The present work was submitted to
the Research Group
Software Construction

of the Faculty of Mathematics,
Computer Science, and
Natural Sciences

Master Thesis

Contract-based Data Flow Validation of BPMN Workflows for Microservices

presented by

Sabine Lüttgen

Aachen, July 26, 2019

Examiner

Prof. Dr. rer. nat. Horst Lichter

Prof. Dr. rer. nat. Bernhard Rumpe

Supervisor

Christian Plewnia, M.Sc.

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Ort, Datum/City, Date

Unterschrift/Signature

Acknowledgment

Zuerst möchte ich mich bei meinem Prüfer Prof. Dr. rer. nat. Horst Lichter für die Chance bedanken, im Lehr- und Forschungsgebiet Informatik 3 Softwarekonstruktion meine Masterarbeit schreiben zu dürfen und für den Input zu meiner Arbeit. Außerdem möchte ich mich bei meinem Zweitprüfer Prof. Dr. rer. nat. Bernhard Rumpe bedanken.

Für sein konstantes Feedback bedanke ich mich herzlich bei meinem Betreuer Christian Plewnia. Er schlug mir das Themengebiet vor, brachte mich mit dem Kooperationspartner des Lehrstuhls zusammen und half mir, mein Thema zu finden. Vielen Dank für all die Meetings, die vielen hilfreichen und interessanten Gespräche und die Korrekturvorschläge.

Außerdem möchte ich mich bei allen Mitarbeitern der Firma Amadeus bedanken, die ich kennenlernen durfte. Vor allem Danke an alle Mitglieder des Minion Teams für die freundliche Aufnahme in ihr Team, den Einblick in ihre Arbeit und die vielen Informationen, die ich erhalten habe. Vielen Dank für die lustige Zeit.
Ein besonderer Dank geht natürlich an alle Teilnehmer meiner Studie. Vielen Dank für eure Zeit und die hilfreichen Ergebnisse und das Feedback.

Bei meinem kleinen Neffen Ben möchte ich mich entschuldigen, dass er in den letzten Monaten nicht die Aufmerksamkeit von mir erhalten hat, die er verdient hätte. Das werde ich jetzt wieder gut machen.
Zu guter Letzt ein sehr großes Danke an meine Familie und Freunde. Vielen Dank für die Korrekturen und die moralische Unterstützung. Danke für die vielen netten Worte, Kaffeepausen und Süßigkeiten. Danke für die Stunden zusammen in der Bib, eure Besuche und alle Anrufe. Ich weiß es war nicht immer leicht, mit mir ständig über die Masterarbeit zu reden.

Danke!

Sabine Lüttgen

Abstract

The *Business Process Model and Notation (BPMN)* is a well established notation for modelling business processes that can be executed by workflow engines.
For the automatic workflow execution, every service task of a BPMN workflow has to be realized as software. This can be done in the form of microservices. In some business domains it is possible to reuse individual tasks in different business processes. However, the task implementations may depend on input data which has to be considered when modelling the workflows.
This combination leads to possible problems at creation and change of microservices and workflows. In this thesis, it is laid out that the data flow definition of such systems might cause problems, since the workflows are created by domain experts without the full needed knowledge of service developers. Errors are often only recognized at runtime with complex test cases for the workflow as a whole or in production.

To support the correct modelling of BPMN workflows, a static data flow validation approach is presented that is implemented in a prototype called *COntracted BAsed data FLOW validation (CobaFlow)*. Every microservice implementing a BPMN service task has to provide a contract specifying their needed input and output. On this basis CobaFlow analyses the workflow's data flow for missing input data. This proposed solution should support the workflow creation and can be implemented into the *Continuous Integration (CI)* process of the microservices.

A small industry case study with six participants was conducted, to get first insights about the validation prototype. The results indicate that the prototype is perceived as helpful for creating new workflows and modifying existing ones.

Contents

List of Tables

List of Figures

List of Source Codes

1. Introduction

Contents

A software system typically reacts to some sort of message from the outside via an interface. This triggers the start of an internal process which can give feedback of intermediate and end states to one or multiple other interfaces. This internal process can be called a workflow. To model these workflows *Business Process Model and Notation (BPMN)* can be used (see 2.1). A BPMN workflow consists of different tasks to be executed in a specific order depending on conditions. A BPMN workflow can model a complete business logic including tasks that are executed or triggered by software or humans.

BPMN is used to build a bridge between software developers, business developers and customers. By that people with different technical background can define and talk about business processes of a software system based on a graphical representation of these processes. These processes are supposed to be directly executable in a software system. Workflow engines are used for this purpose.

Software can be developed in different ways. A monolithic system could be created, including the complete business logic of a system. However, another common way to orchestrate a software is by using microservices that are composed to build the complete business logic.

Combining microservices and the use of BPMN has special requirements and problems to the software architecture and the workflow engines. The goal of this thesis is to extract data flow problems, based on this create an architecture concept and realize a validation that enables users to create multiple different workflows based on the microservices at hand without deeper knowledge of the technical realization of the microservices or the software architecture.

1.1. Scope

The goal of this thesis is to extract data flow problems that occur when using BPMN in a microservice environment. A concept for supporting users and developers needs to be created and requirements to the system need to be defined. The concept is realized in a prototype for data flow error validation. The realization has to be evaluated against the requirements and for usability.

1.2. Related Work

This thesis is embedded in the field of research of BPMN in combination with microservices and the analysis and validation of workflows. Workflows can be analysed regarding different sources of error. These can be semantic and syntactic errors with regards to the process and data flow. In the following research in the field of workflow analysis is presented.

BPMN workflows are used in projects from the beginning of the project. Initial errors can accumulate to bigger issues over the systems development.
Therefore Dijkman et al. [DDO08] introduced a way of semantic checks of BPMN workflows by transferring them into petri nets and using established tests for them. They transform BPMN workflows into *Petri Net Markup Language (PNML)* files and then use existing analysis tools for semantic checks on the validity of the workflow. These tests include tests for dead tasks or proper completion. [DDO08]
Pourmirza et al. [Pou+19] created a *Novel Reference Architecture for Business Process Management Systems (BPMN-RA)* by researching current architectures using BPM. They captured 438 components from 41 primary studies in the academic literature. They categorized nine of these components into "validating and verifying business processes". They therefore state that a BPMS-RA-compliant system must support this functionality. [Pou+19] As examples they give ADONIS (commercial tool) [Ado] which includes a syntactical check and WoPed (open source) [Fre] which includes syntactical and semantic checks based on workflows implemented as petri nets. Both tools validate workflows syntactically. WoPed can be used for syntax highlighting, marking splits and joins belonging together [Rei+11].

Sadig et al. [Sad+04] presented issues in validating data flow issues in workflows at the state of modelling the workflows. By identifying the essential requirements and validation problems they laid the ground for validation tools. Based on this Trčka, van der Aalst and Sidorova classified possible flaws related to data flow in business workflows. Flaws are formulated in the temporal logic *CTL**.[TAS09]
Research in this area states that creating business workflows start by creating the control flow structure and validating this, but that a data flow could be introduced and checked at the workflow creation. However, this means that every created workflow needs to specify explicitly which data every task needs and offers. In this thesis we look at setup, where

multiple workflows coexist and are created by people without deep technical knowledge. This makes it hard for a user setting up workflows to specify data and data formats to be exchanged. Therefore a setup is needed where the services and tasks provide information about data needed and offered and a validation based on this. Currently no research or tool is found for this.

In 2014 von Stackelberg et al. [Sta+14] presented a way of detecting data-flow errors in *BPMN 2.0 standard*. They proposed a transformation of the process model to Petri Nets based on the previously presented method of Dijkman et al. [DDO08]. But as Dijkman et al. only did this on *BPMN 1.0 standard* and therefore without data representation, which was introduced in BPMN 1.0 standard (see chapter 2.1), von Stackelberg et al. [Sta+14] enhanced this procedure to integrate data objects. They did so by performing 2 steps. First they mapped the control flow to the Petri Nets and second they unfolded data of mapped flow elements to according to the execution semantics. By this they were able to find data flow errors in BPMN 2.0 standard workflows. However they did not consider, that the BPMN 2.0 standard does not allow multiple data in- and output for service tasks. This is just allowed for all other types of tasks. Also they did not mention how a graphical or textual feedback to the user could be given. Their solution also lacks the possibility to validate BPMN workflows that are not yet finished or not sound.

1.3. Structure of this Thesis

The thesis will start with an introduction to the background subjects in chapter 2.1. First the concept of BPMN will be presented, with all relevant diagram elements of the specification and file format. The term microservice is defined and an architecture using microservices is given. Based on this BPMN will be placed into the context of a microservice architecture. The term *Continuous Integration (CI)* is introduced and changes to the development process using CI are explained. It is shown how BPMN and microservices can be used with CI.

In chapter 3 problems occurring from using BPMN and microservices are extracted. A concept for reducing these problems with a data flow validation is introduces. Based on this concept, requirements for a system implementing that concept are listed. The concept is then explained in detail with regards to the requirements.

Chapter 4 shows the realization of a prototype based on the concept. Examples of microservices and workflow engines using the prototype are given. It is explained how the prototype could be integrated into an existing architecture using microservices, BPMN and CI.

To evaluate the prototype, an industry case study is presented in chapter 5. First the purpose of the case study is defined and the setup of the study is presented. All results are given. The results are interpreted and possible solutions to occurred findings are suggested. The case study is checked against threats to validity and it is estimated how

expressive the case study is.

Finally a summary of thesis is given in chapter 6, including the concept, realization, evaluation and results. Based on these results a conclusion is given and possible future work is presented.

2. Background

Contents

2.1. Business Process Model and Notation (BPMN)

Business Process Model and Notation (BPMN) is a graphical specification for business processes. BPMN was introduced by Stephan A. White in 2004 [Whi04] and is now under control of the *Object Management Group (OMG)*. The *BPMN 1.0 standard* was published by the OMG in 2007 [Bpma] and they published *BPMN 2.0 standard* in 2011 [Bpmb]. This thesis is based on the BPMN 2.0 standard although most elements used are already a part of BPMN 1.0 standard.

2.1.1. Concept of BPMN

One of the goals of BPMN is "to create a simple and understandable mechanism for creating Business Process models, while at the same time being able to handle the complexity inherent to Business Processes" [Bpmb, p. 27]. It is designed to be used by project managers and process designers as well as developers and at the same time be utilized by software. For this purpose it takes concepts of well known diagrams like flow charts and uses only a small subset of base elements that can then be differentiated into types that all inherit their shape and features by the base element. [Bpmb, p. 21]

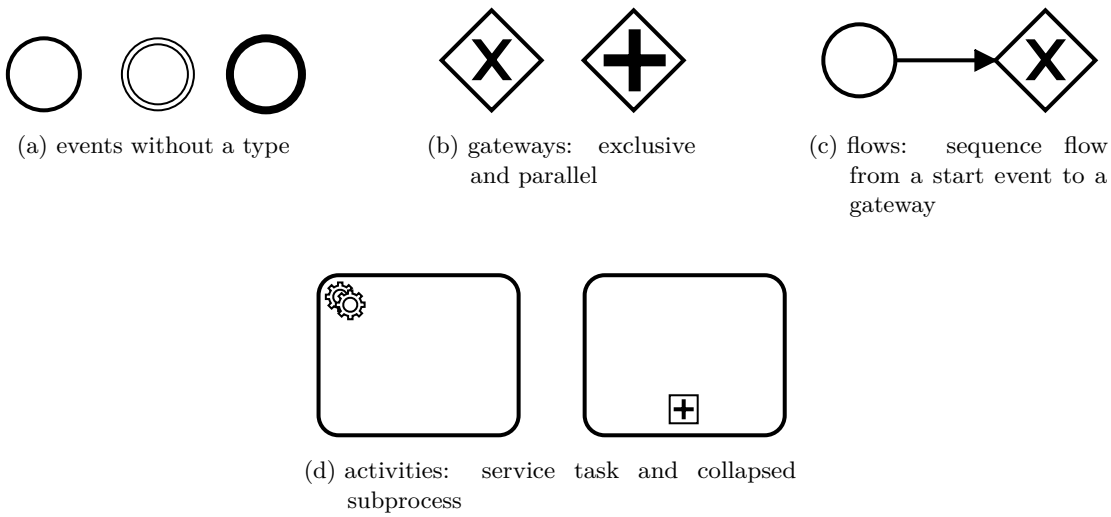
The business process or so called *workflow* is executed from a starting point until end points are reached. For the execution of the workflow (an *instance*) the concept of a `token` is introduced by BPMN 2.0 standard [Bpmb, p. 27]. It is a theoretical concept

that can but does not need to be implemented by BPMN systems. A token is generated at the start of a process and then travels along the workflow. It may be multiplied on and joined throughout the run of an instance. It represents the current position, on which the instance of the workflow is at one stage of the process, and the information at that point in time. An element's behaviour can be defined by defining the interaction with the token. [Bpmb, p. 27]

2.1.2. Elements and Notation

A BPMN workflow consists of *flow objects* that are connected via *connection objects*. There are three different flow objects that can each be distinguished further. In the following a subset and summary of the BPMN 2.0 standard [Bpmb] is given. For a more detailed and definition of all elements refer to the BPMN 2.0 standard.

Activity An *activity* is an executable piece of work in business process. It can either be a *task* or *subprocess*. An activity is represented by a rounded-corner rectangle (see 2.1d). A task is an atomic procedure and has a marker in the upper left corner representing its type. In this thesis we only consider service tasks, which are represented by gears as marker (left task example). A subprocess is a non-atomic activity whose internal details are modelled by activities, events, gateways and sequence flows as well, analogue to a complete BPMN workflow. Subprocesses are either collapsed, which are represented by a task containing a marker with a plus sign on the bottom center of the shape (right task example). Or they are expanded subprocesses, that have the internal process shown in the task rectangle itself (not shown in example).[Bpmb, p. 29]



(a) events without a type

(b) gateways: exclusive and parallel

(c) flows: sequence flow from a start event to a gateway

(d) activities: service task and collapsed subprocess

Figure 2.1.: BPMN flow and connection objects as modeled by the Zeebe modeler[Zeeb]

Event An event is something happening during the business process, that can either have a cause or an impact. [Bpmb, p. 27] This means either something needs to happen in the process for the event to fire, or the event causes something to happen in the process. Events are represented by a circle(see 2.1a). There are three different categories of events (see fiure 2.1a):

1. Start event: circle with single border
2. Intermediate event: double lined circle
3. End event: thick lined circle

Events of every category can have a type, which is represented by an icon inside the circle. The type defines the behaviour of the category in the same way for all categories. [Bpmb, p. 27]
For example a clock inside the circle represents a timer event. This timer event then fires after a specific time. The process is then either started or the flow moves on from the current state. In this thesis types are not considered.
"A Start Event generates a token that MUST eventually be consumed at an End Event (which MAY be implicit if not graphically displayed)" [Bpmb, p. 27]. The start events is the indication of a starting point of a process and the end event is the indication of the end of a process. [Bpmb, p. 29, 83ff.]
In this thesis we expect every workflow and subprocess to have one single start event explicitly given. Also all end events have to be explicit.

Gateway Gateways are used to control how sequence flows converge and diverge within a process. On gateways the process can be merged together on input and/or split apart on output depending on the gateway type. Gateways are represented by a diamond shape including a marker distinguishing the type (see 2.1b).
Exclusive gateways are either explicitly marked with an X (left gateway example) or left empty. *Parallel gateways* are marked by a plus (right gateway example). A gateway can have zero (0) to many (n) incoming and outgoing sequence flows. [Bpmb, p. 29, 90ff] In this thesis at least one (1) sequence flow each is expected.

Exclusive gateways forward a token in a specific instance of the workflow to only one of the outgoing sequence flows. Every outgoing sequence flow has a condition for the decision. One sequence can be the default sequence flow. An instance of the workflow will follow the default sequence flow, if no condition fits. If one condition is evaluated to true, no other condition will be evaluated. [Bpmb, p. 29, 90ff.]
Parallel gateways are used to synchronize parallel flows. Tokens are passed to every outgoing sequence flow. A parallel gateway waits for all incoming flows before it will trigger the token to be passed to its outgoing sequence flows. This means the tokens of all incoming sequence flows will be joined at a parallel gateway. [Bpmb, p. 29, 90ff.]

Sequence Flow A sequence flow is a connection object between two objects of the types gateway, activity or event. A sequence flow indicates the order of execution of the workflow elements. Sequence flows are represented by a single solid line with a solid arrowhead targeting to the target element (see 2.1c). The token is passed from the source element to the target element along the sequence flow. [Bpmb, p. 29, 97ff.]

Data Objects and Data Store BPMN 2.0 standard defines data objects and data store. Data objects represent input and output of an activity and data stores model where the data is stored. Both do not have an effect on the process flow, but are only for analysis of the workflow or for the user's information.
A data object can be associated to multiple activities. In this way the movement of data can be visualized. Data objects cannot be associated to activities of different processes. Therefore data interchanged via processes has to pass through a data store element.
A data object can reference a BPMN `ItemDefinition`. An ItemDefinition defines an import reference, which is by default XML Schema, for a structure definition. Activities can have multiple data objects as input and output. For service tasks this is reduced. A service tasks can have one input and one output at most.
`InputOutputSpecification` define a mapping with at least one `InputSet` and `OutputSet` each, which may reference DataInput and DataOutput elements. [**bmpn20**] Since service tasks may only have one DataInput and one DataOutput, only one of each can be included in the InputOutputSpecification of a service task.
Associations of the data objects are displayed as dotted arrows. Data stores are drawn as a cylindric object. Data objects are represented as a sheet of paper with a bend on the upper right corner.

Not included Elements Additional to the above mentioned elements of BPMN there are sub types of the elements as well as other elements. These elements are not considered in the analysis created in this thesis. Not considered elements include *swimlanes* and *artifacts*.

2.1.3. Example Workflow

With the previously defined BPMN elements business processes can be created. An example of such a process can be seen in figure 2.2. This workflow shows a simplified business process of a booking process on a booking website.
It models that first the booking is initialized and afterwards the customer is validated. This subprocess first checks in parallel, if the given address is valid and the customer is valid and joins this information. This is evaluated afterwards. If the customer is valid, there is a check that the flight is still valid and then the flight is booked. If one of the checks is not evaluated to *true*, an error path is followed.
The first parallel gateway multiplies the token and the second waits for the parallel processes to finish and joins the tokens. The exclusive gateways perform a check for the

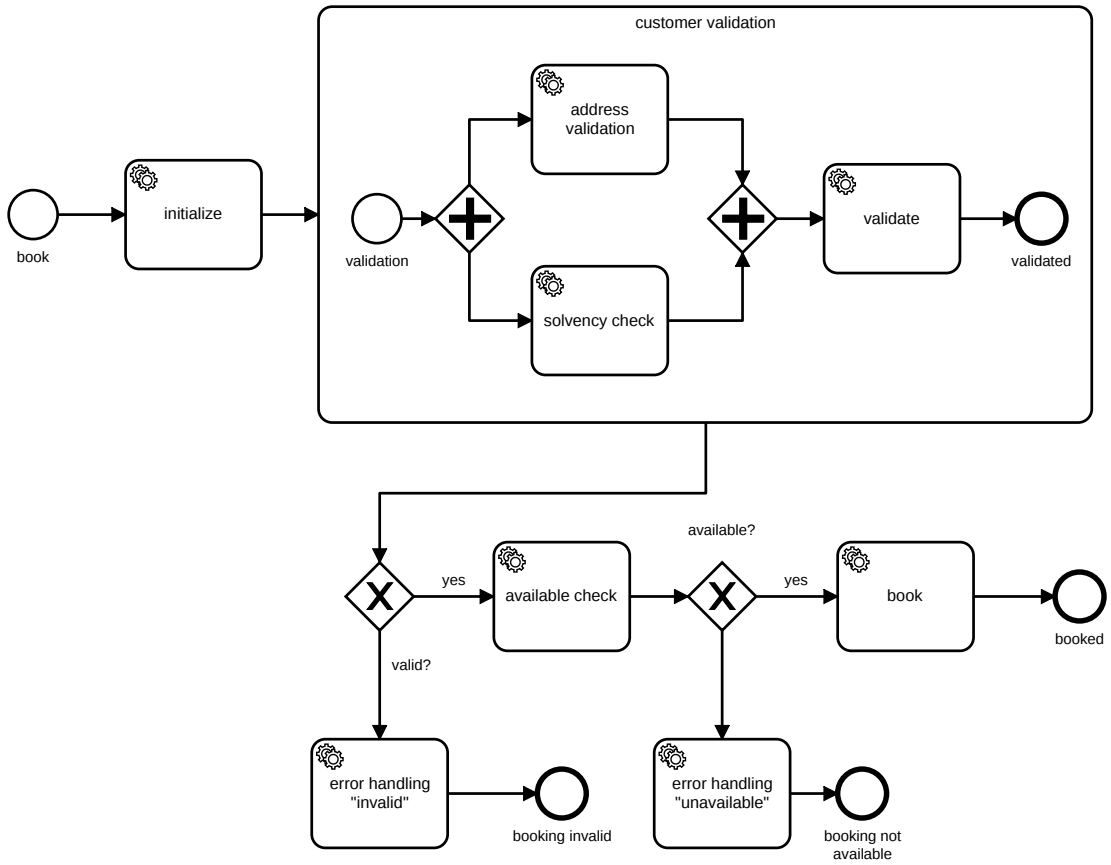


Figure 2.2.: Example workflow of a booking process

condition and only if the condition does not evaluate to true, the default path is chosen leading, to an error. An instance of the workflow is finished, when the token reaches on of the three end events.

2.1.4. XML Representation

BPMN 2.0 standard includes an XML representation for workflows. [Bpmb, p. 475ff.] The representation mainly consists of the semantic definition of the workflow and the graphical representation.

The root element of a BPMN workflow definition must be the `bpmn:definitions` node (see listing 2.1). In this thesis only XML files are considered that define everything without imports.

In a typical workflow definition the `bpmn:definitions` node contains at least one `bpmn:process` node, which defines the process itself. This nodes includes all workflow

```
<?xml version="1.0" encoding="UTF-8"?>
<bpmn:definitions
    xmlns:bpmn="http://www.omg.org/spec/BPMN/20100524/MODEL"
    xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
    xmlns:dc="http://www.omg.org/spec/DD/20100524/DC"
    xmlns:zeebe="http://camunda.org/schema/zeebe/1.0"
    xmlns:di="http://www.omg.org/spec/DD/20100524/DI"
    id="Definitions_11r4rh9"
    targetNamespace="http://bpmn.io/schema/bpmn" exporter="Zeebe
    Modeler" exporterVersion="0.5.0">
  <bpmn:process id="Example" isExecutable="true">
    <bpmn:startEvent id="StartEvent_example">
      <bpmn:outgoing>SequenceFlow_1</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:serviceTask id="ServiceTask_init" name="initialize">
      <bpmn:extensionElements>
        <zeebe:taskDefinition type="initialize" />
      </bpmn:extensionElements>
      <bpmn:incoming>SequenceFlow_1</bpmn:incoming>
      <bpmn:outgoing>SequenceFlow_2</bpmn:outgoing>
    </bpmn:serviceTask>
    <!-- more definitions -->
    <bpmn:sequenceFlow id="SequenceFlow_1"
        sourceRef="StartEvent_example" targetRef="ServiceTask_init" />
    <!-- more definitions -->
  </bpmn:process>
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="Example">
      <bpmndi:BPMNShape id="_BPMNShape_StartEvent_example"
          bpmnElement="StartEvent_example">
        <dc:Bounds x="15" y="89" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="ServiceTask_init_di"
          bpmnElement="ServiceTask_init">
        <dc:Bounds x="1161" y="80" width="100" height="80" />
      </bpmndi:BPMNShape>
    <!-- more definitions -->
      <bpmndi:BPMNEdge id="SequenceFlow_1_di"
          bpmnElement="SequenceFlow_1">
        <di:waypoint x="1065" y="180" />
        <di:waypoint x="1106" y="180" />
        <di:waypoint x="1106" y="145" />
      </bpmndi:BPMNEdge>
    <!-- more definitions -->
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</bpmn:definitions>
```

Source Code 2.1: Example of a XML BPMN definition

elements. Every element of the process is represented by a child node. As seen in the example, a start event is represented by a `bpmn:startEvent` and a service task by a `bpmn:serviceTask`. As all element children of `bpmn:definitions`, every element has a unique id. The elements can have a name, which is displayed in the workflow.
Events, gateways and tasks can have `bpmn:incoming` and `bpmn:outgoing` child elements defining which sequence flows lead to the element and leave it.
Sequence flows are defined as `bpmn:sequenceFlow` elements and contain the redundant information of a source and target reference which references the id of a BPMN element.

The graphical informations can be stored after the process definition in the element `bpmndi:BPMNDiagram`. This node includes the *BPMN Diagram Interchange (BPMN DI)* information. It is the "simplest interchange approach to ensure the unambiguous rendering of a BPMN diagram" [Bpmb, p.367].
The BPMN 2.0 standard defines a BPMN DI ([Bpmb, p.368]). This defines the serialization of the diagram.
Every workflow consists of one *bpmndi:BPMNDiagram* which contains exactly one sub element of the type *bpmndi:BPMNPlane*. A BPMNPlane contains multiple elements of the types*bpmndi:BPMNShape* and *bpmndi:BPMNEdge* representing the different elements of the workflow.
A BPMNEdge and BPMNShape reference exactly one abstract syntax BPMN element. [Bpmb, p368ff.] A BPMNShape for a service task can be seen in listing 13. The attribute *bpmnElement* references the abstract syntax BPMN elements id. The type of the referenced element defines the form of the BPMNShape. The BPMNShape can have elements defining its position, hight and so on.

2.1.5. Workflow Engine

Business processes defined by a BPMN workflow can be executed by software. This software is called a workflow engine. It takes as input a BPMN file and can start this workflow. A workflow engine has an API to start instances of this workflow and executes them.
A workflow engine defines, how tasks can be executed and offers APIs for tasks, user input and other connections. Tasks can either be executed as a monolithic software where every service task calls an internal method or microservices can be used too execute the service task. The workflow engine defines how instances can be started, monitored and checked.

2.2. Microservices

Microservices are a software architecture style in which multiple independent small software parts interact with each other. This architecture is the contrast to a monolithic architecture in which one single unit is responsible for the whole business process. A

microservice architecture is a suite of small services, where every microservice is running independent from the other microservices. For communication every microservice offers a lightweight mechanism (API) to connect to others services or a central system. [JL14]

Fowler and Lewis [JL14] state, that there is not a clear definition of microservices but rather different characteristics of this architecture. They define that a library is a component that is linked into one program. In contrast microservices are rather "out-of-process components with a mechanism such as a web service request, or remote procedure call."[JL14].
In a microservice architecture the microservices are split by business capability, rather than technological layers. This leads to a model where a development team is responsible for its software in production. This increases the contact of the development team to the users. Microservices use smart endpoints but dump pipes. This means they are decoupled and only connected via simple communication protocols that pipe messages between the services.[JL14]

The use of microservices means that a complete software system needs to combine all these pieces of software to a whole system. All components need to be tested and deployed on their own and then the combination needs to be tested again and integrated into the complete system. Here continuous integration and delivery can be used as described in section 2.2.2.

2.2.1. BPMN in the Context of Microservices

As described before in a microservice architecture the microservices connect to each other or a system via APIs. In a BPMN workflow architecture the microservices connect to a workflow engine and the workflow engine distributes the workload to the microservices as needed for started instances. Depending on the support of a workflow engine and the architecture design choices, a microservice can be responsible for one or many service tasks. Both the microservices and the workflow engine can be started with a load balancer, adding instances as needed.

There are different concepts how microservices and the workflow engine can interact. This can be determined by the workflow engine. It can also offer different possibilities.
A workflow engine could search for available microservices for all service tasks via i.e. a service discovery. It could also offer an API and all microservices can connect to the workflow engine and register for service tasks.

Using microservices instead of a monolithic system with BPMN has the disadvantage that it cannot be checked within one system, if every tasks is implemented. Multiple sources have to be checked and multiple systems have to be up and running to test the workflow.
At runtime all services have to be monitored and checked instead of one.
But because of the advantages of microservices this has the advantage of load balancing. Microservices needed in multiple workflows can be started often and so the workload can

be distributed. Also for security issues, microservices accessing private data, can run in a different environment accessing the data, than microservices not dealing with these issues.

2.2.2. Continuous Integration

The practice *Continuous Integration (CI)* is a common working method in agile software development, but also outside agile teams. The term was defined in the Extreme Programming's twelve practices [FF06, p.2]. Its main idea is, that every change to the code should integrate into the main source code and then be tested.
The goal of CI is to always have a running system and all changes are immediately tested. The idea is that when changes are tested early on and are continuously integrated into the whole system, errors are found early on and developers find conflicts with other developers as soon as possible.

In 2006 Martin Fowler wrote an article [FF06] about CI and what is required to do CI. Fowler recommends for every developer to push their code changes at least once a day [FF06, pp.1, 6]. Important for CI is that the code is stored in a source code management tool. A developer always tries to integrate their code to the main line of the source code and build their additions. Although a developer should build their changes, the build should also be automated. In this way it can be also build automatically on an integration system. This is due to the fact that the environment on a developers machine might differ. The build can be done via an CI server. This is a monitor to the repository and new builds are triggered automatically when changes are done to the repository. Every build should also be tested. For this the code needs to be self-testing. All tests need to be run for every build. If tests fail in a build, this means that the whole build should fail. [FF06]

For a BPMN process running microservices this means that the workflows and the sources of the microservices need to be stored in a source code management tool like git and for every change to the microservices or the workflow the according software parts need to be checked. This means the tests for the microservice need to run error free and all tests for the workflow need to be successful.
The CI server needs to check on updates to the BPMN files as well as the source code of the microservices. It can execute test cases and if available, an analysis of the validity of the workflows.
The CI tool might also recognize which microservices belong to which workflow and only run the tests for those workflows, when a microservice is changed. Otherwise it would need to check all workflows on a microservice change.

3. Data Flow Validation of BPMN Workflows

Contents

Executing BPMN workflows with tasks implemented as microservices requires that each task's service is provided with the required inputs and yields the promised outputs. The collaboration of tasks can fail, if a BPMN workflow is modelled such that necessary inputs for one or more tasks are not provided, e.g., other tasks that would provide this information are not part of the workflow or do not precede these tasks. Such errors can be uncovered by analysing and validating the BPMN workflow's data flow.
In the following the concept of such a validation is presented. First, the idea for the validation is laid out and then the requirements to such a validation are presented. Based on this the detailed concept for the validation is defined.

3.1. Data Flow Validation Concept

The data flow of a BPMN workflow can be modelled directly in the workflow. But service tasks can only have one data input and one data output, limiting the expressiveness of the created data flow. Also workflow creators might not have the knowledge of the data structures of service tasks. Including a complete input and output definition in the workflow might also reduce the readability of the workflow.
For this reason in the proposed system of this thesis the microservices define their data input and output themselves outside of the workflow. This is due to the fact, that the microservice developers know what data is needed and, depending on the input, what data can be provided.
The information needs to be gathered for all microservices of a workflow and then validate the workflow for data flow errors. The generated output of this validation can then be used for CI to accept or reject updates or to warn developers or other systems. Also the output could be used for business process designers as immediate feedback to their

created workflows or changes.

3.2. Requirements

A software system is created to fulfil a purpose. Depending on the environment in which it runs and on its purpose different requirements can be defined. These requirements can be grouped into functional and non-functional requirements. In the following the two types of requirements are defined first and then the requirements for the system are laid out.

3.2.1. Functional Requirements

Functional requirements define *what* a system should do. This means it is defined, which behaviour a system should have. Functional requirements can be defined as use cases. Use cases consists of different parts like actors, triggers and preconditions. Since these would be the same for most of the requirements, they are defined first for all requirements and then only the descriptions are given.

The actors for the requirements are the CI system, the microservice developers and the workflow creators. This depends on the trigger of the validation. The trigger is a newly created or changes workflow or microservice. For a new or changed microservice the actor can be a CI system recognizing the changes or a developer starting the validation manually. For a new or changed workflow the actor is either a user or again the CI system. Also a modelling tool could call the validation. The requirements do not change depending on the actor and trigger.

For the system constructed in this thesis different functional requirements are defined. They can be grouped depending of the granularity and the topic.

Validation The validation itself is the main function of the system created in this thesis. Therefore it is the main requirement, that the validation works as intended.
Prerequisites for the following requirements of the validation are that the workflow must be available and the microservices' contracts need to be available.

- The validation needs to support BPMN workflows consisting of the following elements: service tasks, sequence flows, parallel/exclusive gateways and subprocesses defined directly in the BPMN diagram. BPMN elements, which are not mentioned in the list before, must not cause errors in the validation.
- A workflow must be given as a BPMN fill and will be validated for its data flow errors. This includes missing input to a task, wrong conditions to an exclusive gateway and unused data.

- The validation output must contain all found elements that (might) cause an error with the data that causes this error.

Data Input and Output The data defined for the microservices needs to fulfil different requirements. Some of these requirements are based on data definition in BPMN.
Actors for theses requirements are either service developers or a code generator extracting the data from microservice code or generating code out of the data provided. They are triggered whenever a change to a microservice is done.

- A microservice has to provide its input/output specification.
- Multiple data input and output definitions of a microservice must be possible, unlike in BPMN 2.0 standard. Like in BPMN 2.0 standard, a data definition can consist of multiple variables that might be optional.
- Different types and storage possibilities of data must be supported. These types and storage possibilities are depending on the domain the microservices work in.
- Input/output specifications need to be defined only for microservices and conditions on exclusive gateways. Other BPMN elements do not need to specify data.

Other Requirements There are also requirements to the external interfaces, historical data and authorization of a system. For the validation these are the following.

- The validation must be executable by a user or a continuous integration framework.
- The validation output must be reachable for the user or framework executing it.
- The validation does not need to store historical data and can only work with the current state of the workflow and microservices. But using historical data is not prohibited.
- The validation does not need to do authorization of any user. This must be done outside of the system.

3.2.2. Non-functional Requirements

Non-functional requirements define *how* a system should fulfil the functional requirements.

Performance The validation is supposed to run in a continuous integration framework. This means the validation must run within a reasonable time depending on the size of the workflow and the duration of the CI process.

Scalability The system must scale to the number of microservices and workflows. It can run in parallel for multiple workflows, meaning that scaling to more microservices and workflows should have linear runtime growth for the validation. Workflow growth should cause only linear growth in time consumption of a validation run.

Maintainability The validation needs to be maintainable. Adding a new supported element of BPMN needs to be possible. Exceptions in the code must be documented for analysis.

3.3. Validator

A software system might run multiple BPMN workflows created from a pool of service tasks referring to microservices. This means multiple workflows of sub sets of the microservices exist, as service tasks can be combined in different ways.
When creating or changing a workflow the user has to make sure, that the data flow is error free. This means the data requirements of every microservice must be fulfilled. The user needs to know about these requirements, when using building blocks of microservices to create the business process. Since these informations might not be known to the users, a validation concept is shown in the following. This validation analyses the workflow on data flow errors and reports them back to the user, supporting him in his task.

3.3.1. Data Flow in BPMN

A BPMN service task may require input data sets to be executed and can provide output data sets when its execution finished. The data flow can be handled in different ways.
Data can be transported directly from one task to the next via the workflow engine. Data passed on can be called the payload. Depending on the workflow engine this could be stored e.g. in an XML or *JavaScript Object Notation (JSON)* format and passed on. Depending on the workflow engines requirements the data can be stored, read and be overwritten by every task.
Data can also be pushed into some sort of data storage. This can be done in tasks directly or tasks could pass inquiries to the workflow engine which then retrieves or stores the data.

When the data is available to the workflow engine, because it is in the payload or handled via the workflow engine, it can also be used for path decisions in a condition at an exclusive gateway's condition. If the data is not available to the workflow engine, only tasks having access to the data storage can retrieve and write data.

Take the workflow introduced in chapter 2.1.3 figure 2.2. The subprocess `customer validation` (figure 3.1) checks in parallel, if the address is valid and if the customer is solvent. This is then joined and afterwards used in the exclusive gateway. The service task `validate` needs for this as input the result of the two checks before and then combines them into an output which can be afterwards checked. Here it is clear that `validate` needs two informations as input and will offer one as output.

Additionally to the fact, that the data is offered and receivable at all, it is important that the data expected as input for a task matches its expectations. For example the information about address and solvency validation could be a simple boolean or a double

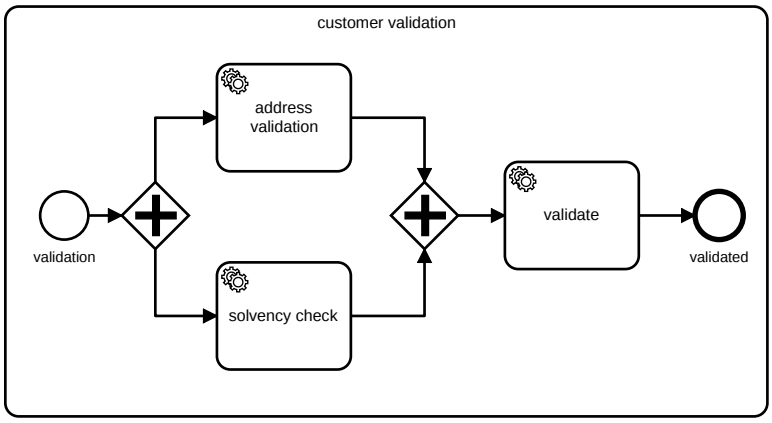


Figure 3.1.: Example workflow: Booking process

representing a probability of how sure the solvency validation is. Also this information could be transported via the workflow or be stored in a database. If the task `solvency check` offers a boolean in a data base and `validate` looks for a double value in the payload, they cannot work together, although the information is accessible to both service tasks.

3.3.2. Data Flow Contracts

To validate the data flow it first needs to be defined, which data is available and necessary. This needs to include information of data structure and data storage.
The concept for this definition is based on *design by contract*. In design by contract *contracts* are defined for all software modules defining preconditions, postconditions and class invariants. A module has to be programmed according to its contract and other modules can test against the contract.
Since the data flow is not validated service to service but the workflow as a whole needs to be validated, the concept of design by contract needs to be adjusted.

As in design by contract every microservice needs to define a contract. It includes which variables are known to the microservice, their type and where the service will retrieve them from or write to. We call these variables properties.
A contract then needs to define which input is expected and what output can be guaranteed according to the input. This is loosely based on *Web Services Description Language (WSDL)*. A contract defines *request* and *response* pairs. Requests and responses are build analogously. They consist of a list of properties. Each property can either be `optional`, `requested` or `forbidden`.
When a workflow reaches the microservice it can only be executed, if the input matches a request. After the microservice's execution it must guarantee that the according response is fulfilled. This means that the properties included in the response are written according

to their definition.

A property has three characteristics, but this could be further extended.
Every property has an id, under which it can be found in the data storage. A property is of a specific type. This means i.e. a boolean or a complex type like a specific object defined in the domain of the workflow. Additionally to this the data storage must be defined. This could either be the payload or some sort of database or data service. Since this is highly domain specific and a general, the data storage is only an id, which must be the same for the same property in every microservice.

The created contracts can be used for testing the microservice or mocking it. This is analogue to design by contract. A microservice can be tested against all request and response pairs. Only if for a request the expected response data is produced, it is implemented as expected. Also a mock can produce test data for a microservice call. This mock can look up which request the call matches and produce a response accordingly.

3.3.3. Static Validation

After the creation and change of a workflow, it can be checked with already existing tools, as described in chapter 1.2, for syntactical correctness. However, these tools will only check the workflow itself without task implementations or information outside the workflow definition. Here the static validation of the contracts defined before can extend the analysis.

Needed Input

The analysis takes the workflow definition in the BPMN file as input, retrieves all contracts defined as described before for every service task and performs the validation on these.
The workflow definition is expected to be a BPMN file in XML format constructed according to the BPMN 2.0 standard [Bpmb]. Also the data expected at the beginning of every instance needs to be defined. This can be done analogue to the contracts described before. Instead of multiple request and response pairs, only multiple data sets as in requests are needed.

Validation Process

As described in chapter 2.1 a workflow exists of multiple BPMN nodes of different types. In this thesis only workflows consisting of the subset of BPMN elements described in that chapter are analysed. The main focus of the validation is set on service tasks. For a complete data flow validation other node types need to be considered as well. Depending

on the node type the validation has to be adjusted.

Recursive Analysis The basic process of the validation starts at the single start event of the workflow and then iterates through all elements connected to it recursively and ends at the multiple end events. For this a workflow has all possible payloads defined alongside the workflow definition as well. These are passed to the start event and from there to all succeeding nodes.

Every workflow node needs a list of incoming payloads from the requests and a list of outgoing payloads connected to the requests, representing the responses. For every node every possible input payload needs to be checked and the corresponding output payload needs to be created and passed to all succeeding nodes (see algorithm 1).
The recursive calls end when an end node is reached and so payloads cannot be passed further on to succeeding nodes. Since workflows can contain loops and therefore nodes can be visited infinitely, there is also the following termination criterion. All incoming payloads are saved in a node. Only unknown payloads to the node are analysed and only if they add new payloads to the list of outgoing payloads these will be passed to succeeding nodes for further analysis. Since the contracts for every task are finite and therefore only a finite number of payloads for the contracts can be created, the analysis will have no infinite runs.

Algorithm 1: Pseudo code for recursive analysis

```
   Data: current node;
   current payloads;
   Result: data flow errors
 1 if payloads already known then
 2     return;
 3 if node is end event then
 4     return;
 5 check new payloads match node;
 6 if no match then
 7     addError("payload does not match contract");
 8     add payload to list outgoingPayloads;
 9 else
10     create payloads according to contract;
11     add created payloads to list outgoingPayloads;
12 for successor node do
13     recursive call with outgoingPayloads ;
```

The analysis and creation of payloads depend on the node type. Therefore this is

described in the following for all considered node types separately. For every type it is described how an incoming payload is analysed.

Service Task The main analysis happens for the service tasks with their contracts. If no contract is available for a service task the node will be considered faulty. Every incoming payload will be considered as not changed in the task and passed directly as output, so that all other nodes can still be analysed although properties might be missing.

When the service is initialized correctly the analysis tries to find a fitting request for the payload. A payload fits to a request, if all required fields in the request are also required in the payload. A payload cannot have an optional or required field that is forbidden in the request.
In case that a fitting request is found, the payload is changed according to the corresponding response. If fields are added in the response or options are changed, this is changed accordingly in the payload and the payload is added to the output payloads.
In the case that no fitting request is found, a mapping is tried out. This means that for a missing variable it is checked, if there might be an unused variable that has a similar name. This could mean that a spelling difference between the service tasks occurred. If a mapping is found, the service task is marked as faulty but an output payload is created as described before.
If the payload fits no request at all, the tasks is marked as faulty and the incoming payload is passed directly to the output without changes, so that all other nodes can still be analysed with the information.

Parallel Gateway In parallel gateway all incoming payloads are passed to all following nodes without changes. However, the BPMN token is multiplied at this point as described in chapter 2.1. This split has to be marked in the payload by this a gateway. When a parallel gateway afterwards joins the parallel flows again it waits for all payloads of parallel processes and then joins the tokens, which contain the information, so here the payloads. If the arriving payloads contain the same property with different options it will take the highest option. This means `optional` fields get the option `required`. If a option is `forbidden`, this is the lowest option. After all payloads are joined into one payload, this will be passed to all following nodes again.

Exclusive Gateway In exclusive gateways the incoming payload is just passed to following nodes. It is checked if the payload can be mapped to an outgoing flow. If so, it will be passed to the according node. If no outgoing flow's condition matches the payload, it will be passed to the default flow if available. If the conditions cannot be analysed, the payload will be passed to all outgoing flows.

Subprocess Subprocesses are handled as a complete workflow. Since a subprocess must have a single start event as well, the payload is passed to this and the analysis starts like in a complete process. It is analysed till it reached all possible end events of the subprocess. If a task of the subprocess is faulty, the complete subprocess is marked faulty. All payloads reaching an end event will be added to the subprocess's outgoing payloads and be passed to the subprocess's following node.

Start/End Event A start event does not analyse the payload at all and will pass the payload to the successor. A payload reaching an end event is saved there and will not be passed any further.

Other Task Types All other types of nodes or tasks are not analysed. If they appear all payloads are just passed on to succeeding nodes.

Output of the Analysis

After the analysis every node can be categorized into error classes. The classes are the following.

- **Not Initialized** Only service tasks can be `not initialized`. This means no contract was provided and therefore no analysis of the node can be done.
- **Not Reached** If the analysis did not reach the node with any payload it is marked as `Not Reached`. This can mean the workflow is faulty or gateways miss outgoing flows for specific payloads.
- **Warning** A node is marked as `Warning` if the analysis reached the node with an payload passed on by a faulty node. The analysis can not be certain that a payload reaching this node would really look like this and therefore cannot be certain it will cause no error.
- **Error** A node is categorized with `Error` if a payload reaches the node that does not fit to any contract.
- **OK** A node is `OK` if it can be reached and all incoming payloads were analysed correctly and outgoing payload could be passed on to succeeding nodes.

The output needs a visualization in the BPMN workflow. All faulty elements need to be highlighted according to their state.
Additionally to a visual output a more detailed textual output is needed to analyse errors. This textual output must include for every element which payloads caused an error and a definition of the error. Possible found solutions to the error need to be included as well. This solution could be for example a found mapping of variables.

3.4. Architecture

The microservice workflow framework should integrate into already established structures with minimal impact to the running system.
Microservice architectures often are highly dependent on speed. Because of this the analysis is static and no changes are made to a running system.

Since CI is used in system for which the validation is created, the workflows are already stored in some kind of workflow registry. This can be i.e. a git repository or a database. This is used for the microservice workflow framework. The additional `Validator` is added to the architecture. The CI system is changed so that the validator is called every time a workflow or service is changed or added. The CI system can decide, depending on the outcome of the static analysis, if a change is rejected, if warnings are published or if the changes are OK. The analysis can be included before integration tests, checking if all parts of the process are available and fit together.

4. Implementation of CobaFlow

Contents

The concept described in chapter 3 can be implemented. For this the validation needs to be implemented and the architecture of the system could be changed accordingly to call the validation. In the following the realization of the validation in a prototype is described and an architecture supporting the validation is shown.

4.1. CobaFlow

The validator is implemented in Java. This was decided because it is a widely used programming language. Workflow engines like *Camunda*, *Zeebe* or *jBPM* support Java and so developers of workflows for these engines will likely be capable of using and developing the analyser. Also Camunda offers the camunda-xml-model ([Cam]), a Java API, used to access BPMN files. This API is used as described in chapter 4.1.4.

The prototype implementation of the validator is called *COntracted BAsed data FLOW validation (CobaFlow)*. CobaFlow is build as a Java program consisting of a main class using different packages for different purposes.
The main class *Validator.java* is responsible for basic checks of the input, retrieving the data and starting the validation. After the validation is finished it also creates the validation's output for the user or possible tools using the result for i.e. CI. This basic procedure can be seen in algorithm 2.

Algorithm 2: Basic procedure of validation

```
Data: BPMN file
Result: graphical and textual validation result
retrieve workflow;
retrieve start payloads;
for service task in workflow do
    retrieve contract;
for payload do
    recursive workflow validation;
generate graphical output;
generate textual output;
```

First the validation is initialized (see chapter 4.1.1) and the contracts are retrieved (see chapter 4.1.4). Based on these informations for every input payload the validation is conducted (see chapter 4.1.4) and finally the output is generated (see chapter 4.1.6).

4.1.1. Initialization

As input the validation needs the BPMN file of the workflow, which is read in at the beginning of the validation. The described workflow is read in with the camunda-xml-model [Cam]. The path to this file can be defined as later on described in section 4.1.7. Additionally to the workflow file, a JSON file with the same name as the workflow must exist, defining all possible incoming payloads.
Each element in the workflow is represented by an instance of a subclass of the abstract class *ValidatorBase*. This class is definde in the *validator* package and has derived classes for the different BPMN elements. Each subclass defines its validation behaviour differently (see chapter 4.1.4). For every service task in this workflow the contract is retrieved (see chapter 4.1.3) and an instance of *ValidatorBase* is created.

4.1.2. Contract Definition

The contracts are defined in JSON format. The contract format is based on the *json-schema* [Jso].
For the task `validate` of the workflow in figure 3.1 an example contract can be seen in listing 4.1.

Every contract can be separated into two main parts. First the `properties` section and then the `contracts`.
The `properties` element, as in json-schema, defines how information is structured and, additionally to the json-schema, where it stored. Like in json-schema the name of a property gets a `type` element defining the type of the property. This can be a common type or referring to a domain specific complex type.

```
{
  "properties": {
    "address_valid": {
      "type": "boolean",
      "source": "payload"
    },
    "solvency_valid": {
      "type": "boolean",
      "source": "payload"
    },
    "valid": {
      "type": "boolean",
      "source": "payload"
    }
  },
  "contracts": [
    {
      "request": {
        "address_valid": "required",
        "solvency_valid": "required"
      },
      "response": {
        "valid": "required"
      }
    }
  ]
}
```

Source Code 4.1: Example of a contract for microservice validate.

```
{
  "input1": {
    "properties": {
      "flight": "required"
      "address": "required"
      "bank": "required"
    }
  },
  "input2": {
    "properties": {
      "bookingNo": "required"
      "address": "required"
      "bank": "required"
    }
  }
}
```

Source Code 4.2: Example of analysis input.

In the example we have three values which are all of type *boolean* and are stored in *payload*. *boolean* refers to a boolean which could be true or false and *payload* is used to mark that it is passed via the workflow engine directly to the service task.
Both fields can contain free text as the content is specific to the domain.
Both attributes are independent of the used workflow engine and the used programming language of the services. In Java the type would refer to the type `boolean` and in Go to the type `bool`. In the workflow engine Zeebe this would mean the data is passed as JSON and can be read or written as `values` by the service. In other workflow engines the properties in the payload could be i.e. an XML document.

After the definition of the properties, the contracts are defined in the JSON list `contracts`.
Every contract consists of a `request` and a `response`. This is loosely based on WSDL. Multiple request/response pairs can be defined per contract. Every incoming message must fit to one of these pairs.
Request and response are constructed in the same way, called `payload` in the following. All properties defined before can be used and marked as either `optional`, `requested` or `forbidden`.
In the example the properties *address_valid* and *solvency_valid* are `required` in the request and then the property *valid* will be guaranteed in the response.

The payloads defined for every workflow as input are designed accordingly. The list of possible payloads could look like listing 4.2. These files contain a JSON list of named payloads that contain a `properties` element defined analogously to a request or response.

4.1.3. Retrieving Contracts

To validate the data flow in a BPMN workflow the BPMN file and all contracts must be available to CobaFlow.
To retrieve the contracts, the names of all microservices are required. BPMN specifies that a service task has an `id` and a `name`. The id is unique in the workflow and the name is an explanation displayed in the diagram for the user to know what is done in the service task. This means both cannot be used to specify the microservice name implementing the service task.
Since some microservices might be called multiple times in a workflow, the unique id is not useful and a human readable name that might change in different workflows although referring to the same microservice cannot be used either.
As explained in section 4.2.2 Zeebe is used as an example workflow engine in this thesis. Zeebe extends the service definition with the node `bpmn:extensionElements` defined in BPMN 2.0 standard. This element has the child node `taskDefinition` with the attribute `type` (see listing 4.1.3).

```
1 <bpmn:serviceTask id="ServiceTask_Init" name="Initialize">
2     <bpmn:extensionElements>
3         <zeebe:taskDefinition type="init" />
4     </bpmn:extensionElements>
5 </bpmn:serviceTask>
```

Source Code 4.3: Definition of the service task init in BPMN with Zeebe extension

In the example listing 4.1.3 the service task `Initialized` is specified. It has the type `init`. Under this name all contracts are retrieved by CobaFlow. For a different workflow engine, the `bpmn:extensionElements` still needs to be included or CobaFlow must be adapted to read in the new workflow engine's specific definition.

When the CobaFlow is initialized all contracts are retrieved and the information is used as described in chapter 4.1.4.

CobaFlow offers two ways to retrieve contracts. Contracts can either be stored in a directory or can be retrieved via an API. How this can be chosen in CobaFlow, is described in section 4.1.7.

Directory When all contracts are chosen to be in a directory they are currently expected to be in the same directory as the BPMN file that is validated. In the directory every contract has to be available under the microservice's name as a JSON file. For the example in listing 4.1.3 this would mean *init.json*.
To retrieve the contracts basic Java functionalities from the java.io package are used.

API The contracts can also be made available via a RESTful API. If this is chosen the contracts will be retrieved from a url defined as

```
http://<server-adress>/<service-name>/contract
```

The API can either be provided by the microservice itself or by another system. On an example how such an API can be realized by the microservice see section 4.2.1. It could also be possible to create one contract server that hosts all contracts. This is not shown in this thesis.
To get the contract the parser provided by Camunda is used (see listing 4.1.3), which internally uses the spring `RestTemplate`.

```
String contract = Parser.LookupService.lookup(this.url);
```

Source Code 4.4: Retrieve contract from API

Parsing The contract structure is represented in the class structure of the validator package `contract` as depicted in figure 4.1. This structure is automatically read in from the JSON file and parsed into Java objects by the Gson library by Google [Gso].

Every Contract is represented by a `Schema` object. This Schema consists of the properties and contracts. A `Property` represents one property that is included in the data available in the workflow. One Property consists of a `type` and a `source`. The type represents the object type of the data set, i.e. an `integer`. The source represents where the data is stored. This can vary between system as described in section 4.1.2. This information can then be used in the service to define data access. This can be considered in code generation described in section 4.2.1.
The list of request/response pairs is represented in a list of `Contract`. The contract class includes a map for requests and a map for responses. Both maps' keys are of type String and refer to a type of a Property. The values are `Options`, an enumeration of Contract. The enumeration's values are `optional`, `mandatory` and `forbidden`.

The other classes of the package are used for the validation and are described in chapter 4.1.4.

4.1.4. Validation

When the initialization is done, the validation algorithm itself is executed. It consists of two parts. First the contracts are checked and then the workflow is validated for every possible incoming payload.

Contract validation When all contracts are read in they are checked for consistency. This means it is checked whether every `Schema` referring to the same `Property` (see figure 4.1) needs to have the same source for the property. If a service provides a data set, i.e. in the payload, and another service expects it to be in another one, i.e. in a

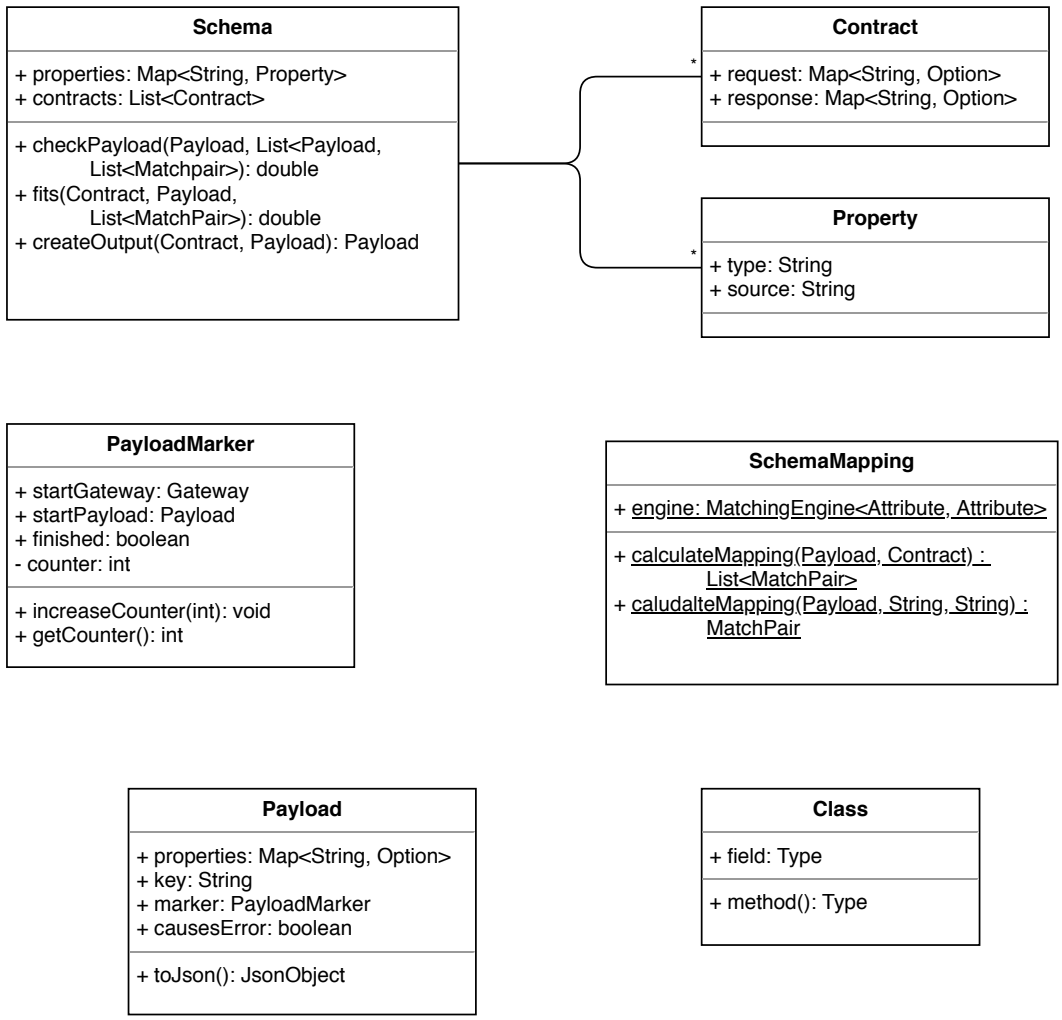


Figure 4.1.: Class diagram: package contract

database, they will not be able to work together.
The algorithm 3 starts by instantiating a map for all known properties with a list of all service tasks known to use this property. Also a list containing all known properties that cause an error is instantiated.

To evaluate which properties are defined incorrect, all services are iterated over. For every service the algorithm iterates over their properties.
If the property is known, it is checked whether it is known to cause errors. If so, the service task is marked as flawed. Else it is checked whether the current property definition matches the already known definition. If there is a difference, it is added to the list of wrong properties and all service tasks using the property are marked as flawed. The task and the property are in any case added to the map of properties.

Algorithm 3: void checkContracts()

```
Data: list of service tasks (static member)
Result: changed list
create map contracts of <property id, list of service tasks>;
create list wrongProperties;
for service task in ValidatorServices do
    for property in service schema do
        if property unkown in contracts then
            if property in wrongProperties then
                create error for service task;
            else
                if property source changed then
                    create error for all service tasks in list;
        else
            add property to contracts;
        add service task to list of property in contracts;
validate model;
write BPMN file;
```

Workflow Validation The main algorithm for the validation is the workflow validation done in the `Validator` function `validate`. This recursive method (algorithm 4) is called for every incoming payload of the workflow. `validate` is based on the abstract class `ValidatorBase` and its derived classes `ValidatorNode`, `ValidatorGateway`, `ValidatorSubProcess` and `ValidatorService` (see figure 4.2).

The method is called once per payload in the list of incoming payloads of the workflow for the start event. After that it will be called for every node recursively with the payloads gathered till that point.

At the beginning of the method a ValidatorBase for the node is searched for. If there is none a ValidatorBase according to the node type is created.
Then the ValidatorBase analyses the payloads. This changes, depending on the node type, as described later on. When there are no new payloads outgoing for this node, the outgoing payloads are returned. Else all succeeding nodes are iterated over. The ValidatorBase decides which outgoing payloads are passed to which succeeding node. This is only split in exclusive gateways. All others pass every outgoing payload to all succeeding nodes. Then the recursive method is called for all succeeding nodes with these outgoing payloads.

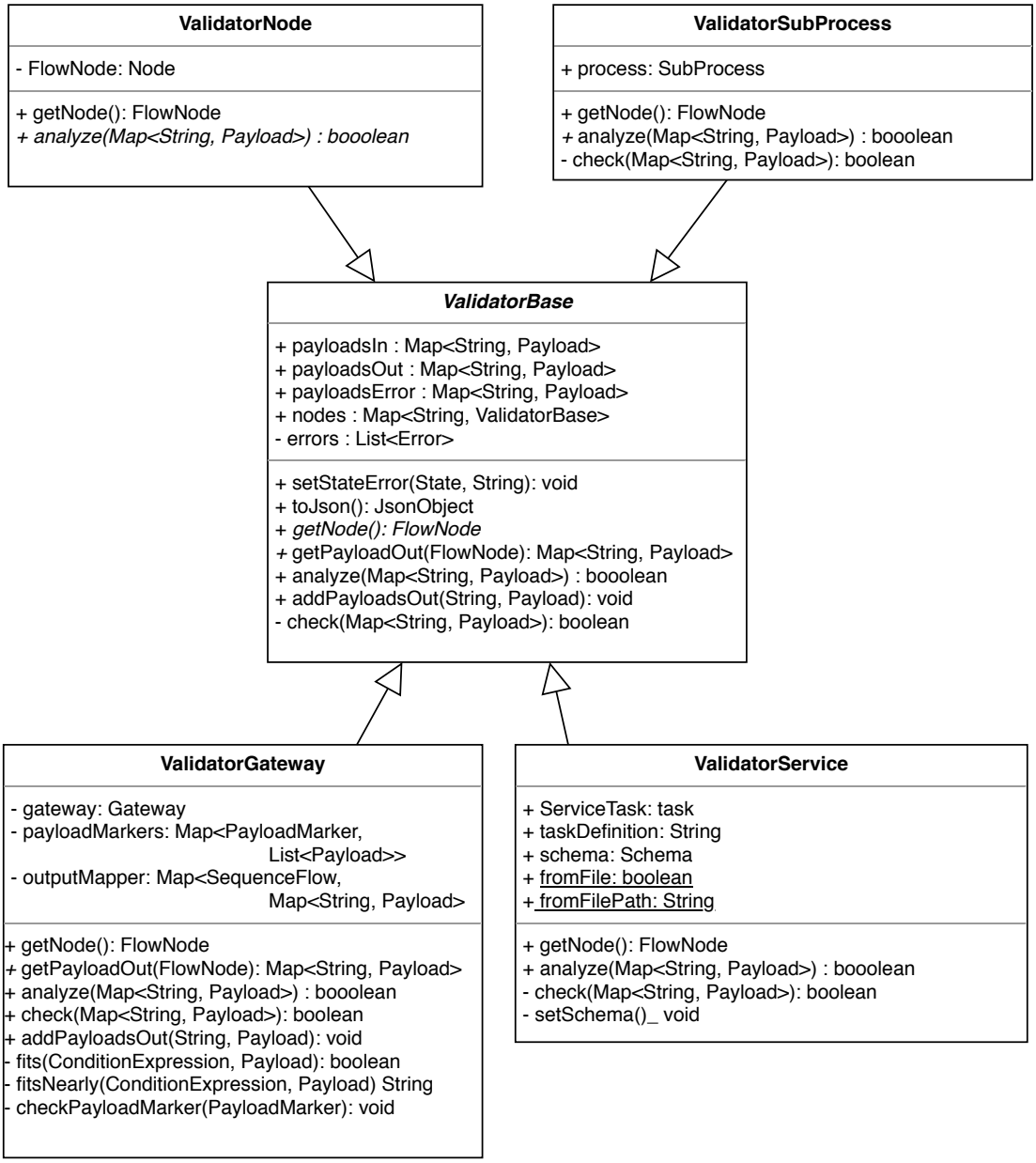


Figure 4.2.: Class diagram: package validation

The ValidatorBase has four derived classes, representing the validated `FlowNode` types in a BPMN file. For gateways, both exclusive and parallel, ValidatorGateway is used. For service tasks ValidatorService, for subprocesses ValidatorSubProcess and for all others ValidatorNode is used. When creating a ValidatorBase in the CobaFlow the according class is used for object creation and the node, returned by the abstract method `getNode()`, is set in the according constructor. All constructors do just that, except

Algorithm 4: Map<String, Payload> validate(FlowNode node, Map<String, Payload> payloads)

```
Data: FlowNode node: Node to be evaluated;
Map<String, Payload> payload: payloads known and available at this point in the workflow;
Result: Map<String, Payload> : payload outgoing for the incoming payloads
1  find ValidatorBase for node;
2  if node unknown then
3      create new ValidatorBase
4  bool newAnalysis = validatorBase.analyze(payloads);
5  if !newAnalysis then
6      return validatorBase.payloadsOut;
7  Map<String, Payload> payloadOut;
8  for FlowNode outgoing in node.SucceedingNodes do
9      Map<String, Payload> payloadsFlow =
         validatorBase.getPayloadOut(outgoing);
10     if payloadsFlow not empty then
11         payloadOut.putAll(validate(outgoing, payloadsFlow));
12 return payloadOut;
```

for ValidatorService. Here the contract is extracted and the schema is set as described in section 4.1.3.

ValidatorBase offers the basic functionalities. It offers the `toJson` method used in section 4.1.6 to create the textual output. It also offers the error handling for all subclasses and defines how the state is set and errors are saved. A basic algorithm for `analyse` is implemented. It only takes all incoming payloads, checks if there are unknown payloads and just hands those to all succeeding nodes.

ValidatorNode is for all BPMN elements that do not have a specific type. This includes for example the start and end event. Therefore it inherits `validate()` and only sets the FlowNode as its node.

ValidatorSubProcess represents the BPMN subprocesses. The functionality of `validate()` is changed. It starts its own validation for the sub workflow analogue to the over all validation. For this it takes all incoming payloads, checks for new payloads and runs a validation for the start event. When the validation is finished it takes all new output payloads gathered till the end and hands them over to all succeeding nodes. If one sub element has another state than `OK` the subprocess inherits that state from its child.

ValidatorGateway handles the validation of the gateways, both exclusive and parallel. `validate()` does not change for exclusive gateways. Here the super method inherited by ValidatorBase is called. For exclusive gateways the handing over of outgoing payloads changes.
ValidatorGateway tries to pass a payload only to the succeeding node, the gateway would choose in a run of the workflow. For this the private function `getCondition` splits the condition of an outgoing sequence flow.
This is then used in `fits` function. A payload fits to a condition, if the value used in the condition is `required` or `optional` in the payload or if it is `forbidden` or not set and the condition needs it to be null. If multiple outgoing sequence flows map, it is handed to all fitting sequence flows.
If there is a condition, with a variable that fits with a ranking higher than 0.5 in the mapping algorithm, it is still passed to that sequence flow. The mapping is described in section 4.1.5.
If the payload does not fit to an outgoing sequence flow, it will be handed to the default flow, if one is available.
If no default flow is available, the payload is handed to every sequence flow, where the condition does not explicitly forbid it. If still no matching flow is found, the payload will be passed to all of outgoing flows and the gateway is marked with an error.

For parallel gateways every payload is handed to every succeeding node. But the payload gets marked with a `PayloadMarker`, if multiple it the gateway has multiple outgoing sequence flows. This saves how many times the payload (the BPMN token) is multiplied to succeeding nodes.
When a payload reaches a parallel gateway this gateway will not pass the payload to succeeding nodes, until the according payload came in from all incoming sequence flows. Then the payloads arriving will me merged to one outgoing payload and passed to all succeeding nodes again. In this merge all variables of the incoming payloads are put into one new outgoing payload.
If one variable arrives from multiple payloads with the same `Option` this will be included once. If it arrives with `optional` and `required`, it will be put in the payload once with option `required`. If it arrives with `forbidden` and another option it will be put into the payload with the other option.

ValidatorService handles all service tasks. Here the contracts are used to validate the payloads. The method `validate()` calls the private function `check` for the payloads (see algorithm 5).

Every payload is checked in method `schema.checkPayload` against the service task's contracts. This method returns a double between `0` and `1` that represents the fitting score of the payload to the contract. The passed object `payloadOut` contains the payloads that are created from the contracts as outgoing for this incoming payload. A fitting value of 1 means that there were no mismatches. 0.5 or higher is considered as fitting, but an error is still produced. Values below 0.5 is an error and the incoming

Algorithm 5: ValidatorService: boolean check(Map<String, Payload> payloads)

```
   Data: payloads: incoming payloads to service
   Result: boolean: new outgoing payloads
 1 for payload in payloads do
 2     if payload known then
 3         continue;
 4     else
 5         newPayloads = true;
 6     payloadsIn.put(payload);
 7     double fitsValue = schema.checkPayload(payloadIn, payloadOut, matches);
 8     if fitsValue == 1 then
 9         put all payloadOut to this.payloadsOut;
10     else if fitsValue <0.5 then
11         setError;
12         put PayloadError;
13         put payloadIn to payloadsOut;
14     else
15         createMappingWarning;
16         setError;
17         put all payloadOut to this.payloadsOut;
18 return newPayloads;
```

payload is just passed as output since no fitting contract could be found.

4.1.5. Mapping

To calculate the fitting score, the method `Schema::fits` is used. It takes all variables used in the payload and checks whether the value is contained in the contract's request and its option matches.
If all fields match, the contract gets a score of `1`. For the case, that a variable does not match in its option, it either gets a value of `0`, if the variable was required in the request but is forbidden in the payload. For every non match with a optional variable a penalty is created. A factor of `0.01` is added for every mismatch but at least `0.1`.
For variables that are required, but are not included in the contract, a matching function is called. The method is called `SchemaMapping.calculateMapping(payload, contract)`. This calls the matching algorithm *WInter.r* [Win] implemented by the University of Mannheim.
For this algorithm all required fields of the request and all fields of the payload are put in a `DataSet<Attribute, Attribute>` each. DataSet and Attribute both are interfaces of WInter.r. Only the variables that are not included in both payload and contract are used. The WInter.r method `engine.runLabelBasedSchemaMatching` is used to calculate the matching. It returns a pairwise matching. Equal variables match with a ratio of 1. For others the `Jaccard` algorithm is used to find a matching ratio. A threshold of 0.5 is used. Pairs with a smaller matching score will not be displayed.
The similarity score is used for a warning text. The scores are added to the text. The sum of these is weight with the number of variables available and the score calculated before. When multiple contracts are defined in the schema of the service task, only the ones with the highest score are considered. A threshold is used to decide, if the payload matches the schema at all as described before.

4.1.6. Output

When the validation itself is finished the output is generated according to chapter 13. First a BPMN file is created, highlighting elements that contain warnings and errors. Second a JSON file is produced including further informations about the elements.

Graphical Output For graphical output a new BPMN workflow is created in the function *void createOutputBPMN()*. Its internal procedure is displayed in algorithm 6.

As described in chapter 2.1.4, every element displayed in the workflow has a BPMNShape element defining its representation. This can be filled with graphic definitions outside the BPMN 2.0 standard. In the example in listing 13 *dc:Bounds* is used for positioning and size. The attribute *bioc:fill* is used to define the colour code of the element in hex colour. In this case the colour is a light red.

Algorithm 6: void createOutputBPMN()

```
Data: workflow (static member)
Result: written BPMN file
for diagram element in workflow do
    if is shape then
        get validator object;
        if validator object found then
            get state;
            safe colour according to state;
        else
            create validator object;
            safe colour warning;
        if colour safed then
            add colour to workflow;
validate model;
write BPMN file;
```

```
<bpmndi:BPMNShape id="ServiceTask_1e3sqbn_di"
    bpmnElement="ServiceTask_init" bioc:fill="#FFCDD2">
    <dc:Bounds x="104" y="21" width="100" height="80" />
</bpmndi:BPMNShape>
```

Source Code 4.5: BPMNShape element defining a service task

The camunda-xml-model is used to iterate over the BPMN workflow plane. Every BPMNShape element is analysed. If the bpmnElement references a validated ValidatorBase object, the state of the validated object is taken. If its state is *ERROR, NOT_INITIALIZED* or *NO_DEFINITION* the colour red (”#FFCDD2”) is saved. If the state is *WARNING* or *INITIIALIZED* the colour is saved as orange (”#FFE0B2”). In case of an element validated as *OK*, no colour is chosen. If no ValidatorBase object is found the colour is also chosen as orange, as this a element was not visited. If a colour is chosen for the element, the attribute *bioc:fill* is set.
When all elements are changed according to their state the workflow is saved where the input workflow file was retrieved from.

A highlighted example workflow could look like in figure 4.3. Here the service task `Initialize` is highlighted red, which could mean for example no contract matched. The service task `book` is highlighted orange, indicating a warning, that might have occurred because of the error in the service task before.

Textual Output Additionally to the graphical output a textual output is created. This output contains further information for BPMN elements that contain an error or a

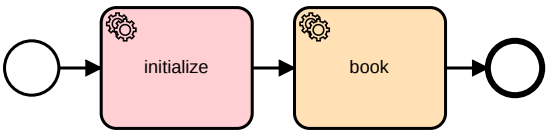


Figure 4.3.: Example of a result workflow.

warning. The method *void resultFile()* (see algorithm 7) creates this file.

Algorithm 7: void resultFile()

```
   Data: workflow (static member)
   Result: result json file
1  for element in ValidatorBases do
2      if state is OK then
3          add element id to list ok;
4      else
5          element to json;
6          add json to list error;
7  add list error to json;
8  add list ok to json;
9  write json file;
```

Here every analysed element is put into a JSON list, depending on whether errors occurred or not. For all valid elements only the name is stored in the JSON list `elements_ok` at the end of the JSON result.
For every element that has at least one error or warning a JSON object is created and put into a list. For JSON creation the Gson library [Gso] is used. For this purpose the ValidatorBase method `public JsonObject toJson()` is called. This creates a `gson::JsonObject` for the element. These objects contain the elements name and state and all messages that were created throughout the validation. Also a list of all payloads coming and and out of the system are displayed for further analysis.
All these JsonObjects are stored in the JSON list `failed_elements` as can be seen listing 4.6.
For all additional warnings the `additional_warnings JsonList` is created. It contains warnings like unused variables. If a variable is not used but a variable with a similar name is optional in a service tasks request, this is marked with a mapping score.

4.1.7. Build and Execution

The validation is a maven project. Therefore it can be build with the command

```
1  $ mvn clean package
```

```
{
  "failed_elements": [
    {
      "node": "ServiceTask_Initialized",
      "state": "ERROR",
      "messages": [
        {
          "ERROR": "no contract matches for payload
              b8c30a80-dcff-4fe1-a00d-495d6a628216"
        }
      ],
      "payloadIn": [
        {
          "id": "b8c30a80-dcff-4fe1-a00d-495d6a628216",
          "causesError": true,
          "properties": {
            "offers": "required"
          }
        }
      ],
      "payloadOut": [
        {
          "id": "b8c30a80-dcff-4fe1-a00d-495d6a628216",
          "causesError": true,
          "properties": {
            "offers": "required"
          }
        }
      ]
    }
  ],
  "additional_warnings": [],
  "elements_ok": [
    "StartEvent_example",
    "EndEvent_example"
  ]
}
```

Source Code 4.6: Example of a json analysis output.

As it is a Java project this can be done on every system running maven.
The built jar can be executed on every system running Java Runtime Environment version 8 or higher. The following command executes one run of the validation.

```
$ java -jar target/analyser-0.1.0-jar-with-dependencies.jar
   -p="/yourpathtoworkflow/" -w="workflowname"
```

The command line options `-p` and `-w` are mandatory. The parameter `-p` defines the path to the workflow to be evaluated. All results will be printed there.
The option `-w` defines the workflow name. The validation assumes in the defined path will be a BPMN file with the workflow name and a JSON file defining the input payloads.
The optional command line option `-f` without a value changes the way the contracts are retrieved. If the option is used, the contracts are retrieved as files from the path. If the option is not used they will be taken by the service task's API as described in section 4.1.3.

4.2. Integration

The validation can be called manually or be executed automatically. It can work with different workflow engines and service tasks can be created in any programming language as long the contracts are created. In the following example usages are described and discussed which changes to a system are necessary or possible.

4.2.1. Service Tasks

As described in chapter 4.1.3, service tasks' contracts can either be retrieved from a directory or from an API. Also there are different ways to create the contracts and how the service can interact with them.

Contract and Microservice Interaction There are three possible ways the contracts and microservices could interact. Every approach can be combined with the validation using the API to retrieve the contracts or get them from a directory.

1. contracts are created manually, service developers must be aware of them and develop against them
2. contracts are created manually, the information can be used for code generation, handling the service's data access
3. contract are generated by the services data access code

In this thesis the first approach was chosen, as it is the easiest and sufficient for a prototype. With this easy approach it can be first checked if concept over all is valid.
The disadvantage of the approach is, that developers must maintain code and contracts in parallel and CobaFlow will fail when contracts and code start to differ. However this was not an important issue in this thesis, as contracts are not included into a productive

system and therefore this disadvantage did not matter.

When using the second approach the contracts could be designed by a software architect, knowing about available data in the system, concerning about compliance rules. Here a developer of the service does not need to bother about where and how to retrieve data but can depend on generated access. A possible way to do this, is to create an abstract base class, offering the data access functionalities and the basic connection to the workflow engine. A developer could derive from this class and only implement the functionality. A disadvantage of this approach is that a developer cannot simply add or change data access but has to change contracts. However this only seems to be more work. Changing a contract might lead to problems on other parts in a system and should be done with the over all system in mind.

The third approach can be difficult to maintain. Extracting only the information which data is used and where it is stored might be fairly manageable. Extracting input and output mapping however could get more difficult. Also this might lead to naming problems between different services when a developer is able to change names and storage of data without other services being involved in the process.

API The services can either offer the contracts via an API or they are stored in a directory. When using the API version this thesis suggest a RESTful API which needs to be available under a url like

```
1 | http://<server-adress>/<service-name>/contract
```

Depending on the used programming language of the service, this API can be created in different ways. A Java service can easily provide such an API when including a spring RESTful web service like described at [Spr].
When Spring is included in a maven build environment, the code could look like in listing 4.2.1. Here the service itself only reads in a contract from a local path and makes this contract available via the API.

```
1  @RestController
2  public class ServiceInit {
3     @RequestMapping("/init/contract")
4      public String contractMapping() {
5          try {
6              byte[] encoded =
                   Files.readAllBytes(Paths.get(getContractPath()));
7              return new String(encoded, StandardCharsets.UTF_8);
8          } catch (IOException ex) {
9              System.out.println("no schema found");
10             return "";
11         }
12
```

```
13      }
14  }
```

Source Code 4.7: RESTful API of a service providing the contract

However, this API can be implemented in different programming languages and libraries. This decision can be made by the developer of the service and does not need to be equal for all microservices evaluated in one workflow.

The validation is independent from the used programming language of the services and how the contracts are produced. It is also possible to combine services in different languages and using different approaches of defining the contracts within one workflow.

The services also need to include the connection to the workflow engine. This differs depending on the chosen workflow engine and is explained in the next section.

4.2.2. Workflow Engine

There are multiple workflow engines available. For this thesis the workflow engine Zeebe [Zeea] in version 0.17 was used. Zeebe is an open source workflow engine initiated by Camunda under the Apache Licence, Version 2.0 and the GNU Affero General Public License (GNU AGPLv3) (only the broker-core part of Zeebe). Zeebe is currently in a developer preview state and was created to focus on microservice orchestration. For this reason it was chosen in this thesis.

The validation can be used with all workflow engines implementing the BPMN 2.0 standard. It currently uses the Zeebe extension tag to specify the services name as described in section 4.1.3. This could be included into a workflow not using Zeebe because it would just be ignored by the workflow engine. Or the validation could be extended to read other engine specific service name definition.

To use Zeebe as a workflow engine, two parts are necessary that are described on the Quickstart page of Zeebe [Zeec]. The workflow engine itself is represented by the Zeebe broker. The broker consist of at least one node and handles all workflows read into the system. Workflows can be read into the broker and then instances of the workflow can be executed. To see how the broker can be started and managed please refer to the guides provided on the Zeebe homepage.

The second part needed to execute workflows are the microservices. A microservice needs to register for every service task type it can execute. A microservice can register for multiple service tasks and multiple microservices can register for a service task. An orchestration tool like Kubernetes could be used to balance the number of microservices. Zeebe will distribute the work to all registered microservices. Zeebe implements a error handling for unavailable microservices, unregistered service tasks, error messages and

timeouts of the microservices.

To create a microservice executing one service task, any programming language able to use gRPC can be used. Zeebe already offers libraries for different programming languages. In this thesis all clients where implemented in Java in a maven project. For Zeebe was included in the microservice's pom file with the dependency information in listing 4.2.2.

```
<dependency>
  <groupId>io.zeebe</groupId>
  <artifactId>zeebe-client-java</artifactId>
  <version>0.17.0</version>
</dependency>
```

Source Code 4.8: Maven dependency for Zeebe client

A Java client needs to define the contact point to the Zeebe broker and then register every service task it can handle. Example listing 4.2.2 shows how the microservice registers for all service tasks of type `init`. The Zeebe broker will then distribute workload for these service tasks to this service.

```
final String contactPoint = args.length >= 1 ? args[0] :
    "127.0.0.1:26500";

final ZeebeClientBuilder builder =
    ZeebeClient.newClientBuilder().brokerContactPoint(contactPoint);

ZeebeClient client = builder.build();
worker.add(
            client
                .newWorker()
                .jobType("init")
                .handler(ServiceInit)
                .timeout(Duration.ofSeconds(10))
                .open()
);
```

Source Code 4.9: Java client registration at Zeebe broker

For every service task the client needs an implementation of type `JobHandler`. This interface provided by Zeebe must implement a `handle` method which includes the functionality for this service task. A simple implementation of this class can be seen in listing 4.2.2. The JobHandler `ServiceInit` used before is implemented in the example. The microservice will only print an information when called.

```
import io.zeebe.client.api.subscription.JobHandler;
public class ServiceInit implements JobHandler {
    @Override
    public void handle(JobClient client, ActivatedJob job) {
        System.out.println("handle task init");
```

```
6           client.newCompleteCommand(job.getKey())
7                       .send()
8                       .join();
9       }
10  }
```

Source Code 4.10: JobHandler for java client for Zeebe

Zeebe can pass data from element to element in JSON format. This data can be set or read via the ActivatedJob job.

4.2.3. Continuous Integration

The validation can be used in a system using CI.
Since on workflow defines the business process of one system it can be validated in the CI tool for this system. Every time a new system is integrated in the companies pipelines CobaFlow can be part of the pipeline.
The textual output of CobaFlow can be evaluated. If all elements are evaluated `OK` the pipeline can run without any problems. For validation results including warnings or errors thresholds can be defined. Since warnings and errors are showing, that the workflow is likely to run into problems, a changed or new workflow should not be delivered if an error or warning is recognized. However it could be desirable to still give it into test with the validation result as remark.

CobaFlow can also be included for changes on a service task's contracts. This depends on the usage described in section 4.2.1.
There are two possibilities. A service tasks creates its contracts from its source code or it is programmed against a contract.
If a service tasks creates its contract automatically from its sources, then the validation must be executed every time a service is changed and the created contract is changed. The CI pipeline can check whether the contract of a newer version of the service has changed. If so the validation for every workflow using this service tasks has to run again. This means the CI tool needs to know which service tasks are used in which workflow. This information can either be extracted from the BPMN workflow or from the validation output. This information is also needed for orchestration tools as Kubernetes, since these tools will need to know which services are need to run for a working system.
If the service tasks is created against an existing contract, the validation needs to be re-run for every workflow, too. But here the trigger for the CI tool is on the contract. The CI tool could also use this trigger to re-run the service task's tests or at least tests for the interface, if they are categorized in this way.

4.2.4. Versioning

The validation and architecture described before to include the validation does not include a versioning of workflows, services and contracts. There are different ways an architecture could deal with this problem.
A BPMN workflow does not include a versioning of a service task. One reason for this is, that it does not mention how a service task is called. A workflow engine can use extensions to define this. For example the workflow engine Zeebe uses an extension element in the BPMN element to define the service tasks name.

When a software system uses multiple workflows depending on the same service tasks or CI tools, a way versioning is necessary.
One way to deal with changes to the contracts, is to always have one valid version of contracts throughout the complete architecture. But as with the use of a DevOps approach or only using CI tools this is normally not possible. Different versions need to coexist.
As long as a workflow engine does not support versioning of service tasks one way of dealing with this matter is to name the services accordingly to the version of the contracts. However, versioning of service tasks does not only belong to changing contracts but also behaviour. This means a version could be changed even though a contract is not changed. But at least a contract change needs to be represented in a version change.

An approach to do this versioning is to include the version in the name of the service task in the CI workflow. In Zeebe the extension could therefore include as type a version. Instead of `init` this could mean that it is called `init1.0.0`.
A service task could then register at the workflow engine with the same extended service name and be called accordingly. A orchestration tool like Kubernetes could then start services in all versions needed by the workflows running. This could mean that one service tasks could be needed in different version, as different workflows might use different versions or even a workflow could be running in different versions, as possible with Zeebe, and the versions use different service task versions.

5. Industry Case Study

Contents

The BPMN validation prototype CobaFlow was evaluated in a qualitative case study. The following chapter will lay out the idea of the case study, how it was designed, what the results are and how they can be interpreted. At the end threats to validity are discussed.

5.1. Study Design

CobaFlow is a prototype to support the work of microservice developers, workflow creators and the CI process of a software system using BPMN workflows in a microservice environment.
To evaluate if the prototype does this, a case study is conducted. It is designed to show whether the concept itself is helpful to the different user groups. It should also indicate where the prototype currently lacks usability and where potential for improvement in the validation and output of it lies.

5.1.1. Participant Selection

CobaFlow allows users with different levels of technical knowledge to compose workflows with the tasks implemented as microservices, as well as giving feedback to service developers about possible errors to existing workflows caused by changes to the microservices' code.
To get feedback from both user groups, developers and users, the selection of participants is based on their previous knowledge of BPMN and their level of technical knowledge. For both they are asked to rate themselves on a scale from `none (1)` to `professional (5)`.

The participants where selected to have non professional and professional knowledge in both fields. A mixture of professionalism in both areas was tried to be fulfilled.

All participants are employees of an industry partner of the Research Group Software Construction at the RWTH Aachen University. The company is specialized in leisure IT. Some of the participants where part of the team this thesis was created with. Others did not know of the thesis or the author beforehand.

5.1.2. Setup

The setup of the case study was the same for all participants. One participant at a time was asked into a room and only the participant and the experiment supervisor were attending. The participant was provided with a laptop connected to an additional external display, a keyboard and a mouse. These tools where needed to perform a series of tasks.
Any deviations from the setup are noted in the case study's protocol.

The supervisor was only allowed to read the introductions and task descriptions to the participants. The conductor may only speak freely, if the participant had problems understanding the tasks or was exceeding the given time per task. Every additional explanation is noted. The study was conducted in German.
The participant worked in an Ubuntu VM with the *Zeebe modeler* to create or change workflows.
The user got a bash command for every task to start CobaFlow or load the current state of the workflow into the Zeebe workflow engine and execute test instances.

Every test started with an introduction that was read to the participant (see appendix A.1). This introduction explains how the following test is set up and that the answers given will be included anonymised in this thesis and that the results will not influence the grade given for this thesis.

After that participant got an introduction to BPMN read out to him/her (see appendix A.1). This includes the following BPMN elements, which are later on used in the case study. They are introduced with a small workflow describing the function of a vending machine (see figure 5.1).
The introduced elements are

- start event
- service tasks
- parallel and exclusive gateways
- end event

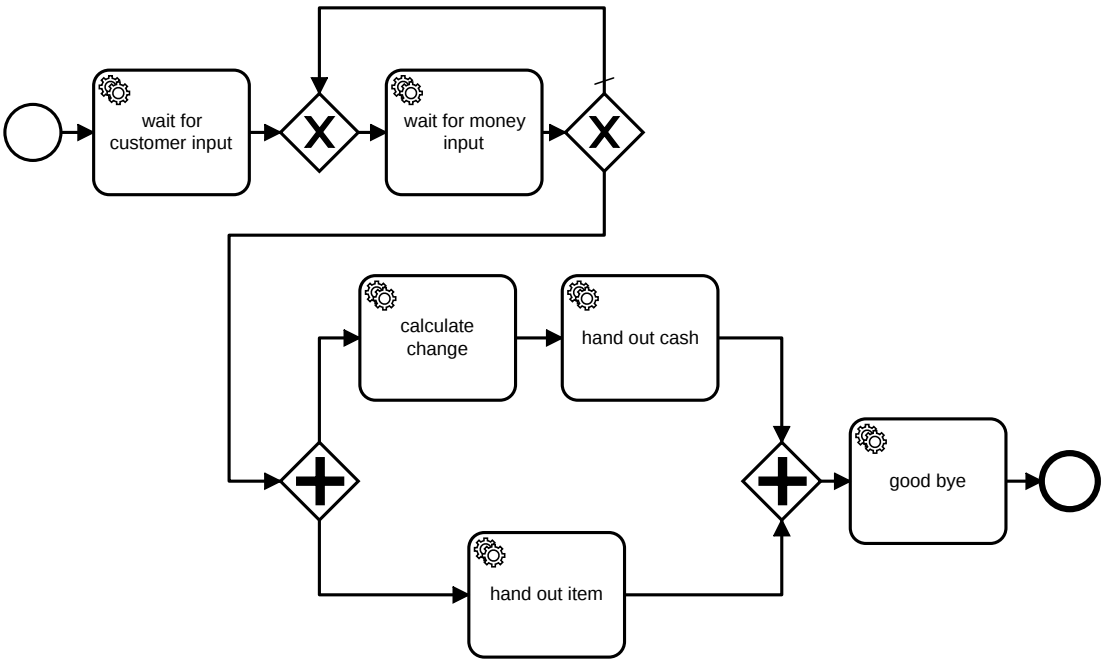


Figure 5.1.: BPMN introduction: example workflow

After that the participant was asked how familiar he/she is with BPMN, programming, XML and JSON (see questionnaire A.2).
Technical and BPMN knowledge were asked for as described before. The participants were asked if the programmed in the last year, ever of never. They could state if they knew XML and JSON at all.

Then the participant got an introduction to CobaFlow (see appendix A.1) and the concept of service contracts (see appendix A.2). Here, the basic abilities of CobaFlow were presented. This included that it checks for data flow errors in every BPMN element. Then the concept of contracts was explained and how they are involved in the validation of CobaFlow.

The participants were asked to perform three tasks (see section 5.2). For each task, they were given 15 minutes to execute the task.
After every task they needed to rate the difficulty of the tasks from `easy (1)` to `difficult (5)` and give a free answer about what was helpful to execute the task and where problems appeared.

The participants were divided into two groups. For this the supervisor tried to rate the technical and BPMN knowledge of each participant beforehand and grouped them, so that professionals and non professionals were represented in both groups. Depending on the group they could use CobaFlow in task 1 (Group 1) or in task 2 (Group 2).

After the third task the participants were asked final questions (see A.7). They were asked if they understood the concept of BPMN and if they thought, people without deep technical knowledge where able to create workflows without further support like CobaFlow and if contracts and CobaFlow could help microservice developers in their work (answer possibilities for all four: `yes, no, I don't know`).
They were asked how helpful the workflow engines output to the test instances, the textual and graphical output of CobaFlow were with answers ranging from `not helpful (1)` to `very helpful (5)`.
In the end they could give a free text feedback of how CobaFlow could be improved.

When all questions were answered, the supervisor and participant went through all answers and make sure everything is readable and clear to the supervisor.

5.2. Tasks

The participants were asked to execute three different tasks. First they needed to parallel a sequential workflow to make it faster, then they needed to find errors and in the end create a workflow from scratch from ten given service tasks. In the following the tasks are described in more detail. Each task ended after 15 minutes independent of whether the participant completed it or not.

Task 1: Make workflow parallel. In the first task, (see appendix A.4) the participant was presented with a sequential workflow in the Zeebe modeler (figure 5.2, shown without line breaks).
The participant was asked to parallel the workflow as much as possible. Group 1 was allowed to use CobaFlow while the others were not.
The user needed to change the workflow in the Zeebe modeler.
The participant was provided with one test instance and the command to execute it and the *Zeebe simple monitor* could be used to see the result. The task was finished if the participant said there is no better solution and the test instance ran to the end error free.

Task 2: Find errors in a workflow. In the second task (see appendix A.5), a workflow was shown to the participant (figure 5.3) and the contracts of the used service tasks were given.
There are three different errors included in the workflow:

1. Unreachable path after an exclusive gateway: the variable used in the condition is misspelled in therefore the path with the condition will never be chosen.

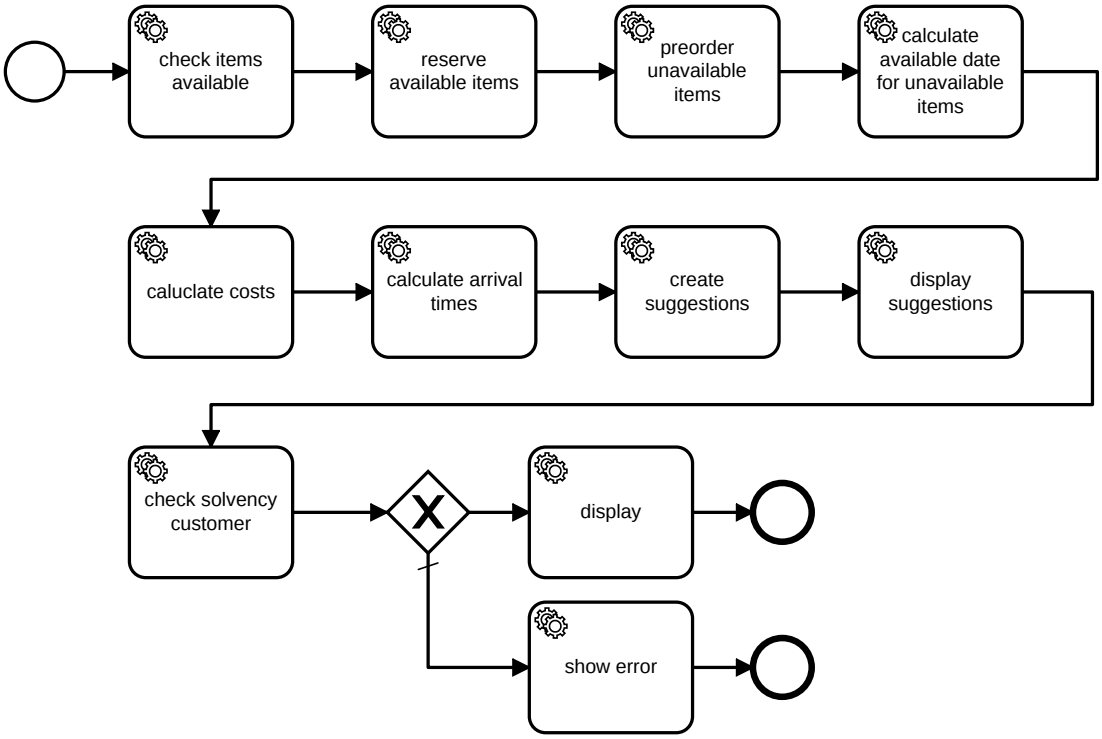


Figure 5.2.: Sequential workflow of task 1

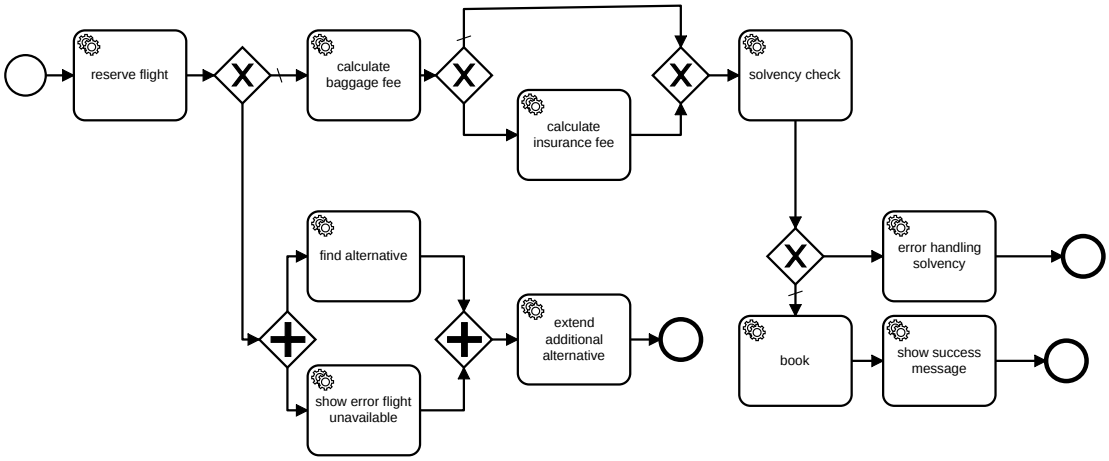


Figure 5.3.: Booking process that has errors for task 2

2. A tasks needs a missing variable, that was never available: Alternative flight search needs to have a price range from another task

3. Optional field is misspelled and therefore never used: Service task *booking* misspelled

the optional input field *insurance* so it will never book the insurance.

The participant needed to recognize as much errors as possible and give a possible solution verbally.
Group 2 could use the output of CobaFlow from the beginning. Participants of group 1 could only use it after not finding further errors or explanations without it. Both groups were presented with result of three test instances in the Zeebe simple monitor.Every error is represented in one of the test instances.
The task was finished when the participant told the supervisor he/she could not find further errors with the help of CobaFlow or all errors were found.

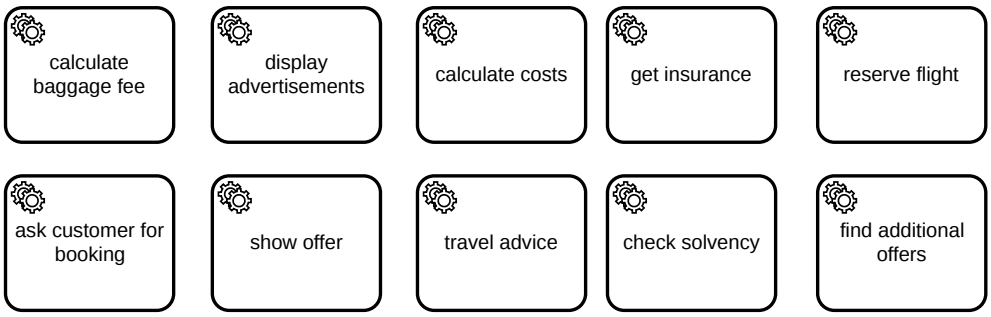


Figure 5.4.: Service tasks for the booking process tasks for task 3

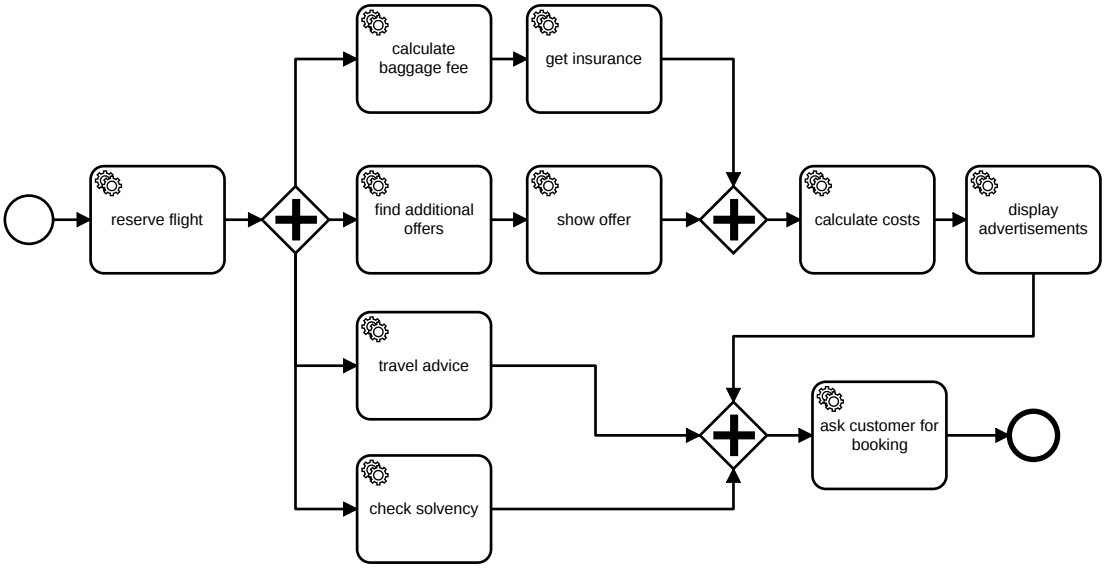


Figure 5.5.: A reference optimal solution for task 3

Task 3: Create a workflow from scratch. In the third task (see appendix A.6), the participant needed to create a workflow from scratch.
All possible service tasks were pre defined (figure 5.4) and opened in the Zeebe modeler by the supervisor.
The user needed to order all tasks and create additional sequence flows and parallel gateways. The participant was asked to use parallel processes as much as possible to speed up execution time.
A possible solution can be seen in figure 5.5. The participant could execute a bash command, which starts CobaFlow on the created workflow and a test instance result which the participant could see in the Zeebe simple monitor.
The task was finished when the participant said there was no better solution and all test were successful or the user stated he/she could not find a working solution.

5.3. Study Execution

The case study was conducted with 6 participants, 3 participants per group. Participants 1,2 and 5 where put into group 1, participants 3, 4 and 6 into group 2. This was done based on BPMN and technical knowledge level as described before.
In the following the results are presented and after that interpreted. The participants are referred to as *P1* for participant 1 and so on. All answers and notes, that are not included in the next chapter can be found in appendix A.2.

Most participants did not have prior knowledge of BPMN, so help with creating sequence flows and gateways was needed. The supervisor showed how to create an exclusive gateway and how to delete or create sequence flows in the first task. Gateways and sequence flows where only created by the supervisor on explicit order.

5.3.1. Results

The participants always needed the 15 minutes per task. Some did not finish with the tasks 1 and 3 within the given time but could always tell where the remaining, not yet fixed, errors were by using the output of CobaFlow and could give right solutions verbally for them.

The participants had different backgrounds. P1 has very basic technical knowledge (5.6), the others are all used to programming (5.1), although this might be longer ago (P3) or it is not they're current field of expertise. All where familiar with XML and only P3 did not use JSON before (A.1).

The participants had different levels of BPMN knowledge (5.7). Two were not familiar at all (P1, P3), two worked with BPMN before (P2, P5) and two only had very basic

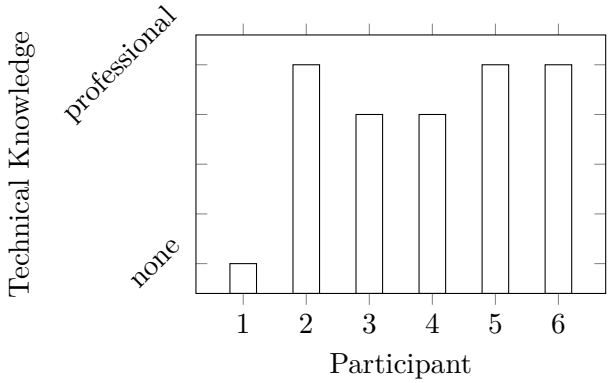


Figure 5.6.: Self-assessed technical knowledge of the participants

Participant	1	2	3	4	5	6
Yes, programmed in the last year		X		X	X	X
Yes, programmed over a year ago			X			
Never	X					

Table 5.1.: Has the participant programming experience

knowledge (P4, P6).

The tasks where perceived as increasingly difficult (see figure 5.8).
Using the Zeebe modeler was not causing this problem, as most had little to no problems using it (see figure 5.9).
Understanding BPMN seemed not to be a problem, as all stated, that they think they understood BPMN after the tasks (see A.2) and one participant also stated that the introduction and task formulation was helpful.

In the free text feedback after each task the participants stated, that the contracts where most helpful executing the task.Also the visual validation output was mentioned by the participants, especially in the second task. The textual output of CobaFlow was said to be helpful but overloaded. One participant said, that you need to get used to the text and then it is helpful.

The answers to what was helpful, show that the visual output was the most helpful (see 5.10. The Zeebe modeler is not ranked as helpful, as most did not use it. The textual output was also often perceived as less helpful, as the participants stated in the free text answers, that it was too hard to understand and to much information to understand.
P6 suggested, it would get more helpful, the more he would work with it, as he then would know, what the fields mean and how they should be read.

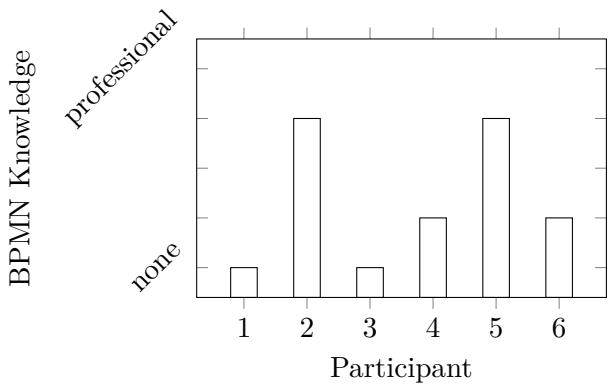


Figure 5.7.: Self-assessed BPMN knowledge of the participants

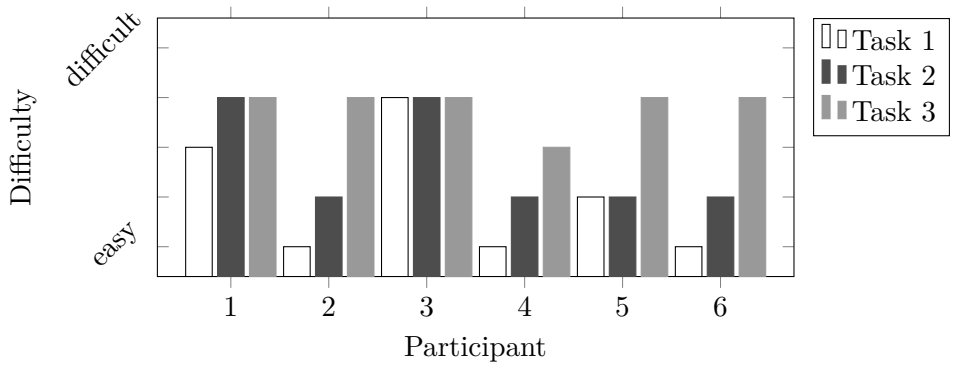


Figure 5.8.: Ranking of the difficulty of each task

The participants all stated that they understood the concept of BPMN as used in the tasks (A.2). P4 and P5 stated workflows could be created by people with little technical knowledge without further help like the validation. The others all stated it could not (A.3). P1, the participant without technical background, even stated verbally this could not be possible at all.
Except for P1 all participants stated that contracts (A.4) and the validation (A.5) could support software developers. P1 chose answer *I don't know* as he was not a software developer and could not image what was helpful for them.

When asked if people outside of a technical department would be able to create workflows on basis of service tasks provided by the software department as done in task 3, P1, P2 and P3 rated the confidence with 4 out of 5 and the others with 5, with 5 being the highest confidence (5.11).

In the free text answer to how CobaFlow could be improved, participant 1 suggested, that changes can be made directly in the marked visual output. Participant 5 suggested

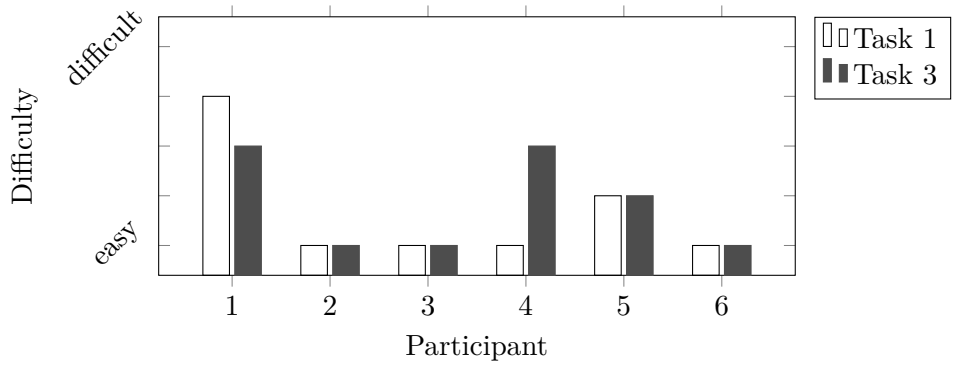


Figure 5.9.: Difficulty using the Zeebe modeler

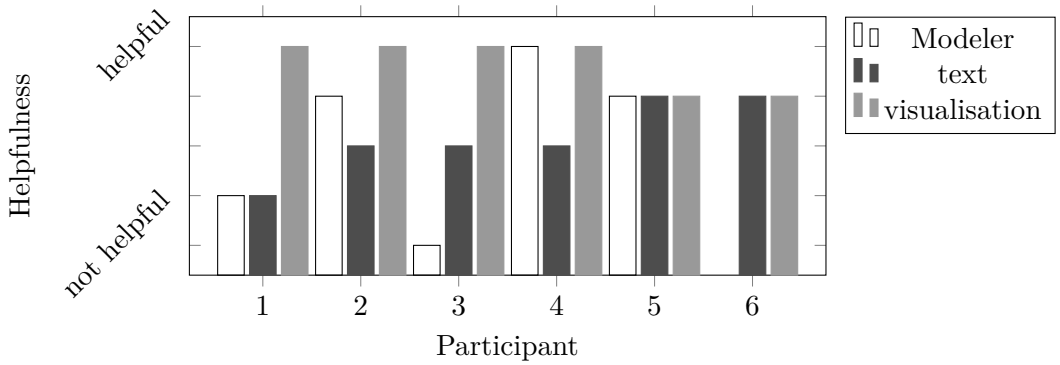


Figure 5.10.: What was helpful?

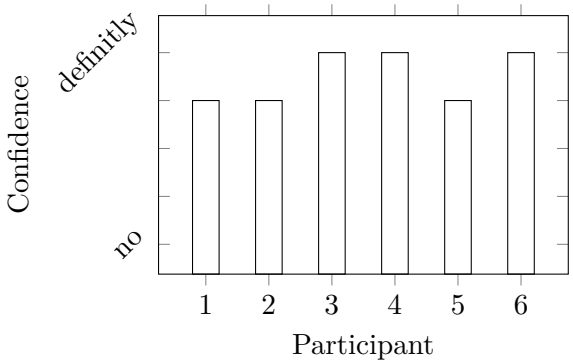


Figure 5.11.: Could people outside of technical departments create workflows on basis of service tasks provided by the software department (as in task 3)?

that the validation could be executed while working on the workflow.

The understandability of the validation between the tasks was received very mixed (see 5.12). 4 out of the 6 ranked the understandability of the validation in the second tasks as the highest.

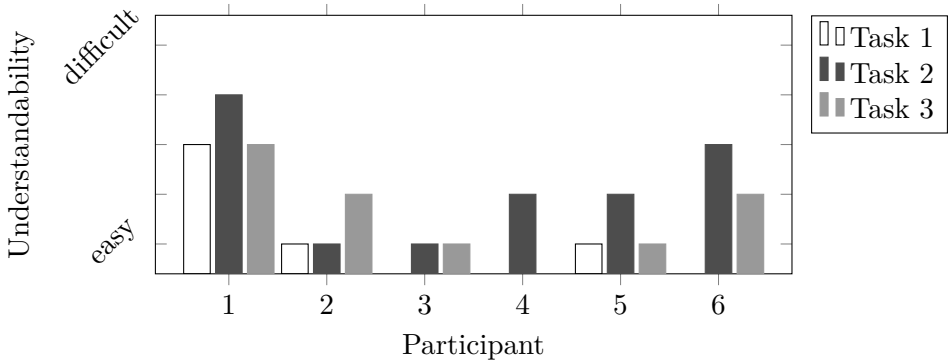


Figure 5.12.: Understandability of validation (if used)

5.4. Interpretation

The understandability of the validation was the highest in task 2. This could be due to the fact, that the participants used it there the most. In tasks 1 and 3 the users only used the validation at the end of the tasks and did not solely work with it.

Since the participants over all only used the validation at a finished workflow, it seems that a constant validation with graphical output in the current modelled workflow would be helpful. Constant validation could increase the understandability as the amount of data would be less, since consequential errors could be avoided and only the current built flow is analysed.

The case study was conducted with users that saw the system for the first time. Except for 2 participants all others only new little or none of BPMN before. These two factors might lead to a higher difficulty of understanding the validation. Still the validation could be improved by a clearer textual structure and a minimized content, since the participants stated the textual output was to overloaded and hard to understand. Instead of giving every information, it could be divided into a shorter error description as an overview with a deeper information for further information.

The graphical validation was ranked as the most helpful part while executing the tasks. However the information content is small. A combination of the textual and graphical output could close the gap between the helpfulness. A short error or warning description could be included into the BPMN diagram.

This could be done by comment element which is part of BPMN 2.0 standard and is shown in graphical tools. It could also be done by an extension field for BPMN elements, and a i.e. the Zeebe modeler could be adjusted to show this information, when hovering over an erroneous element.

The users where overall confident, that the workflows could be created by a business department with the help of the validation. Without a validation this was seen as not possible by 4 out of 6. This shows that the validation could be a helpful tool supporting an microservice environment, where workflows can be created by people, who do not also create the microservices.

5.5. Threats to Validity

There are different threats to the validity of the conducted case study. This means that the setup might cause misinterpretation or not represent all possible users opinions. In the following the threats to validity are given and explained.

Small Sample Size A small group of participants might not represent the majority of users. If the sample size is to small only some problems might occur, by chance the participants only are members of a group only having specific problems and results only a few users might have will likely be not included in the sample.
The case study was conducted with 6 participants and needed to represent a wide field of technical and BPMN knowledge as well as software developers and workflow users.

Selection Bias This threat to validity appears during the selection of the participants. If they are not equally selected from the user groups this might lead to failure in evaluation which results occurred and how often they appear for users.
The participants were chosen from one company and some were members of the same team. This means that it is hard to represent the vast field of microservice developers and workflow users. Even representing the field of leisure travel domain is hard. Since there were no candidates available with high knowledge in BPMN but low technical knowledge this group was not represented.
Using workflows with leisure travel as a theme could have led to selection bias as well. The tasks might have been easier, if the workflow was as expected from previous knowledge. Or the tasks might have been harder, if they were not as expected. Some participants even stated that their previous knowledge led to misinterpretation of service task names and made it harder for them to find solutions.

Experimenter Bias If the experimenter knows about the participants background, he might interpret results depending on this knowledge. It might also cause him to group the participants into wrong groups.

The study was conducted by the author and therefore experimenter bias cannot be ruled out, although the supervisor stuck to the script as much as possible.

Participant Bias The participants might know the experimenter. This can cause biased results as the participants might answer and act in favour of what they think helps the experimenter.
The participants knew the supervisor before the study and might have thought that the results of the thesis might cause a different grade which could lead to a biased answering of the questions.

As can be seen from these threats to validity, it cannot be ruled out, that the results of the case study might lead into a wrong direction.
However, the case study is supposed to give a first impression of whether the basic concept of CobaFlow and moving the data definition outside of BPMN to the microservices is a path worth for further investigation.
Therefore the case study is valid to give a first impression and show possible enhancements of the concept. For deeper insights a study with a larger group from different domains is needed.

6. Conclusion

Contents

Based on the concept and results of the case study a conclusion can be found. In this chapter the idea, concept, realization and results are summarized. This summary leads to a discussion of the results and possible future work is laid out.

6.1. Summary

Business processes can be designed in multiple ways. BPMN is a well established notation to create such business processes and directly execute the created models. There are many different workflow engines available, reading in BPMN files and executing the workflow.
Microservices are an software architecture style where a software is split into different, independent software parts, which are combined to a complete system. Each microservice offers an API to interact with and has his specific business logic it is responsible for. Multiple microservices in the same system might be written in different programming languages and might store data independently.

Combining BPMN and microservices leads to special requirements. In BPMN a service tasks is represented by a microservice. Domain experts want to create different workflows from buildings blocks based on the microservices. However, there are data dependencies between these building blocks. Although these can be created within a BPMN workflow, domain experts might not have the knowledge of the data dependencies. Developers of services have this knowledge instead. Also defining data dependencies for every new workflow again is costly as it needs to be done for every workflow again, although this information did not change.

To solve these problems a data flow validation based on contracts for every microservice is proposed. For every microservice a contract can be created, defining its input and output. The created validation CobaFlow (COntracted BAsed data FLOW validation) collects all contracts and checks for all elements of BPMN. If the given data is sufficient, there are unreachable paths and whether data might not be used or named and stored in different ways between different elements.

The created prototype CobaFlow was tested with a industry case study. Six participants of a software company in the leisure travel market where asked to use CobaFlow. They had to execute three tasks each independently and where asked about their previous knowledge. After doing the tasks their feedback and suggestions was asked for.
The general feedback was that the CobaFlow output is currently to much and unstructured and that the output could be created earlier in the background, the information could be directly displayed in the workflow that is changed currently and more information in the graphic output might help. Most of the participants did not think that a domain expert could create workflows without help of a tool like CobaFlow.
This results lead to the conclusion that general concept could be useful and is worth following but that the created output needs to be improved.

6.2. Discussion and Future Work

The results of the case study show that the concept of data definition outside the workflow and data flow validation might be helpful for microservice developers and workflow users alike. To increase the usability and output of the validation different approaches could be followed.

It is possible to combine the data flow error detection of von Stackelberg et al. [Sta+14] with the setup of the validation presented in this thesis.
Here the contracts could be still defined on the microservice side and the information could be translated to data information in the BPMN workflow. The workflows could then be validated with the analyser of von Stackelberg et al. [Sta+14] instead of CobaFlow. These two data flow validations could then be compared.
Von Stackelberg et al. [Sta+14] did not specify the graphical or textual output of the error detection. It is only stated, that for every found data flow error, the tool returns the data object and task causing the error [Sta+14] p. 12. Here the current or improved output of CobaFlow could be used.
The combination of both systems would not comply to the BPMN 2.0 standard. The possibility of multiple data in- and output for service tasks is ruled out in the BPMN 2.0 standard. But this is crucial for a microservice architecture using BPMN.
Also the error detection does only work on “well-formed and sound BPMN process model“ [Sta+14, p.12]. CobaFlow can work with under specified workflows and deal with minor errors, thus supporting an early on validation. As shown in the evaluation a early and constant feedback might be helpful for creators of workflows. Therefore a combined solution should try to inherit this behaviour from the validation.

Based on the findings of the user study the framework could be improved as suggested in chapter 5.4. This means CobaFlow could be continuously be executed in the background when creating or changing workflows. In this way the user could always see the current state of data flow validity of his changes.

Also the graphical output could be enriched with deeper information of the validation with shorted information that is already present in the textual output.

To support the work while creating and changing workflows, not only a continuous validation without might be helpful. The information could also be used to suggest tasks that are allowed at a specific point of the workflow or suggest tasks that might solve missing data problems. Therefore a BPMN modeler could be developed or enhanced, that supports the developer by suggesting these or forbidding tasks that will not have enough input data at a specific point in a workflow.

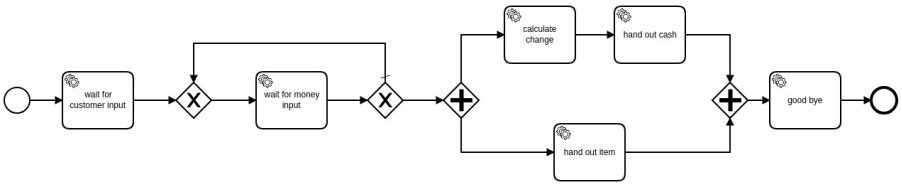
A. Evaluation

A.1. User Study Setup

Einführung

Einführung in BPMN

Diese Nutzerbefragung soll helfen zu verstehen, wie Nutzer mit BPMN Workflows arbeiten und wie man sie in ihrer Arbeit unterstützen kann. Deine Antworten und alle Notizen werden in der Masterarbeit anonymisiert enthalten sein und ausgewertet. Ich werde Notizen machen, wie du vorgehst und wo ich dir zusätzliche Hilfestellung geben muss.
Du wirst 3 Aufgaben bekommen und jeweils 15 Minuten Bearbeitungszeit haben. Daran angeschlossen ist jeweils ein kurzer Fragebogen, den du ausfüllen musst. Deine Antworten haben keine Auswirkung auf die Benotung der Arbeit.
Du wirst jetzt erst eine Einführung in das Thema BPMN erhalten, selbst wenn du BPMN schon kennen solltest. BPMN ist viel umfangreicher, als das, was ich dir gleich zeigen werden. Bitte nutze in den Aufgaben nur die BPMN Bausteine, die ich dir vorstelle.



BPMN steht für Business Process Model and Notation. Es ist eine graphischer Weg, Business Prozesse, sogenannte Workflows zu gestalten. Sie können nach der Erstellung in sogenannten Workflow Engines eingelesen werden und können dann automatisch ausgeführt werden.
Sehen wir uns jetzt diesen Workflow hier an. Er soll den technischen Ablauf in einem Getränkeautomaten darstellen. Der Automat fragt zunächst den Kunden nach seinem Wunsch. Dieser kann dann ein Getränk auswählen. Wenn er das getan hat, wird er nach Geld für das Getränk gefragt. Nach jeder Münze, die eingeworfen wurde, wird geprüft, ob genug Geld eingeworfen wurde. Wenn genug gezahlt wurde, werden zwei Dinge parallel ausgeführt. Zum einen wird ausgerechnet, die viel der Kunde zuviel gezahlt hat und dann das Wechselgeld zurück gegeben und zum anderen das Getränk ausgegeben. Wenn beides ausgeführt wurde, erhält der Kunde noch einen Abschiedsgruß auf dem Display und der Vorgang ist beendet.
Hätte der Automat eine Workflow Engine könnte der Ablauf so eingelesen werden und pro Aktion könnte Software hinterlegt werden, die ausgeführt wird, wenn der Workflow an dieser Stelle angelangt.
Schauen wir uns den Workflow noch mal an. Jeder Workflow hat einen **Startpunkt**, das ist der runde Kreis auf der linken Seite. Immer wenn ein Kunde neu zum Automaten kommt, wird dieser Ablauf an diesem Punkt neu gestartet.
Diese Kästchen sind **Servicetasks**. Sie stehen für ein Stück Software, dass im Hintergrund ausgeführt wird, wenn der Workflow an diese Stelle gelangt. Wenn die Software ausgeführt wurde, wird der Workflow, entlang der Pfeile weiter ausgeführt. Jeder dieser Servicetasks benötigt bestimmte Daten als Input und gibt andere als Output aus. In und Output sind optional. Der erste Task "wait for customer input"würde zum Beispiel keine Input erhalten, aber der Ablauf erhält von außen durch den Nutzer dann die Nummer des Getränks und gibt diese weiter an den Workflow. Dieser Output wird dann im nächsten Task genutzt um für das Getränk zu berechnen, ob der gesamte Preis bezahlt wurde.
Dieses Symbol, der Diamant, ist ein **Gateway**. Er kann entweder ein X enthalten oder ein +.
Enthält er ein X wie hier, ist es ein *Exklusiver Gateway*. Das bedeutet, nur einer der ausgehenden Pfade wird ausgeführt. Nach dem Einwurf einer neuen Münze wurde berechnet, ob der nutzer genug eingeworfen hat. Ist das der Fall, wird der Ablauf weiter geführt. Wenn nicht, wird der *Default* Weg gewählt. Den erkennst du an dem Strich am Pfeil. Das heißt hier wird so lange eine Schleife durchgeführt, bis genug Geld eingeworfen wurde.
Enthält der Gateway ein + ist er ein *Parallel Gateway*. Das bedeutet, dass alle ausgehenden Pfade parallel abgearbeitet werden. Später werden diese parallelen Wege wieder zusammengeführt in einem weiteren Gateway. Dieser blockiert so lange, bis alle eingehenden Pfade abgearbeitet wurden. Danach wird der Workflow weiter ausgeführt.

Figure A.1.: Introduction to the user study (page 1 of 3)

Erreicht der Workflow den **Endpunkt**, das ist der Kreis mit dem dicken Rand, ist der Workflow beendet. Ein Workflow kann verschiedene Endpunkte besitzen, sollte zum Beispiel durch so einen Gateway der Ablauf geteilt werden und diese nicht mehr zusammen geführt werden.
Dieser Workflow könnte so in einer Workflow Engine eingelesen und automatisch gestartet werden. Es können auch mehrer der Abläufe parallel ausgeführt werden.

Du wirst gleich 3 Aufgaben erhalten, die jeweils auf einem Workflow basieren, der in einer Workflow Engine eingelesen und ausgeführt werden kann. Ich habe dir dafür jeweils ein Kommando geschreiben. Dieses liest den Workflow in die Engine ein und wenn das erfolgreich war, wird eine Testinstanz ausgeführt. Du kannst dir das Ergebnis davon im Browser Fenster jeweils ansehen.
Dieses Beispiel habe ich schon für dich eingelesen und einmal ausgeführt. Du siehst hier an den Zahlen, dass dieses Stück der Software bereits 1 mal ausgeführt wurde. Die Zahl an den Elementen bedeutet, dass der Workflow dieses Stück so oft schon hier angelangt ist und durchgeführt wurde. Ist die linke Zahl höher als 0, stehen derzeit so viele ausgeführte Instanzen an dieser Stelle und es wird ggf gewartet oder es ist ein Fehler aufgetreten.
Jeder dieser Kästen, also Tasks hat Daten die er als Input benötigt und Daten, die er ausgeben kann als Output. Die Workflow Engine gibt diese Daten an alle Elemente weiter und erhält sie, falls sie an einer Stelle hinzugefügt werden als Output. Diese Daten werden uns unserem Fall auch als Bedingung ausgewertet. Sollte der Datensatz an dieser Stelle nicht vorhanden sein, blockiert die Workflow Engine und es kann nicht weiter ausgeführt werden.

Fragen

1. **Bewerte, wie technisch versiert du bist.**
   gar nicht □—□—□—□—□ professionell
2. **Wie gut kennst du BPMN?**
   gar nicht □—□—□—□—□ professionell
3. **Kennst du dich mit JSON und oder XML aus?**
   □ JSON □ XML
4. **Hast du schon mal programmiert?**
   □ Ja, im letzten Jahr □ Ja, vor über einem Jahr □ Nein

Einführung in die Validierung

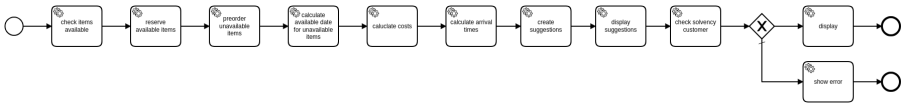
In der Arbeit geht es darum, sicherzustellen, dass in ein Workflow der erstellt wurde valide ist und dass nach Änderungen an der Software hinter den Service Tasks der Workflow noch durchlaufen kann.
Wie vorher erklärt benötigt jeder Task spezielle Input Daten und kann dazu Output Daten generieren. Das kann man mit BPMN darstellen. Hier kann ein Datensatz als Input und ein Datensatz als Output für einen Servicetask definiert werden. Um verschiedene Kombinationen aus Input und Output zu erlauben und um diese Information nur auf der technischen Seite zu belassen, wurden Contract erzeugt. Jeder dieser Service Tasks definiert seine benötigten Daten selber. Du kannst dir immer die Contracts zu jedem Service Task ansehen. Im oberen Teil des Contracts steht, mit welchen Daten überhaupt gearbeitet wird. Im unteren Teil wird aufgeführt, welche Request Response Paare es gibt. Das heißt, zu welchem Input welcher Output generiert wird. Du kannst davon ausgehen, dass sollten zusätzliche Daten im Input vorhanden sein, dass diese einfach ignoriert werden. Wird ein Service Task aufgerufen, erwartet er, dass die Daten die er erhält zu einem dieser Requests passt und erzeugt eine Response die dann entsprechend aussieht. Jeder Input wird weiter gereicht.
Du musst in dieser Arbeit nie den Contract erstellen, das wurde pro Service Task bereits gemacht. Du könntest ihn dir aber immer ansehen. Diese Contracts werden in der Validierung genutzt.
Die Validierung nimmt diese Contracts und guckt, ob bei einem Ablauf alle Daten vorhanden sein werden. Das heißt es wird berechnet, ob immer alle Inputdaten vorhanden sind, welche Felder genutzt werden und wo nicht genug Daten vorhanden sind. Es wird auch getestet, ob auf Grund der Daten bestimmte Pfade nie eingeschlagen werden.
In manchen Aufgaben wirst du diese Validierung nutzen dürfen. Ich gebe dir jeweils das Kommando dazu, sodass du sie ausführen kannst. Du bekommst dann einen graphischen und einen textuellen Output. Ich zeige dir, wo du diesen einsehen kannst. Bei der graphischen Ausgabe werden die Elemente, die einen Fehler enthalten rot angezeigt, gelb bei Warnungen und ansonsten weiß. Sollte alles gut sein, sieht das Ergebnis so aus, wie der Workflow selber. In der textuellen Ausgabe siehst du eine Liste der Elemente, die keine Fehler generieren und

Figure A.2.: Introduction to the user study (page 2 of 3)

eine Liste der fehlerhaften Elemente mit einer Liste der Ein- und Ausgeben dieses Elements und einen Fehlertext.

Figure A.3.: Introduction to the user study (page 3 of 3)

Aufgabe 1: Parallele Workflows



Hier siehst du den Businessprozess eines Online Kaufs als Workflow abgebildet.
Derzeit ist der Workflow sequentiell. Das heißt alle einzelnen Aufgaben des Systems werden nacheinander ausgeführt. So ein Kauf Prozess soll natürlich möglichst schnell passieren. Das bedeutet, wenn Abläufe parallel ausgeführt werden können, wird der gesamte Prozess schneller. Deine Aufgabe ist, den Workflow so weit neu zu ordnen, dass möglichst viel parallel ausgeführt wird. Die Ordnung momentan garantiert, dass jeder Task den Input erhält, den er braucht. Wenn jetzt ein Task vor einem anderen ist, bedeutet das aber nicht, dass das zwangsläufig so sein muss.
Du kannst dir alle Contracts einsehen. Wenn du etwas änderst kannst du den Workflow in die Workflow Engine einlesen und eine Testinstanz laufen lassen. Gruppe 1: Die Validierung kannst du auch nutzen.

Fragen

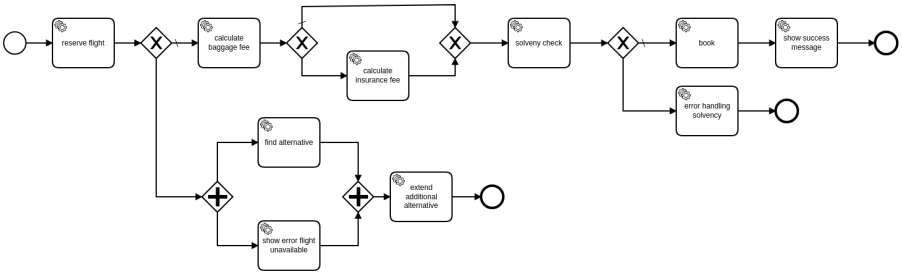
1. **Wie schwer war die Aufgabe?**
   einfach □—□—□—□—□ sehr schwer
2. **Wie schwierig war die Änderung des Workflows im Modeller?**
   einfach □—□—□—□—□ schwer
3. **Wie verständlich war die Validierung (falls genutzt)**
   einfach □—□—□—□—□ schwer
4. **Was war hilfreich bei der Bearbeitung der Aufgabe?**

   _____
   _____
   _____
   _____
5. **Wo lagen Probleme bei der Bearbeitung der Aufgabe?**

   _____
   _____
   _____
   _____

Figure A.4.: Task 1: Description and questionnaire

Aufgabe 2: Fehlersuche



Hier siehst du den Workflow eines online Buchunssystems. Der Workflow könnte dem ähnlich sein, den du durch deine Arbeit kennst, muss er aber nicht.
Stell dir vor, dieser Ablauf wurde geändert. Services, die die Tasks ausführen wurden verändert und Teile des Workflows wurden umgestellt. Nun kommt es zu Fehlern und du sollst diese identifizieren.
Um das zu bewerkstelligen, kannst du dir den Workflow im Modeller ansehen, die Contracts einsehen und die Ausgabe einer Testinstanz anzeigen lassen. Gruppe 1: Du kannst dir die Validierungsergebnisse ansehen.
Versuch mir alle Fehler zu nennen, die du erkennen kannst und sag mir, wie sie behoben werden könnten. Sag mir, wenn du denkst, es gibt keine weiteren Fehler. Du hast höchstens 15 Minuten für die gesamte Aufgabe.
Gruppe 1: Hier ist das Ergebnis der Validierung. Denkst du, sie gibt noch weitere Fehler, als die von dir gefundenen an?

Fragen

1. **Wie schwer war die Aufgabe?**
   einfach □—□—□—□—□ schwer
2. **Wie schwierig war es, die Validierung zu verstehen?**
   einfach □—□—□—□—□ schwer
3. **Was war hilfreich zum Bearbeiten der Aufgabe?**

   _____
   _____
   _____
   _____
4. **Wo hattest du Probleme beim Bearbeiten der Aufgabe?**

   _____
   _____
   _____
   _____

Figure A.5.: Task 2: Description and questionnaire

Aufgabe 3: Erzeugung eines Workflows

Stell dir vor, du müsstest den Buchungsablauf für einen Kunden zusammenbauen. Und zwar den Teil von Flug reservieren bis der User bestätigen muss, dass er wirklich buchen will. Dem User sollen verschiedene Angebote gemacht werden und es sollen schon ein paar Tests für die Buchung durchgeführt werden. Teile sind wie im vorherigen Diagram, aber es wird keine exclusive gateways (Symbol X) geben und du musst keinen Fehlerfall betrachten. Von der Software Abteilung hast du die folgenden 10 Services bereit gestellt bekommen. Sie sind alle nötig um den Ablauf darzustellen, den der Kunde sich wünscht. Das heißt, sie müssen alle immer ausgeführt werden. Wie in der ersten Aufgabe, soll der Ablauf so schnell wie möglich sein, das heißt es sollte möglichst viel parallelisiert sein.
Auch hier gilt wieder, der Ablauf könnte so aussehen, wie die, die du kennst und die gleichen Bedingungen haben, das muss aber nicht so sein!
Du kannst wieder einen Test starten. Dieser kann allerdings nur laufen, wenn der Workflow vollständig ist und ohne Fehler. Hier kannst du die Validierung starten. Sie kann auch schon genutzt werden, wenn der Workflow noch nicht vollständig ist.

Fragen

1. **Wie schwer war die Aufgabe?**
   einfach □—□—□—□—□ sehr schwer
2. **Wie schwierig war die Änderung des Workflows im Modeller?**
   einfach □—□—□—□—□ schwer
3. **Wie verständlich war die Validierung**
   einfach □—□—□—□—□ schwer
4. **Was war hilfreich zum Bearbeiten der Aufgabe?**

   _____
   _____
   _____
   _____
5. **Wo hattest du Probleme beim Bearbeiten der Aufgabe?**

   _____
   _____
   _____
   _____

Figure A.6.: Task 3: Description and questionnaire

Abschließende Fragen

Fragen

1. **Hast du verstanden was BPMN ist und wie man es nutzt?**
   □ Ja □ Nein □ Weiß nicht
2. **Denkst du Workflows können von Menschen ohne technisches Hintergrundwissen erstellt werden ohne zusätzliche Hilfsmittel wie die Validierung?**
   □ Ja □ Nein □ Weiß nicht
3. **Wie hilfreich war der Output der Workflow Engine im Browser Fenster?**
   nicht hilfreich □—□—□—□—□ hilfreich
4. **Wie hilfreich war die textuelle Validierung?**
   nicht hilfreich □—□—□—□—□ hilfreich
5. **Wie hilfreich war die grafische Validierung?**
   nicht hilfreich □—□—□—□—□ hilfreich
6. **Denkst du Contracts können Softwareentwickler der Services unterstützen?**
   □ Ja □ Nein □ Weiß nicht
7. **Denkst du die Validierung könnte Softwareentwickler der Services unterstützen?**
   □ Ja □ Nein □ Weiß nicht
8. **Könnten Mitarbeiter außerhalb der technischen Abteilung Workflows erstellen auf Basis Tasks der Software Entwicklung (wie in Aufgabe 3)?**
   auf keinen Fall □—□—□—□—□ auf jeden Fall
9. **Wie könnte die Validierung verbessert werden?**

Figure A.7.: Final questionnaire

A.2. Evaluation Results

Participant	1	2	3	4	5	6
XML	X	X	X	X	X	X
JSON	X	X		X	X	X

Table A.1.: Knowledge of XML and JSON

Participant	1	2	3	4	5	6
Yes	X	X	X	X	X	X
No						
I don't know						

Table A.2.: Did the participant understand BPMN?

Participant	1	2	3	4	5	6
Yes				X	X	
No	X	X	X			X
I don't know						

Table A.3.: Did the participant think workflows can be created by people without technical knowledge without further aids like the validation?

Participant	1	2	3	4	5	6
Yes		X	X	X	X	X
No						
I don't know	X					

Table A.4.: Did the participant think the contracts can support software developers?

Participant	1	2	3	4	5	6
Yes		X	X	X	X	X
No						
I don't know	X					

Table A.5.: Did the participant think the validation can support software developers?

A.2.1. Textual Answers and Notes

Task 1

Was war hilfreich bei der Bearbeitung der Aufgabe?

- **Participant 1** Die visuelle Validierung
- **Participant 2** Die Contracts helfen zu erkennen, was die Taks zur Verfügung stellen und was sie benötigen
- **Participant 3** Contracts
- **Participant 4** Gute Aufgabenstellung
- **Participant 5** Meine Vorkenntnisse in BPMN und dem Modeler
- **Participant 6** Contracts; Selbsterklärende/sprechende Beschreibungen für Services; Einführung und recht intuitive UI

Wo lagen Probleme bei der Bearbeitung der Aufgabe?

- **Participant 1** -
- **Participant 2** -
- **Participant 3** Grundverständnis welche Info steht wo.
- **Participant 4** -
- **Participant 5** Zu wenig Bildschirme
- **Participant 6** -

Notes

- **Participant 1** 3 mal validiert, Fehler immer direkt gefunden. Änderungen im grafischen
- **Participant 2** sieht sich alle Contracts an und verschiebt erst dann entsprechend. wenig parallelisiert. Validierung ist korrekt, keine weitere Parallelisierung.
- **Participant 3** liest zuerst alle Items durch und fragt bei 3 die Bedeutung. Gruppiert ohne contracts. Sieht Contracts an und ist erst etwas verwirrt. Erzeugt dann alles und nutzt Monitor Output: Alles in Ordnung
- **Participant 4** versteht schnell die Aufgabe, guckt in Contracts kurz nach, was benötigt wird und macht alles entsprechend. Einmal starten der Testinstanzen: alles richtig

- **Participant 5** guckt kurz Contracts an. Macht alles fertig, will nicht testen, sondern denkt lieber nach. Erkennt "alles richtig", sagt richtig wo es noch paralleler ginge, aber Zeit ist um

- **Participant 6** sehr schnell richtige Lösung gefunden, wenig Contracts benötigt, direkt richtig und durch Tests bestätigt.

Task 2

Was war hilfreich bei der Bearbeitung der Aufgabe?

- **Participant 1** Die visuelle Darstellung

- **Participant 2** Visuelle Darstellung der Validierung sowie Contracts

- **Participant 3** Farbkodierung der Fehler. Contracts

- **Participant 4** farbliche Kodierung. Das textuelle war hilfreich, aber schwieriger zu lesen, als das graphische

- **Participant 5** siehe Aufgabe 1

- **Participant 6** Contracts, Visualisierung mit Farbcodes; Text-Output, wird hilfreicher, je mehr man sich damit beschäftigt

Wo lagen Probleme bei der Bearbeitung der Aufgabe?

- **Participant 1** Nichts verstaden bei dem Text. Bei der visuellen Darstellung den Ort des Fehlers gesehen, keine Idee diesen zu beheben.

- **Participant 2** Fehler übersicht ist teilweise etas unübersichtlich

- **Participant 3** Langsam auf Grund von "neuer" Methode

- **Participant 4** "calculate insurance fee" war nicht farblich markiert, obwohl das Programm eine mögliche Fehlerquelle entdeckt hat.

- **Participant 5** Anzeige ungleich Text (Validierung)

- **Participant 6** "not used" grafisch darstellen, war nicht so klar im Text-Outpput zu verstehen. "Mouse Over" Kontextinfo aus Textdatei in grafischer Darstellung wäre hilfreich.

Notes

- **Participant 1** ohne Validierung: erkennt grün im Monitor zeigt Fehler, keine Ahnung, woher Fehler. Keine Ahnung woher Fehler kommen könnte.
  mit Validierung: Text nicht angesehen, Zitat "Der bringt mir nichts". Sieht Fehler, aber "Orange ist kein Fehler". Ohne Text nur erkannt, wo es Fehler gibt, nicht welche genau.
- **Participant 2** Ohne Validierung: erkennt Fehler "price range" direkt und schnell Begründung durch Contracts.
  Mit Validierung: erkennt direkt weitern Fehler und durch contracts erkannt, dass es falsch geschrieben ist. Dritten Fehler nicht gesehen.
- **Participant 3** Mit Valdierung: klickt im grafischen und sucht erst da weitere Informationen. Sieht Fehler bei "find alternative" denkt aber Parallelisierung ist Problem (Konzept parallel erst nicht ganz klar).Ist sehr auf das rote fokusiert "gelb ist nur Warnung". Textueller Input hilft wenig. Contract hilft: findet falsche Schreibweise. 3. Fehler nicht gefunden.
- **Participant 4** erkennt direkt Fehlerorte und erkennt, dass orange ggf Folgefehler sind. Fehler "find alternative" und falsche Schreibweise sehr schnell gefunden durch Text und Contracts. Ungenutztes Feld dauert etwas länger bis dieser Teil gefunden wird.
- **Participant 5** Ohne: Fehler direkt erkannt, durch Contracts direkt Fehler benannt und Lösung korrekt gesagt. Erkennt außerdem direkt, dass Pfad hinten nicht abgelaufen wird und erkennt Fehler schnell durch Contracts.
  Mit: sieht sich nicht den Text richtig an, erkennt dadurch nicht den 3. Fehler
- **Participant 6** direkt textuellen Output durchgegangen und Fehler direkt erkannt und verstanden, Folgefehler richtig erkannt.
  Falsche Schreibweise direkt erkannt
  Ungenutztes Feld: Zunächst verwirrt von Beschreibung, aber Fehler richtig erkannt.

Task 3

Was war hilfreich bei der Bearbeitung der Aufgabe?

- **Participant 1** visuelle Darstellung, Contracts
- **Participant 2** Visualisierung + Contracts
- **Participant 3** Contracts
- **Participant 4** Contracts
- **Participant 5** siehe 2
- **Participant 6** Contracts

Wo lagen Probleme bei der Bearbeitung der Aufgabe?

- **Participant 1** mit der Bezeichnung der Service Tasks. Vor allem durch Vorwissen, da nicht klar war welche Kosten bspw. gemeint sind.
- **Participant 2** Die Menge der Tasks macht es schwer zu überblicken, welche Taks die Requirements anderer Tasks erfüllen.
- **Participant 3** Es dauert, bis die Business-Funktionen der Tasks klar sind.
- **Participant 4** Übersichtlichkeit für Input/Output hat im Modeller gefehlt. *Anmerkung:* Contractdaten im Grafischen wünschenswert
- **Participant 5** -
- **Participant 6** Service-Namen/Semantic bei Workflows nicht ausreichend, um Abhängigkeiten zu erkennen; Anmerkungen von Frage 3.

Notes

- **Participant 1** zieht alle Tasks an Stelle und testet. Findet direkt Fehler und korrigiert, versteht die Semantik nicht, warum ein Task etwas braucht, aber versteht durch die Validierung, dass es so ist und wie es geändert weren muss. 3 mal validiert.
- **Participant 2** Sehr lange ansehen Contracts. Dann alles geordnet. 3 mal Validierung genutzt. Jeweils schnell den Fehler gefunden und direkt behoben und nächste Validierungsrunde.
- **Participant 3** arbeitet intuitiv richtig. sieht 3 Contracts an und nutzt diese. Führt eine Validierung aus und findet direkt den Fehler durch Contracts und grafisches Output und korrigiert diesen.
- **Participant 4** Validierung nicht genutzt, alles richtig durch Contracts und Intuition
- **Participant 5** startet direkt ohne Contracts. Sortiert falsch wegen Vorkenntnissen im Buchungsablauf. Setzt fast alles zusammen und guckt dann erst in Contracts. Dann getestet. Fehler direkt in Validierung gefunden und korrigiert
- **Participant 6** erst grob sortiert mit Contracts. Durch Fehler, dass reserve erst ans Ende gehört viel falsch sortiert. Hinweiß erhalten: "reserve" ansehen. Direkt verstanden und umsortiert. Keine Validierung oder Tests genutzt zwischendurch. Fehler direkt verstanden und erklärt, was noch geändert werden müsste.

Final Questions

Wie könnte die Validierung verbessert werden?

- **Participant 1** Man sollte direkt im grafischen Output Änderungen vornehmen können bzw diesen in en Modeler einbinden/anwenden.
  Service Task mit kurzer Textbeschreibung versehen
- **Participant 2** Wenn man visualisieren könnte, welche Taks zu welchen passen, würde das bei komplexen Workflows sehr helfen. *Anmerkung:* anklicken und alle Tasks die passen würden anzeigen
- **Participant 3** Grafische Validierung sollte Detail-Infos zu Fehlern bereitstellen
- **Participant 4** Input/Output in den Modeler. Alle Infos der textuellen Validierung in die grafische einfügen
- **Participant 5** siehe Aufgabe 2; Analyse zur Laufzeit (während rumschubsen im Modeler); Anzeige der Contracts im Modeller
- **Participant 6** Intellisense UI: Textinfos besser verknüpfen; Recommendation auf Basis der Cotracts für mögliche Verbindungen der Services

Bibliography

[Ado] *ADONIS*. https://uk.boc-group.com/adonis/. Accessed: 2019-03-31 (cited on page 2).

[Bpma] *BPMN 1.0*. https://www.omg.org/spec/BPMN/1.0/. Accessed: 2019-05-07 (cited on pages 5, 81).

[Bpmb] *BPMN 2.0*. https://www.omg.org/spec/BPMN/2.0/. Accessed: 2019-05-07 (cited on pages 5–9, 11, 20, 81).

[Cam] *camunda xml model*. https://github.com/camunda/camunda-xml-model. Accessed: 2019-05-07 (cited on pages 25, 26).

[DDO08] R. M. Dijkman, M. Dumas, and C. Ouyang. "Semantics and analysis of business process models in BPMN". In: *Information and Software Technology* 50.12 (2008), pp. 1281 –1294. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2008.02.006. URL: http://www.sciencedirect.com/science/article/pii/S0950584908000323 (cited on pages 2, 3).

[FF06] M. Fowler and M. Foemmel. "Continuous integration". In: *Thought-Works) http://www. thoughtworks. com/Continuous Integration. pdf* 122 (2006), p. 14 (cited on page 13).

[Fre] T. Freytag. *WoPeD–Workflow Petri Net Designer*. https://woped.dhbw-karlsruhe.de/. Accessed: 2019-03-31 (cited on page 2).

[Gso] *Gson*. https://github.com/google/gson. Accessed: 2019-07-09 (cited on pages 30, 39).

[JL14] M. F. James Lewis. *Microservices*. https://martinfowler.com/articles/microservices.html. Accessed: 2019-07-22. 2014 (cited on page 12).

[Jso] *JSON Schema*. https://json-schema.org/. Accessed: 2019-03-12 (cited on page 26).

[Pou+19] S. Pourmirza et al. "BPMS-RA: A Novel Reference Architecture for Business Process Management Systems". In: *ACM Trans. Internet Technol.* 19.1 (Feb. 2019), 13:1–13:23. ISSN: 1533-5399. DOI: 10.1145/3232677. URL: http://doi.acm.org/10.1145/3232677 (cited on page 2).

[Rei+11] H. A. Reijers et al. "Syntax Highlighting in Business Process Models". In: *Decis. Support Syst.* 51.3 (June 2011), pp. 339–349. ISSN: 0167-9236. DOI: 10.1016/j.dss.2010.12.013. URL: http://dx.doi.org/10.1016/j.dss.2010.12.013 (cited on page 2).

[Sad+04] S. Sadiq et al. “Data Flow and Validation in Workflow Modelling”. In: *Proceedings of the 15th Australasian Database Conference - Volume 27*. ADC ’04. Dunedin, New Zealand: Australian Computer Society, Inc., 2004, pp. 207–214. URL: http://dl.acm.org/citation.cfm?id=1012294.1012317 (cited on page 2).

[Spr] *Building a RESTful Web Service with Spring Boot Actuator*. https://spring.io/guides/gs/actuator-service/. Accessed: 2019-07-09 (cited on page 42).

[Sta+14] S. von Stackelberg et al. “Detecting Data-Flow Errors in BPMN 2.0”. In: *Open Journal of Information Systems (OJIS)* 1.2 (2014), pp. 1–19. ISSN: 2198-9281. URL: http://nbn-resolving.de/urn:nbn:de:101:1-2017052611934 (cited on pages 3, 62).

[TAS09] N. Trčka, W. M. P. van der Aalst, and N. Sidorova. “Data-Flow Anti-patterns: Discovering Data-Flow Errors in Workflows”. In: *Advanced Information Systems Engineering*. Ed. by P. van Eck, J. Gordijn, and R. Wieringa. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 425–439. ISBN: 978-3-642-02144-2 (cited on page 2).

[Whi04] S. A. White. “Introduction to BPMN”. In: (2004) (cited on page 5).

[Win] *Web Data INTEgRation Framework (WInte.r)*. https://github.com/olehmberg/winter. Accessed: 2019-05-20 (cited on pages 37, 81).

[Zeea] *Zeebe*. https://zeebe.io/. Accessed: 2019-07-09 (cited on page 43).

[Zeeb] *zeebe-modeler*. https://github.com/zeebe-io/zeebe-modeler. Accessed: 2019-05-15 (cited on page 6).

[Zeec] *Zeebe Quickstart*. https://docs.zeebe.io/introduction/quickstart.html. Accessed: 2019-07-09 (cited on page 43).

Glossary

BPMN Business Process Model and Notation

BPMN 1.0 standard is the standard of BPMN in version 1.0 from 2007 published by the OMG [Bpma].

BPMN 2.0 standard is the standard of BPMN in version 2.0 from 2011 published by the OMG [Bpmb].

BPMN DI BPMN Diagram Interchange

Camunda is a company mostly responsible for creating the workflow engine Camunda BPM.

CI Continuous Integration

CobaFlow COntracted BAsed data FLOW validation

design by contract is a concept for software development based on contracts defining interfaces.

jBPM is a jBoss workflow management system.

JSON JavaScript Object Notation

OMG Object Management Group

PNML Petri Net Markup Language

WInter.r is a mapping algorithm designed at the University of Mannheim [Win]

WSDL Web Services Description Language

Zeebe is an open source workflow engine initiated by Camunda.

Zeebe modeler is an open source BPMN modeler created for the workflow engine Zeebe. It was used to create all workflows in this thesis.

Zeebe simple monitor is an open source monitor for the workflow engine Zeebe.