Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Resilient Continuous Delivery Pipelines Based on BPMN

Oliver Kabierschke

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	M. Sc. Thomas F. Düllmann
Commenced:	October 2, 2018
Completed:	April 2, 2019

Abstract

The “Continuous Delivery” (CD) approach promises to significantly reduce the time-to-market of new features in a software development project, enabling faster feedback for the developers and in effect higher customer satisfaction. The key of implementing CD is automating the delivery process, which involves building, testing, and deploying the application to the target platform. With an increasing application size, the delivery process becomes more complex and is likely to fail at various points. A failure can interrupt the development process and should consequently be prevented whenever possible. The “Business Process Model and Notation” (BPMN) specifies approaches to introduce reliability increasing structures in a business process, which prevent failures or compensate their effects during process execution. The goal of this master’s thesis is to support the design of resilient CD processes by bringing concepts from BPMN to the CD domain. A graphical representation of a CD pipeline can make complex processes more manageable by abstracting relevant structures. Particularly the compensation concept in BPMN is a promising way of abstracting complicated parts of a CD pipeline. In order to create compatibility between BPMN and the CD domain, the “Domain Specific Language” (DSL) StalkCD has been invented. It stores the CD process of a Jenkinsfile and combines it with information that is relevant when generating a BPMN model. Transformations from Jenkinsfiles via StalkCD to BPMN and back have been developed, enabling to depict Jenkinsfiles using BPMN. Evaluation results show that the support of the provided implementations for the declarative syntax of Jenkinsfiles is good. Jenkinsfiles that have been translated to BPMN models can be visualized and edited comprehensively, however, the graphical user support is still improvable. Because of the lack of sufficiently complex real-world-examples, this thesis could not prove that there actually result resilience benefits from using BPMN as abstraction layer when designing CD pipelines.

Kurzfassung

Das Konzept „Continuous Delivery“ (CD) verspricht signifikant kürzere Entwicklungszeiten neuer Funktionen in einem Software-Entwicklungsprojekt. Dies ermöglicht schnellere Rückmeldungen an Entwickler und letztlich auch höhere Kundenzufriedenheit. Der Schlüssel zur Umsetzung von CD ist die Automatisierung des Auslieferungsprozesses, welcher aus dem Bauen, Testen und Installieren der Anwendung besteht. Mit zunehmender Anwendungsgröße steigt auch die Komplexität des Auslieferungsprozesses und es entstehen an vielen Stellen Fehlerquellen. Da ein Fehlschlag der automatischen Auslieferung den Entwicklungsfluss unterbricht, ist es sinnvoll, Fehler zu verhindern, wo immer es möglich ist. „Business Process Model and Notation“ (BPMN) enthält Konzepte, um Zuverlässigkeits-Konstrukte in Prozesse zu integrieren, welche Fehler verhindern oder deren Folgen während der Prozessausführung kompensieren. Das Ziel dieser Masterarbeit ist die Unterstützung der Entwicklung von resilienten Auslieferungsprozessen durch die Verwendung von BPMN-Konzepten in der CD-Domäne. Solche Prozesse sollen in der Lage sein, sich ohne menschliches Zutun von Fehlern zu erholen. Eine grafische Repräsentation einer CD-Pipeline kann komplexe Prozesse handhabbarer machen, indem sie relevante Strukturen abstrahiert. Besonders das „Compensation“-Konzept in BPMN ist ein vielversprechender Weg, um komplizierte Teile eines Auslieferungsprozesses zu abstrahieren. Um Kompatibilität zwischen BPMN und der CD-Domäne herzustellen, wurde die „Domänen-Spezifische Sprache“ (DSL) StalkCD entwickelt. Sie speichert den CD-Prozess einer Jenkinsfile und zusätzlich Informationen, die zur Visualisierung eines daraus generierten BPMN-Modells relevant sind. Es wurden Transformationen von Jenkinsfiles über StalkCD zu BPMN und umgekehrt entwickelt, welche es ermöglichen, eine Jenkinsfile in BPMN darzustellen. Evaluierungen haben gezeigt, dass die entwickelte Implementierung eine gute Abdeckung von Jenkinsfile-Sprachelementen aufweist. In BPMN überführte Jenkinsfiles lassen sich außerdem vollständig darstellen und bearbeiten, jedoch ist die grafische Benutzerunterstützung noch ausbaufähig. In dieser Arbeit konnte nicht bewiesen werden, dass der Einsatz von BPMN als Abstraktionsschicht beim Entwickeln von Auslieferungsprozessen diesen tatsächlich zu höherer Resilienz verhilft, da entsprechend komplexe praktische Beispiele fehlten.

Acknowledgment

This work would not have been possible without the great people around me, providing support and inspiration during the thesis and my study. I feel a deep sense of gratitude:

- to Dr.-Ing. André van Hoorn for giving me the opportunity to do this master's thesis at the Reliable Software Systems Group. I very much appreciate his invaluable and kind support at any time during the thesis.
- to M. Sc. Thomas Düllmann for his detailed and constructive feedback and guidance, always helping me to stay focused on the objectives.
- to all researchers at the RSS and IAAS who provided me with know-how, for their friendly and productive support.
- to my friend Daniel Vietz for countless inspiring discussions and for enjoyable work breaks at the university.
- to my friend Fabian Keller for proofreading and for motivating me always to seek for the best solution.
- to my wife Carolin for her love, encouragement, and patience, giving me strength and confidence.
- to my parents Jutta and Rainer who made my study possible and always supported me.
- to my friends and family, who are always there when I need them.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

ANTLR “ANother Tool for Language Recognition”. 20

BPM “Business Process Management”. 11

BPMN “Business Process Model and Notation”. iii, 2, 12

CD “Continuous Delivery”. iii, 1, 8

CI “Continuous Integration”. 1, 8

DevOps “Development and Operations”. 1, 7

DSL “Domain Specific Language”. iii, 3

IaC “Infrastructure as Code”. 9, 41

JVM “Java Virtual Machine. 57

OMG “Object Management Group”. 12

PDL “CD Pipeline Definition Language”. 2

UML “Unified Modeling Language”. 12

VCS “Version Control System”. 9

XML “Extensible Markup Language”. 18

XPDL “XML Process Definition Language”. 12

1 Introduction

As an introduction to the topic of this master's thesis, this chapter defines the research context and states the problems it is addressing. Furthermore, it outlines the chosen solution approach and the used methodology for evaluating it. The chapter will be concluded by an outline of the document structure.

1.1 Research Context & Problem Statement

Each software development project eventually reaches the point in time, where the product is to be delivered to the customer. The day of going live is often perceived as the day of truth, where all mistakes of the involved developers from the past days, weeks or months come to light. When searching for the causes of problems, mutual accusations between teams are not uncommon, according to Humble and Farley [HF10]. They point out, that especially the development and operations teams often have conflicting interests and lack communication.

The "Development and Operations" (DevOps) movement [EGHS16; Hüt12; SNP15] aims to bring together these two parties by establishing the requirements for an effective collaboration between them. A critical prerequisite to reaching the goal of aligning the interests of development and operations is automation [Hüt12]. It enables to reliably repeat and measure tasks, which has numerous benefits for the software development process: By repeatedly executing and testing the delivery process, the risk of each single delivery is minimized. This allows to release more often, generating feedback for new features more quickly. Furthermore, by measuring and monitoring the delivery and operation of an application, the sources of errors can be detected quickly and precisely, thus the responsibilities are clear and conflicts are avoided.

The "Continuous Delivery" (CD) approach is an important part of the DevOps movement. In the past decade, it has increasingly gained importance as it is being widely adopted in industry [HF10; SBZ17]. It can be seen as an extension to the already widely practiced "Continuous Integration" (CI) approach. CD can help to significantly decrease the time to market of new software-features by shortening release cycles from several months or even years down to few weeks or even days [Che15]. Consequently, customers can benefit earlier from the developers' work and provide them with feedback. This allows incomplete or irrelevant requirements to be detected soon so that the development effort can be focused on the relevant tasks. Therefore, more frequent releases improve the product's quality, create higher value, and increase customer satisfaction [Che15].

As applications grow, the complexity of their CD pipelines increases, which makes it challenging to maintain reliable deliveries. The CD process can fail at various points, in some instances for trivial reasons, which requires the valuable work of a developer to inspect and fix the delivery process. Laukkanen et al. [LIL17] conducted a systematic literature review on problems that challenge many organizations when adopting CD. They found that the development flow gets broken by failed builds, preventing developers from doing value-adding activities. Thus, a CD process should be designed to fail as seldom as possible. Ideally, it should be resilient, i. e., recover from failures without manual intervention, wherever possible.

A promising way to reduce the complexity of a CD pipeline and at the same time make it more resilient is to use "Business Process Model and Notation" (BPMN) as an abstraction layer for designing the delivery process. The notation not only offers proven approaches to visualize processes of any kind, but also defines concepts to ensure their operability.

The goal of this master's thesis is to provide an approach to use BPMN for the definition of CD pipelines. Transformations have to be developed, capable of converting a BPMN model into the "CD Pipeline Definition Language" (PDL) of a CD tool. This enables to execute CD pipelines defined in BPMN using existing CD tools. Additionally, in order to increase the practical applicability of the implemented approach, transformations from PDLs to BPMN are developed, enabling to depict existing CD pipelines using BPMN. The following research questions define the main focus of this master's thesis:

RQ 1: What features do CD tools provide to ensure operability of the CD process?
The first research question aims to investigate the current state of existing CD implementations, regarding their features to ensure the operability of CD pipelines. Analyzing existing CD tools for their capabilities of recovering from failures enables comparing the results of the thesis with existing approaches.

RQ 2: What BPMN features can be used to increase the resilience of CD pipelines?
BPMN has concepts that can help designing robust business processes. A goal of this master's thesis is to investigate if these concepts can be used in the domain of continuous delivery. Concrete concepts and implementations should be developed to transform relevant BPMN structures to CD pipelines that can be executed using existing CD tools.

RQ 3: What feature coverage does the transformation from Jenkinsfiles to StalkCD provide?
An important aspect of the transformations from PDLs to BPMN is comprehensiveness. An ideal implementation would be capable of processing any language construct that the concerned CD tool allows. The goal of RQ 3 is to evaluate the practical usefulness of the provided implementation by measuring its feature coverage.

RQ 4: What are the resilience benefits of the chosen approach?
Finally, it is of special interest to investigate if the main motivation of this master's thesis has been solved. The evaluation should show whether the use of the implemented transformations results in actual benefits to the resilience of a CD pipeline.

1.2 Approach & Evaluation Methodology

As part of the contributions of this master's thesis and to answer RQ 1, the CD pipeline syntax of Jenkins as a representative of popular CD tools has been analyzed. The result of this analysis is a meta-model of the Jenkinsfile syntax, which can be found in Chapter 4. Based on this meta-model, the "Domain Specific Language" (DSL) *StalkCD* has been introduced. Its language design can be found in Chapter 5. It is supposed to serve as an intermediate representation of a CD pipeline to create a compatibility layer between the CD domain and BPMN. StalkCD supports all relevant features from the CD domain and additionally extends the CD components by properties that facilitate depicting them in BPMN.

Figure 1.1 shows the main contributions of this master's thesis and their corresponding chapters or sections. The connecting arrows represent transformations, described in detail in Chapter 6. Transformation ① enables to translate the CD pipeline inside a Jenkinsfile to the StalkCD DSL. This makes it possible to apply the approaches of this master's thesis to existing pipeline definitions. Transformation ② brings a StalkCD pipeline back into the Jenkinsfile syntax, which allows to execute the CD process using Jenkins. These two transformations could be exchanged or extended to support other CD tools as well.

Figure 1.1: Overview of the contributions of this master's thesis

A StalkCD pipeline can be depicted in a BPMN model using transformation ③. For details on the mapping of CD features to BPMN elements and how layout information is assigned to the generated BPMN elements, see Section 6.3. The created BPMN model in XML format can be displayed and modified in a BPMN-editor, such as the Camunda Modeler[1]. After having made changes, transformation ④ allows to translate the pipeline back into the StalkCD language, as described in Section 6.4.

For evaluating transformations ① and ② from Jenkinsfiles to StalkCD and back, an empirical study has been conducted, as described in Section 7.2. The results reveal that the coverage of Jenkinsfile features is sufficient to correctly process 70 % of sample Jenkinsfiles, collected from publicly available GitHub repositories. This is a good result when taking into account that numerous sample files contain syntax errors or violate the principle of declarative programming.

[1]The Camunda BPMN / DMN Process Modeler – https://camunda.com/download/modeler/

Transformations ③ and ④ were evaluated using the Camunda Modeler as BPMN tool. Investigations show that all relevant properties of the modeled CD process can be reached using the controls provided by the editor. However, the accessibility still is improvable, since not all relevant data is visible in the BPMN model and has to be searched at a rather hidden place in the properties panel.

In order to evaluate the resilience benefits, the CD process of the Kieker monitoring framework[2] has been investigated for common causes of failure. The results show that almost none of the evaluated failures could have been prevented by introducing compensation structures in the pipeline. Consequently, the developed approach brings no resilience benefits in the case of Kieker.

[2]Kieker – http://kieker-monitoring.net/

1.3 Outline

The remainder of this master's thesis is structured as follows:

Chapter 2 – Foundations introduces the fundamental concepts that are required in order to understand the contents of subsequent chapters. It outlines the main goals of the DevOps movement and presents the continuous delivery approach as part of it. Jenkins is introduced as popular tool, used in connection with CD. In addition, the chapter explains the term resilience and how it can be measured. Also the concept of BPM is introduced, as well as the related notation BPMN. Finally, the parser generator tool ANTLR and the concepts it is based on are explained.

Chapter 3 – Related Work outlines the findings of Willig when bringing BPMN to the CD domain in his master's thesis. Furthermore, two approaches of introducing DSLs for describing CD pipelines are presented.

Chapter 4 – Analysis of the Jenkinsfile Meta-Model conducts a detailed analysis of the textual representation of a CD pipeline for Jenkins, the Jenkinsfile. It defines a meta-model for the Jenkinsfile DSL, containing all relevant language elements, and explains their structure and semantics.

Chapter 5 – StalkCD presents the domain specific language that has been developed based on the Jenkinsfile meta-model. All language elements and their purpose are explained here.

Chapter 6 – Transformations presents the developed transformations from Jenkinsfiles to StalkCD and back, as well as from StalkCD to BPMN and back.

Chapter 7 – Evaluation defines an evaluation approach to find answers to the remaining research questions. It present experiment setups and the implementation strategy, as well as the evaluation results.

Chapter 8 – Conclusions and Future Work recaptures the results of this master's thesis and discusses them. Furthermore, it lists connection points that can be addressed by future research.

2 Foundations

This master's thesis brings together various fields in the domain of software engineering and business processes. The following sections convey basic knowledge on terms, techniques, and tools that are required to understand the remainder of the thesis. Sections 2.1 to 2.3 cover the field of continuous software delivery and Sections 2.4 and 2.5 introduce the basics of business processes. The aspect of resilience, which is aimed to be improved by bringing together both fields is defined in Section 2.6. Besides from conceptual foundations, Section 2.7 provides technical knowledge on parsing structured data, which is important in order to understand later implementation chapters.

2.1 DevOps

Humble and Farley [HF10] state that collaboration between different teams of an organization is one of the main challenges when it comes to delivering software. They found that many projects use a siloed approach, where developing, testing, and deploying software is strictly separated, leading to communication issues and unclear responsibilities. In their opinion, this problem can be solved by following the principles of the "Development and Operations" (DevOps) movement. Everyone should be involved in the delivery process to increase its transparency and clearness.

Hüttermann [Hüt12] defines DevOps as a modern way of bringing together development and operations to assure serving a customer fast and reliably, while delivering high-quality products. The approach involves various aspects, such as improving communication, automating, and measuring the delivery process. Additionally, Ebert et al. [EGHS16] highlight that also monitoring the running application is an important part in the movement. The central goal is to align the incentives of the different teams, collaborating on development, delivery, and operation of software.

One key requirement to successfully move to DevOps is automation [EGHS16]. It is vital to make deliveries fast and reliable, as manual steps often are time-consuming and error-prone. This includes finding the right tools for building, deploying, and operating an application.

2.2 Continuous Delivery

In the past decade, the approach named "Continuous Delivery" (CD) has increasingly gained attention as it is being widely adopted in industry [HF10; SBZ17]. It can be seen as element of the DevOps movement and extension to the already widely practiced "Continuous Integration" (CI) approach . The frequent integration of software in development aims to reduce the risk of incompatibilities at the time of its release [FF06]. Usually, each developer integrates his work at least once per day, which triggers an automated test. If it fails, the developer can quickly fix compatibility issues, if needed in collaboration with other team members.

CD promises to significantly decrease the time to market of new software-features by shortening release cycles from several months or even years down to few weeks or even days [Che15]. Consequently, customers can benefit earlier from the developers' work and provide them with feedback. This allows incomplete or irrelevant requirements to be detected soon so that the development effort can be focused on the relevant tasks. Therefore, more frequent releases improve the product's quality, create higher value, and in consequence increase customer satisfaction [Che15].

2.2.1 The deployment pipeline

The central element of the CD approach is the CD pipeline. According to Humble and Farley [HF10, p. 103 ff.], it consists of stages, transforming source code into a deliverable product and finally deploy it to a production environment. In general, there are four stages, as depicted in Figure 2.1. Depending on the individual requirements of a software development process, the actual number of stages in a deployment pipeline and their order can vary.

The *commit stage* assures that the code has no obvious defects by running unit tests and calculating code metrics. The fully automated *acceptance test stage* covers the functional and non-functional level of testing, making sure that all user requirements are fulfilled. The manually performed *user acceptance testing stage* assures that the software is usable and valuable. During the manual tests, defects not found by automated tests can be detected. The *capacity testing stage* is generally independent of the *user acceptance testing stage* and can be executed in parallel. Here, the software is assessed according to non-functional requirements, e. g., performance and memory usage.

Only if the software has passed all previous stages, it is automatically released. It has to be distinguished between continuous delivery and continuous deployment [SBZ17]. In case of continuous delivery, it is ensured that the software that passed the testing stages is at a production-ready state by installing it to a production-like environment. Continuous deployment goes one step further and deploys the software to the production or customer environment. Which of both approaches is more relevant to an actual software development process depends on the type of software being developed. If there will be many installations that are not directly accessible, the continuous delivery approach can be preferable. If there is only one installation on a server, automating the deployment process makes sense.

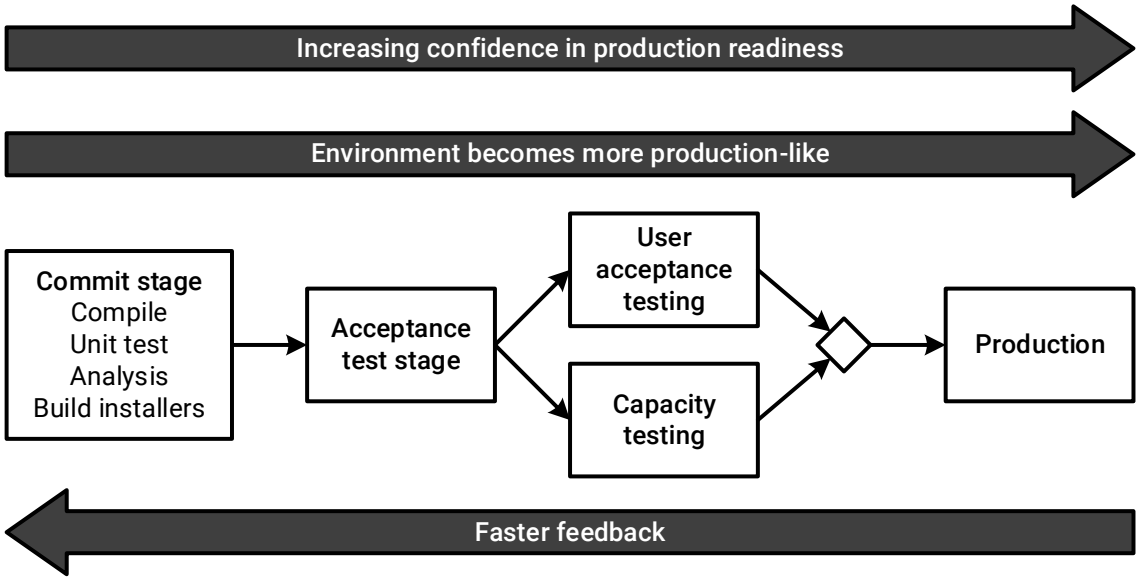


Figure 2.1: General CD pipeline [HF10]

2.2.2 Infrastructure as Code

An important prerequisite of CD is a high degree of automation. To achieve this goal, it is common that everything required to execute an application in a testing or production environment exist as code, also referred to as *"Infrastructure as Code" (IaC)* [Hüt12]. This includes a list of required software packages that are part of the operating system together with their configuration files, technical users, network settings and so on. One benefit of having all the infrastructure as code is the possibility to set up a new production environment from scratch without having to manually configure the system.

Modern CD tools apply this concept by relying on a configuration file located in the "Version Control System" (VCS). This also makes it possible to determine time and author of any change to the specification of the production environment and to introduce reviews and approvals. This is important in order to prevent common mistakes, e. g., random package upgrades, which would create an undefined, error-prone state of the system [HF10, p. 283 ff.].

2.3 Jenkins

Many existing software tools support the development of CD pipelines. In their master's thesis, Bergsteinsdóttir and Edholm [BE18] conducted a case study to identify popular tool chains used in the industry for implementing CD. They found that Jenkins[1] is considered to be the de facto industry standard when it comes to continuous integration and delivery.

[1] Jenkins CI – https://jenkins.io

The authors of Jenkins claim that their product is “the leading open-source automation server”. Indeed, it is often mentioned in literature and referred to as de facto standard for continuous integration and delivery tools [BE18; Beh12]. The application written in Java is especially popular because of over a thousand plug-ins making it highly flexible to fit in almost every use case. Some of its benefits are an easy installation and intuitive usage, extensibility through a REST interface, compatibility to virtually any programming language, the possibility to distribute it, its highly active and professional community and that it is open-source and free of charge [Beh12].

When Jenkins is freshly installed, it only supports a basic set of features like pulling source code from a repository, building it, visualizing build results and sending notifications to developers on build events. These features can be used to continuously test and integrate a software project, but are not sufficient to implement continuous delivery. The *Pipeline* plug-in (ID: workflow-aggregator) adds features to Jenkins allowing to automate the delivery process of an application. At the time of this thesis, the most recent version of the plug-in is 2.6. All following explanations refer to this version.

The plug-in’s central concept is the *Jenkinsfile*, a text file that defines a CD pipeline using a DSL derived from the Groovy programming language[2]. There are two approaches of describing CD pipelines[3]: It can be written using the *scripted* or *declarative* syntax.

A *scripted* pipeline is initiated by the `node` keyword followed by curly braces defining an environment, as can seen in Listing 2.1. Inside of it, Groovy source code describes the deployment process. It can use methods provided by the pipeline plug-in, most importantly steps such as calls to shell scripts or programs, file operations, sending emails, and many more. In the example, the *docker* statement is used to execute deployment steps inside a docker container. The *stage* method, provided by the pipeline plug-in, enables to define segments of the CD process that logically build upon each other. The simple *sh* step executes shell commands in the environment of the docker container. Line 10 in the example listing contains an if-statement that is used to conditionally execute steps. This highlights that a scripted pipeline can access most features of the Groovy language. This freedom allows to implement virtually any requirement of a CD process.

As of version 2.5, the pipeline plug-in additionally supports the *declarative* syntax, which is particularly easy to understand. It builds upon pre-defined structures, following the principle of declarative programming [Llo94]. This is useful in projects where it is important for employees to be capable of easily modifying the deployment process without having to learn a programming language. An example declarative pipeline implementing the same functionality as the presented scripted pipeline (Listing 2.1) can be found in Listing 4.1 on page 31. A detailed analysis of the declarative syntax of Jenkinsfiles can be found in Chapter 4.

[2]The Groovy programming language – http://groovy-lang.org
[3]Jenkins Pipeline Syntax – https://jenkins.io/doc/book/pipeline/syntax/

Listing 2.1 An example scripted Jenkinsfile

```
node {
  docker.image('node:7-alpine').inside {
    stage('Build') {
      sh 'npm install'
    }
    stage('Test') {
      sh 'npm test'
    }
    stage('Deploy') {
      if (currentBuild.result == null || currentBuild.result == 'SUCCESS') {
        echo 'Deploying....'
      }
    }
  }
}
```

2.4 Business Process Management

The term "Business Process Management" (BPM) [CT12; DLMR+18; JN14] has many interpretations and definitions from people with different interests and points of view [JN14]. According to Dumas et al. [DLMR+18], it denotes the management of chains of events, activities, and decisions, creating the value of an organization. As depicted in Figure 2.2, a business process consists of *events*, *activities* and *decision points* [DLMR+18]. An *event* is distinguished by having no duration. It can occur at any point of time and trigger the execution of activities. An *activity* defines necessary steps that have to be executed in order to achieve business goals. It is called *task*, when it is a single unit of work that cannot be split into parts.

Based on the result of an activity, a *decision point* can define which activity has to follow in the process. A typical business process also involves actors, e. g., human actors or organizations. This can be both, organization's staff, supplier or customers. Furthermore, there are objects, that can be split into physical objects, such as equipment, material or products, and information objects, i. e., digitally stored data. A business process delivers *outcomes*, which can be positive and give value to, e. g., a customer. But a process can also have (partial) outcomes that do not contribute to the value-add chain of the organization, e. g., cancellations or goods return.

As BPM is very generic, it can be applied to various domains including the CD domain. The software delivery process can be seen as business process, its stages as activities, the delivered software package or a failure message is an outcome. When mapping deployment pipelines to business processes, one can benefit from numerous existing solutions for modeling, depicting, and analyzing them.

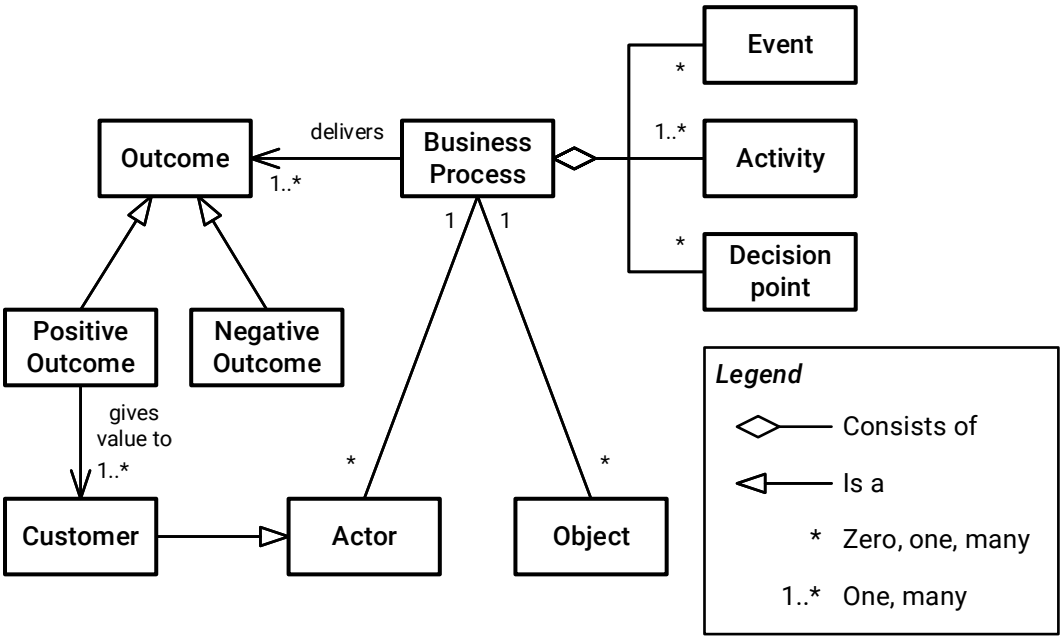


Figure 2.2: Ingredients of a business process [DLMR+18]

2.5 Business Process Model and Notation

The "Business Process Model and Notation" (BPMN) is a rich modeling language that has become the de facto standard for modeling any kind of business process [Rec10]. It inherits and combines elements from proven notations such as "XML Process Definition Language" (XPDL) and "Unified Modeling Language" (UML) activity diagrams [DDO08]. BPMN has been designed to be an expressive modeling language that is easily understandable for both, business managers and staff, but also to contain enough details for a technical operator [CT12]. After ten years of development and wide adoption in the industry, version 2.0 of the "Object Management Group" (OMG) standard has been released in 2011. Until today, this is the most recent version of the standard.

2.5.1 BPMN Elements

The modeling language BPMN consists of numerous elements, each representing a component of a business process [OMG11]. There is a very large variety of elements that can each have a high degree of detail. On one hand, this richness allows to use BPMN in almost any scenario. On the other hand, the complexity prevents a comprehensive implementation, which also leads to the fact that available implementations differ in their support for BPMN language features. In the following, some common BPMN elements are introduces, that are likely to be supported by tools. The remainder of the master's thesis will refer to only those named elements.

Processes

Most likely, the *process* is the most important concept in BPMN, also known as workflow [OMG11]. A BPMN process is a finite graph of flow elements, that describe a business process. Each process begins with a start event (empty circle with single thin line) and ends with an end event (empty circle with single thick line). In between, activities, other events, gateways, and sub-processes can be placed. Sequence flows connect the elements in the graph.

The example in Figure 2.3 shows a BPMN process, defining a procedure to handle book requests in a library. When it is triggered, a book request is received. A service task gets the current status of the requested book. Based on the result, a gateway decides whether the customer can proceed to checkout or if the customer has to be informed that the desired item is on loan. In this case, he can decide to put the book on hold, or to decline hold, resulting in a cancellation of the book request. The process also defines time-based events for automatically cancelling requests, if the customer does not reply.

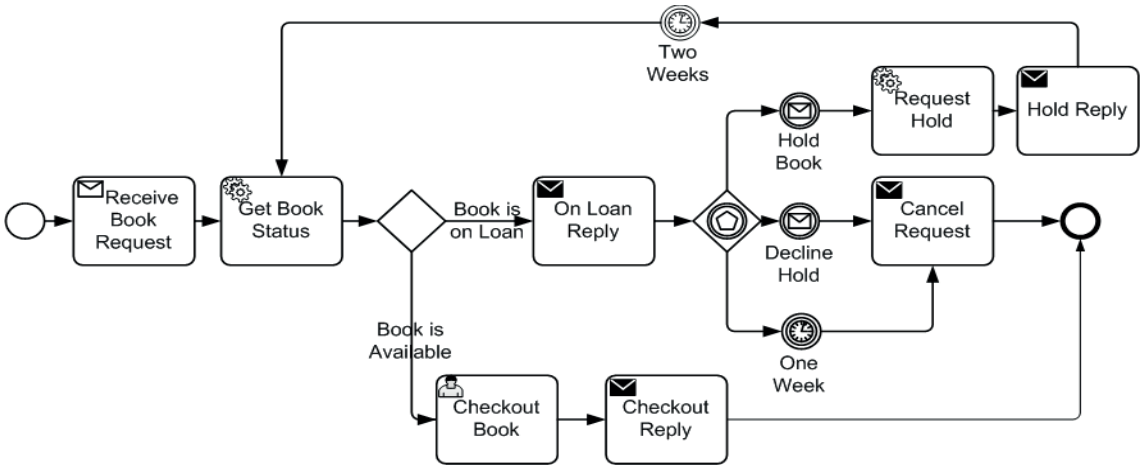


Figure 2.3: An example BPMN process [OMG11]

Activities

Activities represent the work steps that are part of the modeled process. The types of activities available inside a process are tasks, sub-processes and call activities. In general, an activity can have multiple incoming and outgoing sequence flows to other elements. This makes it possible to realize uncontrolled sequence flows, where multiple paths trigger the execution of an activity. However, in a regular scenario, it has exactly one incoming and one outgoing sequence flow. For the implementation of decisions and parallelization, the gateway element should be used.

Tasks are atomic pieces of work, that can not be split into more fine-grained activities. The most important types of tasks are depicted in Figure 2.4. Simple tasks have no special semantic and are meant for generic use. User tasks indicate that a human has to perform

work as part of the process. Manual tasks can be used to model manual steps in a partially automated application. A service task is any automated application, such as a database or web-service. Script tasks contain code that is executed by a BPMN engine.

Figure 2.4: Some important types of tasks

Sub-processes are activities, serving as a container for further activities. This concept can be used for grouping logically related pieces or work. As depicted in Figure 2.5, a sub-process can be either collapsed or expanded. In its collapsed state, the containing tasks and activities are not visible in the graphical representation, but can be extracted from the underlying data structure.

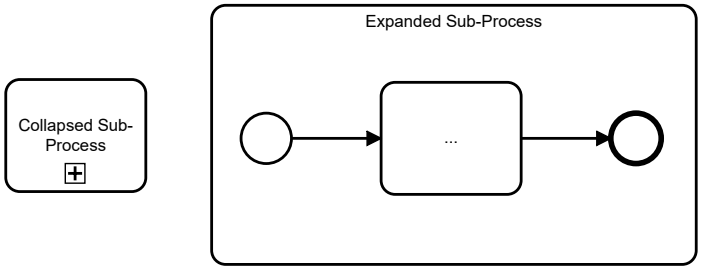


Figure 2.5: The BPMN sub-process

Call activities are used to include reusable BPMN elements. Their graphical representation is similar to the one of a collapsed sub-process, but differs in its border thickness, as seen in Figure 2.6. Included elements have to be defined outside the process, they are referenced from. Only processes and global tasks can be included by a call activity.

Figure 2.6: The BPMN call activity

Gateways

The sequence flow of a process can be controlled using gateways. A gateway has a mechanism to determine whether the sequence flow has to be split or merged and which of the different paths have to be executed. In general, a gateway either controls diverging or converging sequence flows, i. e., to define multiple branches in the process, one gateway is required to split the execution into multiple sequence flows and another to merge the branches again. There are several types of gateways. The most relevant ones are exclusive and parallel gateways.

Exclusive gateways are depicted as diamond shape, containing a bold "X", as seen in Figure 2.7. Alternatively, they can be drawn as empty diamond. An exclusive gateway represents a decision in the process, resulting in alternative paths, of which only one can be taken. Outgoing sequence flows are associated condition expressions, which determine if a path can be taken, or not. If none of the conditions match, the optionally defined default sequence flow is taken. If there is no default path and additionally no condition evaluates to true, a runtime exception is thrown. The path to take is determined by iterating all outgoing sequence flows in order. The first one with a matching condition is taken.

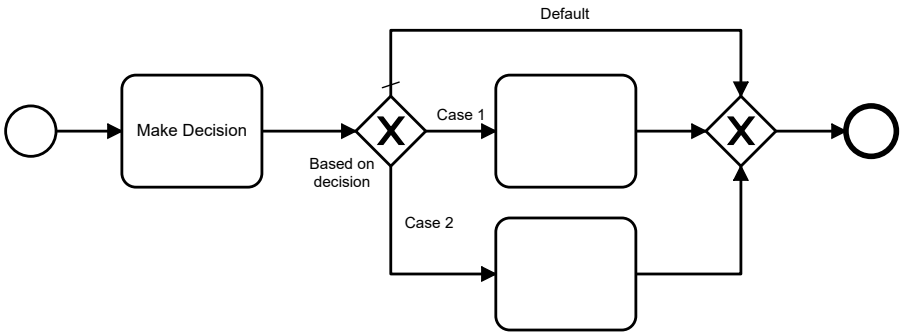


Figure 2.7: The BPMN exclusive gateway

Parallel gateways are drawn as diamond shape, containing a bold "+", as can be seen in Figure 2.8. A diverging parallel gateway indicates, that all outgoing sequence flows are taken at the same time, which results in multiple branches, executed in parallel. A converging parallel gateway synchronizes the incoming sequence flows and continues the execution, when all branches are done.

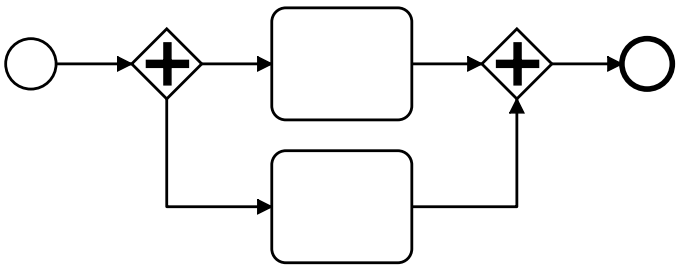


Figure 2.8: The BPMN parallel gateway

Events

Events are visualized in BPMN as circles, containing the symbol that corresponds to their event type. They can be used to show the occurrence of messages and exceptions that have an impact on the process flow. Events can be handled by dedicated activities, i. e., a handler event is triggered upon the occurrence of the concerned event.

There are three main categories of events, namely start events, end events and intermediate events. Start events indicate the beginning of a process as soon as they catch a trigger. End events occur when a process is done and throw a result to their caller. They differ from start events in their border, where a thick border indicates an end event. Intermediate events may catch triggers or throw results based on their concrete type. They can occur in between a start and end event during the execution of the process. In the following, only the event types will be introduced that are relevant to this master's thesis.

The **error event** handles exceptions that occur during the execution of the process. It is an interrupting event, i. e., the execution of the process or activity in which it occurs, is canceled and continued at the activity handling the exception. Figure 2.9 contains an example, showing an intermediate boundary error event that is attached to an activity. The error handler associated to is will be triggered, if an exception occurs during the execution of the activity. The activity has a sequence flow to an error throw event. This propagates to the caller of the process, which can handle the thrown exception.

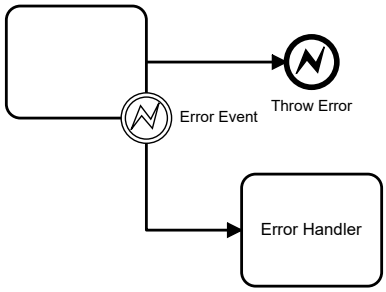


Figure 2.9: The BPMN error event

The **conditional event** is triggered upon the change of a specific variable in the scope of the event. The variable it listens to is defined in the meta-data of the event. In Figure 2.10, the conditional event is present as interrupting and non-interrupting boundary event and as start event. The interrupting conditional boundary event gets triggered upon the first change to the specified variable. In consequence, the execution of the activity it is attached to, is canceled and continued with the handler activity. The non-interrupting conditional boundary event is triggered every time, the observed variable changes. Since the execution is not canceled, this can happen any number of times.

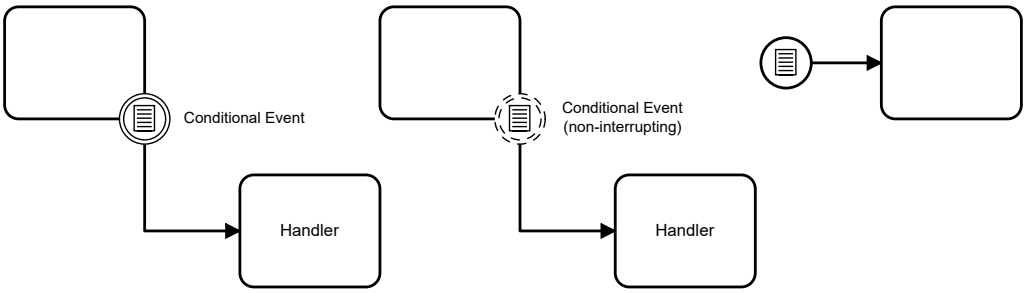


Figure 2.10: The BPMN conditional event

Compensation

The compensation concept in BPMN allows rolling back already successfully completed activities in case of subsequent exceptions. An activity can have associated a compensation handler that gets executed when the effects of the activity need to be rolled back. Figure 2.11 shows an example of a compensation scenario. The task "Perform Task" has a boundary compensation event, which has attached the compensation handler "Roll Back". If the validation done by the task "Verify Correctness" fails, the compensation procedure is initiated by the compensation end event. In consequence, all compensation handlers in scope are triggered, which results in the case of the given example in rolling back the initially performed task. If there were other tasks with compensation handlers in the same or a higher-level process, these would be triggered as well.

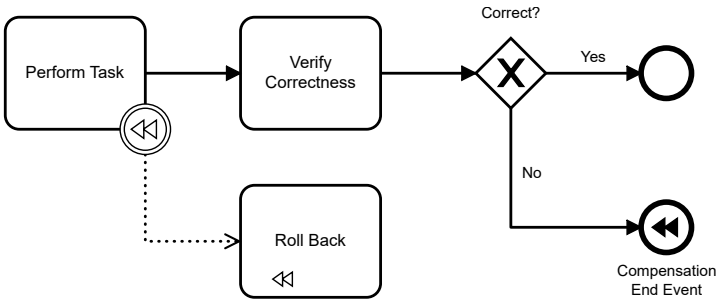


Figure 2.11: The BPMN compensation event

2.5.2 BPMN Engines

A BPMN process can either simply serve for documentation and visualization purposes, or contain information detailed enough to be executable. In general, a BPMN process can be executed using one of many available BPMN engines. However, the BPMN standard is highly complex and allows individual implementations to make own decisions and assumptions at many points. In addition, Geiger et al. [GHL+15] found that many implementations have poor coverage of the BPMN standard and partially even lack support for basic language elements. This limits the portability of BPMN processes and makes it necessary to carefully select the engine that is used in a project. Executable process definitions can often only be executed by one specific engine.

The Camunda Services GmbH[4] is specialized in developing software solutions for modeling and executing business processes. According to their official website, their tool stack is used and trusted by numerous well-known companies, such as Allianz[5], Deutsche Bahn (German Rail)[6] and Zalando[7] to name some examples. Among their products, there is an

[4]Camunda BPM – https://camunda.com
[5]Allianz – german financial services – https://www.allianz.de/
[6]Deutsche Bahn – german rail – https://www.bahn.de/
[7]Zalando – https://www.zalando.com/

open-source BPMN modeling tool, the Camunda Modeler[8]. The editor is both, easy to use and powerful. It offers good support for the BPMN 2.0 standard and can export diagrams as vector graphics, which in total makes it the ideal choice as editing and visualizing solution for this master's thesis.

Flowable[9] is another popular BPMN engine, developed and published as open-source tool by the Flowable AG. They also provide extensive enterprise solutions for process automation. Flowable offers a web-based BPMN editor as well as an Eclipse plug-in. Activiti[10] focuses on the open-source community and aims to be a light-weight tool that is scalable and cost-effective at the same time. Its cloud version uses the web-based bpmn.io editor, which is also the foundation of the Camunda Modeler. Additionally, there is an Eclipse plug-in for the process design.

There are numerous other BPMN engines, most of them open-source and Java-based. Obviously, the technology of currently available tools is rather similar and their most important difference is the support of BPMN constructs. Investigations have shown that the Camunda Modeler offers a good BPMN support, which is why this master's thesis focuses on this tool.

2.5.3 BPMN XML files

A BPMN model can be interchanged using the "Extensible Markup Language" (XML) format, defined in the official BPMN specification [OMG11]. An example file can be found in Listing 2.2. The corresponding graphical representation is depicted in Figure 6.30b on page 65. The root element of a BPMN file has to be the `<bpmn:definitions>` element, containing all elements of the modelled process. In the example, the definitions include one process, which has a start- and end event, as well as a script task element and connecting sequence flows. Furthermore, in a separate XML element, the layout information of the BPMN diagram are stored: The node `<bpmndi:BPMNDiagram>` contains an entry for each BPMN element, defining its visual bounds.

The BPMN specification allows the extension of a model by custom elements. The example in Listing 2.2 contains a start event (lines 4-11), which has the extension element `camunda:properties`, in this case provided by the Camunda editor. This allows to include any information in a BPMN model, adapting it to individual use-cases, where the available properties of existing BPMN elements are not sufficient.

[8] Camunda Modeler – https://camunda.com/products/modeler/
[9] flowable—Java Business Process Engines – https://www.flowable.org/
[10] Activiti – Open Source Business Automation – https://www.activiti.org/

Listing 2.2 An example BPMN file containing a minimal process – “`minimal-process.bpmn`”

```
<?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:di="http://www.omg.org/spec/DD/20100524/DI"
    ↪ xmlns:dc="http://www.omg.org/spec/DD/20100524/DC" id="Definitions_1q01txf"
    ↪ targetNamespace="http://bpmn.io/schema/bpmn" exporter="Camunda Modeler"
    ↪ exporterVersion="2.0.3">
  <bpmn:process id="Process_1" isExecutable="true">
    <bpmn:startEvent id="StartEvent_1">
      <bpmn:outgoing>SequenceFlow_0nlehpk</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:endEvent id="EndEvent_0avv273">
      <bpmn:incoming>SequenceFlow_0nlehpk</bpmn:incoming>
    </bpmn:endEvent>
    <bpmn:sequenceFlow id="SequenceFlow_0nlehpk" sourceRef="StartEvent_1"
    ↪ targetRef="EndEvent_0avv273" />
  </bpmn:process>
  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
    <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="Process_1">
      <bpmndi:BPMNShape id="_BPMNShape_StartEvent_2" bpmnElement="StartEvent_1">
        <dc:Bounds x="173" y="102" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNShape id="EndEvent_0avv273_di" bpmnElement="EndEvent_0avv273">
        <dc:Bounds x="259" y="102" width="36" height="36" />
      </bpmndi:BPMNShape>
      <bpmndi:BPMNEdge id="SequenceFlow_0nlehpk_di" bpmnElement="SequenceFlow_0nlehpk">
        <di:waypoint x="209" y="120" />
        <di:waypoint x="259" y="120" />
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</bpmn:definitions>
```

2.6 Resilience

There is a large variety of definitions and interpretations of the term resilience in literature. It is being used in different domains with different meanings. In the following, its meaning in the domain of CD will be explained, together with a consistent definition of the terminology that will be used throughout this thesis.

Bishop et al. [BCFM11] identified the lack of a single, generally accepted understanding of the term resilience and propose a security-related definition that is meant to make future collaboration and communication consistent and unambiguous. They highlight the importance of a consistent usage of terminology in general and that already slightly differing interpretations can lead to misinterpretations of a work’s results.

Bishop et al. deduce the term resilience from its meaning in natural language and transfer it to the computer science domain. According to them, the non-technical definition of resilience is either "resuming the original shape or position after being bent, compressed, or stretched" or "rising readily again after being depressed; hence, cheerful, buoyant, exuberant". Following their explanations, this means for a resilient computer system that it may suffer from capacity loss under stress or after an impact, but that it still has to provide some essential functionality.

Most importantly, it has to be capable to recover or reconfigure itself to return to normal operation after an impact [BCFM11]. Bishop et al. highlight, that this is not only valid for the attributes performance and availability, but also for confidentiality and integrity. This in particular is a challenging problem, since data that became public is not easily made private again. For this reason, they believe that resilient computer systems will not be adopted in near future with few possible exceptions.

Obviously, resilience is not easily measurable. There are different approaches to ensure a system's functionality despite failures or attacks. One could introduce redundancy or do preparations in order to reduce the time to recover from failure. But which approach makes a system more resilient? Musman and Agbolosu-Amison gave a definition of resilience, enabling to objectively compare the resilience of systems in similar fields:

> **Resilience is:**
>
> *The persistence under uncertainty of a system's mission-oriented performance in the face of some set of disturbances that are likely to occur given some specified timeframe.*
>
> (Musman and Agbolosu-Amison [MA14])

This definition is a risk-oriented metric, focusing on the probability of undesirable events during a given timeframe. The higher the probability of disturbances is, the more uncertain the system's performance becomes. In this case, the term *performance* does not necessarily refer to processing a certain amount of work per specified time interval, but rather means the system's general contribution to achieving mission requirements, outcomes or objectives. Here again, an important part of the definition is *persistence*, meaning not only the maintenance of normal operation, but the recovery from a failure as well.

2.7 ANTLR

As mentioned before, a part of this master's thesis is the transformation of Jenkinsfiles, which requires an approach of parsing them. "ANother Tool for Language Recognition" (ANTLR)[11] is an open-source parser generator that can be used to parse, translate, process or execute any kind of structured input data, such as text or binary files—in other words, the utility simplifies the development of compilers, translating a source language into a target language. It is being developed at the University of San Francisco since 1989. At the time of this master's thesis, its latest version available is 4.7.2.

[11] ANTLR – https://www.antlr.org/

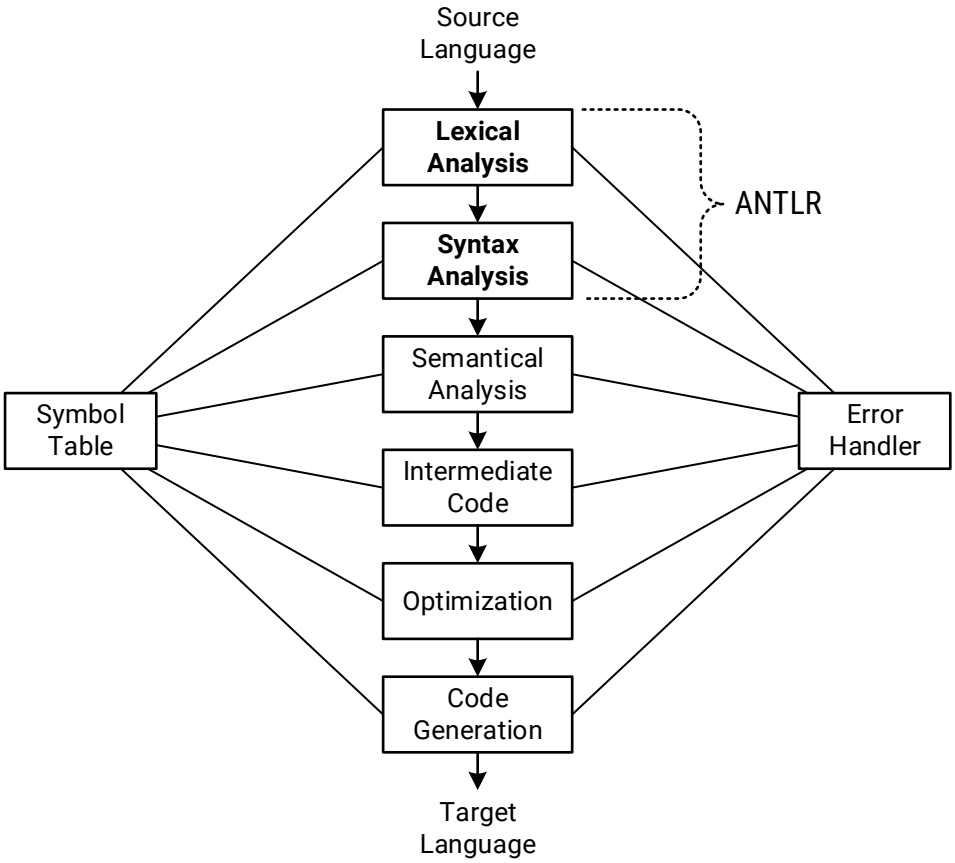


Figure 2.12: The stages of a compiler [ASU99]

In general, a compiler consists of six stages, as depicted in Figure 2.12 [ASU99]. The first three stages represent the *analysis* phase, splitting the input into its components and construct an internal representation. The remaining three stages form the *synthesis* phase, constructing the target language. The purpose of ANTLR is performing the first two compiler stages, the lexical and syntactical analysis of the source language. In the following, these stages will be explained in more detail according to Aho et al. [ASU99]. All remaining stages are not relevant to this master's thesis.

2.7.1 The Lexical Analysis

The lexical analysis, also known as *scanning*, forms logical groups of input characters and assigns symbols to them [ASU99]. It converts the input source code into a stream of tokens, which is processed in the subsequent stages of the compilation process.

The component of the compiler responsible for performing the lexical analysis is called *scanner* or *lexer*. It recognizes symbols in the input source code based on patterns, defining the sequence of characters that has to be assigned a specific symbol. For the definition of patterns, regular expressions are ideally-suited. For instance, given the rules in Figure 2.13, the input text in Figure 2.14 gets assigned the depicted symbols. The set of rules forms a

```
IDENTIFIER  ⟶  [a-zA-Z]+
    NUMBER  ⟶  [0-9]+
    ASSIGN  ⟶  ':' '='
      PLUS  ⟶  '+'
     TIMES  ⟶  '*'
```

Figure 2.13: Example lexer rules

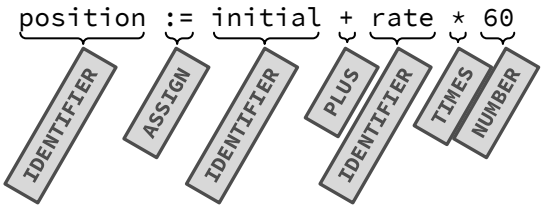


Figure 2.14: An example input with assigned tokens

grammar, describing the language elements that can be detected by the lexer. The order of rules is relevant, as the first matching one determines the symbol that is assigned to the concerned sequence of input characters.

The lexer is the first component of a compiler that reads the source code. This makes it well-suited for the task of cleaning the input from irrelevant information. For this reason, one of its tasks is removing whitespace and tab characters, line breaks, and comments. This step is important, because the implementation of the syntactical analysis and other stages would be significantly more complex, if they had to deal with characters, only existing to increase readability.

2.7.2 The Syntax Analysis

The syntactical analysis is done by the *parser* component of the compiler [ASU99]. It uses rules to build a parser tree from the symbol stream it gets from the lexer. The rules generally are given as context-free grammar, which precisely describes the input language and its allowed constructs. Each rule defines the structure of a language element, consisting of other parser rules (non-terminals) or lexer symbols (terminals). One rule has to be defined as start symbol, which is used as root for the parsing process.

Figure 2.15 shows an example parser grammar, which recognizes simple arithmetic expressions, such as the one given in the example before (see Figure 2.14). The `assignment` rule is defined as start symbol. When parsing the token stream that results from performing a lexical analysis on the given example input expression, the parser would construct the syntax tree given in Figure 2.16.

```
assignment   ⟶   IDENTIFIER ASSIGN expression
expression   ⟶   expression op expression
expression   ⟶   IDENTIFIER
expression   ⟶   NUMBER
        op   ⟶   TIMES
        op   ⟶   PLUS
```

Figure 2.15: Example parser grammar

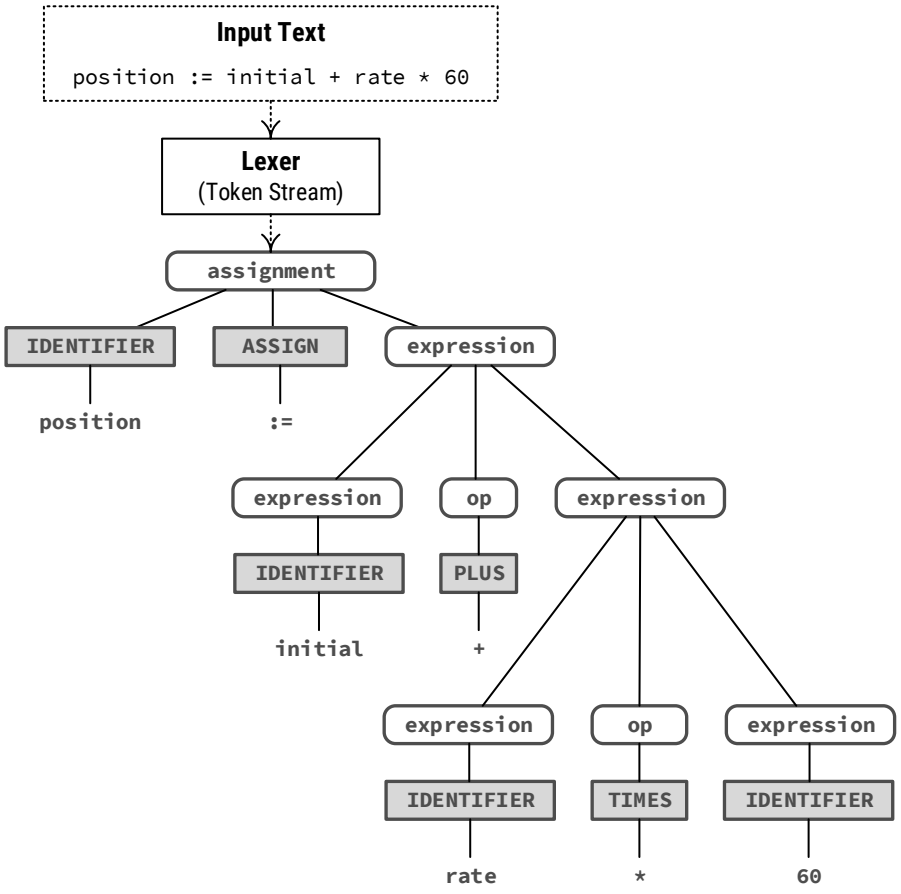


Figure 2.16: The syntax tree, resulting from parsing the given input string

2.7.3 The ANTLR Grammar File

ANTLR is configured by a file with the extension `.g4`, containing the grammar of the language to parse [Par13]. The file has to start with the keyword `grammar`, followed by the name of the grammar. Below this initial line, grammar rules can be defined, where the name of a rule determines its type. If it starts with a small letter, it is interpreted as parser rule. If its first letter is a capital letter, it is a lexer rule. While the order of rules generally is not important, by convention, parser rules should be placed at the top of the file, followed by lexer rules. As explained above, lexer rules are sensitive to their order among each other.

In ANTLR, the name of a rule has to be unique, i. e., it is not possible to define alternatives by defining the concerned rule multiple times. Instead, the vertical pipe character has to be used to indicate alternatives (see lines 5 and 6 in Listing 2.3). Static terminals, such as explicit characters and strings have to be surrounded by single quotation marks (see line 12). Sequences of multiple characters can be written as continuous string, i. e., `'e' 'n' 'd'` and `'end'` are equivalent. Each rule definition must end with a semicolon.

Fragment rules can be used to increase the readability of lexer rules. They can be included in other rules and will never produce symbols themselves. Fragment rules are initiated by the keyword `fragment` and the rule name (see line 11 in Listing 2.3). By convention, they should be named according to the CamelCase naming convention.

The partial example grammar in Listing 2.3 recognizes assignments as given in examples above. Its start symbol is the parser rule `assignment`, which matches an `IDENTIFIER`, followed by an `ASSIGN` symbol and an `expression`. The first two components are lexer rules, recognizing a specific sequence of characters. In the case of the `IDENTIFIER` rule, any coherent sequence of small or capital letters gets assigned the according token. The `ASSIGN` rule only matches a single equals sign. If multiple subsequent equals signs would occur in the input string, each single one of them would get an `ASSIGN` token.

The `expression` rule is a recursive parser rule, recognizing either an `IDENTIFIER`, a `NUMBER` or two nested expression, divided by an operator `op`. The `NUMBER` lexer rule matches a coherent sequence of digits, defined in a fragment rule `Digit`. The fragment itself never leads to the assignment of a token to a matching input string, i. e., single digits as recognized by the `Digit` rule do not get assigned a token. This means that a fragment rule cannot be used in a parser rule directly and has to be included in a lexer rule.

Listing 2.3 An example ANTLR file – "`example-grammar.g4`"

```
1  grammar ExampleGrammar;
2
3  // Parser Rules
4  assignment: IDENTIFIER ASSIGN expression ;
5  expression: expression op expression | IDENTIFIER | NUMBER ;
6  op: TIMES | PLUS ;
7
8  // Lexer Rules
9  IDENTIFIER: [a-zA-Z]+ ;
10 NUMBER: Digit+ ;
11 fragment Digit: [0-9] ;
12 ASSIGN: '=' ;
13 // ...
```

3 Related Work

In order to show the relevance of this master's thesis' topic, this chapter presents previously published scientific work that brings BPMN to the CD domain or develops a DSL for CD processes. Furthermore, connecting factors in existing work are presented, on which this master's thesis can build upon.

3.1 BPMN in the CD Domain

In his master's thesis, Willig [Wil18] determined that state-of-the-art CD systems have deficits when it comes to defining CD pipelines. He explains that PDLs often use declarative concepts which restricts their functional abilities to the implementation of the underlying system. As a result, those languages tend to lack expressiveness since the CD system has to support all of their features. According to Willig, in some cases, this is compensated by allowing imperative languages inside declarative constructs making the CD pipeline definitions unclear and hard to maintain. He claims that especially YAML-based PDLs lead to unclear results, because of poor support for structure and modularisation elements.

Furthermore, he found that it is hard to map conditional flows to the PDLs of current CD systems. He explains that as a result, many projects have to use multiple CD pipelines in order to cope with all circumstances. However, these pipelines share common fragments that have to be synchronized upon changes which is error-prone and expensive.

He figured out that BPMN fulfills all necessary requirements to be used as modeling language for CD pipelines. He maps BPMN elements to the CD process domain to point out which subset of the notation is suitable for a graphical representation of CD processes. Based on these results, he extends the software delivery system JARVIS [Dör18] to support BPMN.

3.2 Android Continuous Delivery DSL Engine

Some authors have found out that the design of CD pipelines is a time-consuming activity and that developing an abstraction layer to simplify this task can help organizations adopting CD. For instance, Fischer [Fis18] developed a DSL engine in his master's thesis, which aims to support the design of CD pipelines for Android applications. He found, that implementing CD for mobile platforms bears challenges, making it interesting focusing on this domain. According to his thesis, some challenges that distinguish the development of conventional desktop or web applications from mobile ones are: There is a high diversity of operating systems and device specifications, making it impossible to perform comprehensive tests. This increases the risk of a release since the probability for the new version not to work on a

Listing 3.1 Sample Android CD DSL script [Fis18]

```
1  .repo("https://github.com/M-Fischer/android-playground.git")
2
3  .buildServer("serverUser", "serverPassword")
4    .signing("keystorePw", "keyPw")
5    .ftpCredentials("ftpUser", "ftpPw")
6
7  .pipeline("My Android Pipeline")
8    .stage(COMMIT_STAGE).
9      .step(COMPILE_RELEASE)
10     .step(UNIT_TEST)
11     .step(LINT)
12   .stage(ACCEPTANCE_TEST_STAGE, Trigger.AUTOMATIC)
13   .parallel()
14     .seqStage(PERFORMANCE_TESTING_STAGE, Trigger.AUTOMATIC)
15       .eventCount(1500)
16     .seqStage(CUSTOM_DEPLOYMENT_STAGE, Trigger.AUTOMATIC)
17       .location("localhost")
18     .stage(TESTING_AS_A_SERVICE_STAGE, Trigger.MANUAL)
19   .endParallel()
20   .stage(ALPHA_RELEASE_STAGE, Trigger.AUTOMATIC)
21   .stage(BETA_RELEASE_STAGE, Trigger.MANUAL)
22   .stage(PRODUCTION_STAGE, Trigger.TIMETRIGGER, 7, DAYS)
23
24 .generate()
```

specific type of device is not negligible. Additionally, publishing a new version of an app to the store takes time because it has to be reviewed by the app store vendor. This also restricts the frequency of publishing new versions and prevents to do incremental releases.

Fischer determined that for these reasons, the development of CD infrastructures requires much time and know-how and keeps developers from performing their main activities. To simplify the creation process of CD infrastructures, he developed a DSL engine. It provides a Java API that can be used by Android developers to describe their individual CD process in so-called *DSL scripts*. An example script can be found in Listing 3.1. The *Android CD DSL Engine* generates XML files out of such a script to configure Jenkins which executes the designed CD process.

The meta-model of the Android CD DSL, depicted in Figure 3.1, sticks closely to the definition of CD pipelines by Humble and Farley [HF10]. Fischer aimed to keep it generic enough not to be limited to the Android domain, however the implementation is highly specific, which prevents to use it for different kinds of applications without making fundamental changes. The meta-model consists of *stages* that are arranged in sequential or parallel order. A stage can only be executed if all predecessors have been successfully completed. Parallel stages can be executed in arbitrary order. A stage can be triggered (a) automatically when all previous ones are completed, (b) manually upon decision of a developer, (c) periodically, e. g., every night, or (d) by commits in the VCS.

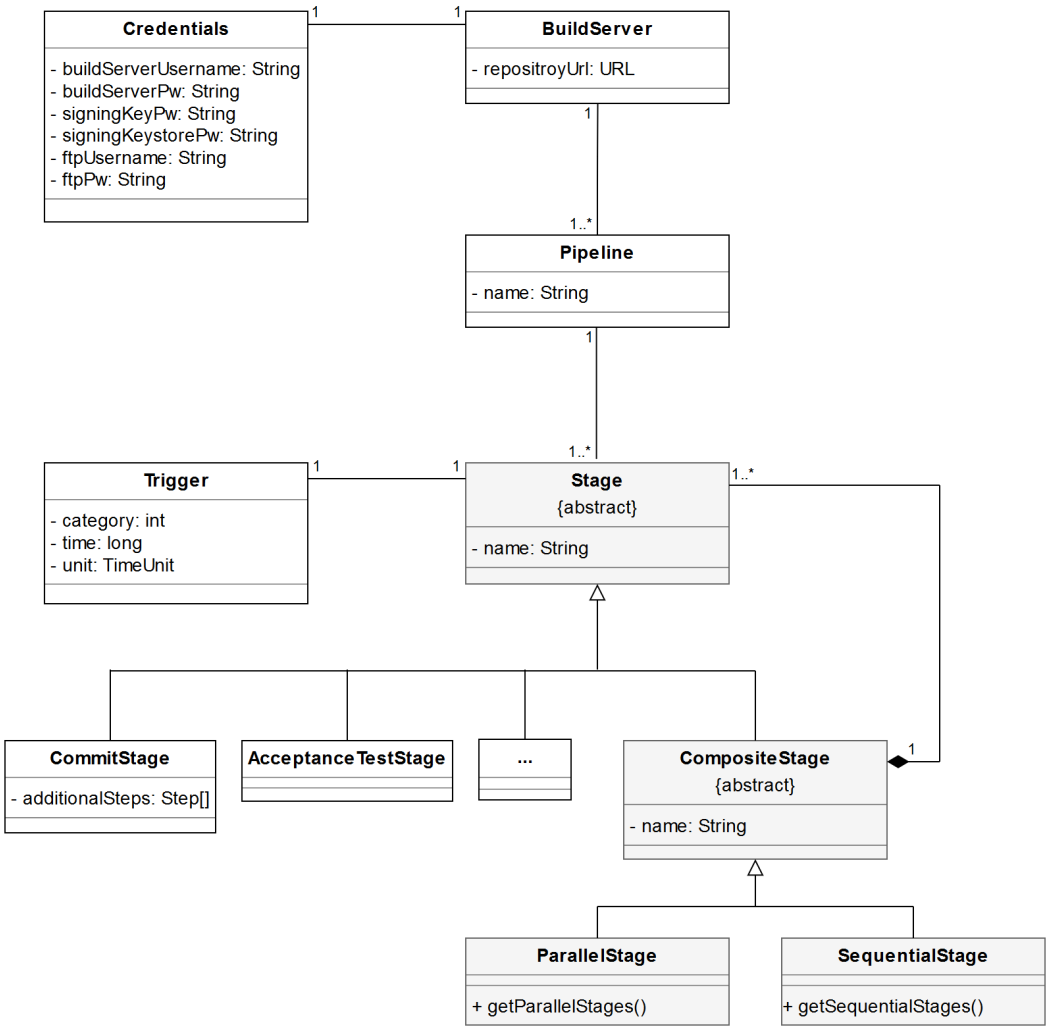


Figure 3.1: Semantic model of the Android CD DSL [Fis18]

The DSL is based on the assumption that all Android app development processes use the same technical steps for deploying the application, e. g., Gradle[1] has emerged as the standard build system. For this reason, there is no possibility to adapt the technical implementation of the steps inside a CD stage. The developer can choose from a built-in set of stages and configure them with few parameters, e. g., the address of the Git repository and credentials for the build server to use. However, he cannot add custom steps or modify the implementation of existing ones. Additionally, the DSL does not include features for error handling or roll-back after failures. Hence, it would be necessary to write additional code in order to improve the fault tolerance of the CD process.

[1] https://gradle.org/

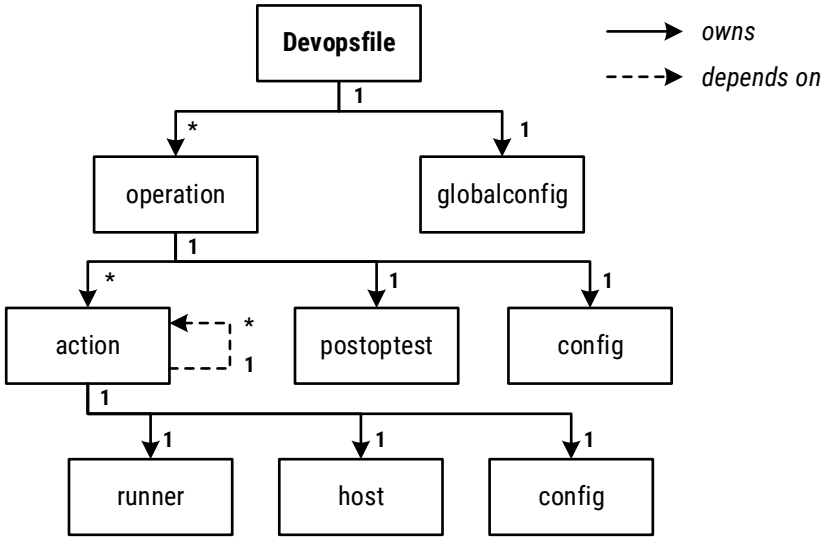


Figure 3.2: DevOps Slang—structure of a Devopsfile [WBL14]

Since the current implementation of the DSL is very limited to the Android domain, it cannot be used as generic basis for further research. However, the semantic model of Fischer (see Figure 3.1) can serve as a first starting point for the development of a generic metamodel for the CD domain.

3.3 DevOps Slang

Wettinger et al. [WBL14] determined that state-of-the-art solutions for automating CDs processes have one major problem: they are not usable as a holistic approach to collaborate on automating deployment and operation of applications, independently of specific providers or tools: On one hand, there are existing approaches to automate the infrastructure of applications that are often bound to specific providers or tools, e. g., Amazon AWS[2]. Some solutions support the definition of higher levels of applications, but here, additional imperative logic is required, making it hard to get a high-level view on the deployment process. On the other hand, there are UML deployment diagrams, which are well-suited for collaboration and communication, but are not executable. They found that there are no solutions to bridge this *DevOps gap* that exists between the low-level implementation of a DevOps process and its high-level description.

Wettinger et al. tackle the DevOps gap by inventing a DSL called *DevOpsSlang*. It enables the definition of DevOps processes on a high level, while containing enough information to be executable. The central goal of DevOpsSlang is to enable and support an efficient collaboration between developers and operations staff. As depicted in Figure 3.2, it consists of JSON[3]-based text files, called *Devopsfiles*, that contain the operations of the DevOps

[2] https://aws.amazon.com
[3] https://www.json.org/

Listing 3.2 An example Devopsfile

```
"deploy": {
  "actions": {
    "deploy-mongodb": {
      "runner": "chef-solo-runner",
      "config": {
        "files": { "mongodb.tgz": "http://.../mongodb.tgz" },
        "runlist": [ "recipe[mongodb::default]" ]
      }
    }
  },
  "postoptest": {
    "runner": "command-runner",
    "config": {
      "command": "export RESCODE=$(curl -sL -w \"%{http_code}\\n\" \"http://localhost
:3000\" -o /dev/null) && [[ \"$RESCODE\" == \"200\" ]] && true || false"
    }
  }
}
```

process to automate. Each operation can contain actions that define the steps necessary to deploy and operate the application. Actions can depend on each other defining an order in which the execution has to take place. An action is performed by one specific runner on one specific host. The action can choose from a pool of runners that have individual implementations to perform a dedicated type of task. For example, one runner can be designed to execute Ruby scripts, another may run a console command on the destination host. To retain a high level of abstraction of Devopsfiles, the implementation of runners can be highly application-specific. This allows hiding the complexity of the process while remaining flexible enough to easily change the overall behavior of the DevOps process.

Listing 3.2 shows an example Devopsfile containing the operation “deploy”, which has set the “postoptest” attribute. It demonstrates how a successful execution of an operation can be validated using a shell command. In the example, the web server running on the local port 3000 is being queried. If the resulting code equals 200, meaning that the server is up and responding, the postoptest action succeeds. If the result code differs, the test fails.

4 Analysis of the Jenkinsfile Meta-Model

To bring BPMN to the CD domain, first of all it is necessary to precisely understand and define popular concepts in the field of continuous delivery. In Section 2.2, CD has already been introduced based on the definitions by Humble and Farley [HF10], which represent the foundation of any CD implementation. This chapter will go into detail by analyzing the concepts of a specific CD implementation and also answer RQ 1.

Jenkins and its numerous plug-ins are being actively developed by contributors from various fields which makes the tool highly flexible and suitable for most projects. But this also makes it a good choice as a starting point for further development in the CD domain. The following will present an analysis the DSL of Jenkinsfiles to find out about the concepts of the implementation of CD in Jenkins.

Listing 4.1 An example declarative Jenkinsfile

```
pipeline {
    agent {
        docker { image 'node:8-alpine' }
    }

    stages {
        stage('Build') {
            steps {
                sh 'npm install'
            }
        }
        stage('Test') {
            steps {
                sh 'npm test'
            }
        }
        stage('Deploy') {
            when {
                expression {
                    currentBuild.result == null || currentBuild.result == 'SUCCESS'
                }
            }
            steps {
                echo 'Deploying...'
            }
        }
    }
}
```

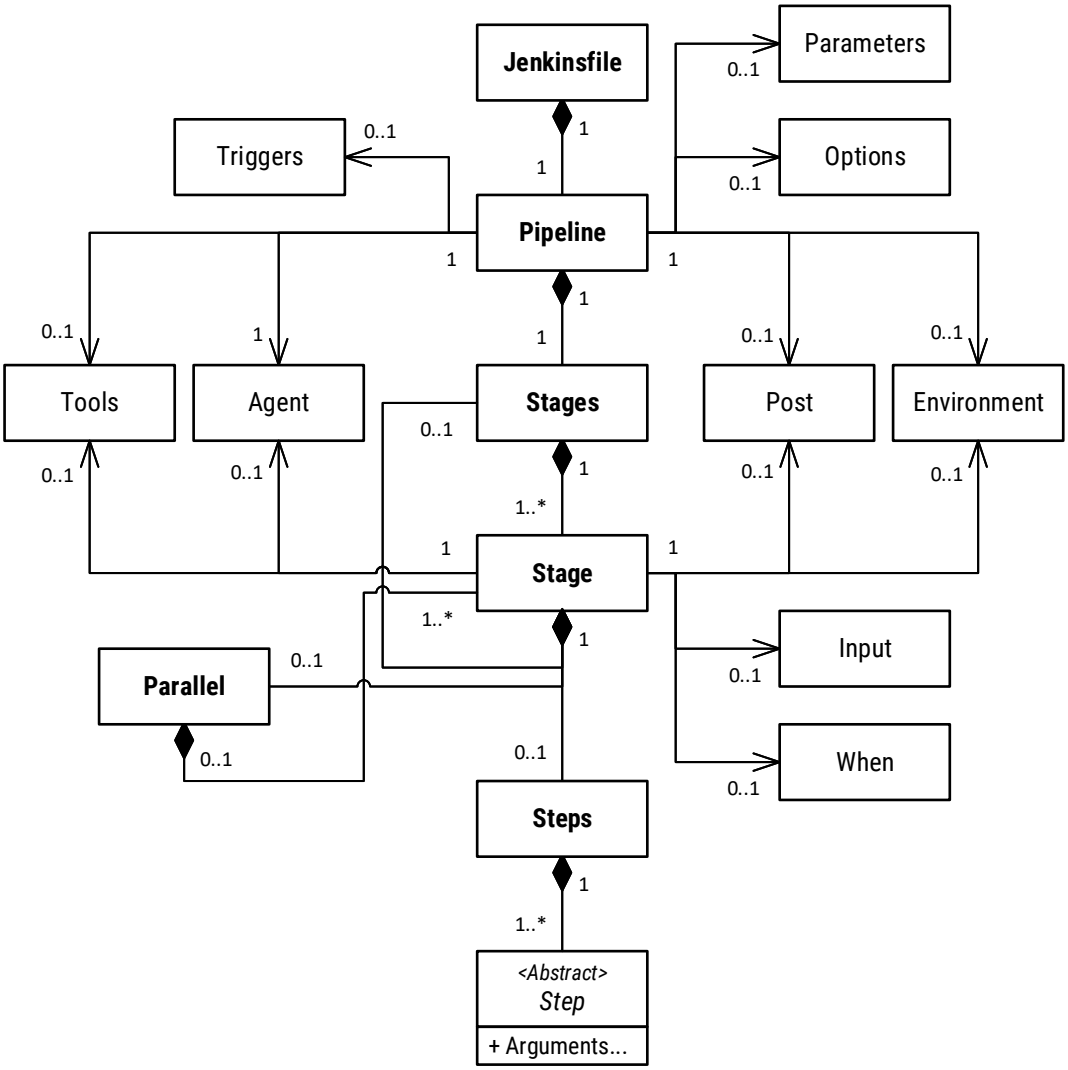


Figure 4.1: Meta-model of declarative Jenkinsfile

In Section 2.3, the two languages a Jenkinsfile can be written in are introduced. As the scripted variant at its core is an imperative programming language, it is not well-suited to be transformed to different languages. In contrast, the number of possible constructs allowed in the declarative syntax is much lower, making it relatively easy to understand the semantics and to transfer the implemented functionality to different languages. For this reason, this master's thesis will focus on the declarative pipeline syntax of Jenkinsfiles. Based on our findings, the scripted syntax is considered to be out-dated and has no meaning for this master's thesis. An example of a declarative Jenkinsfile is given in Listing 4.1.

Figure 4.1 shows the meta-model of the declarative syntax of Jenkinsfiles. It has been developed based on the official documentation of Jenkinsfiles[1]. Some properties, such as the name of stages, have been omitted in the figure for the sake of simplicity. In the following, the entities of the meta-model will be explained from top to bottom, in order to get a detailed understanding of the essential parts of a Jenkinsfile.

4.1 The Pipeline Section

Each declarative Jenkinsfile has to contain exactly one `pipeline` element (line 1 in the example). A pipeline must have an `agent` and can define `triggers`, `options`, `parameters`, `tools` and `environment` variables. The **agent** section determines the environment in which the CD pipeline is to be executed. A common example is a docker image, in which Jenkins should start the CD process (line 3 in the example). A **trigger** can be either a time-schedule which periodically initiates the pipeline process or it can specify a list of jobs and a threshold to re-trigger the pipeline if any listed job completes at least with a certain status. This allows to automate building a software that is split into multiple repositories. In the **options** section, pipeline-specific parameters can be defined, such as a number of `retry` or a `timeout` period. Particularly the `retry` option is a good way to avoid temporary failures, e. g., communication errors, ensuring operability of the CD pipeline. The **parameters** directive can be used to define values that a user has to provide when triggering the pipeline. These can be used in steps during the execution. Using the **tools** section, Jenkins can be instructed to automatically install *maven*, *jdk* or *gradle* within the agent before launching the pipeline. This option is only usable, if there has been defined an *agent* other than *none*. Within the **environment** section, variables can be specified, that are available to steps within scope, i. e., variables of a stage will not affect steps defined on upper levels.

4.2 Post Steps

The **post** section can contain conditionally executed steps. The available conditions are:

always	Regardless of the completion status
changed	If the completion status differs from the previous run
fixed	If the completion status changed from unstable or failed to successful
regression	If the completion status is not successful, but was so before
aborted	If the pipeline has been manually aborted
failure	If the completion status is failed
success	If the completion status is success
unstable	If the completion status is unstable
unsuccessful	If the completion status differs from success
cleanup	Always, but after any other step in the post section

[1] Pipeline Syntax – https://jenkins.io/doc/book/pipeline/syntax/

Listing 4.2 An example of a `post` section in a Jenkinsfile.

```
1  pipeline {
2      agent any
3      stages { ... }
4      post {
5          always {
6              echo 'Pipeline finished'
7              sh './clean-tests.sh'
8          }
9          regression {
10             sh './regression.sh'
11         }
12     }
13 }
```

Listing 4.2 shows an example pipeline containing a simple post section. Its stages are omitted for the sake of simplicity. The steps inside the `always` section are executed regardless of the completion status of the stages in the pipeline. This for instance allows to perform cleanup steps or to collect statistical data. Conditionally executed steps such as these are a limited approach to ensure the operability of the CD pipeline. Upon a failure, post steps could ensure that subsequent stages are able to operate and modify the completion status so that Jenkins does not abort the execution. In theory it is also possible to control the sequence flow and to retry the failed stage. However this would require to exit the declarative Jenkinsfile syntax and implement the functionality using Groovy code. Since this would violate the principle of declarative programming, this method is not desirable.

4.3 Stages

The **stages** section inside a pipeline has to contain at least one stage. All stages here are executed sequentially. As soon as the first stage returns a non-successful completion status, the pipeline itself fails. A **stage** can overwrite the sections *agent*, *environment* and *tools*. If they were set in parent stages or in the pipeline section, those previous definitions will be replaced for this stage and any nested ones.

A stage has to contain either one `stages` section itself, one `parallel` section or one `steps` section. A stages section contains nested stages, as does the parallel section. The difference between both is that the stages inside a parallel section can be executed at the same time. The overall result of in parallel executed stages is determined, when all of them have terminated. This behavior can be altered by setting the `failFast` attribute in the stage containing the `parallel` section. It will let the stage fail as soon as the first nested one returns a non-successful completion status. Remaining stages that are still running, will be aborted immediately. There are two important restrictions when using the `parallel` section: 1. Nested stages cannot contain further parallel sections and 2. stages containing parallel steps cannot contain an `agent` or `tools` section. These are usable again in nested stages containing steps.

In addition, one `post` section can be appended, similarly to the one available in the *pipeline* section. Furthermore, a stage can specify an `input` section which allows manual intervention before the execution of the stage. Here, the developer of the pipeline can request additional information or decisions from the user. The execution will be paused until the requested information is entered in the user interface of Jenkins.

4.4 Conditional Execution of Stages

The `when` section can contain nested conditional steps that are executed based on the result of the stage it is contained in. The following set of conditions can be used:

branch	Only execute the stage if building specific branches
buildingTag	Only execute the stage if a specific tag is associated to the respective commit
changelog	Only execute the stage if the commit message matches a given pattern
changeset	Only execute the stage if the specified files have been changed
changeRequest	Only execute the stage if it has been triggered by a change request (aka pull request)
environment	Only execute the stage if a specified environment variable has a given value
equals	Only execute the stage if a variable has a specified value
expression	Only execute the stage if a given Groovy expression evaluates to `true`
triggeredBy	Ony execute the stage if has been initiated by a given trigger

The `expression` condition allows implementing decisions using any Groovy expression. This can be useful if the available functionality does not suffice to implement the desired behavior. Expressions can be combined using the keywords `not`, `allOf` and `anyOf`. An example for the use of the `when` section can be found in Listing 4.3. It shows the stage “Demo Stage: When”, which is only executed if the branch *master* is currently being built and additionally if either the environment variable `HOSTNAME` has the value “prod3” or the environment variable `PRODUCTION` is set to “YES”.

Listing 4.3 Example of a `when` section in a Jenkinsfile.

```
pipeline {
    agent any
    stages {
        stage('Demo Stage: When') {
            when {
                allOf {
                    branch 'master'
                    anyOf {
                        environment name: 'HOSTNAME', value: 'prod3'
                        environment name: 'PRODUCTION', value: 'YES'
                    }
                }
            }
            steps { ... }
        }
    }
}
```

Command	Description	Example
`sh`	Execute a shell command	`sh 'echo \'Test\''`
`echo`	Print a message	`echo 'Test'`
`dir`	Change the current directory	`dir '/home/me'`
`deleteDir`	Recursively delete a directory	`deleteDir '/tmp/build'`
`script`	Execute Groovy code	`script { def x }`
`error`	Throw en exception	`error 'Something happened.'`
`mail`	Send an email	`mail to:'some@one.com', subject:'A Mail!'`
`sleep`	Pauses the execution	`sleep 5`

Table 4.3: Basic Jenkinsfile steps using a simplified syntax

4.5 Steps

The most fundamental concept in both, the declarative and scripted syntax are steps. Jenkins' pipeline plug-in has a large set of build-in steps that can be used out-of-the-box. Some examples are given in Table 4.3. Additionally, a large number of highly specific steps are provided by plug-ins that can easily be integrated into Jenkins, e. g., for performing tasks with the AWS[2] or the Telegram Bot API[3].

[2]Amazon Web Services – https://aws.amazon.com
[3]Telegram Bot API – https://core.telegram.org/bots/api

4.5.1 Steps are Function Calls

Basically, a step can be seen as the call to a pre-defined function. Most steps allow using the simplified method call syntax of the Groovy language. This syntax does not require parentheses as long as at least one parameter is passed to the method. A method can be called without passing parameters by appending an opening and closing parenthesis to the method identifier. When passing multiple parameters, these have to be named, as seen in Listing 4.4. The parameters, represented by key-value-pairs, have to be separated by commas. Appending a trailing comma after the last parameter is allowed. To increase the readability, a step definition can span multiple lines.

Listing 4.4 Multi-line step definition in a Jenkinsfile using the simplified syntax

```
1 mail to: 'some@one.com',
2     subject: "Failed Pipeline: ${currentBuild.fullDisplayName}",
3     body: """
4 Hi Someone,
5
6 Something is wrong with ${env.BUILD_URL}!
7 """
```

Some steps, however, do not define a symbol that allows using the simplified syntax. These have to be included in the pipeline using the *general build step*. It uses a Java-style method call syntax with parentheses, as demonstrated in Listing 4.5. The example shows the instantiation of a step for capturing code coverage reports[4]. The parameter `$class` defines the name of the plug-in class to use for building the step. All remaining parameters are specific to the step and can be found in the individual documentation.

Listing 4.5 The general build step

```
1  step([
2      $class: 'CloverPublisher',
3      cloverReportDir: env.WORKSPACE + '/build/reports/clover',
4      cloverReportFileName: 'clover.xml',
5      healthyTarget: [
6          methodCoverage: 70,
7          conditionalCoverage: 80,
8          statementCoverage: 80,
9      ],
10 ])
```

[4] Clover Plugin – https://wiki.jenkins.io/display/JENKINS/Clover+Plugin

Listing 4.6 The script environment in a Jenkinsfile

```
1 script {
2     def branches = ['master', 'dev-1', 'dev-2']
3     for (i = 0; i < branches.count(), i++) {
4         println "Testing branch ${branches[i]}"
5     }
6 }
```

4.5.2 String Values

Most parameter values are provided as strings, which are opened and closed by single or double quotation marks. Quotation marks can be used inside a string by escaping them with a backwards slash. Double quotation marks allow to output variables within the string using a dollar sign followed by an environment, opened and closed by curly braces (see Listing 4.4, lines 2 and 6). This is not allowed inside a string using single quotation marks.

A regular string cannot span multiple lines. In Groovy, multi-line strings are surrounded by three single or double quotation marks (see Listing 4.4, lines 3-7). Here, too, double quotation marks allow outputting variables.

Additionally, it is possible to concatenate strings using a plus sign. A string can also be concatenated with variables, numbers, and the results of function calls.

4.5.3 Groovy Scripts

Despite the large variety of available step implementations, some requirements may only be solved by individual implementations. For this purpose, the script environment can be used as a step inside a Jenkinsfile. A script can contain any valid Groovy program as demonstrated in Listing 4.6.

4.6 Resilience Features

The given analysis of the declarative Jenkinsfile syntax gives answers to RQ 1: a pipeline or stage can have assigned options, including a number of retry, telling Jenkins to repeat the concerned element for a given number of times. Using this option, it is possible to handle temporary failures. For instance, if a link in the network connection fails but recovers a few moments later, it is not necessary to trigger a failure of the pipeline. Repeating the failed action after a short waiting time is sufficient.

Another approach of ensuring the operability of the CD pipeline is using conditionally executed stages. Alternative paths can be defined that compensate failures of each other. For example, multiple deployment stages could be configured for using different registry servers. If the regular deployment stage fails, an alternative stage can try pushing the application to a fallback registry.

Conditionally executed post steps are not usable for ensuring operability without restriction. It is possible to control the sequence flow in order to retry a stage or to trigger another one upon a failure, but this would require to violate the principle of declarative programming. Thus, post steps can perform clean-up actions upon a failure but are not capable of implementing more complex decisions regarding the sequence flow.

5 StalkCD

To achieve the goal of transforming BPMN to a PDL of a CD tool and vice versa, both fields have to be made compatible to each other. Obviously, BPMN is more generic and by far richer than any PDL, which are designed for one specific task, namely automating software delivery processes. Therefore, it is necessary to select a subset of BPMN that fits the concepts of the CD domain. Not all elements of the notation have equivalent features in CD tools and a feature from a CD tool can be formulated in BPMN by using various elements. This chapter will present a solution of bringing together the two domains by providing a reusable DSL that is helpful when implementing transformations from BPMN to PDLs and vice versa, as explained in Section 1.2.

A new DSL *StalkCD* has been developed with the purpose of bridging the functional gap between BPMN and the CD domain. It should be generic enough to be modeled using BPMN without being too restrictive. At the same time, it should be capable of representing a Jenkinsfile without information loss.

StalkCD is based on YAML [BEI05], a popular, easy-to-read markup language that is well-suited to serve as a data store in conformance with the concept of "Infrastructure as Code" (IaC). The language has been developed based on the concepts found in the Jenkinsfile (see Chapter 4). However, it is less restrictive to be easily transformable to BPMN, which makes it necessary to validate an instance of it before it can be transformed back to a Jenkinsfile. But since BPMN is richer than Jenkinsfiles, a validation of BPMN models as source for Jenkinsfiles is necessary anyway, hence this does not introduce much additional complexity.

5.1 Relation to Jenkins Pipelines

Most concepts found in a Jenkinsfile can be found in StalkCD, too. The only conceptual difference from the Jenkinsfile DSL is the design of parallel stages. Instead of using a dedicated section for the parallel execution of stages, in StalkCD, a stage can have the attribute *parallel* to indicate that the contained stages are to be executed in parallel. This simplifies the meta-model and also the transformation to BPMN.

The information contained in many Jenkinsfile sections is atomic in terms of the StalkCD DSL, i. e., it is not necessary to parse the instructions found there in order to transform them to StalkCD. For example, triggers have no dedicated corresponding data structure in StalkCD and are stored as an array of strings. Equally, there are numerous other entities in StalkCD, distinguishable by their data type `string[]`. The array of strings contains the unmodified instructions from the Jenkinsfile.

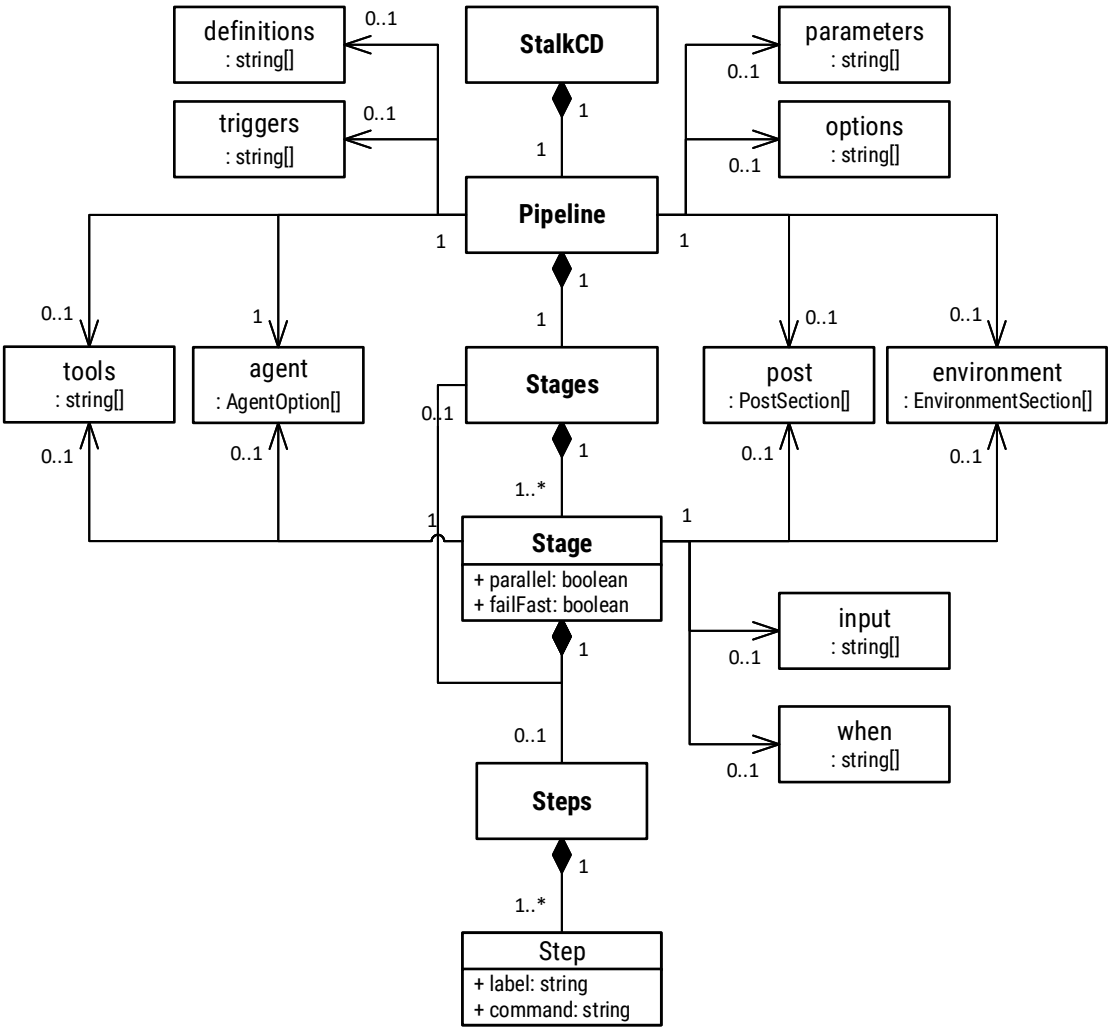


Figure 5.1: The meta-model of the StalkCD DSL

Obviously, the central CD elements such as the various sections, stages or steps, have corresponding structures in StalkCD. These are explained in more detail in the remainder of this chapter.

5.2 Language Design

In Figure 5.1, the meta-model of the StalkCD DSL can be found. Its entities and their purpose will be explained in the following.

5.2.1 Pipeline Definitions

Each StalkCD file has to contain exactly one pipeline. An example is given in Listing 5.1. Note that in YAML, quotation marks are escaped by duplicating them. A pipeline has the atomic sections *definitions*, *triggers*, *parameters*, *options* and *tools*, each implemented as array of strings as explained above. The items of the *definitions* section contain Groovy definitions that can be used in the steps of the pipeline. In the Jenkinsfile, they have to be placed in front of the actual pipeline definition. All remaining atomic sections retain the semantics of their equivalent Jenkinsfile elements (see Chapter 4). The remaining non-atomic sections, having dedicated structures in StalkCD, are described in the following.

Listing 5.1 An example StalkCD file

```
definitions:
  - def exampleFunction() { return "Example" }
  - def exampleProperty = true
agent:
  - name: docker
    options:
      - name: image
        value: '''openjdk:8-jdk-alpine'''
tools:
  - maven 'apache-maven-3.0.1'
environment:
  - name: XL_DEPLOY_CREDENTIALS
    value: credentials("xld-credentials")
  - name: XL_DEPLOY_USERNAME
    value: '"${env.XL_DEPLOY_CREDENTIALS_USR}"'
options:
  - 'timeout(time: 30, unit: "MINUTES")'
parameters:
  - 'string(name: "Name", description: "What is your name?")'
triggers:
  - pollSCM('H */4 * * 1-5')
stages:
  - name: Build
    when: branch 'master'
    steps:
      - label: echo
        command: echo "Build step."
post:
  always:
    - label: Remove test output
      command: deleteDir '/tmp/test'
```

5.2.2 Agent Section

The environment in which a pipeline or stage has to be executed can be specified in their corresponding agent sections. In StalkCD, such a section is designed as array of `AgentOption` elements (see Figure 5.2). A concrete option can either be an `AgentValue` having the string properties `name` and `value`. Or it can implement the class `AgentSection` containing nested values in the `options` array. This allows modeling both, simple values such as `any` or `none`, and complex structures as seen in Figure 5.1 (lines 4 to 8).

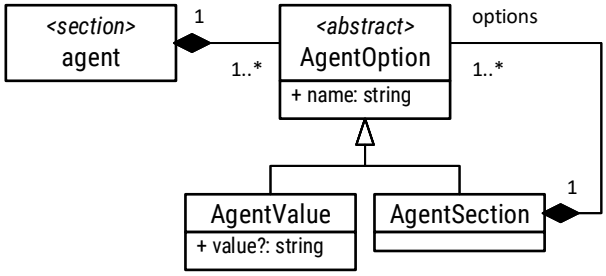


Figure 5.2: Class diagram of the agent section in StalkCD

5.2.3 Environment Section

The environment section contains an array of assignments to variables that are made available to the programs called by steps in the concerned pipeline or stage. An assignment consists of a variable name and a value, as seen in Figure 5.1 (lines 11 to 15). The value of an assignment contains an expression that is evaluated by Jenkins during the execution of the pipeline. This means that constant string values have to be surrounded by quotation marks. Note that the YAML notation requires quotation marks itself if a value contains such characters. This might make it necessary to escape them, e. g., the value `'some string value'` would be represented in YAML as `'''some string value'''`.

5.2.4 Post Section

The post section contains steps that are conditionally executed based on the result of the concerned pipeline or stage. For each post condition available in the Jenkinsfile, it has an array of steps (see below), as depicted in Figure 5.3.

5.2.5 Step

The StalkCD step is a wrapper around a Jenkinsfile step, i. e., the complete Jenkinsfile expression is contained in the equivalent StalkCD file without modifications. This preserves the core features of Jenkins and makes them usable in StalkCD without the need for parsing and translating step instructions. In addition, a StalkCD step can contain further information

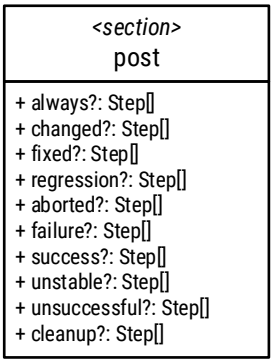


Figure 5.3: Class diagram of the post section in StalkCD

that is helpful when depicting it in a BPMN model. The `name` property (see Figure 5.1) contains a summary of the step's purpose that can be displayed more space-efficiently than the original expression.

6 Transformations

Based on the analysis of the Jenkinsfile syntax in Chapter 4 and the corresponding definition of the StalkCD DSL in Chapter 5, this chapter presents transformations from Jenkinsfile to StalkCD, as well as the reverse direction. Furthermore, it describes an approach of mapping a StalkCD pipeline to a BPMN model, which also results in an answer to RQ 2. Moreover, it shows how to convert a BPMN model back into a StalkCD pipeline. Figure 6.1 highlights that the transformations developed in this master's thesis enable a bi-directional conversion from Jenkinsfiles to BPMN and back. The software containing the implementation of all presented transformations is written in the TypeScript programming language and can be found in the supplementary material of this master's thesis [Kab19]. Its concepts are explained in the following sections.

6.1 Jenkinsfile to StalkCD

In order to transform a Jenkinsfile into a StalkCD file, it is necessary to process its content and to build corresponding data structures enabling to recognize and translate its language elements. Here, parsing Jenkinsfiles and understanding their semantics is the main challenge that has to be met. Literature review has returned no existing solutions for transforming Jenkinsfiles to another format or to process their contents. Additionally, even though Jenkins is open-source, its code is not reusable for the purposes of this master's thesis: The functionality of the pipeline plug-in is distributed across several repositories and there is no generic processor providing an API or other possibilities to transform a Jenkinsfile into a structured model.

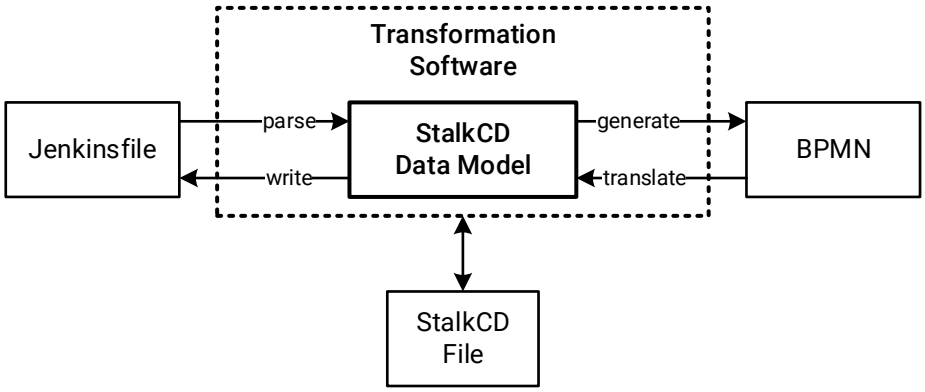


Figure 6.1: An overview over the implemented transformations

In this master's thesis, a text-parsing approach has been chosen to read and transform Jenkinsfiles. Based on Chapter 4, a context-free grammar has been developed that describes the exact structure of a Jenkinsfile and enables parsing and processing it. The grammar is written for ANTLR v4 [Par13], a popular open-source parser generator that helps reading and translating text files. With help of the antlr4ts utility[1], TypeScript code is generated out of the grammar, which can be used according to the visitor design pattern to process the elements that have been recognized. More precisely, the ANTLR parser is given a visitor object which has methods for possible components of the parsed data, defined as parser rules by the ANTLR grammar. For each matching rule, the parser calls the corresponding method on the visitor object, so that this can process the found element.

In the developed generator for transforming Jenkinsfiles to StalkCD files, the visitor object passed to the ANTLR parser is a StalkCD builder. It accepts Jenkinsfile elements and creates corresponding StalkCD objects. For instance, if the parser detects a stage section in the processed Jenkinsfile, it calls the corresponding method of the StalkCD builder. This creates a stage object with the given properties and appends it to the list of stages of the previously found pipeline section.

As soon as the parser has finished processing the input Jenkinsfile, the built data structure can be serialized. This is done by passing the cleaned data object built by the StalkCD builder to the library "js-yaml"[2]. The resulting string can be saved to a text file. The YAML library also serves as parser for the saved file, i. e., serializing and de-serializing StalkCD files is trivial in terms of the implemented transformation.

The grammar that serves as source for the generated Jenkinsfile parser consists of rules, defining the components of the input files to read. The capitalization of a rule name determines its type: If it starts with a small letter, then it is a parser rule. All rules starting with a capital letter are read as lexer rules. By convention, all parser rules of the developed grammar solely use small letters and an underscore as word delimiter, e. g., `groovy_definition`. Lexer rules are named using only capital letters, e. g., `STRING_LITERAL`. Fragment rules are named according to the CamelCase naming convention, e. g., `StringLiteralSingle`. Such fragments can be included in lexer rules, but are never counted as tokens themselves, i. e., their only purpose is to simplify the grammar, increasing its readability. The following subsections will present the rules of the developed grammar.

6.1.1 Section: Pipeline

The first element inside a Jenkinsfile is the pipeline section. Consequently, the entry point of the grammar is the `pipeline` rule, depicted in Figure 6.2. It starts with an arbitrary number of Groovy definitions or a title, followed by the pipeline keyword and opening and closing curly braces. The braces can contain various sections, defining the properties of the pipeline. The implementation of the components this parser rule consists of, will be presented in the following.

[1] antlr4ts - TypeScript/JavaScript target for ANTLR 4 – https://github.com/tunnelvisionlabs/antlr4ts
[2] JS-YAML - YAML 1.2 parser / writer for JavaScript – https://www.npmjs.com/package/js-yaml

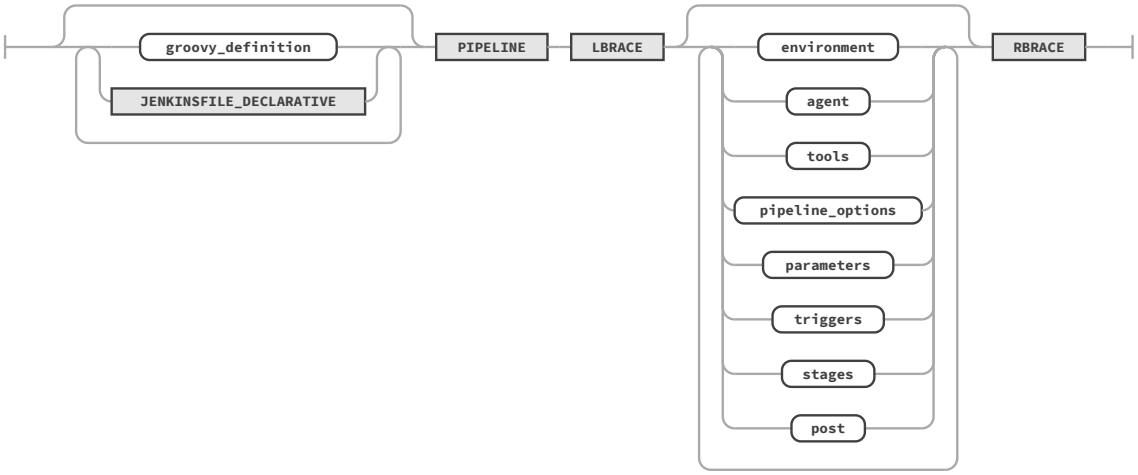


Figure 6.2: The `pipeline` rule as railroad diagram

Groovy definitions

Groovy definitions can be library imports, function specifications or variable declarations. Steps inside the pipeline can use these definitions in order to reduce their complexity by reusing functionality. The `groovy_definition` rule (see Figure 6.3) aims to exactly persist the sequence of characters, forming a Groovy definition. The use of lexer rules for detecting Groovy statements makes it possible to precisely store and reproduce the found statements, without loosing information about whitespace or other elements that are consumed by the lexer. This does not enable processing or executing the individual components of the statements, however this is not a requirement of the developed transformation. Basic support of Groovy definitions is sufficient to transform a Jenkinsfile to StalkCD, hence implementing a comprehensive parser for the Groovy language would not create additional value.

Figure 6.3: The `groovy_definition` rule as railroad diagram

A detectable Groovy definition has to consist of either a `DEF_LITERAL` or a `LIBRARY_LITERAL`. A `DEF_LITERAL` (see Figure 6.4) has to start with the word “def”, followed by whitespace and an identifier, represented by the fragment rule `DefLiteralId`. This identifier can be followed by an equals sign, indicating that the definition is a variable declaration. If an opening parenthesis is next to the identifier, a method declaration is found. Parameter definitions (fragment `DefLiteralParam`) may be inside the found parentheses, which is implemented in the grammar as sequence of any characters except a closing parenthesis. If a closing parenthesis is part of a parameter definition, the grammar rule is likely to fail, i. e., this case is not supported. Jenkinsfiles using such constructs will not be read correctly.

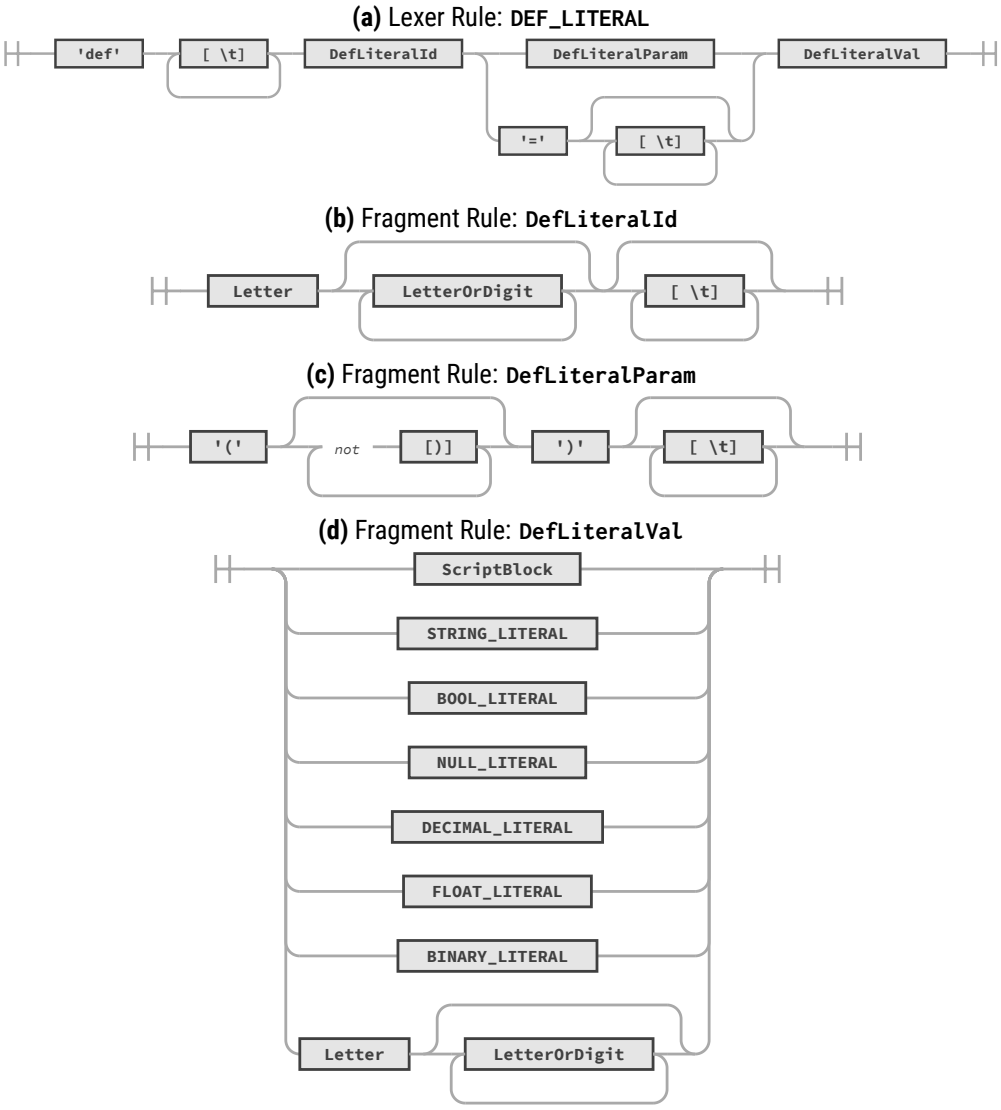


Figure 6.4: The `DEF_LITERAL` lexer rule as railroad diagram

The `LIBRARY_LITERAL` rule (see Figure 6.5) matches Groovy library imports in the parsed Jenkinsfile. They have to start with the string "@Library" followed by parentheses containing a string literal (see Section 6.1.13) that specifies the name of the imported library. The statement can be followed by whitespace and an underscore character. This indicates the import of all modules inside the specified library.

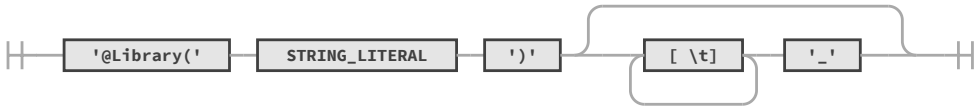


Figure 6.5: The `LIBRARY_LITERAL` lexer rule as railroad diagram

Jenkinsfile Title

Investigations have shown that many Jenkinsfiles start with the line “Jenkinsfile (Declarative Pipeline)”. This title is not a valid part of the Jenkinsfile and obviously originates from a mistake by the developers of the pipelines when copying examples from the official web site of Jenkins. Because this mistake can be found in numerous files, the transformation is designed to ignore it. For this reason, the lexer rule `JENKINSFILE_DECLARATIVE` (see Figure 6.6) matches the concerned line of text.

Figure 6.6: The **`JENKINSFILE_DECLARATIVE`** lexer rule as railroad diagram

6.1.2 Section: Stages

The stages section inside the pipeline or a stage contains a sequence of stages. As depicted in Figure 6.7, the corresponding parser rule expects either the keyword “stages” or “parallel” followed by curly braces. Inside the braces, stage definitions have to be placed, which are explained in more detail in Section 6.1.3.

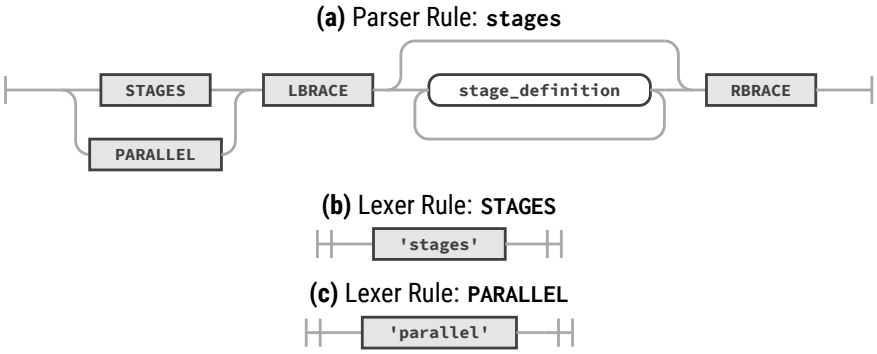


Figure 6.7: The **`stages`** parser rule as railroad diagram

6.1.3 Section: Stage

A stage can have a name inside parentheses after the initial keyword “stage”, as seen in Figure 6.8. The corresponding parser rule `stage_name` is an alias for `STRING_LITERAL`, which is explained in more detail in Section 6.1.13. By using a dedicated rule for matching the name of the stage, a Jenkinsfile visitor can distinguish between the string literal specifying the stage name and such string literals located elsewhere in the stage definition. Inside of curly braces, all relevant properties can be specified using appropriate sections and statements. The relevant rules will be presented in later sections.

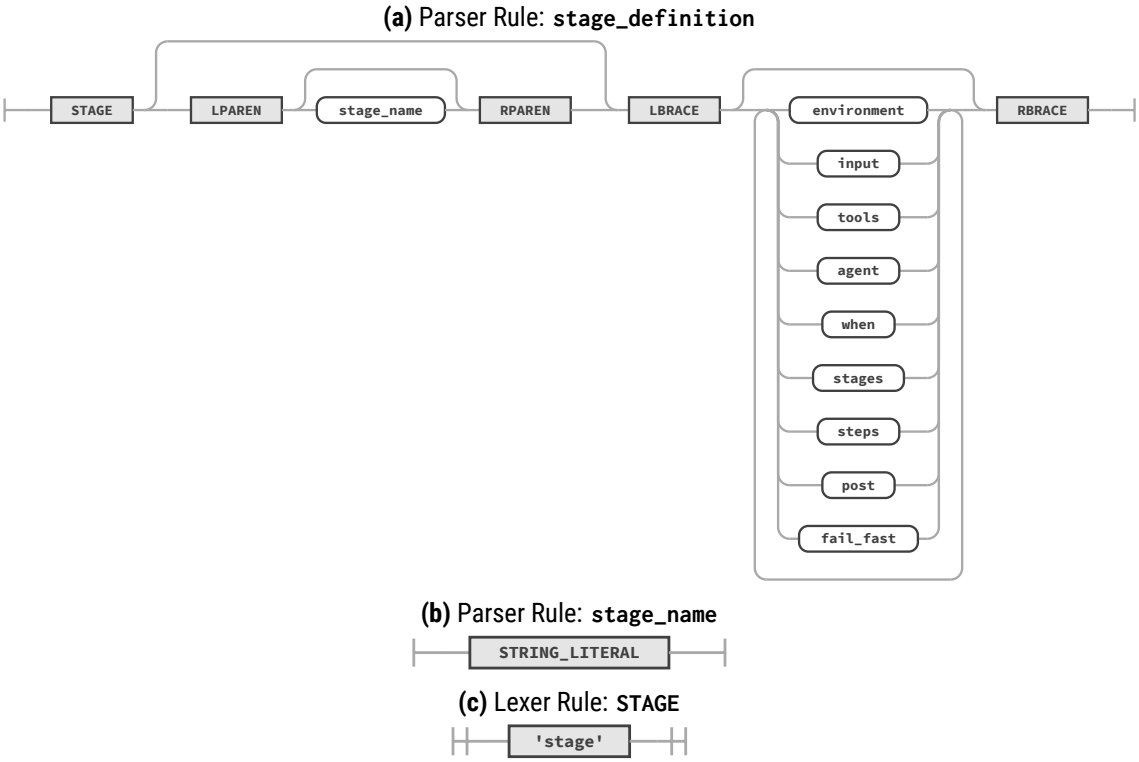


Figure 6.8: The `stage_definition` parser rule as railroad diagram

A special statement, only available inside of a stage, is the `failFast` attribute. It tells Jenkins not to wait for the result of concurrent stages as soon as one in parallel executed one failed. The parser rule `fail_fast` (see Figure 6.9) matches the keyword "failFast" and expects a subsequent boolean expression. The lexer rule `FAIL_FAST` accepts the keyword with and without the capital letter "F".

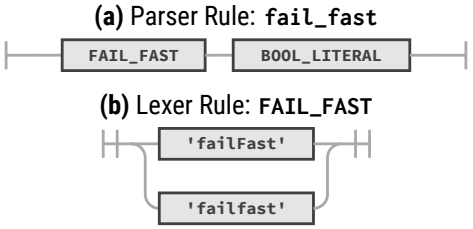


Figure 6.9: The `fail_fast` parser rule as railroad diagram

6.1.4 Section: Steps

The steps section contains any number of step definitions, each performing an action. The `steps` parser rule, shown in Figure 6.10, matches the keyword "steps" followed by curly braces containing steps (see Section 6.1.5).

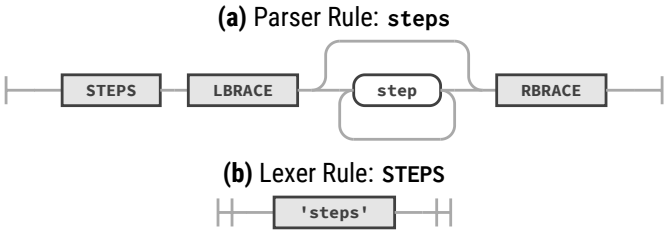


Figure 6.10: The `steps` parser rule as railroad diagram

6.1.5 Step Definitions

The syntax of a step definition in general equals the simplified method call syntax of Groovy. For this reason, the method call rule, described in Section 6.1.11 is used in the `step` rule shown in Figure 6.11.

Figure 6.11: The `step` parser rule as railroad diagram

Additionally, a step can be a Groovy script, which is recognized by the `script` rule, depicted in Figure 6.12. It matches the token produced by the lexer rule `SCRIPT_LITERAL`. The recognition of scripts is fully implemented using lexer rules in order to preserve whitespace characters and other structures that are removed during the lexical analysis. A script step has to be initiated by the keyword “script”, followed by a script block.

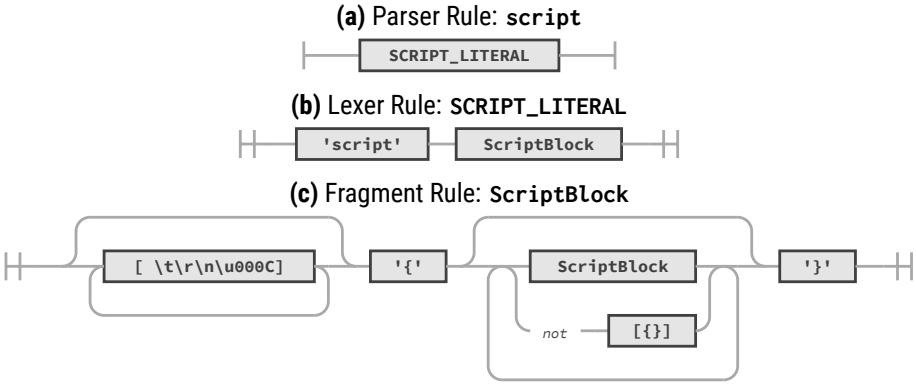


Figure 6.12: The `script` parser rule as railroad diagram

The `ScriptBlock` fragment matches a sequence of any characters surrounded by curly braces. The initial expression `[ \t\r\n\u000C]` allows whitespace characters to be placed in front of the opening brace of the code block. The code block itself either has to consist of any characters except for opening and closing curly braces, or of other nested script blocks, i. e., the number of opening and closing curly braces inside the script has to be even. This enables recognizing functions, if-statements and other Groovy constructs that are surrounded by

curly braces. However, this method might lead to problems, if a single curly brace without closing counterpart is contained in the script, e. g., as part of a string or comment. In such a case, the designed lexer would not recognize the code block correctly. This drawback is accepted at this point, as the only solution would be the development of a comprehensive Groovy parser, which is not in scope of this master's thesis.

6.1.6 Section: Environment

The `environment` rule (see Figure 6.13) reads the environment settings of a pipeline or stage. It is initiated by the keyword "environment" and opening and closing curly braces. These contain any number of assignments.

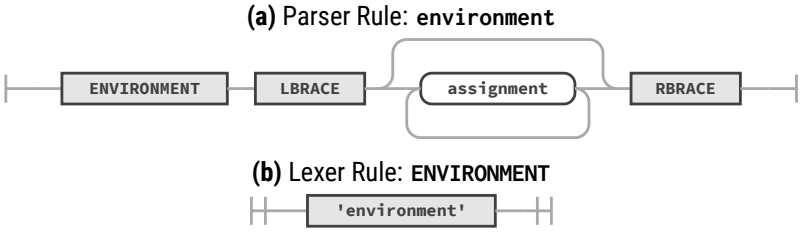


Figure 6.13: The **environment** parser rule as railroad diagram

The `assignment` rule (see Figure 6.14) consists of a key (rule `assignment_key`), the assign sign "=" and an expression. The `assignment_key` rule is an alias for the `identifier` rule which is explained in Section 6.1.12. The implementation of the `expression` rule can be found in Section 6.1.10.

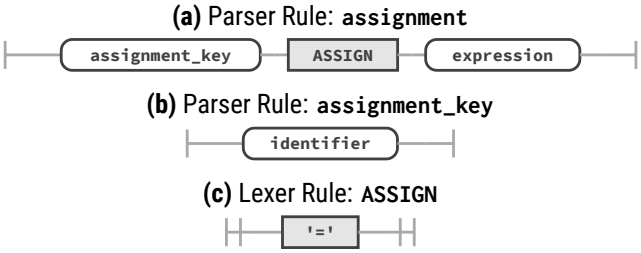


Figure 6.14: The **assignment** parser rule as railroad diagram

6.1.7 Section: Agent

The agent of the pipeline or a stage can be defined by either one word, e. g., "any" or by using a section containing more settings, as seen in the example in Listing 6.1. An agent section is matched by the parser rule `agent_section` (see Figure 6.15). It either consists of method calls or a named section of method calls, where the section name defines the type of agent to use, e. g., "docker". The concept of method calls can be used at this place because of the syntactical equality of the agent options and the simplified method call syntax of

Listing 6.1 Example agent sections

```
// Simple agent specification
agent any

// Only method calls
agent {
    docker 'python:3.5.1'
}

// Named section
agent {
    docker {
        image 'python:3.5.1'
    }
}
```

Groovy (see Section 6.1.11). The rule `agent_type` is an alias for the `identifier` rule (see Section 6.1.12). By introducing a dedicated rule for matching the agent type, a Jenkinsfile visitor can determine if an identifier is part of a method call or specifies the agent type.

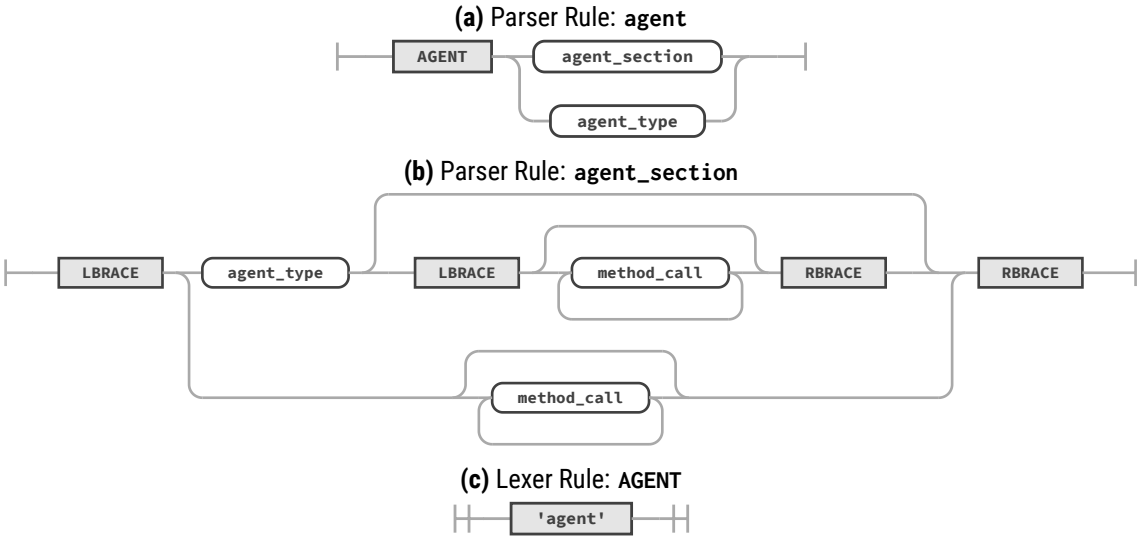


Figure 6.15: The `agent` parser rule as railroad diagram

6.1.8 Sections: Tools, Options, Parameters, Triggers

The sections `tools`, `options`, `parameters` and `triggers` in a Jenkinsfile pipeline have equivalent syntactical layouts. They are initiated by their corresponding keyword, followed by curly braces containing method calls. Hence, their grammar rules are almost equivalent and only differ in the initial keyword. As representative of the mentioned sections, the `tools` parser rule is depicted in Figure 6.16.

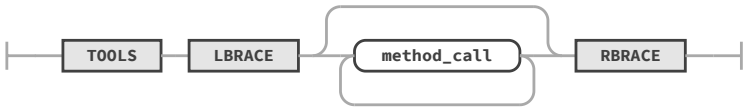


Figure 6.16: The `tools` parser rule as railroad diagram

6.1.9 Section: Post

The post section contains steps that are conditionally executed based on the result of the stage or pipeline they are assigned to. The parser rule `post` matches the keyword "post" and a subsequent section containing post conditions, as seen in Figure 6.17. The `post_condition` rule expects an identifier (see Section 6.1.12), followed by curly braces containing steps (see Section 6.1.5).

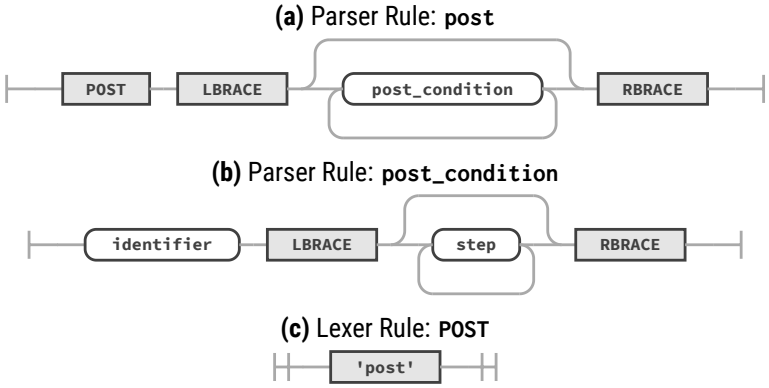


Figure 6.17: The `post` parser rule as railroad diagram

6.1.10 Expressions

The expression is a core element of any programming language. It can be a static value, such as a number or text. It can be the assignment to a variable that consists of nested expressions defining target and value. An expression can be the call to a function, a condition query, the increment of a variable and so on. In order to cover all possible constructs, the grammar rule accepting expressions has to be comparatively complex.

A comprehensive ANTLR grammar for Groovy expressions can be found in the official source code repository of the programming language[3]. However, this grammar heavily relies on Java extensions, which are included in the grammar files. Using rules out of it in the developed grammar would require to add a large amount of code, which would unnecessarily increase its complexity. The solution depicted in Figure 6.18 has been derived from the expression grammar rule for the Java programming language[4], which is by far

[3] Apache Groovy – https://git-wip-us.apache.org/repos/asf?p=groovy.git;a=tree;f=src/antlr

[4] https://github.com/antlr/grammars-v4/blob/ce7cd542c77add5fd0949f36a250d35b14e61ac9/java/JavaParser.g4#L468

Listing 6.2 Example of an expression list in a Jenkinsfile

```
step([
    $class: 'CloverPublisher',
    cloverReportDir: env.WORKSPACE + '/build/reports/clover',
    healthyTarget: [
        methodCoverage: 70,
    ],
])
```

less complex. This is possible, because Groovy is based on Java and is executed in the "Java Virtual Machine (JVM). Most constructs allowed in Java are possible in Groovy as well. It is acceptable if some Groovy expressions cannot be recognized by the Jenkinsfile parser.

The `expression` parser rule depicted in Figure 6.18 matches expressions allowed inside a Jenkinsfile. An important component is the `primary` rule, as seen in Figure 6.19. It enables parenthesizing expression parts to form logical groups. Furthermore, it matches literals and identifiers, which are essential elements of any Jenkinsfile. For instance, a static string value as argument of a method call is resolved as expression, primary, and finally as string literal.

The `expression` rule allows some constructs that are not permitted by the Java syntax: An expression can start with a square bracket to begin a list of expressions, delimited by commas (see rule in Figure 6.20 and corresponding example in Listing 6.2). This enables recognizing Groovy map literals[5]. Furthermore, the rule allows to provide key-value-pairs, separated by colons, as also seen in Listing 6.2, which may be part of a map literal. Literals and identifiers are introduced in Section 6.1.13 and Section 6.1.12, respectively.

The implementation of expression lists might allow constructs that would not be accepted by the Groovy compiler, because it allows expressions of different types to be part of the same expression list. For instance, the expression list `[a: 'A', x++]` would be recognized by the presented grammar rule, but is no valid Groovy code. However, since the purpose of the transformation is persisting existing functionality and not syntax validation or execution, a permissive parser is acceptable.

Note, that method calls as part of an expression have to use the Java-style syntax. The simplified syntax of Groovy is not permitted here, because it would lead to ambiguities. The remaining alternative constructs in the depicted `expression` rule are also available in Java and will not be explained in detail here.

6.1.11 Method Calls

The method call is a fundamental concept inside a Jenkinsfile. Steps and the properties of most sections are in general specified using the simplified method call syntax of the Groovy programming language. Some examples of method calls are given in Listing 6.3.

[5]The Groovy Map literal – http://groovy-lang.org/groovy-dev-kit.html#_map_literals

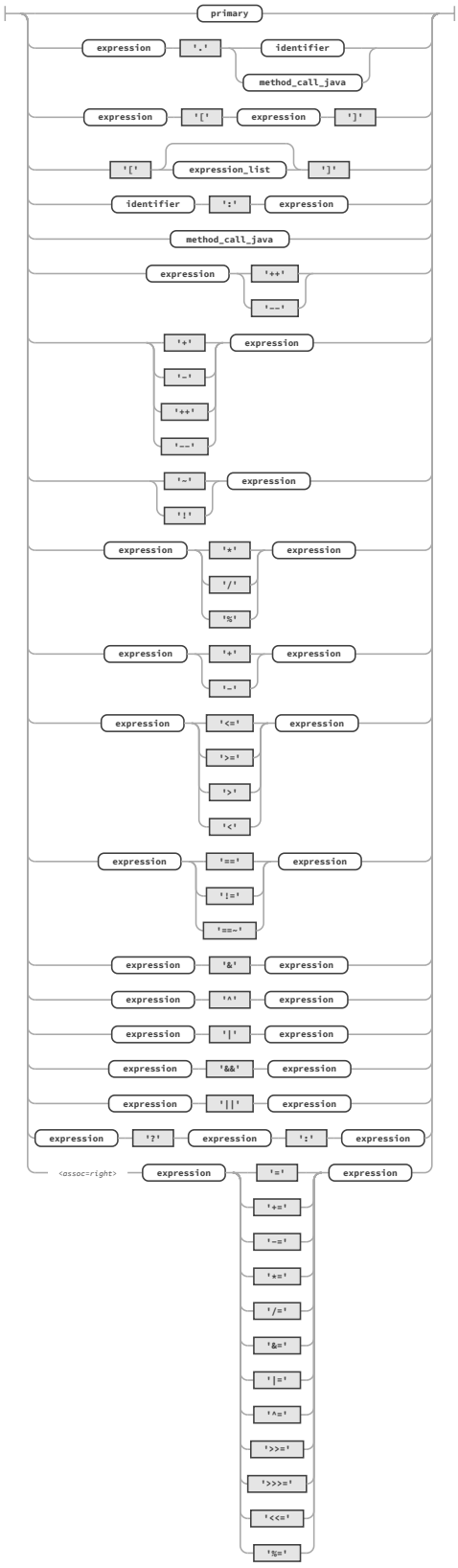


Figure 6.18: The `expression` parser rule as railroad diagram

Figure 6.19: The `primary` parser rule as railroad diagram

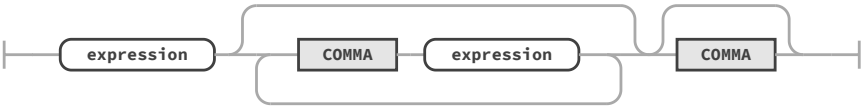


Figure 6.20: The `expression_list` parser rule as railroad diagram

Example 1 shows the most primitive type of method calls, only having one parameter without parentheses. Example 2 contains a simplified method call with multiple named parameters. In Example 3, a more complex method call with parentheses is presented. Finally, Example 4 shows a method environment, containing further nested method calls. All these structures have to be supported by the developed grammar rule.

Figure 6.21 shows the parser rule `method_call`, which defines the basic structure of a method call. It can have the simplified or Java-style syntax or it can be a method environment. Optionally, it can be terminated with a semicolon.

The parser rules for detecting the simple and Java-style syntax of method calls are depicted in Figure 6.22. Both start with an identifier, specifying the name of the called method. The rule `method_call_simple` accepts an immediately following list of method arguments

Listing 6.3 Example method calls in a Jenkinsfile

```
1  // Example 1: Simple step
2  echo 'Hello World!'
3
4  // Example 2: Named parameters
5  mail to: 'some@one.com' subject: 'Hello World!'
6
7  // Example 3: Extended Java-style method call
8  // ␣␣ with optional trailing semicolon
9  step([$class: 'TestingClass', param: 'Value']);
10
11 // Example 4: Method environment
12 withCredentials([
13 ␣␣usernamePassword(
14 ␣␣␣␣credentialsId: "docker",
15 ␣␣␣␣usernameVariable: "USER",
16 ␣␣␣␣passwordVariable: "PASS"
17 ␣␣)
18 ]) {
19 ␣␣sh 'docker login -u $USER -p $PASS'
20 }
```

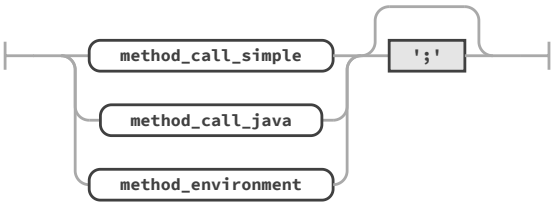


Figure 6.21: The `method_call` parser rule as railroad diagram

(divided by whitespace characters, which have been removed during the lexical analysis). The `method_call_java` rule expects the list of arguments to be surrounded by parentheses. Only the Java style allows to call a method without passing an argument.

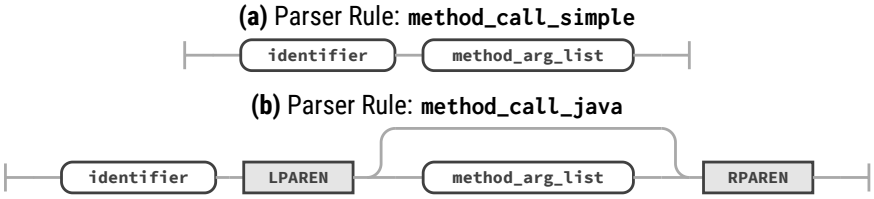


Figure 6.22: The parser rules for the different method call styles as railroad diagrams

A list of method arguments (see rule `method_arg_list` in Figure 6.23) has entries divided by commas and optionally is ended by a comma. Method arguments are recognized by the parser rule `method_arg`, which expects either an expression representing the argument value (see 6.1.10) or a named argument consisting of a `method_arg_key` (alias for `identifier`), a colon and an expression.

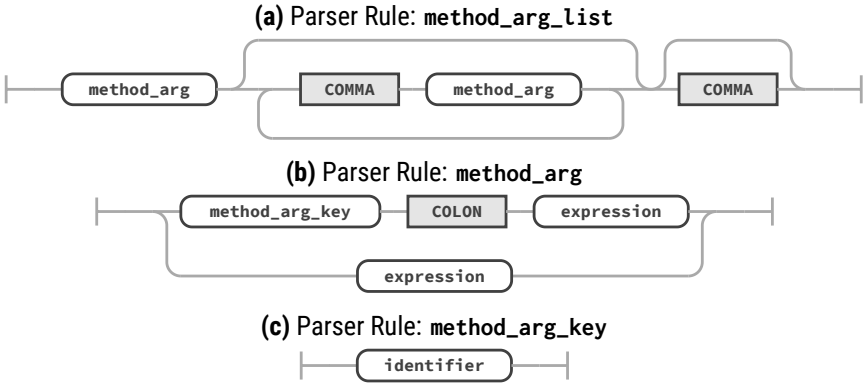


Figure 6.23: The `method_arg_list` parser rules as railroad diagram

Figure 6.24 shows an example syntax tree, which results when parsing the given input string using the previously presented grammar rules. In this specific example, the grammar detects a Java-style method call with one argument. The argument is recognized by the `expression` rule as expression list surrounded by square brackets. The list consists of two key-value-pairs representing named arguments.

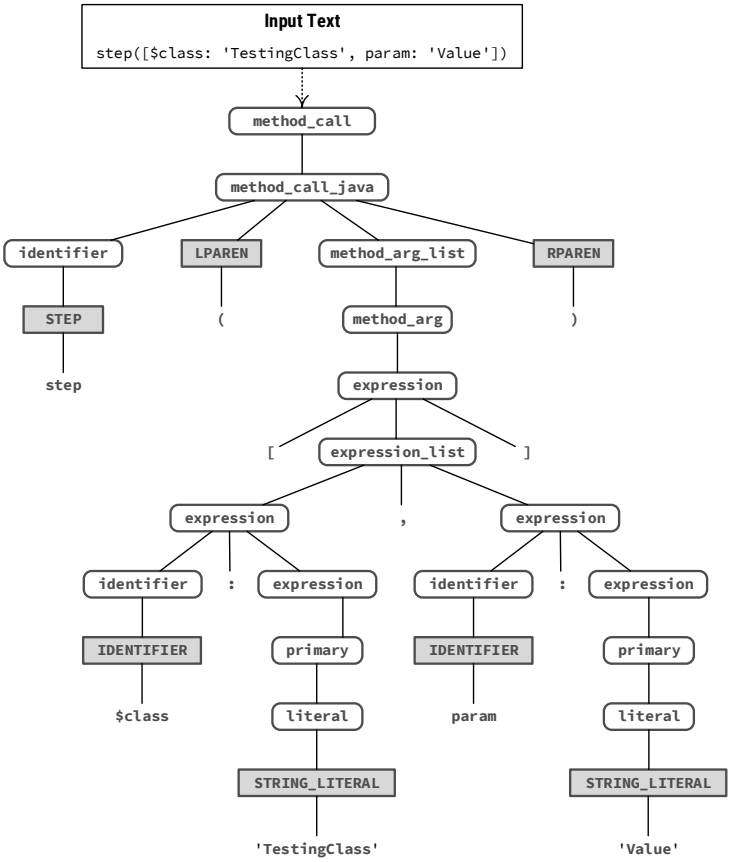


Figure 6.24: An example syntax tree demonstrating the functionality of the grammar

Besides from simple and Java-style method calls, the `method_call` rule also recognizes method environments. The `method_environment` parser rule, depicted in Figure 6.25, matches a Java-style method call, followed by curly braces. These define an environment containing steps, which can make use of the preparations of the initial method call. An example for a method environment can be found in Listing 6.3 on page 59. Here, the "withCredentials" method loads credential data that can be used by steps inside the method environment.

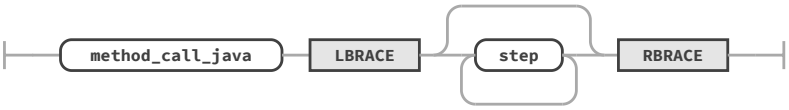


Figure 6.25: The **`method_environment`** parser rule as railroad diagram

6.1.12 Identifiers

An identifier is the name of a variable, function or field and may consist of small and capital characters, underscores, and digits. Additionally, in Jenkinsfile, some identifiers may start with a dollar sign. In most programming languages, identifiers are not allowed to equal language specific keywords. This however is possible in Jenkinsfiles, which makes it

necessary to implement the identifier rule using a parser rule. The lexer cannot determine if a found keyword is part of an expression and has to be assigned the identifier token. For this reason, the `identifier` parser rule matches the `IDENTIFIER` token and all available keywords, such as `PIPELINE`, `STAGES`, `STAGE`, `ENVIRONMENT` and so on. For a comprehensive listing of allowed keywords, see Appendix A on page 93.

6.1.13 Literals

String values, numbers, and similar expressions inside a Jenkinsfile, which are static and defined by the developer, are also known as literals. There are several types of literals, as seen in Figure 6.26. Most grammar rules for the detection of literals equal the definitions in the ANTLR grammar for Java[6], except for the string and regex literals.

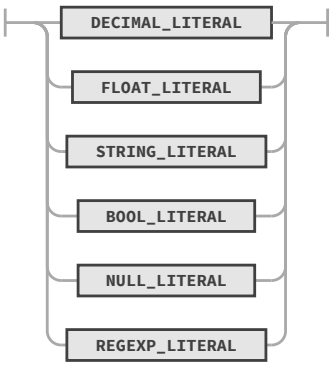


Figure 6.26: The `literal` parser rule as railroad diagram

In Groovy, string literals started with double quotation marks enable outputting variables using an escape sequence, e. g., `"Name: ${name}"`, whereas this is not permitted inside single quotation mark string literals. Additionally, it is possible to define multi-line strings using three quotation marks as starting and ending sequence. This is valid for both, single and double quotation marks. The grammar rules recognizing string literals in Jenkinsfiles are depicted in Figure 6.27. The rules `StringLiteralSingle` and `StringLiteralDouble` recognize sequences of any characters except for the quotation mark they were initiated with and line feed characters. To enable including quotation marks in the string literal, that normally would terminate it, the lexer rules also match escape sequences, defined in the fragment rule `EscapeSequence`, which has been taken from the Java ANTLR grammar as well (see Appendix A on page 93).

Furthermore, the Groovy syntax specification allows literals for regular expressions. They are started and ended with a forward slash and contain a sequence of any character, except for forward slashes and line feeds. The grammar rule recognizing regular expressions is depicted in Figure 6.28. Escaped forward slashes are matched by the included fragment rule `EscapeSequence`.

[6] https://github.com/antlr/grammars-v4/blob/9931f78ad75b4d0fbbba65c77d5981edb01b7f44/java/JavaLexer.g4#L85

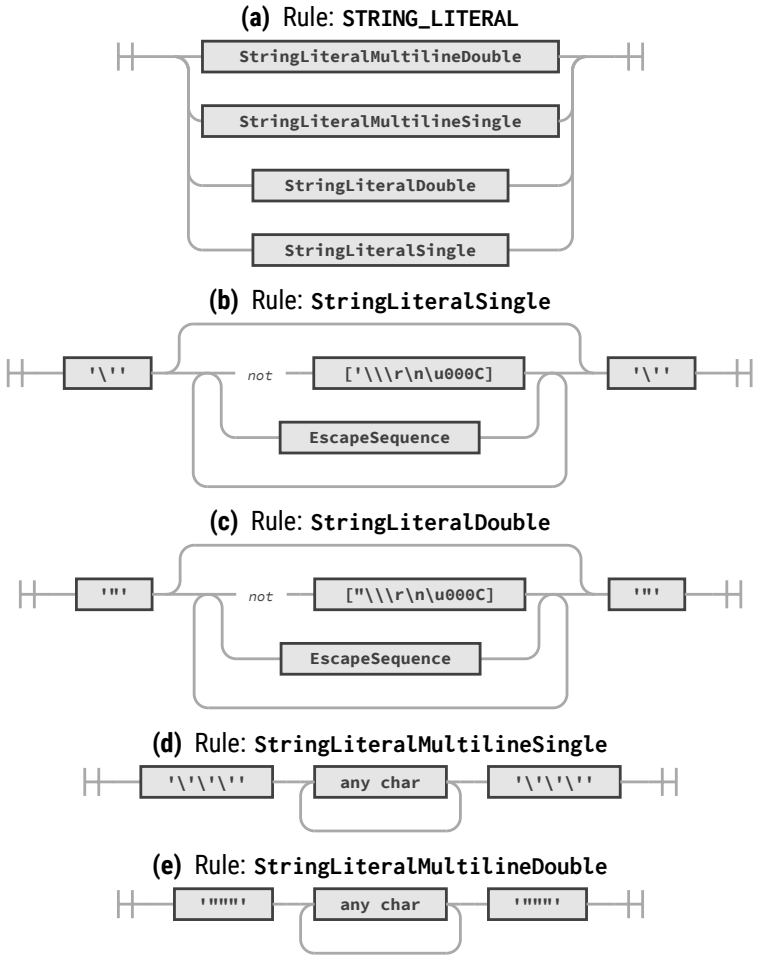


Figure 6.27: The `STRING_LITERAL` lexer rule as railroad diagram

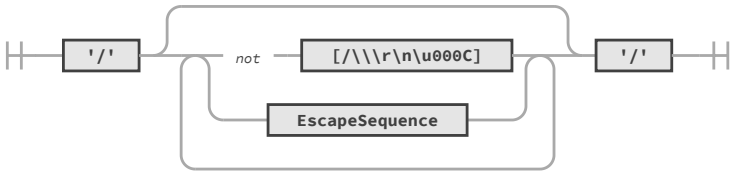


Figure 6.28: The `REGEXP_LIERAL` lexer rule as railroad diagram

6.2 StalkCD to Jenkinsfile

In order to transform a StalkCD file into a Jenkinsfile, the first step is to load the source file into the developed transformation software. As mentioned in Section 6.1 on page 48, parsing a StalkCD file is trivial, as the library “js-yaml” does the job of translating a StalkCD file written in YAML into data objects. The objects have the structure of the input YAML file and can be directly used in within the software.

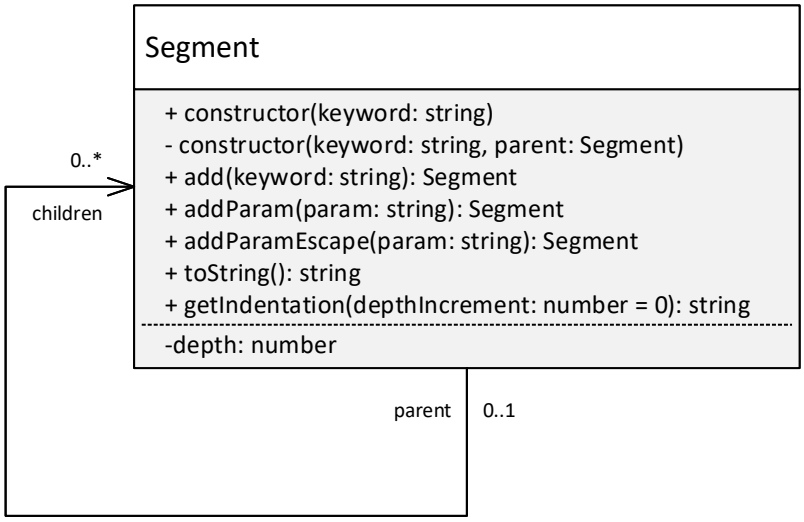


Figure 6.29: The `Segment` class as part of the Jenkinsfile builder

Writing a Jenkinsfile is comparatively trivial as well, since all data objects can simply be iterated and converted to their string representation. The developed Jenkinsfile writer takes a StalkCD pipeline object as input and passes all relevant properties to a Jenkinsfile segment builder, depicted as class diagram in Figure 6.29. If an element has child elements, these are added to its corresponding segment object. More precisely, the writer creates one root segment with the keyword "`pipeline`". All properties of the given StalkCD pipeline are added to the segment using its method `add()`. Some of the properties have nested components themselves, e. g., a stages section has stages. These, again, are added to the corresponding segment and so on.

The segment class allows to add parameters to the keyword, which can be used to build a method call that uses the simplified method call syntax of Groovy. When calling the method `addParam()`, the segment's keyword is appended by a space and the passed parameter. The method `addParamEscape()` additionally surrounds the given value with quotation marks and replaces such quotation marks inside the value by escape sequences, if applicable.

When all information of the StalkCD pipeline has been processed, the root segment is transformed to its textual representation by calling `toString()`. The result starts with its keyword and is appended by opening and closing curly braces. These braces are filled with the string representation of all child segments, each on a separate line. During their transformation to string, the child segments themselves add curly braces to their keyword if their children property has entries. If a segment has no children, then its string representation only contains its keyword.

Each segment has the property depth, which is the by one increased corresponding value of the parent segment. This property enables prepending indentation to the resulting string representation.

6.3 StalkCD to BPMN

To transform a StalkCD file into a BPMN model, matching BPMN constructs have to be defined that can be used to model StalkCD features. The goal is to use BPMN as graphical representation of a StalkCD file. The resulting BPMN model is not required to be executable by a general-purpose BPMN engine, which in principle makes it possible to use the full feature set of BPMN without the restrictions of a concrete implementation. However, since BPMN editors also often only support a subset of BPMN features, the constructs have to be selected such that they are compatible with an editor. For this master's thesis, the open-source Camunda Modeler[7] has been selected as editor. The chosen transformation only produces elements that can be visualized by this tool.

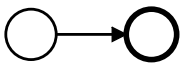
The following sections will present the developed transformation from StalkCD to BPMN. For all StalkCD features, matching BPMN constructs will be selected that can serve as representation in a BPMN model.

6.3.1 Pipeline

A StalkCD pipeline serves as section for stages and can have several properties. The BPMN process is ideally-suited to represent a pipeline in a BPMN model. It can contain sub-processes and other structures, representing its stages and properties. Figure 6.30 shows a minimal StalkCD pipeline and the corresponding BPMN model. It only consists of a start and end event, because the pipeline does not have any stages. The XML representation of this pipeline is shown in Listing 6.4. Initial comments, XML namespace definitions and layout information are excluded from the listing. The subsequent sections will show how the properties of a pipeline can be included in the BPMN process.

```
1 stages: { }
```

(a) The StalkCD example



(b) The resulting BPMN model

Figure 6.30: Mapping a pipeline to the BPMN process

6.3.2 Stage

A stage is a sequence of steps or further nested stages, which perform a logically separated piece of work. In BPMN, sub-processes provide a distinction between different sub-sets of activities, which makes them the logical choice for modeling a stage. For each stage in the transformed pipeline, a sub-process is created during the transformation. An example can be found in Figure 6.31, where a pipeline containing the stage "Build-Stage" is transformed into a sub-process. The sequence flow of the pipeline is directed top-to-bottom for space

[7] The Camunda BPMN / DMN Process Modeler – https://camunda.com/download/modeler/

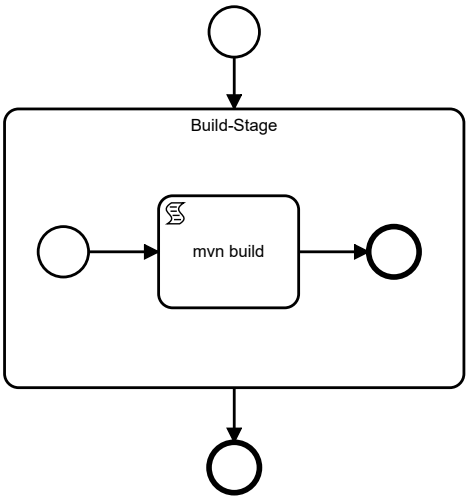
Listing 6.4 The XML representation of the example pipeline in BPMN

```
<bpmn:definitions>
  <bpmn:process id="Process_0000001" isExecutable="true">
    <bpmn:startEvent id="StartEvent_0000002">
      <bpmn:outgoing>SequenceFlow_0000004</bpmn:outgoing>
    </bpmn:startEvent>
    <bpmn:endEvent id="EndEvent_0000003">
      <bpmn:incoming>SequenceFlow_0000004</bpmn:incoming>
    </bpmn:endEvent>
    <bpmn:sequenceFlow id="SequenceFlow_0000004" sourceRef="StartEvent_0000002"
    ↪ targetRef="EndEvent_0000003" />
  </bpmn:process>
</bpmn:definitions>
```

efficiency reasons. In a regular BPMN model, it would be directed left-to-right. The sub-process has dedicated start- and end events, surrounding the sequence flow of the stage, which in this case consists of one step. The properties of a stage will be discussed in later sections.

```
- stages:
    - name: Build-Stage
      steps:
        - label: mvn build
          command: sh 'mvn build'
```

(a) The StalkCD example



(b) The resulting BPMN model

Figure 6.31: Mapping a stage to the BPMN sub-process

6.3.3 Step

The most primitive construct in StalkCD is the step. It represents an atomic piece of work that is part of a CD process. The equivalent BPMN feature is the task. Particularly the script task is well-suited to represent a CD step, as it can contain unstructured source code. When transforming a BPMN model to StalkCD and into a Jenkinsfile, the exact content of a script is preserved. This allows using the core features from Jenkins directly in BPMN.

In the graphical representation of BPMN, the source code of a script task is not visible, because it obviously would quickly lead to visual clutter. Instead, the code is only contained in the XML file and can be displayed by an editor, for instance, in a properties panel. The purpose of a StalkCD step can be summarized in its label property, which is used as name for the BPMN task. This name is visible in the BPMN model. Figure 6.32 shows an example StalkCD step and its corresponding BPMN representation. The model as XML code can be found in Listing 6.5.

```
1 - label: Run NPM Test
2   command: sh 'npm test'
```

(a) The StalkCD example

(b) The resulting BPMN model

Figure 6.32: Using BPMN tasks to model StalkCD steps

Listing 6.5 The XML representation of the example step in BPMN

```
1 <bpmn:scriptTask id="ScriptTask_0000001" name="Run NPM Test" scriptFormat="jenkins">
2     <bpmn:script>sh 'npm test'</bpmn:script>
3 </bpmn:scriptTask>
```

6.3.4 Agent

The agent settings of a pipeline or stage can be configured in a BPMN model using extension elements in the start event of the respective BPMN process or sub-process. The implemented transformation uses the `properties` extension element, defined by the Camunda Modeler. It is necessary to convert the agent settings from StalkCD into key-value pairs, which can be inserted in the properties list of the extension element. For each leaf node in the agent options hierarchy, a property key is constructed by concatenating "`agent`" with the names of all nodes in its path, separated by dots. For instance, an the agent option with the path `docker > image` and the value `'sample.agent'` would be converted into the key-value-pair `agent.docker.image: 'sample.agent'`.

An example of agent settings in a BPMN model can be seen in Listing 6.6. It shows the start event of a stage of the Kieker CD pipeline. The docker image, its label and additional docker arguments are specified in multiple property nodes. Also note, that variables can be used as value of a property, which makes it necessary to surround static string values by quotation marks.

6.3.5 Conditional Stages

The execution of a stage can be bound to `when` conditions. Only if the given condition evaluates to true, the execution of the stage is initiated, else it is skipped. In BPMN, exclusive gateways are well-suited to model this behavior. Figure 6.33 shows a conditional

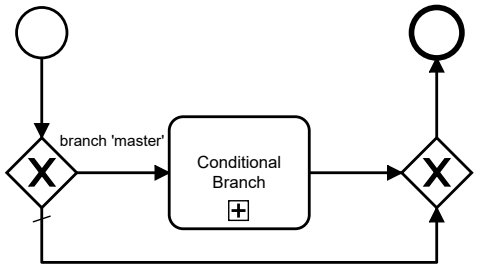
Listing 6.6 The XML representation of agent settings

```
<bpmn:startEvent id="StartEvent_0000005">
  <bpmn:extensionElements>
    <camunda:properties>
      <camunda:property name="agent.docker.image"
    ↪ value="'kieker/kieker-build:openjdk8'"/>
      <camunda:property name="agent.docker.args" value="env.DOCKER_ARGS"/>
      <camunda:property name="agent.docker.label" value="'kieker-slave-docker'"/>
    </camunda:properties>
  </bpmn:extensionElements>
  <bpmn:outgoing>SequenceFlow_0000007</bpmn:outgoing>
</bpmn:startEvent>
```

stage, which is only executed if the *master* branch of the VCS is the origin of the CD process. The stage's steps are not contained in the example for the sake of simplicity. The process starts with an exclusive gateway, which only routes the execution to the sequence flow leading to the conditional stage, if the condition "`branch 'master'`" evaluates to true. Else, the default sequence flow is taken, which results in skipping the conditional stage. In the figure, the stage "Conditional Branch" is depicted as collapsed sub-process. It is followed by a converging exclusive gateway, joining both execution branches. Subsequent stages would be placed after this gateway.

```
# Stage:
name: Conditional Branch
when: branch 'master'
steps:
  - ...
```

(a) The StalkCD example



(b) The resulting BPMN model

Figure 6.33: Modeling a conditional stage using exclusive gateways in BPMN

6.3.6 Post Steps

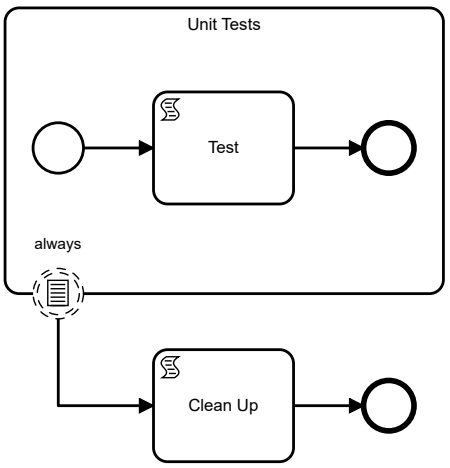
Post steps are executed based on the result of a stage or the pipeline. The conditions available in a post section of StalkCD could be mapped to BPMN using gateways. However, this would create inconsistencies with the BPMN specification. There are conditions like `always` and `failure` that are executed when an exception occurs. In BPMN, the regular sequence flow is left when an error occurs during the execution of a step. An exception handler is invoked to perform activities that handle the exception. This especially makes implementing the `always` condition problematic: Steps that are to be executed without regard of the according stage would have to be appended to both, the regular sequence flow and the exceptional flow. This would introduce extra complexity in the model which is not desirable.

For this reason, it is preferable to find a simpler approach. The BPMN specification provides various boundary events that could serve as starting point for a sequence of conditional steps. The obvious choice is to use the conditional event. It is triggered, when a variable in scope is changed to a value that matches the specified condition. In the case of the interrupting conditional event, which is depicted as a circle with a continuous, double line, the execution of the activity or sub-process, where the variable change occurred, is aborted. Instead, the sequence flow following the conditional event is initiated. If there are multiple interrupting conditional events, only one of them can be triggered. This conflicts with the semantic of post conditions, since it is possible that multiple conditions are met after the execution of the stage. For instance, upon a failure of a stage, not only its "failed" post steps are executed, but also the ones labeled with the "always" and "cleanup". For this reason, the interrupting conditional event cannot be used for representing post steps in a BPMN model.

Instead, the non-interrupting conditional event has to be used. Its specification intends the regular sequence flow to continue when the event is triggered. This allows to trigger multiple boundary events during the execution of the concerned activity or sub-process. Figure 6.34 shows an example of a stage, having conditional post steps with the condition "always". In the equivalent BPMN diagram, the sub-process representing the stage, has a conditional boundary event with the condition expression "always". The event is followed by a task, representing the step in the stage's post section.

```
1 stages:
2   - name: Unit Tests
3     steps:
4       - label: Test
5         command: npm test
6     post:
7       always:
8         - label: Clean Up
9           command: ssh registry.local
      ↪  rm -f /opt/next-version
```

(a) The StalkCD example



(b) The resulting BPMN model

Figure 6.34: Modeling conditional post steps using conditional events in BPMN

When depicting the post steps with the condition "failed" in BPMN, intuition would imply using the error boundary event. However, here again, this event is interrupting and would prevent other boundary events to be triggered. In fact, the Jenkinsfile specification does not allow for constructs that would imply an interruption of the regular execution of a stage, preventing post steps with conditions such as "always" or "cleanup" to be executed. Only the scripted Jenkinsfile syntax would allow to define try-catch-blocks, having an interrupting semantic.

Listing 6.7 The XML representation of a conditional event in BPMN

```
<bpmn:subProcess id="SubProcess_1" name="Unit Tests">
  <bpmn:startEvent id="StartEvent_2">
    <bpmn:outgoing>SequenceFlow_3</bpmn:outgoing>
  </bpmn:startEvent>
  <bpmn:sequenceFlow id="SequenceFlow_3" sourceRef="StartEvent_2" targetRef="Task_6" />
  <bpmn:endEvent id="EndEvent_4">
    <bpmn:incoming>SequenceFlow_5</bpmn:incoming>
  </bpmn:endEvent>
  <bpmn:sequenceFlow id="SequenceFlow_5" sourceRef="Task_6" targetRef="EndEvent_4" />
  <bpmn:scriptTask id="Task_6" name="Test">
    <bpmn:incoming>SequenceFlow_3</bpmn:incoming>
    <bpmn:outgoing>SequenceFlow_5</bpmn:outgoing>
    <bpmn:script>npm test</bpmn:script>
  </bpmn:scriptTask>
</bpmn:subProcess>
<bpmn:sequenceFlow id="SequenceFlow_7" sourceRef="BoundaryEvent_10" targetRef="Task_8" />
<bpmn:scriptTask id="Task_8" name="Clean Up">
  <bpmn:incoming>SequenceFlow_7</bpmn:incoming>
  <bpmn:outgoing>SequenceFlow_10</bpmn:outgoing>
  <bpmn:script>ssh registry.local rm -f /opt/next-version</bpmn:script>
</bpmn:scriptTask>
<bpmn:endEvent id="EndEvent_9">
  <bpmn:incoming>SequenceFlow_10</bpmn:incoming>
</bpmn:endEvent>
<bpmn:sequenceFlow id="SequenceFlow_10" sourceRef="Task_8" targetRef="EndEvent_9" />
<bpmn:boundaryEvent id="BoundaryEvent_11" name="always" cancelActivity="false"
    ↪ attachedToRef="SubProcess_1">
  <bpmn:outgoing>SequenceFlow_7</bpmn:outgoing>
  <bpmn:conditionalEventDefinition id="ConditionalEventDefinition_12">
    <bpmn:condition xsi:type="bpmn:tFormalExpression">always</bpmn:condition>
  </bpmn:conditionalEventDefinition>
</bpmn:boundaryEvent>
```

6.3.7 Layouting

Up to this point, the generated BPMN model only consists of the logical elements, defining the components of a process, such as sub-processes, script tasks, sequence flows and events. However, they have no graphical information, determining where and with what size each element has to be drawn when depicting the BPMN process. In order to generate such data, the general-purpose graph layout library “dagre”[8] is used.

The process of generating layout information consists of three main stages. In stage one, the BPMN model is converted into a generic graph data structure, which is extended by layout information by the mentioned JavaScript library. In stage two, the generated layout

[8] dagre – Graph layout for JavaScript – https://www.npmjs.com/package/dagre

information have to be modified, in order to implement specific features of BPMN, which have not been regarded by the dagre library. Finally, stage three applies the graphical information to the given BPMN model.

Stage 1: Generating Layout Information

The “dagre” library expects a simple graph as input, consisting of nodes and edges. Edges in BPMN are sequence flows, connecting two elements, which is trivial to map to the generic graph data structure. Nodes can have width and height properties, defining the specific dimensions of the node. The dimensions of the individual elements have been taken from the Camunda Modeler, since the model should be visualized using this tool. Thus, all events, gateways, and tasks have fixed sizes that are the same for all models. The Camunda Editor does not even offer a feature to resize these elements, hence it is safe to assume static sizes.

The size of sub-processes, however, depends on the elements it contains. Since the used library does not support nested nodes, this problem has been solved using a recursive approach. When the layout generator finds a sub-process, it initiates a new layout calculation process. When this recursive function call is done, the dimensions of the resulting sub-graph are copied to the node representing the concerned sub-process.

Boundary events are inserted in the generic graph data structure as normal nodes, having an edge to the sub-process they are attached to. This representation is only temporary and will be corrected in Stage 2. When a layout generation process is done creating the graph data structure, it calls the dagre library, which creates layout information for all nodes and edges.

Stage 2: Fix BPMN-Specific Features

The recursive approach makes it necessary to fix all coordinates of the nodes inside a sub-process. Both, the x and y coordinates of each node and edge have to be added by the corresponding offset, resulting from the position of the sub-graph in the overall graph. The offset has to be calculated by adding up the offsets of all parent graphs.

Furthermore, boundary events have to be moved to the boundary of the sub-process they are attached to. This is done by limiting the x and y coordinates of the event to the boundary coordinates of the sub-process. This also requires adjusting the coordinates of the outgoing edge from the event to the subsequent task. The edge between the sub-process and the boundary event is irrelevant and will not be considered in Stage 3.

Stage 3: Apply Layout

Now that the layout information in the graph data structure is complete, it can be applied to the BPMN XML representation. The data structure of the BPMN process is extended by a “BPMNDiagram” element, containing “BPMNShape” nodes for all BPMN elements and “BPMNEdge” nodes for all sequence flows. When inserting the position and dimension

information into these elements, the x and y coordinates of all elements are swapped to create a left-to-right directed layout, instead of the top-to-bottom directed output from the dagre library. An example for a BPMN diagram with layout information in XML format can be found in Listing 2.2 on page 19.

6.4 BPMN to StalkCD

The first step when converting a BPMN file into a StalkCD pipeline is parsing the source BPMN file, given in XML format. The utility "xml2js"[9] performs this step and returns an iterable data structure, containing all information from the XML document. The transformation of the source data structure into StalkCD pipeline is done by iterating over all elements in the BPMN process, following the sequence flow. Figure 6.35 shows a UML class diagram of the `ProcessTranslator` class. It has to be initiated with a StalkCD pipeline builder, which is used to construct the pipeline.

The translation process is started by calling the method `translate` with the source BPMN process as parameter. The method performs a depth-first search on the given graph, while using the property `nextElementsStack` as list of elements that are yet to be processed. If the current BPMN element has no outgoing sequence flow, then the end of a branch is detected and the next element from the `nextElementsStack` is processed next. If the stack is empty upon the end of a branch, then the end of the BPMN process has been reached.

When the method finds a sub-process, the `PipelineBuilder` is instructed to create a new stage in the StalkCD data model. Script tasks are inserted as steps in the active stage. Gateways are expected to be followed by sub-processes, indicating conditional or parallel stages. Conditional boundary events attached to a sub-process are identified as post conditions. Their condition is extracted from the expression property of the BPMN event. When the translation process is done, the StalkCD file can be generated using the utility "js-yaml", as already described in Section 6.1.

[9] xml2js – https://www.npmjs.com/package/xml2js

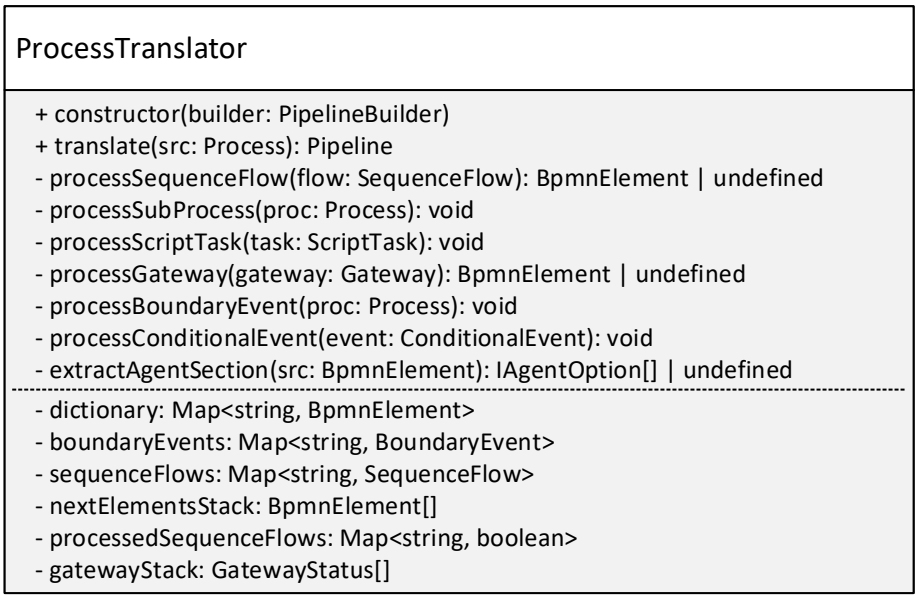


Figure 6.35: The class `BpmnParser` as class diagram

7 Evaluation

The evaluation investigates the concepts and implementations developed in this master's thesis. It starts by explaining the main goals of the evaluation. Next steps will be the definition of experiments to find answers to the research questions and to present and discuss their results. The chapter is concluded by an analysis of the threats to the validity of the conducted experiments. This section will outline possible sources of error and list assumptions and limitations of the chosen method of evaluation.

7.1 Evaluation Goals

In order to find answers to the remaining research questions defined in Section 1.1, appropriate evaluation methods have to be found. RQ 1 and RQ 2 have been answered in the previous chapters. RQ 3 aims to investigate the feature coverage of the implemented transformation from Jenkinsfiles to StalkCD. In Section 7.2, a method of measuring this coverage is given. RQ 4 covers the main motivation of this master's thesis by raising the questions concerning the resilience benefits, the given solutions provide. This aspect is handled in Section 7.3.

Aside from finding answers to these research questions, an additional goal of this chapter is evaluating the practicability of depicting Jenkinsfiles in BPMN diagrams. Section 7.4 demonstrates that it is possible to comprehensively visualize and edit generated BPMN diagrams.

7.2 Jenkinsfile Language Coverage

To answer RQ 3, the transformation from Jenkinsfile to StalkCD has to be investigated. Ideally, a formal analysis should prove that all features that are available in a Jenkinsfile are supported by the transformation. However, there is no comprehensive specification of Jenkinsfile features because there is a large amount of plug-ins that can be used. For this reason, a good approach to quantify the language coverage metric is conducting an empirical analysis. It consists of collecting sample data and analyzing the transformation for each sample. The exact procedure is described in the following.

7.2.1 Data Gathering

In preparation for the analysis, sample Jenkinsfiles have been collected. The GitHub REST API with its code search endpoint served as data source here. Since many Jenkinsfiles are using the scripted syntax, a simple search for all files with the name "Jenkinsfile" would return many unusable results. The fact that each declarative pipeline starts with a pipeline element, containing a mandatory agent section can be used to generate better search results.

Accordingly, the search term "pipeline agent filename:Jenkinsfile in:file" has been used to get a list of Jenkinsfiles contained in any publicly accessible GitHub repository, which returned more than 60,000 entries. However, the GitHub API does not offer more than approx. 1,000 search results, which is why the used sample set is limited to a total amount of 1,134 files. The files used as input for the evaluation can be found in the supplementary material of this master's thesis [Kab19].

Due to the primitive search method, there are still some false positives among the results. Hence, in a next step, the list of files to use for the analysis has to be filtered in two stages. First, a file is considered to represent a declarative pipeline, if it contains a line that only consists of the word "pipeline", an opening curly brace and, if applicable, whitespace characters. Second, all files that contain multiple instances of a pipeline start element are excluded from the final data set. By applying this method of filtering, 107 Jenkinsfiles were removed from the set of samples.

7.2.2 Experiment Setup

In order to validate the correctness of the transformation from Jenkinsfiles to StalkCD and back, each sample Jenkinsfile J is transformed to a StalkCD file. This result is then transformed back to a Jenkinsfile J'. The transformation is complete for J, if $J \equiv J'$. A high percentage of correctly transformed samples proves a high coverage of language features of Jenkinsfiles.

J' can differ in certain aspects from J while having the same functional semantic. Such differences can be single-line and multi-line comments, whitespace (e. g., indentation and blank lines) as well as trailing semicolons and commas. This freedom makes it necessary to normalize both, J and J' to make them comparable.

Normalization

A comprehensive normalization, preserving all functional semantics of a Jenkinsfile would require a complex grammar implemented by a lexer and parser. However, this grammar itself could introduce faults into the normalization result, hiding errors in the evaluated transformation. For this reason, the procedure used here is chosen to be as simple as possible. Some degree of information loss, e. g., a missing whitespace character is acceptable because the main evaluation goal is to validate the support of all high-level constructs of

the transformation. The normalization is implemented as sequence of replacement rules based on regular expressions. The procedure consists of the following steps, executed sequentially:

1. **Indentation:**
   Remove whitespace characters at the beginning of all lines.
   `/^\s+/gm` ⇒ `''`
2. **Head comment:**
   The file might start with a head comment, specifying the interpreter to use when executing it. It is not relevant and removed during the transformation, hence it has to be removed during normalization, too.
   `/^#.*\$/gm` ⇒ `''`
3. **Single-line comment:**
   A line comment is introduced with two slash characters. This, though, is not valid if they are located inside a string literal. The following regular expression only considers single-line strings. If a multi-line string contains two adjacent slash characters, they are recognized as comment, which is not desirable, but not solvable using simple regular expressions.
   `/^(([^'"\r\n]|"[^"]*"|'[^']*')*?)\/\/.*?$/gm` ⇒ `'$1'`
4. **Multi-line comment:**
   Multi-line comments are surrounded by `/*` and `*/`. This, again, is only valid, if the start or end sequence does not occur inside a string literal. Additionally, the end sequence could have been already removed, if it is preceded by a single line comment. These issues, too, would only be solvable using a more complex lexer.
   `/^(([^'"\r\n]|"[^"]*"|'[^']*')*?)\/\*(.|\n)*?\*\//gm` ⇒ `'$1'`
5. **Copy & paste mistakes:**
   Some Jenkinsfiles begin with the line “Jenkinsfile (Declarative Pipeline)”. This is no valid language element of a Jenkinsfile and has no effect to the execution of the defined pipeline. The reason, why this line is present in many files is probably that the authors copied an example pipeline from the official Jenkins web site, where every code example has a title equal to the line. Since this mistake can be found in numerous repositories, the transformation is designed to ignore it and hence, the normalization has to remove it, too.
   `/^\s*Jenkinsfile \(Declarative Pipeline\)/` ⇒ `''`
6. **Semicolons:**
   During the transformation, multiple statements on one line, divided by semicolons are expanded so that each statement is on its own line. For this reason, semicolons that are not part of a string literal have to be removed during the normalization process.
   `/(([^'"\r\n]|"[^"]*"|'[^']*')*?);\s*/g` ⇒ `'$1\n'`
7. **Spaces before and after brackets:**
   As preparation for the removal of trailing commas, space characters before and after brackets have to be eliminated.
   `/\s*([,()\[\]])\s*/g` ⇒ `'$1'`

8. **Quotation marks around the stage name:**
   The name of a stage can be wrapped by either single or double quotation marks. In order to eliminate any differences here, double quotation marks are replaced by single ones.
   `/stage\("([^"]*)"\)/g ⇒ 'stage(\'$1\')'`
9. **Trailing commas:**
   Commas, ending a list of arguments are allowed in a Jenkinsfile, but not generated during the transformation from StalkCD to Jenkinsfile.
   `/,([)\]}])/g ⇒ '$1'`
10. **Whitespace**
    Whitespace characters, including line breaks, are ignored when comparing two Jenkinsfiles. It is assumed that bugs in relation with missing spaces are detected using a manual test procedure.
    `/\s/g ⇒ ''`

```
1  #!/usr/bin/env groovy
2
3  // Comments should be eliminated
4  pipeline {
5      agent { node { label 'aws' } }
6
7      /* This block will disapear
8      environment { any="example" }
9      // Including this comment
10     */
11
12     stages {
13         stage('Test Stage') {
14             steps {
15                 sh 'echo "This is an example! //
       ↪ This is no comment!"'
16             }
17         }
18     }
19 }
```

(a) Before normalization

```
1 pipeline{agent{node{label'aws'}}stages{
    ↪ stage('TestStage'){steps{sh'echo'
    ↪ Thisisanexample!//Thisisnocomment!
    ↪ ''}}}}
```

(b) After normalization

Figure 7.1: Normalization example of a Jenkinsfile

Figure 7.1 shows an example of the normalization of a Jenkinsfile. Note, that all comments are removed during normalization while similar expressions like URLs inside strings are preserved. The resulting string does not contain any whitespace characters anymore, i. e., it consists of only one line. Furthermore, it is not executable anymore because of missing spaces. However, all instructions, blocks, and the order of all elements are preserved.

Failure Classification

If a (normalized) sample file that passed the transformation process differs from its (normalized) original version, a transformation failure has occurred. To find the reason for this failure, a failure classification is being performed. In preparation of the classification, the difference between the original sample file and the transformation result has to be calculated.

This task is done with help of the *jsdiff* utility, found in the npm package manager[1]. The method `JsDiff.diffWords` is called with the contents of the original sample file and the resulting file after transformation, both normalized using the previously presented procedure. The utility returns an array, containing the differences between both files, divided in added and removed passages. Added parts are not interesting for the classification, since their origin is in general an unsupported construct, not present any more in the transformation result.

Each sequence that has been removed during the transformation is tested for certain properties to determine the failure class of the analyzed sample file. The failure classes together with their condition and explanations can be found in Table 7.1. A removed passage is tested for the listed conditions in the given order. The class of the first matching condition is assigned to the passage. A file is classified by all failure classes of its passages. The class unknown is only assigned if no passage has a concrete classification.

7.2.3 Results & Discussion

After applying the filter rules, 1,027 Jenkinsfiles remain as input for the evaluation. 717 of these files (70%) could be successfully transformed to StalkCD and back to Jenkinsfile without information loss, i. e., the normalized resulting Jenkinsfile equals the normalized original one. This also means that 310 samples (30%) could not be transformed back to their original semantic, which indicates that they use constructs that are not supported by the transformation. All transformation results are available in the supplementary material of this master's thesis [Kab19].

Figure 7.2 shows a classification of the transformation errors. A large portion (90 samples) of all failures could not be classified. Reasons for this are, among others, syntax errors in the sample Jenkinsfile, multiple sections of the same type overriding previous ones, e. g., multiple agent sections inside the pipeline block, non-ascii quotation marks, extra brackets, empty sections or complex Groovy code at the top of the file. Hence, the majority of transformation failures are caused by mistakes by the developers of the Jenkinsfiles or because of an extensive use of the Groovy language, which intentionally is unsupported by the transformation (see Section 6.1).

However, some failure classes indicate lacking support of some features available in the Jenkinsfile: 22 sample files contain environments that could not be transformed, e. g., the *aws* section which is specific for Amazon Web Services steps or the rarely used *lockThenSteps*

[1] https://www.npmjs.com/package/diff

section. Further elements that were not processed by the transformation are complex steps (39 occurrences), such as the parallel step, wrapping multiple sections of nested steps and expression environment within other aggregation sections such as the *anyOf* and *allOf* environments.

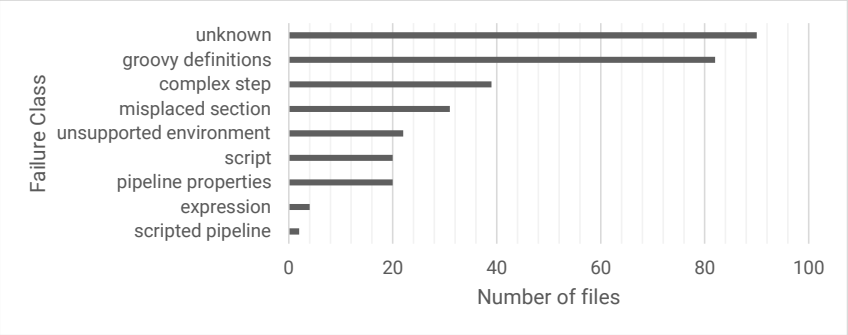


Figure 7.2: Failure classification of the evaluated Jenkinsfiles

7.3 Resilience Benefits

One focus of this master's thesis was creating resilience benefits for CD pipelines by using BPMN as an abstraction layer. This is evaluated in the following at the example of the Kieker monitoring framework[2]. Its source code can be found on GitHub[3]. We chose this project because it uses the declarative Jenkinsfile syntax for the definition of its CD process and because the build results are publicly accessible[4]. Kieker uses Jenkins to automate its continuous delivery process. The Jenkinsfile is written in the declarative pipeline syntax, which is a requirement to be transformed to BPMN by the previously introduced approach. In order to find problems in the delivery process, which would benefit from resilience increasing measures, the messages in pipeline failures for the `master` branch are analyzed.

At the time of the evaluation, the most current build ID is #273. The total amount of build results that are accessible through the web-interface of Jenkins is 144. Of these builds, 78 (54 %) completed successful, 12 (8 %) were canceled and 54 (38 %) failed. A review of the console output of the failed builds showed, that 29 (54 %) of the failures were caused by configuration mistakes, as seen in Figure 7.3. 19 failures (35 %) could be classified as permanent I/O errors, such as no more free space on the disk. Only 8 (15 %) failures were caused by temporary failures, such as connection problems or disk write failures. These could have been avoided by simply repeating the failing stage. Two error logs did not contain enough information to give answers about the cause of failure.

[2]Kieker – http://kieker-monitoring.net/
[3]Kieker (source code) – https://github.com/kieker-monitoring/kieker
[4]Jenkins instance of the Kieker monitoring framework – https://build.se.informatik.uni-kiel.de/jenkins/job/kieker-monitoring/

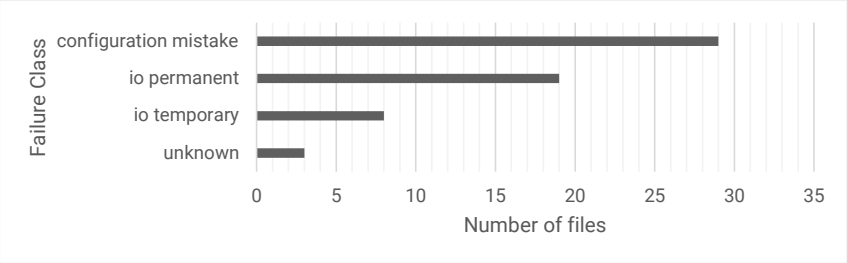


Figure 7.3: Failure classification of the Kieker CD pipeline

The results show that 89 % of the evaluated failures could not possibly have been avoided automatically. They were caused by human mistakes or required human intervention. The greatest part of the remaining failures could have been avoided by simply setting the `retry` option in the Jenkinsfile, telling Jenkins to re-trigger the delivery process upon a failure for the given amount of times. Thus, in the case of the Kieker CD pipeline, introducing BPMN as abstraction layer only has the advantage of visualizing the delivery process. This can help, finding mistakes in the process design, but brings no resilience advantages.

7.4 BPMN Visualization

Visualization is an essential purpose of BPMN. It helps the user to understand and adapt a process in a comfortable way. For this reason, this section will evaluate, if BPMN models generated by the developed transformations can be visualized comprehensively. Furthermore, it will check, if it is possible to read and write all relevant properties using a BPMN editor.

Figure 7.4 shows the Camunda Modeler after manually designing a BPMN process. In the top part of the window, a standard toolbar enables performing file operations and visually arranging selected elements. The left part of the editor window contains a toolbox with tools for navigation, selection, and re-arrangement. Furthermore, it enables creating the major types of BPMN elements. When selecting an existing element, its precise type can be chosen, as demonstrated in Figure 7.5 at the case of a boundary event. The displayed options vary depending on the selected BPMN element. On the right, there is the properties panel, providing access to meta-data of the selected element, which is not visible in their graphical representation.

In order to access agent settings, options, and other properties of BPMN elements that are stored in extension elements, the tab “Extensions” in the properties panel has to be used. Here, all defined key-value-pairs of the currently selected graph node are visible and can be modified, as seen in Figure 7.6. Note, that the values entered here are copied without modifications into a StalkCD- and Jenkinsfile. This allows to access variables in the scope of the Jenkins pipeline plug-in, but also requires to surround string literals with quotation marks.

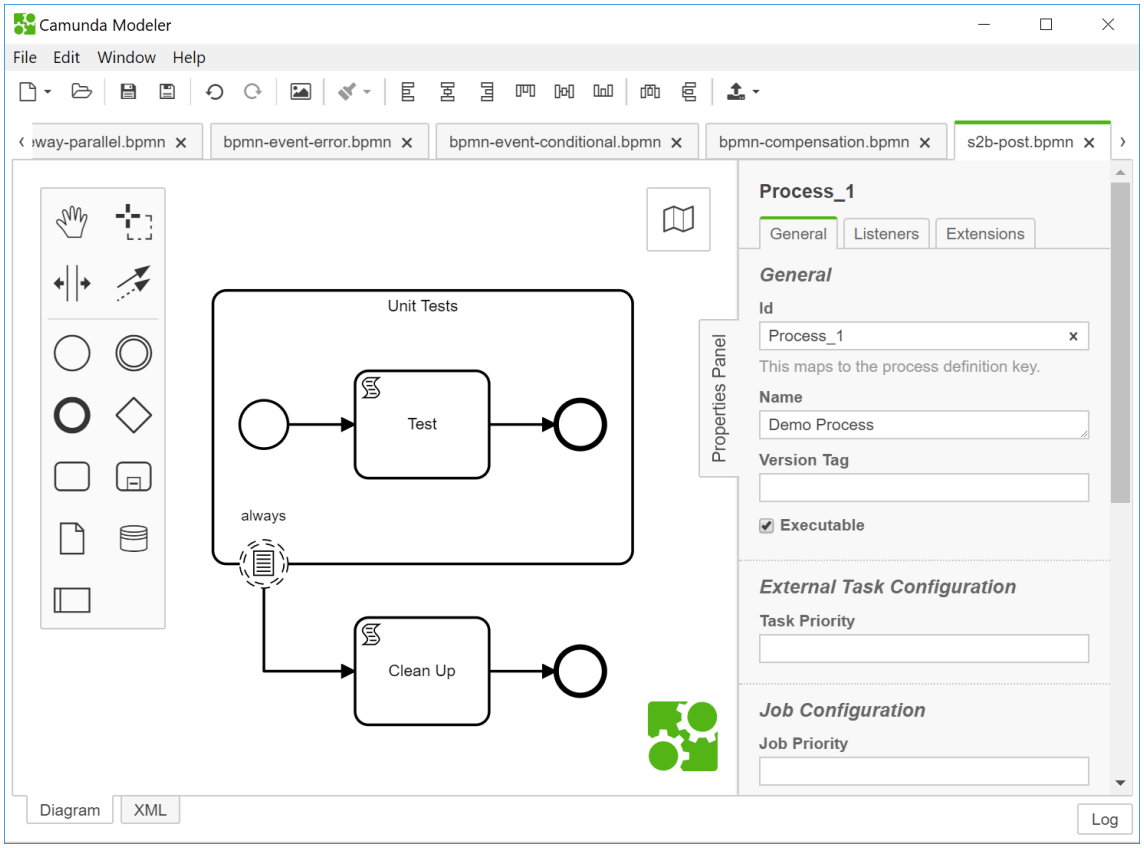


Figure 7.4: The Camunda Modeler

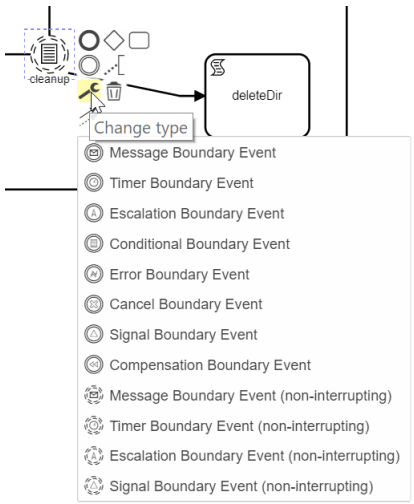


Figure 7.5: The type selection menu of the Camunda Modeler

Failure Class	Condition
complex step	Passage starts with “parallel(” or “wrapCommands(”. Complex steps are not supported by the transformation. An example for a complex step is the parallel step, containing multiple subsections having further nested steps (not to be confused with the parallel section inside a stage).
misplaced section	Passage starts with a well-known section name (e. g., agent, options, steps) and an opening culy brace. The support of some common sections of a Jenkinsfile are known to be supported by the transformation. If one of them is missing in the transformation output, it is assumed to be misplaced or a duplicate.
pipeline properties	Passage starts with “properties(”. Properties that are specified as prefix to the pipeline section are unsupported.
script	Passage starts with “script{”. Sections are unsupported inside aggregation environments, such as *anyOf* or *allOf*.
expression	Passage starts with “expression{”. Sections are unsupported inside aggregation environments, such as *anyOf* or *allOf*.
groovy definitions	Passage starts with one of “def”, “import” or “@Library” or is located at the beginning of the sample file. Not all Groovy expressions can be parsed by the used grammar.
scripted pipeline	Passage starts with “node”. Some files contain both, declarative and scripted pipeline definitions. These cannot be parsed.
unsupported environment	Passage starts with an identifier followed by “{”. Not all environments that can be used in a Jenkinsfile are supported by the transformation.
unknown	All remaining passages, removed during the transformation that can not be classified by one of the rules above.

Table 7.1: Failure classes and their conditions

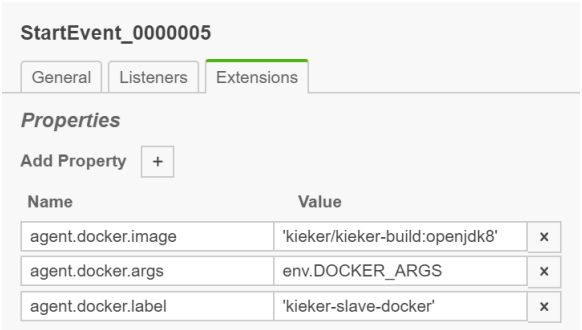


Figure 7.6: Agent settings in the properties panel of the Camunda Modeler

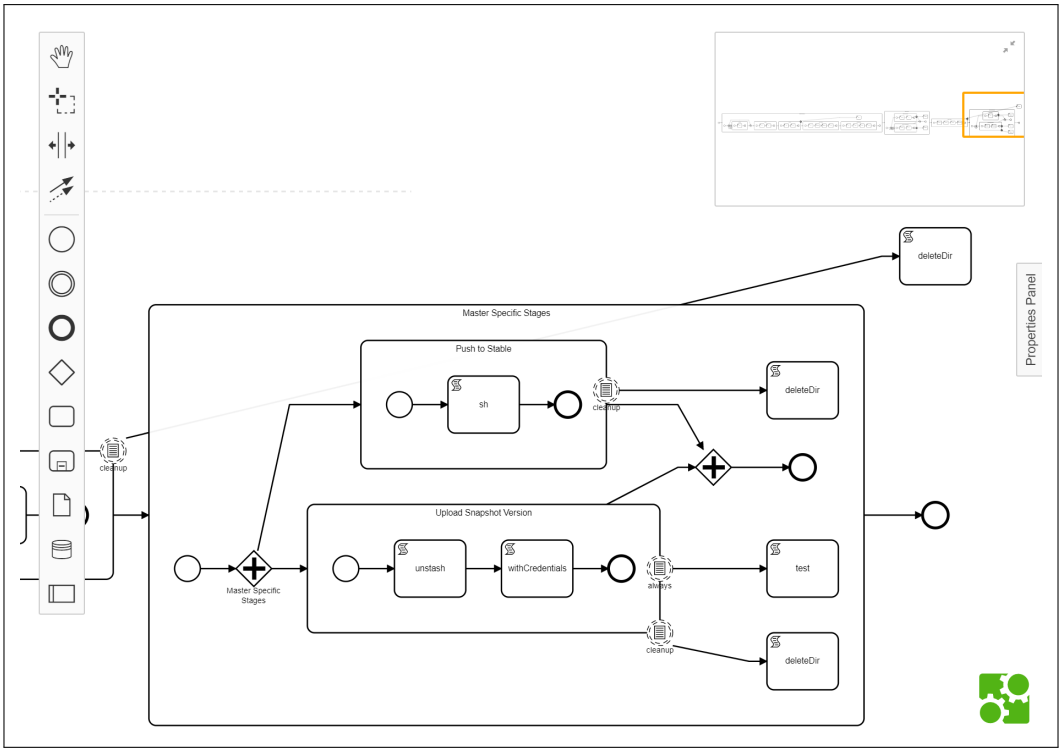


Figure 7.7: The Kieker Jenkinsfile depicted as BPMN diagram in the Camunda Modeler

Figure 7.7 shows a part of the Jenkinsfile of Kieker[5], as it is being generated by the developed transformations from Jenkinsfile via StalkCD to BPMN. It is striking that the layout differs from the manually generated model in Figure 7.4, as sequence flows often do not have right angled direction changes. This is caused by the specific implementation of the used layout algorithm, but in most cases does not affect the model's readability.

However, the position of conditional post steps, depicted as boundary events and script tasks, often is not ideal. The missing support of the layout algorithm for structured nodes containing sub-graphs in some cases leads to large distances between a sub-process and the tasks of its boundary events. The reason for this is that the chosen layout library manages layers of nodes regardless of their size. Only one node per branch is allowed to be located on a layer, where the largest node defines the size of the layer.

In the example of the Kieker CD pipeline, depicted in Figure 7.7, the original position of the boundary event "cleanup" in the left of the figure is on the same level as the sub-process "Master Secific Stages". The following script task "deleteDir" is placed on the next available level, together with the end event of the process. When the transformation algorithm moves the boundary event to the outline of the corresponding sub-process, the original place remains empty, which in case of the example significantly increases the distance between the boundary event and the following task. The same behavior can also be seen at the

[5]Kieker monitoring framework – http://kieker-monitoring.net/

boundary events inside the sub-process “Master Specific Stages”. In these cases, however, the resulting additional distance is smaller because the largest element in the concerned graph level is a parallel gateway node.

The visualization of the Kieker pipeline also shows that BPMN models representing CD pipelines quickly get very large and hard to overview. The “minimap” in the top right corner of Figure 7.7 reveals that the visible part of the opened model is very small. Even though the features for navigating provided by the Camunda Modeler are very useful, knowing the exact position in the pipeline can be challenging with increasing size.

At large, the features provided by the Camunda Modeler suffice to display all BPMN elements that are required when depicting a CD pipeline. Special extension properties are accessible in the properties panel, which allows the user to read and write such information, but still leaves room for improvement, as their existence is not visible in the displayed BPMN model.

7.5 Threats to Validity

The analysis of the transformation from Jenkinsfiles to StalkCD in Section 7.2 relies upon a data set that has been collected from GitHub repositories. The chosen approach has some restrictions, which potentially threaten the external validity of the evaluation results. Firstly, the extent of the source data set of Jenkinsfiles is limited by the maximum number of search results that are provided by the GitHub API. In consequence, it cannot be guaranteed that the evaluated Jenkinsfiles provide a good coverage of the available language features. However, as the search results are not ordered by repository name but by relevance, it can be assumed that they contain all kinds of projects, from small to large.

Secondly, due to the fact that only publicly accessible repositories can be searched using the GitHub API, software projects that potentially would be of special value to the evaluation are excluded from the sample data set. The larger an application is, the higher its business value and the lower the probability that the manufacturer publishes its source code. But particularly large applications require complex CD pipelines, using a large set of the CD tool’s features.

Finally, the presented normalization approach could hide errors in the tested transformation. Missing spaces and wrong quotation marks or escape sequences inside strings may not be detected when comparing the transformation result with the original Jenkinsfile. However, this does not affect the fulfillment of the main requirement for the evaluation. The general file structure with all important elements is preserved, which allows to detect missing or additional elements like steps or stages, as well as their position and order.

8 Conclusions and Future Work

In this master's thesis, an approach of bringing BPMN to the CD domain has been presented. Section 8.1 summarizes the contributions of the thesis. In Section 8.2, the results are discussed and put into relation to the expected outcomes. Finally, Section 8.3 names points of connection that can be addressed by future work.

8.1 Summary

Chapter 1 showed the importance of resilient CD processes and named BPMN as possible means to reduce the frequency of CD pipeline failures. The notation specifies constructs that can make processes more reliable or compensate the effects of failures. As a preparation for the use of BPMN as modeling language for CD processes, a meta-model of the DSL of Jenkinsfiles has been developed in Chapter 4.

StalkCD as Technology Bridge

This created the foundation for the development of the DSL StalkCD, specified in Chapter 5. It aims to bridge the functional gap between the CD domain and BPMN. StalkCD supports all relevant features from the Jenkinsfile DSL as representative of popular PDLs. By adding more properties to existing language elements or by introducing new ones, support for more CD tools can be added easily. This makes the language a generic, extensible modeling language for CD processes.

Additionally, the DSL can serve as a data store in accordance with the concept of IaC. In contrast to BPMN, it does not contain visual position data, which could lead to unnecessary commits in the VCS. The available information is sufficient to generate a graphical BPMN model, enabling to benefit from the capabilities of BPMN. At the same time, a StalkCD pipeline can be easily converted into a Jenkinsfile for execution.

Bi-Directional Transformations

Chapter 6 defined approaches to transform Jenkinsfiles via StalkCD to BPMN and back. Parsing Jenkinsfiles was a demanding task since there is no re-usable approach for parsing and processing them. An ANTLR grammar has been developed, which recognizes the language constructs of a Jenkinsfile and allowed to implement a translator without much effort. As it is highly re-usable, the grammar could support future projects in parsing and processing declarative Jenkinsfiles.

For the transformation from StalkCD to BPMN, the features of StalkCD were mapped to compatible BPMN elements, defining an approach to depict a CD pipeline in BPMN. The transformations from BPMN to StalkCD and from StalkCD to Jenkinsfiles were comparatively trivial and did not require complex implementations.

At large, the provided solutions allow even existing CD pipelines to take advantage from BPMN as powerful modeling notation. The two-way transformation from Jenkinsfiles to BPMN and back enables converting virtually any given Jenkinsfile into a graphical BPMN model. After having done modifications to the graphical representation, the CD pipeline can be translated into StalkCD in order to be kept in a VCS for collaboration. When the pipeline is to be executed, it can be simply transformed into a Jenkinsfile.

Evaluation Strategy and Results

The main task of Chapter 7 was the evaluation of the quality of the implemented transformations. A set of sample Jenkinsfiles has been acquired from the GitHub API, which was transformed to StalkCD and back to the Jenkinsfile DSL. Source and result have been normalized in order to make them comparable. The results showed that 70 % of the input files could be translated to StalkCD and back without losing information. Many transformation failures were caused by bad sample files, containing syntax errors or violating the concept of declarative programming by extensively using the Groovy programming language.

Further evaluation showed that failures of the CD pipeline of the Kieker monitoring framework often are caused by permanent errors, that cannot be resolved automatically. A small minority of the evaluated failures could have been resolved by repeating the process. Thus, introducing BPMN as abstraction layer would bring no resilience benefits in case of Kieker.

8.2 Discussion

During this master's thesis, numerous decisions have been made that impact the chosen approaches and the results. In the following, these decisions will be recapitulated.

StalkCD

The decision to develop a DSL for bridging the functional gap between the CD domain and BPMN originally was based on the goal of supporting multiple CD tools. The implemented transformations only support Jenkinsfiles as target PDL, but since other tools such as Travis CI[1] or GoCD[2] also use the concept of stages and steps, the effort for extending the transformations to support them can be expected to be manageable.

[1] Travis CI – https://travis-ci.com/
[2] GoCD – https://www.gocd.org/

Additionally, StalkCD helps minimizing the data that has to be managed in a VCS. Using a BPMN XML file as single data store would also mean that layout information has to be kept under version control. Every position change of a BPMN element would require a new commit in the VCS, which obviously is not desirable. With StalkCD as intermediate abstraction layer, only information that cannot be automatically generated into a BPMN model is included in the data store. For instance, the summary of a step that is displayed in the graphical representation of the BPMN model is contained in the StalkCD file. At large, StalkCD as extension to the PDLs of CD tools helps storing information that is relevant when depicting them using BPMN.

Jenkins as CD Tool

The complex DSL of Jenkinsfiles and the lack of an existing solution for automatically processing them required an extensive implementation for parsing and translating them to StalkCD. Other CD tools use YAML-based files that would have been comparatively trivial to read and process. Even though this prevented more detailed work on increasing the resilience of CD pipelines, this contribution of the thesis is still valuable, as literature shows that Jenkins even today is a popular CD tool with a high relevance [BE18].

Transformation of Jenkinsfiles to StalkCD

The design of an ANTLR grammar has significantly simplified the implementation of the transformation from Jenkinsfiles to StalkCD. The generated parser and its interface according to the visitor design pattern is re-usable and not limited to the generation of StalkCD files. In principle, it can be used for further applications, such as tools for validating or visualizing Jenkinsfiles.

The implemented transformation has a good coverage of Jenkinsfile features. However, not all sample files could be parsed, which is caused by the permissiveness of the Jenkinsfile syntax. The possibility to use the Groovy programming language inside a declarative Jenkinsfile makes it hard to develop comprehensive solutions.

Transformation of BPMN to StalkCD

A BPMN model can contain countless structures that cannot be processed by the developed transformation. This is in the nature of BPMN, as the notation is designed to be as flexible as possible to be capable of representing virtually any kind of process. For this reason, the provided implementation intentionally only allows constructs that would also result when transforming StalkCD to BPMN. Any other structures are ignored or lead to failures in the transformation process or the resulting StalkCD pipeline. However, this issue can be avoided by improving the graphical support of the user.

Resilience

The evaluation of resilience benefits only considers the Kieker monitoring framework, a research-driven application of medium size and complexity. Its delivery process involves a single call to a script, which uploads the built artifacts to a registry. A more complex application, e. g., depending on server clusters for database, logging, and so on, would require a more complex delivery process, having numerous potential points of failure. Here, it could be interesting to investigate the introduction of compensation actions, for instance to recover from failures during the testing stage, preventing the pipeline to fail for trivial reasons.

However, such an example application was not available for this master's thesis. The reasons for this are two-fold: on one hand, investigations have shown that large software systems often depend on the scripted Jenkinsfile syntax. On the other hand, complex software can be found mainly in enterprise projects, which often are not publicly accessible. This lack of real-world examples hinders the evaluation whether there actually is the need for the introduction of an additional layer of abstraction to CD processes.

Additionally, Jenkins itself already provides some basic features to ensure operability of a CD pipeline. However, complex decisions regarding the sequence flow of stages are not implementable without violating the principle of declarative programming. We conclude, that the use of BPMN as an abstraction layer for Jenkinsfiles does not lead to an increased resilience. But without the restrictions of the declarative Jenkinsfile syntax, more complex decisions could be implemented, ensuring the operability of the CD process.

8.3 Future Work

Various results of this master's thesis can serve as points of connection, where future research can start at. These points are listed in the following.

Better support of CD tools: The implementation provided in this thesis only supports Jenkins as source CD tool. The support of more tools can bring additional opportunities to demonstrate the benefits of bringing BPMN to the CD domain, as it would increase the number of analyzable CD pipelines. Tools like Travis CI[3] or Gitlab[4] are promising examples, because they use the YAML format as data store.

Improved BPMN layout generation: The BPMN layout generation approach in some cases produces unclear results. The handling of boundary events and sub-graphs can be improved in order to optimize the space usage and to prevent clutter and overlapping elements.

[3] Travis CI – https://travis-ci.com/
[4] Gitlab – https://about.gitlab.com/

Better BPMN tooling: The Camunda Modeler, which was used as editing solution in the thesis, does not provide solutions to depict CD related properties in the graphical model. An extension to the editor could not only increase the amount of visible information, but also simplify the development process of a CD process in BPMN. By defining re-usable snippets and restrictions, the user can be supported.

Split StalkCD pipelines: With increasing complexity of the delivery process, corresponding BPMN model increases significantly in size. The example of the Kieker Jenkinsfile depicted as BPMN process demonstrates that keeping an overview and navigating in a model can be challenging. A possible solution to this problem can be splitting up a StalkCD pipeline into re-usable parts. Smaller BPMN models can be easier managed and modified and support collaboration.

Statical analysis of BPMN: Van der Aalst [Van98] presents an approach of mapping BPMN to Petri nets. According to him, many analysis techniques exist for the domain of Petri nets, which could also be helpful in the CD domain: in combination with the approach of mapping CD pipelines to BPMN that has been developed in this master's thesis, problems of a delivery process in development could be detected already at design-time.

A Jenkinsfile Parser Grammar

```
grammar jenkinsfile;

/**
Parser
*/

pipeline                :
                        (
                            groovy_definition
                            |
                            JENKINSFILE_DECLARATIVE
                        )*
                        PIPELINE
                        LBRACE
                        (
                            environment
                            |
                            agent
                            |
                            tools
                            |
                            pipeline_options
                            |
                            parameters
                            |
                            triggers
                            |
                            stages
                            |
                            post
                        )*
                        RBRACE
                        ;

groovy_definition       :
                        DEF_LITERAL
                        |
                        LIBRARY_LITERAL
                        ;

environment             :
                        ENVIRONMENT
                        LBRACE
                        assignment*
```

```
45                          RBRACE
46                          ;
47
48  parameters              :
49                          PARAMETERS
50                          LBRACE
51                          method_call*
52                          RBRACE
53                          ;
54
55  agent                   :
56                          AGENT
57                          (
58                              agent_section
59                              |
60                              agent_type
61                          )
62                          ;
63
64  agent_section           :
65                          LBRACE
66                          (
67                              agent_type
68                              (
69                                  LBRACE
70                                  method_call*
71                                  RBRACE
72                              )?
73                              |
74                              method_call*
75                          )
76                          RBRACE
77                          ;
78
79  agent_type              : identifier ;
80
81  tools                   :
82                          TOOLS
83                          LBRACE
84                          method_call*
85                          RBRACE
86                          ;
87
88  pipeline_options        :
89                          OPTIONS
90                          LBRACE
91                          method_call*
92                          RBRACE
93                          ;
94
95  triggers                :
96                          TRIGGERS
97                          LBRACE
```

```
                        method_call*
                        RBRACE
                        ;

stages                  :
                        (
                            STAGES
                            |
                            PARALLEL
                        )
                        LBRACE
                        stage_definition*
                        RBRACE
                        ;

stage_definition        :
                        STAGE
                        (
                            LPAREN
                            stage_name?
                            RPAREN
                        )?
                        LBRACE
                        (
                            environment
                            |
                            input
                            |
                            tools
                            |
                            agent
                            |
                            when
                            |
                            stages
                            |
                            steps
                            |
                            post
                            |
                            fail_fast
                        )*
                        RBRACE
                        ;

stage_name              :
                        STRING_LITERAL
                        ;

fail_fast               :
                        FAIL_FAST
                        BOOL_LITERAL
                        ;
```

```
151
152 steps                   :
153                         STEPS
154                         LBRACE
155                         step*
156                         RBRACE
157                         ;
158
159 step                    :
160                         script
161                         |
162                         method_call
163                         ;
164
165 script                  :
166                         SCRIPT_LITERAL
167                         ;
168
169 input                   :
170                         INPUT
171                         LBRACE
172                         (
173                             method_call
174                         )*
175                         RBRACE
176                         ;
177
178 when                    :
179                         WHEN
180                         LBRACE
181                         (
182                             method_call
183                             |
184                             when_aggregation
185                             |
186                             when_expression
187                         )*
188                         RBRACE
189                         ;
190
191 when_aggregation        :
192                         when_aggregation_type
193                         LBRACE
194                         (
195                             method_call
196                             |
197                             when_aggregation
198                             |
199                             when_expression
200                         )*
201                         RBRACE
202                         ;
203
```

```
when_aggregation_type   :
                        ALLOF
                        |
                        ANYOF
                        |
                        NOT
                        ;

when_expression         :
                        EXPRESSION
                        LBRACE
                        expression
                        RBRACE
                        ;

post                    :
                        POST
                        LBRACE
                        post_condition*
                        RBRACE
                        ;

post_condition          :
                        identifier
                        LBRACE
                        step*
                        RBRACE
                        ;

assignment              :
                        assignment_key
                        ASSIGN
                        expression
                        ;

assignment_key          :
                        identifier
                        ;

method_call             :
                        (
                            method_call_simple
                            |
                            method_call_java
                            |
                            method_environment
                        )
                        ';'?
                        ;

method_environment      :
                        method_call_java
                        LBRACE
```

```
                        step*
                        RBRACE
                        ;

method_call_simple      :
                        identifier
                        method_arg_list
                        ;

method_call_java        :
                        identifier
                        LPAREN
                        (
                            method_arg_list
                        )?
                        RPAREN
                        ;


method_arg_list         :
                        method_arg
                        (
                            COMMA
                            method_arg
                        )*
                        COMMA?
                        ;

method_arg              :
                        method_arg_key
                        COLON
                        expression
                        |
                        expression
                        ;

method_arg_key          : identifier ;


expression              : primary
                        | expression bop='.'
                            (
                                identifier
                                | method_call_java
                            )
                        | expression '[' expression ']'
                        | prefix='[' expression_list? postfix=']'
                        | identifier bop=':' expression
                        | method_call_java
                        | expression postfix=('++' | '--')
                        | prefix=('+'|'-'|'++'|'--') expression
                        | prefix=('~'|'!') expression
                        | expression bop=('*'|'/'|'%') expression
```

```
                        | expression bop=('+'|'-') expression
                        | expression bop=('<=' | '>=' | '>' | '<') expression
                        | expression bop=('==' | '!=' | '==~') expression
                        | expression bop='&' expression
                        | expression bop='^' expression
                        | expression bop='|' expression
                        | expression bop='&&' expression
                        | expression bop='||' expression
                        | expression bop='?' expression ':' expression
                        | <assoc=right> expression
                        bop=('=' | '+=' | '-=' | '*=' | '/=' | '&=' | '|=' | '^=' |
    ↪ '>>=' | '>>>=' | '<<=' | '%=')
                        expression
                        ;

expression_list         : expression
                        (
                            COMMA
                            expression
                        )*
                        COMMA?
                        ;

primary                 : '(' expression ')'
                        | literal
                        | identifier
                        ;

literal                 : DECIMAL_LITERAL
                        | FLOAT_LITERAL
                        | STRING_LITERAL
                        | BOOL_LITERAL
                        | NULL_LITERAL
                        | REGEXP_LITERAL
                        ;

identifier              : IDENTIFIER
                        | PIPELINE
                        | STAGES
                        | PARALLEL
                        | STAGE
                        | STEPS
                        | ENVIRONMENT
                        | INPUT
                        | TOOLS
                        | PARAMETERS
                        | OPTIONS
                        | TRIGGERS
                        | AGENT
                        | POST
                        | WHEN
                        | ANYOF
                        | ALLOF
```

```
                        | EXPRESSION
                        | FAIL_FAST
                        | NOT
                        ;

/**
Lexer
*/

// Literals

DECIMAL_LITERAL:    ('0' | [1-9] (Digits? | '_'+ Digits)) [lL]?;
HEX_LITERAL:        '0' [xX] [0-9a-fA-F] ([0-9a-fA-F_]* [0-9a-fA-F])? [lL]?;
OCT_LITERAL:        '0' '_'* [0-7] ([0-7_]* [0-7])? [lL]?;
BINARY_LITERAL:     '0' [bB] [01] ([01_]* [01])? [lL]?;

FLOAT_LITERAL:      (Digits '.' Digits? | '.' Digits) ExponentPart? [fFdD]?
             |       Digits (ExponentPart [fFdD]? | [fFdD])
             ;

HEX_FLOAT_LITERAL:  '0' [xX] (HexDigits '.'? | HexDigits? '.' HexDigits) [pP] [+-]?
    ↪ Digits [fFdD]?;

BOOL_LITERAL:       'true'
            |       'false'
            ;

NULL_LITERAL:       'null'
            ;

STRING_LITERAL: (
    StringLiteralMultilineDouble
    | StringLiteralMultilineSingle
    | StringLiteralDouble
    | StringLiteralSingle
    ) ;
fragment StringLiteralMultilineDouble: '"""' .*? '"""';
fragment StringLiteralMultilineSingle: '\'\'\'' .*? '\'\'\'';
fragment StringLiteralDouble: '"' (~["\\\r\n\u000C] | EscapeSequence)* '"';
fragment StringLiteralSingle: '\'' (~['\\\r\n\u000C] | EscapeSequence)* '\'';

// Keywords
PIPELINE                : 'pipeline';
STAGES                  : 'stages';
PARALLEL                : 'parallel';
STAGE                   : 'stage';
STEPS                   : 'steps';
ENVIRONMENT             : 'environment';
INPUT                   : 'input';
TOOLS                   : 'tools';
PARAMETERS              : 'parameters';
OPTIONS                 : 'options';
TRIGGERS                : 'triggers';
```

```
414 AGENT                   : 'agent';
415 POST                    : 'post';
416 WHEN                    : 'when';
417 ANYOF                   : 'anyOf' | 'anyof';
418 ALLOF                   : 'allOf' | 'allof';
419 NOT                     : 'not';
420 EXPRESSION              : 'expression';
421 FAIL_FAST               : 'failFast' | 'failfast';
422
423 // Literals
424 SCRIPT_LITERAL          :
425                         'script'
426                         ScriptBlock
427                         ;
428
429 DEF_LITERAL             :
430                         'def' [ \t]+
431                         DefLiteralId
432                         (
433                             DefLiteralParam
434                             |
435                             '=' [ \t]*
436                         )
437                         DefLiteralVal
438                         ;
439
440 fragment DefLiteralId   :
441                         Letter
442                         LetterOrDigit*
443                         [ \t]*
444                         ;
445
446 fragment DefLiteralParam:
447                         '('
448                         ~[)]*
449                         ')'
450                         [ \t]*
451                         ;
452
453 fragment DefLiteralVal  :
454                         (
455                             ScriptBlock
456                             |
457                             STRING_LITERAL
458                             |
459                             BOOL_LITERAL
460                             |
461                             NULL_LITERAL
462                             |
463                             DECIMAL_LITERAL
464                             |
465                             FLOAT_LITERAL
466                             |
```

```
467                             BINARY_LITERAL
468                             |
469                             Letter
470                             LetterOrDigit*
471                         )
472                         ;
473
474 JENKINSFILE_DECLARATIVE :
475                         'Jenkinsfile (Declarative Pipeline)'
476                         ;
477
478 LIBRARY_LITERAL         :
479                         '@Library('
480                         STRING_LITERAL
481                         ')'
482                         (
483                             [ \t]+
484                             '_'
485                         )?
486                         ;
487
488 IMPORT_LITERAL          :
489                         'import'
490                         [ \t]
491                         (
492                             Letter
493                             LetterOrDigit
494                             |
495                             '.'
496                             |
497                             '*'
498                         )+
499                         ;
500
501 fragment ScriptBlock    :
502                         [ \t\r\n\u000C]*
503                         '{'
504                         (
505                             ScriptBlock
506                             |
507                             ~[{}]
508                         )*
509                         '}'
510                         ;
511
512 // Separators
513 LPAREN:             '(';
514 RPAREN:             ')';
515 LBRACE:             '{';
516 RBRACE:             '}';
517 LBRACK:             '[';
518 RBRACK:             ']';
519 COLON:              ':';
```

```
COMMA:              ',';
DOT:                '.';
EQUALS:             '==';

// Operators
ASSIGN:             '=';

// Whitespace and comments
MULTI_COMMENT:      '/*' .*? '*/'    -> skip;
WS:                 [ \t\r\n\u000C]+ -> skip;
LINE_COMMENT1:      '#' ~[\r\n\u000C]*    -> channel(HIDDEN);
LINE_COMMENT2:      '//' ~[\r\n\u000C]*    -> channel(HIDDEN);

// Identifiers
REGEXP_LITERAL:     '/' (~[/\\\r\n\u000C] | EscapeSequence)* '/';
IDENTIFIER:         '$'? Letter LetterOrDigit*;

// Fragment rules
fragment ExponentPart
    : [eE] [+-]? Digits
    ;
fragment EscapeSequence
    : '\\' [btnfr"'\\]
    | '\\' ([0-3]? [0-7])? [0-7]
    | '\\' 'u'+ HexDigit HexDigit HexDigit HexDigit
    ;
fragment HexDigits
    : HexDigit ((HexDigit | '_')* HexDigit)?
    ;
fragment HexDigit
    : [0-9a-fA-F]
    ;
fragment Digits
    : [0-9] ([0-9_]* [0-9])?
    ;
fragment LetterOrDigit
    : Letter
    | [0-9]
    ;
fragment Letter
    : [a-zA-Z$_] // these are the "java letters" below 0x7F
    | ~[\u0000-\u007F\uD800-\uDBFF] // covers all characters above 0x7F which are not a
    ↪ surrogate
    | [\uD800-\uDBFF] [\uDC00-\uDFFF] // covers UTF-16 surrogate pairs encodings for
    ↪ U+10000 to U+10FFFF
    ;
```

Bibliography

[ASU99] A. V. Aho, R. Sethi, J. D. Ullmann. *Compilerbau Teil 1*. 2. Auflage. Oldenbourg Verlag München Wien, 1999 (cit. on pp. 21, 22).

[BCFM11] M. Bishop, M. Carvalho, R. Ford, L. M. Mayron. "Resilience is More Than Availability". In: *Proceedings of the 2011 New Security Paradigms Workshop*. NSPW '11. Marin County, California, USA: ACM, 2011, pp. 95–104. ISBN: 978-1-4503-1078-9 (cit. on pp. 19, 20).

[BE18] E. M. Bergsteinsdóttir, H. H. Edholm. "Designing Continuous Toolchains - Using Proposed Guidelines and Tool Capabilities". MA thesis. University of Gothenburg, 2018 (cit. on pp. 9, 10, 89).

[Beh12] M. Behrendt. *Jenkins kurz & gut*. O'Reilly Germany, 2012 (cit. on p. 10).

[BEI05] O. Ben-Kiki, C. Evans, B. Ingerson. *Yaml ain't markup language (yaml™) version 1.1*. 2005 (cit. on p. 41).

[Che15] L. Chen. "Continuous delivery: Huge benefits, but challenges too". In: *IEEE Software* 32.2 (2015), pp. 50–54 (cit. on pp. 1, 8).

[CT12] M. Chinosi, A. Trombetta. "BPMN: An introduction to the standard". In: *Computer Standards & Interfaces* 34.1 (2012), pp. 124–134 (cit. on pp. 11, 12).

[DDO08] R. M. Dijkman, M. Dumas, C. Ouyang. "Semantics and analysis of business process models in BPMN". In: *Information & Software Technology* 50.12 (2008), pp. 1281–1294 (cit. on p. 12).

[DLMR+18] M. Dumas, M. La Rosa, J. Mendling, H. A. Reijers, et al. *Fundamentals of business process management*. Second Edition. Springer, 2018 (cit. on pp. 11, 12).

[Dör18] J. S. Döring. "An Architecture for Self-Organizing Continuous Delivery Pipelines". MA thesis. RWTH Aachen, 2018 (cit. on p. 25).

[EGHS16] C. Ebert, G. Gallardo, J. Hernantes, N. Serrano. "DevOps". In: *IEEE Software* 33.3 (2016), pp. 94–100 (cit. on pp. 1, 7).

[FF06] M. Fowler, M. Foemmel. "Continuous integration". In: *Thought-Works http://www.thoughtworks.com/ContinuousIntegration.pdf* 122 (2006), p. 14 (cit. on p. 8).

[Fis18] M. Fischer. "A DSL for Configuring Continuous Deployment Pipelines for Android Apps". MA thesis. Friedrich-Alexander-Universität Erlangen-Nürnberg, 2018 (cit. on pp. 25–28).

[GHL+15] M. Geiger, S. Harrer, J. Lenhard, M. Casar, A. Vorndran, G. Wirtz. “BPMN conformance in open source engines”. In: *2015 IEEE Symposium on Service-Oriented System Engineering*. IEEE. 2015, pp. 21–30 (cit. on p. 17).

[HF10] J. Humble, D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010 (cit. on pp. 1, 7–9, 26, 31).

[Hüt12] M. Hüttermann. “DevOps for Developers”. In: Apress, Berkeley, CA, 2012. Chap. Infrastructure as Code, pp. 135–156 (cit. on pp. 1, 7, 9).

[JN14] J. Jeston, J. Nelis. *Business Process Management: Practical Guidelines to Successful Implementations*. Third edition. Routledge, 2014 (cit. on p. 11).

[Kab19] O. Kabierschke. *Resilient Continuous Delivery Pipelines Based on BPMN – Supplementary Material*. 2019. DOI: 10.5281/zenodo.2602927. URL: https://doi.org/10.5281/zenodo.2602927 (cit. on pp. 47, 76, 79).

[LIL17] E. Laukkanen, J. Itkonen, C. Lassenius. “Problems, causes and solutions when adopting continuous delivery—A systematic literature review”. In: *Information and Software Technology* 82 (2017), pp. 55–79 (cit. on p. 2).

[Llo94] J. W. Lloyd. “Practical Advtanages of Declarative Programming.” In: *GULP-PRODE (1)*. 1994, pp. 18–30 (cit. on p. 10).

[MA14] S. Musman, S. Agbolosu-Amison. *A measurable definition of resiliency using mission risk as a metric*. Tech. rep. MITRE Corporation, 2014 (cit. on p. 20).

[OMG11] OMG. *Business Process Model And Notation (BPMN 2.0)*. May 2011. URL: http://www.omg.org/spec/BPMN/2.0 (cit. on pp. 12, 13, 18).

[Par13] T. Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013 (cit. on pp. 23, 48).

[Rec10] J. Recker. “Opportunities and constraints: the current struggle with BPMN”. In: *Business Proc. Manag. Journal* 16.1 (2010), pp. 181–201 (cit. on p. 12).

[SBZ17] M. Shahin, M. A. Babar, L. Zhu. “Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices”. In: *IEEE Access* 5 (2017), pp. 3909–3943 (cit. on pp. 1, 8).

[SNP15] J. Smeds, K. Nybom, I. Porres. “DevOps: A Definition and Perceived Adoption Impediments”. In: *Agile Processes in Software Engineering and Extreme Programming*. Ed. by C. Lassenius, T. Dingsøyr, M. Paasivaara. Cham: Springer International Publishing, 2015, pp. 166–177. ISBN: 978-3-319-18612-2 (cit. on p. 1).

[Van98] W. M. Van der Aalst. “The application of Petri nets to workflow management”. In: *Journal of circuits, systems, and computers* 8.01 (1998), pp. 21–66 (cit. on p. 91).

[WBL14] J. Wettinger, U. Breitenbücher, F. Leymann. “DevOpSlang -- Bridging the Gap Between Development and Operations”. In: *Proceedings of the 3rd European Conference on Service--Oriented and Cloud Computing (ESOCC 2014)*. Lecture Notes in Computer Science (LNCS). Springer--Verlag, 2014, pp. 10 8–1 22 (cit. on p. 28).

[Wil18] N. Willig. “Implementing Continuous Delievery through BPMN”. MA thesis. RWTH Aachen, 2018 (cit. on pp. 5, 25).

All links were last followed on March 17, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature