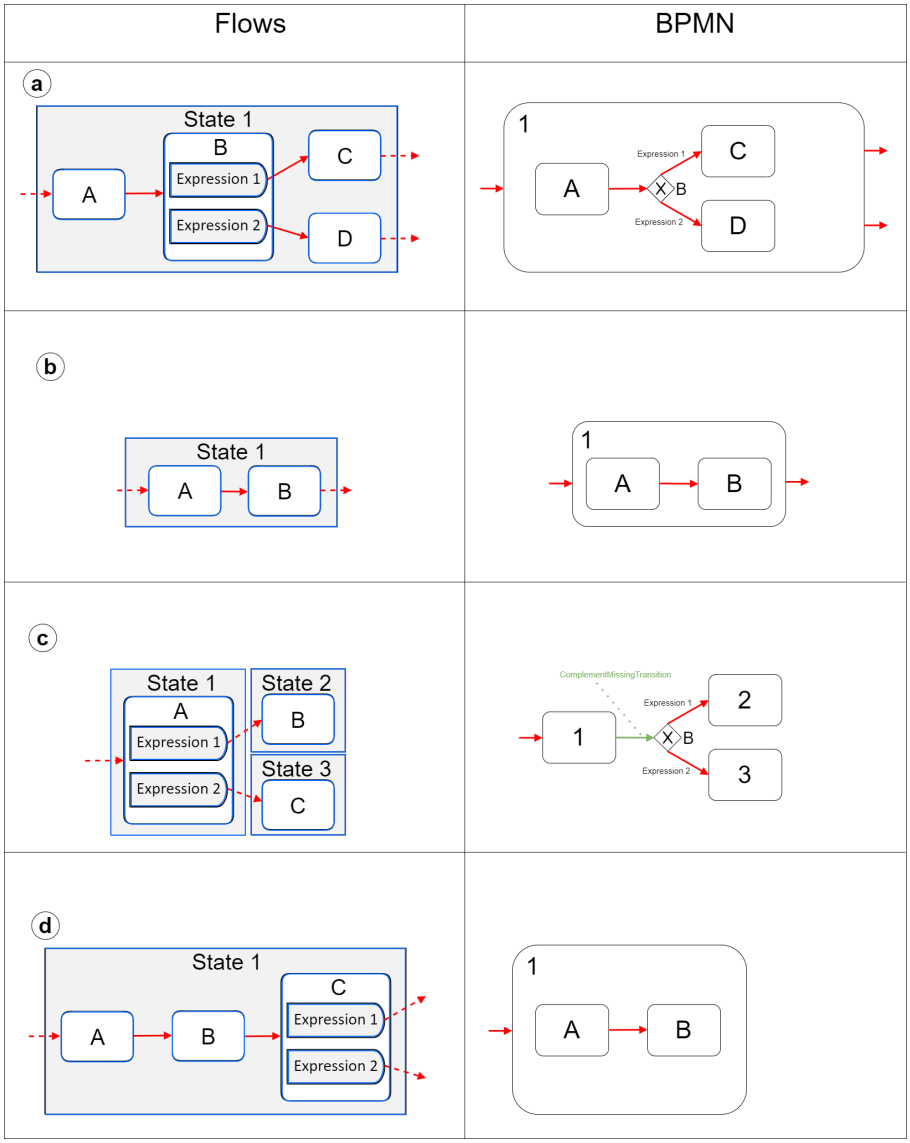

Abbildung 4.3: Initial example

Furthermore, as a result from Algorithm TBD, activities may have multiple incoming sequence flows. Consequently, for each such case, exclusive gateways are subsequently generated to group multiple incoming sequence flows of a BPMNElement. For this purpose, the set of such sequence flows must be specified (cf. Def. TBD).

Definition 4.1 (Multiple Target Sequence Flows).
Let $\delta = (\mathrm{E}, \mathrm{A}, \mathrm{T}_\delta, \mathrm{K}, \mathrm{N})$ be a 5-tuple of BPMN process elements. Further, let $\Upsilon = \mathrm{A} \cup \mathrm{K} \cup \{\mathrm{E}.\epsilon_{start}\}$. Then:
$\mathrm{T}_\delta^{\mathbf{mltplTrgt}} := \bigcup_{\upsilon \in \Upsilon} \mathrm{T}_\upsilon^{mltplTrgt} \subset \mathrm{T}_\delta$ with:

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

Finally, Algorithm TBD completes the semantically correct transformation of transitions by describing the generation of exclusive gateways as mentioned above, as well as the adaptation of the associated sequence flows in pseudo code. More specifically, Algorithm TBD 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 an output. Note that given a BPMNElement υ, the function SetTarget(υ) sets the target of a sequence flow τ_δ to υ.

Algorithm 5 Complement 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** $\tau_\delta \in \mathrm{T}_{mltplTrgt}^{\upsilon}$ **do**
5: τ_δ.SetTarget(κ_{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**

4.2.4 Backwards Transition Transformation

Backwards transitions allow to reset the execution of a micro process by jumping back to a previous state. In turn, in BPMN, jumping back to a previous activity and reseting its execuction cannot be represented by a single process modeling element. For example, jumping back to a previous activity within the BPMN process model can be achieved by a loop, but the execution of activities is not necesserialy reseted. Such a reset, however, can be represented by the data objects associated to the corresponding activities, since the repeated execution of an activity within the loop will write a new data object in the corresponding state. Since by Def. TBD an object can only be in one state at a time, it appears as the execution of an activity has been reseted. Altogether, a backwards transition is transformed in terms of BPMN as specified in Algorithm TBD. Note that besides a backwards transitions ψ as input, Algorithm TBD 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 and therefore does not provide any direct output. Further, given a BPMNElement υ, SetTarget(υ) sets the target of a sequence flow τ_δ to υ. 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 SetSource, Successor and SuccessorSqncFlw are defined analogously.

Algorithm 6 Create Loop

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

4.3 User Integration

4.4 Relation Integration

4.5 Robotic Process Automation

4.6 Transformation Algorithm

Algorithm 7 Step One (S1)

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

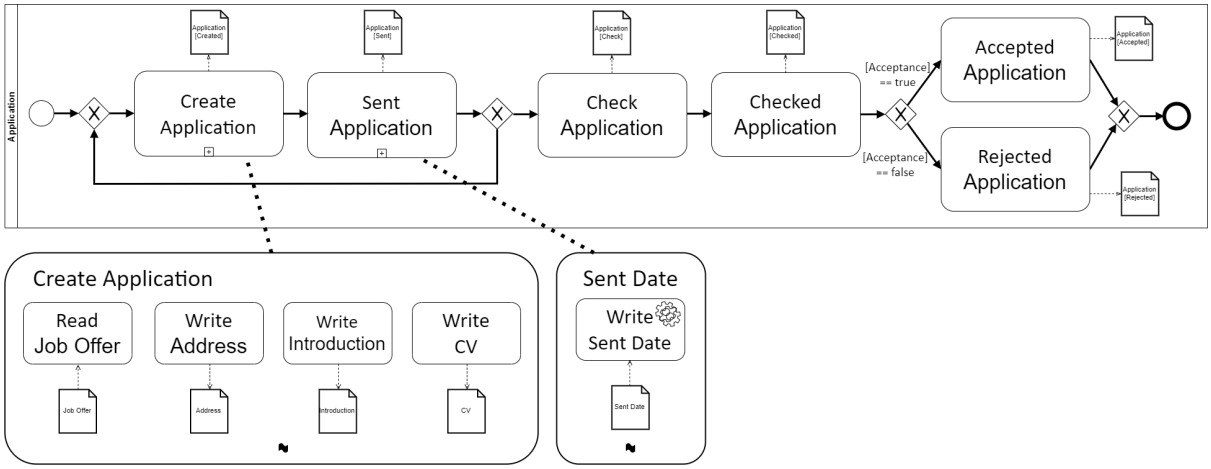

Abbildung 4.4: Initial example

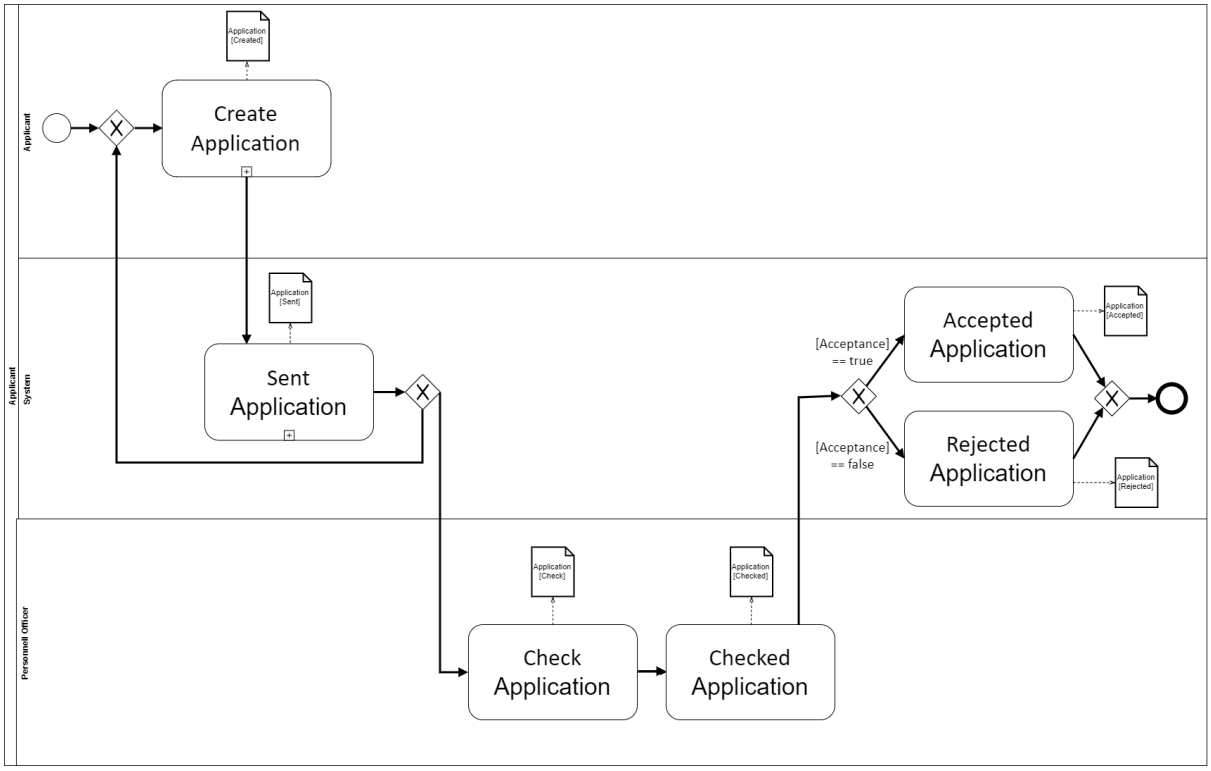

Abbildung 4.5: Initial example

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.

```
1 public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello World");
4     }
5 }
```

Literatur

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

[2] Frank Mittelbach, Michel Goossens und 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