TU WIEN Informatics

Cloud Foundry Config File Generation Using JetBrains MPS and DSLs

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Alexander Grieshofer
Registration Number 01625732

to the Faculty of Informatics

at the TU Wien

Advisor: Ass.Prof. Dr. Dominik Bork
Assistance: Gabriel Morais, MSc

Vienna, 23rd October, 2023

Alexander Grieshofer	Dominik Bork

Erklärung zur Verfassung der Arbeit

Alexander Grieshofer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. Oktober 2023

Alexander Grieshofer

Abstract

In the realm of cloud application deployment in the context of DevOps, a significant challenge emerges concerning the effective configuration of microservices. This challenge appears from the necessity for multiple components, such as deployment manifests and continuous deployment pipelines, along with tool-specific configuration files, leading to complexity and an elevated risk of errors.

The Configuration File Generation (ConF Gen) language for Cloud Foundry deployments offers an innovative solution to the complicated world of system configuration. In a landscape where system setups grow increasingly complex, this domain-specific language (DSL) provides an efficient alternative to manual configuration processes.

The DSL uses a single model, the *SystemModel*, to generate various outputs, making configuration quicker and more reliable. Consequently, JetBrains MPS's capabilities in handling diverse outputs with minimal user input will be explored.

A solution designed to enhance the configuration of Cloud Foundry deployments will be introduced, including its strengths, limitations regarding MPS and other challenges that unveiled during the research process.

Contents

CHAPTER 1

Introduction

This work uses Model-Driven Software Engineering (MDSE) in order to address the complexity of cloud software systems. MDSE relies on models which guide the development process by bridging the gap between the core implementation and the high-level representation of software artifacts [BCW17]. Domain-specific languages (DSLs), as an essential component of the MDSE framework, are tailored to address a certain domain or context to simplify the descriptive task of people for that domain. These DSLs operate with a more abstract model with the objective of generating source code in order to create a working application. Notably, the scope of code generation is not restricted to programming languages. It facilitates the transformation of diverse models into various of software artifacts such as test cases, documentation, or in the context of this study, configuration files.

"Model-driven Software Engineering in Practice" by Brambilla et al. [BCW17] stands as an inspiring work in the field of software engineering. This comprehensive book provides invaluable insights and guidance on the practical application of model-driven software engineering principles. By offering a deep understanding of MDSE's key concepts, methodologies, and real-world applications, this book has played a pivotal role in shaping the landscape of modern software development practices, illustrating the impact of MDSE in improving the efficiency and quality of software development processes.

In the context of cloud applications, the challenges that emerge when deploying a microservice in a DevOps environment were considered. To properly configure a microservice, typically a deployment manifest, which the cloud platform uses at runtime, and a continuous deployment pipeline, a fundamental component for DevOps practices, are required. Furthermore, the complexity of this setup may involve the automatic generation of additional configuration files tailored to different tools and platforms. For instance, when deploying a microservice, a domain-specific language can streamline this process. It can automatically generate configuration files for the deployment manifest and continuous deployment pipeline, ensuring that they adhere to best practices and

platform requirements. Moreover, it can create additional configuration files needed to comply with specific platform management tools. This level of automation not only simplifies configuration but also greatly reduces the risk of errors.

To address these challenges, the approach selected involved the development of a domain-specific language designed for generating configuration files tailored to Cloud Foundry[1] deployments. The Configuration File Generation (ConF Gen) language aims to make Cloud Foundry deployments easier by utilizing a single model, called the *SystemModel*, where everything begins to unfold. It's a central file from which various outputs or generation targets originate, ensuring consistency across multiple outputs. By using the language and its integrated generation targets (Manifest, Pipeline-CD, and Moneysaver), users don't have to spend time manually creating configuration files for each part of their system. Instead, they can utilize the *SystemModel* and its core concepts – *Application*, *System*, and *Service* – making the setup process quicker and simpler.

While JetBrains MPS[2] is capable of building DSLs, the capabilities of handling different outputs based on a singular model still had to be explored. The objective was to achieve this functionality while concurrently minimizing the user's input effort.

Adhering to the Design Science Research cycle [Wie14], this work aims to deliver a valuable DSL solution that enhances the generation of Cloud Foundry configuration files, thereby enhancing the efficiency of cloud application deployments. In the evaluation phase of this research cycle, a comparison was conducted between the user-provided files and the DSL-generated outputs to ensure accuracy. Additionally, an experienced system architect evaluated the DSL and its real-world performance.

In this work, a solution designed to enhance the configuration of Cloud Foundry deployments will be introduced. This solution showcases its strengths while unveiling limitations regarding MPS and other challenges that appeared during the research process. Subsequent chapters will provide an explanation of how and why this solution works, showing how it makes system configuration more straightforward, faster, and less prone to errors.

[1] https://www.cloudfoundry.org
[2] https://www.jetbrains.com/mps/

CHAPTER 2

Background

This section provides the background needed understand this work. Cloud Foundry and domain-specific languages (DSLs) workbenches will be described, including and overview of JetBrains MPS, a projectional editor.

2.1 Cloud Foundry

"Cloud Foundry is a platform for running applications, tasks, and services. Its purpose is to change the way applications, tasks, and services are deployed and run by significantly reducing the develop-to-deployment cycle time. As a cloud-native platform, Cloud Foundry directly uses cloud-based infrastructure so that applications running on the platform can be infrastructure unaware. Cloud Foundry provides a contract between itself and your cloud-native apps to run them predictably and reliably, even in the face of unreliable infrastructure." [Win17, p. 1]

Cloud Foundry supports the full application development lifecycle [Hat], allowing developers to build, deploy, and run containerized applications. It employs a container-based architecture, enabling the execution and management of applications in various programming languages across different cloud service providers, both public and private. This multi-cloud environment enables seamless workload migration without altering the application code.

2.2 DSL Workbenches

Domain-specific languages (DSLs) are specialized languages crafted to describe specific aspects of software systems, such as algorithms, configuration specifications, or domain-specific processes [BCCP21]. Supporters of DSLs argue that using a combination of single-purpose DSLs can bring numerous benefits, including increased abstraction levels,

reduced code errors, decreased technology lock-in, and improved communication between developers and non-technical stakeholders.

DSL workbenches, such as JetBrains MPS and Xtext[1] are tools for developing domain-specific languages. As discussed in [Bet13], key difference between them is the projectional editing versus the textual editing. MPS uses projectional editing, where developers manipulate the Abstract Syntax Tree (AST) directly. This approach does not rely on parsing text, which allows for highly customized notations and language structures. It can be more powerful but may have a steeper learning curve and might feel unfamiliar to novice users. The textual editing of Xtext is common to developers who are used to working with code as text. Developers create DSLs by specifying grammars in a text-based format, and Xtext generates code based on these grammars.

2.2.1 Jetbrains MPS

Domain-specific languages have gained prominence in software development due to their ability to enhance productivity and reduce errors by providing specialized notations tailored to specific problem domains [BCCP21]. JetBrains Meta-Programming System (MPS) is an open-source language workbench designed to facilitate the creation, management, and tooling of DSLs [BCCP21].

JetBrains MPS is a language workbench with an emphasis on DSLs, developed by JetBrains since the early 2000s. The acronym "MPS" stands for Meta-Programming System, highlighting its core mission of enabling meta-programming [BCCP21], which involves the creation of languages and comprehensive tooling for programming. Meta-programming stands for the manipulation or transformation of languages [She01].

As presented in [BCCP21], MPS revolutionizes code editing by employing a projectional editor, a concept originating from the 1970s and mainly adopted by non-mainstream programming tools. This method allows direct manipulation of the in-memory code representation, in contrast to traditional character-based typing. This method is similar to editing math formulas in text processors. MPS aims to make projectional editing more widely applicable by introducing node transformations. When a user presses a key, it triggers an event within the Abstract Syntax Tree (AST) wherever the cursor is currently located, rather than inserting a character into a text document. Registered listeners on the relevant AST node respond to this event by transforming the AST to represent the character, enhancing the user-experience.

[1] https://projects.eclipse.org/projects/modeling.tmf.xtext

CHAPTER 3

Related work

The project in [Joh22] aligns with this work regarding the approach of using model-driven development in order to automate and abstract complex configuration tasks. To deploy software as microservices, it advocates the use of containers, particularly within a container cluster like Kubernetes. However, it acknowledges the fundamental challenge: the complexity and repetitiveness of writing Kubernetes deployment files. In response to this challenge, the research project explores the potential of model-driven development to simplify the creation of Kubernetes deployment files with the core objective of designing and implementing a domain-specific language.

Both projects, this work and the related research project [Joh22], use model-driven development to enhance automation and abstraction. However, the specific target domains differ, with the related research tackling Kubernetes deployment complexities for microservices versus this approach of streamlining Cloud Foundry configuration. Both methods try evolving through the landscape of software configuration in the digital age by leveraging innovative approaches to address modern challenges.

The research of Morais et al. [MA20][MBA21] recognized the changing landscape of architectural styles, with a specific focus on the emerging Microservices Architecture (MSA) continuously replacing monolithic systems. As MSA gains popularity, there is a growing need to promptly discover its fundamental principles and commonly accepted patterns and anti-patterns to practitioners. The driving force behind this effort is to provide a human and machine-readable representation of MSA's core concepts including relationships among them.

To meet this challenge, they have employed an ontology-based approach to refine the representation of MSA concepts and principles, ultimately creating OMSAC (Ontology of Microservices Architecture Concepts) [MA20]. The primary objective of OMSAC is to facilitate the development of support tools aimed at enhancing the exploration, understanding, and utilization of Microservices Architecture. As such, the scope of this

ontology is tailored to capture and represent the concepts and principles relevant to MSA.

Their research aligns with this study in the realm of knowledge representation, offering various perspectives on microservices within a more abstract framework. However, a notable distinction is that this work focuses on runtime configuration and abstracting diverse cooperating microservices into a higher-level language.

Previous research has also delved into related domains like the development of a domain-specific language for diverse mobile target platforms. These research approaches in [SŢCS13] and [Man11] share a common objective with this work: primarily, the reduction of development time and complexity, elevating the level of abstraction and enhancing efficiency.

The approach outlined in [SŢCS13] employs model-driven development to amplify the level of abstraction within the mobile application development process, thereby achieving platform independence. Utilizing this model, code generators are employed to construct platform-specific, native applications for all targeted platforms. This form of DSL enhances productivity and platform independence while limiting a complex and powerful configuration compared to platform-specific programming languages. Within this context, this work of Steiner et al. employs the Xtext workbench to design a DSL that remarkably reduces source code, up to 86 percent, particularly for simple applications.

A notable distinction between this work and the aforementioned research lies in the emphasis on generating deployment configuration files for the deployment of applications on Cloud Foundry.

CHAPTER 4

Research Design

This work employs the Design Science (DS) method [Wie14] as its research framework, following a structured research cycle to develop a domain-specific language for generating configuration files tailored for Cloud Foundry deployments. This DS research cycle is composed of the following incremental and iterative steps:

1. Problem Identification and Motivation: Defining the problem to solve
2. Defining Objectives of a Solution: Shaping the ideal artifact
3. Design and Development: Crafting the DSL artifact
4. Demonstration: Practical application of the artifact
5. Evaluation: Assessing effectiveness and efficiency
6. Communication: Sharing the artifact

By adhering to this structured Design Science Research cycle, this work aims to contribute a valuable DSL solution to enhance Cloud Foundry configuration file generation, ultimately advancing the efficiency and reliability of cloud application deployments.

4.1 Problem Identification and Motivation

Users may possess a clear understanding of the desired output, yet they may lack proficiency in domain-specific languages and JetBrains MPS techniques required for efficient creation. Consequently, there is an opportunity to enhance the efficiency and user-friendliness of the output generation process by bridging the gap between user expertise and the generation of abstract DSLs using JetBrains MPS. This opportunity

emerged from a deep engagement with a highly experienced system architect with over 20 years of expertise in crafting cloud manifests and deployment and management pipelines for various systems based on microservices.

This architect manually creates configurations for a cloud platform, utilizing Tamzu as the cloud provider and Concourse as the tool for pipeline orchestration. Currently, this process is labor-intensive and involves distributing critical information across multiple configuration files. The motivation is to streamline and automate this task, with a primary goal of combining all the essential data into a single document. By doing so, the architect not only anticipates an increase in efficiency but also recognizes the advantages of centralizing vital information for better system management.

However, the architect has raised a concern regarding the portability of the technical stack, which is in a continuous evolving state. Therefore, the solution needs to rely on a standard and multi-platform approach to ensure that it can seamlessly adapt to evolving technologies and platforms.

The architect has provided examples of the configuration files he routinely constructs. These exemplars serve as a fundamental reference for the development of abstract DSLs within the JetBrains MPS framework, thus contributing to the evolution of an adaptable, and user-friendly solution that aligns with the constantly evolving technical landscape.

Therefore, the research question to be addressed is as follows:

How can templates be used in JetBrains MPS to generate outputs and abstract DSLs?

In this context, templates refer to suitable outputs or the term "generation-targets", which will be introduced in Section 4.3.

4.2 Defining Objectives of a Solution

From the problem identification, as discussed in the previous step, I have carefully considered the capabilities and limitations of JetBrains MPS resulting from the insights of sources such as [Voe], [VKS+19], [Fowa], and [Fowb]. Notably, the installation of MPS comes bundled with numerous samples, in order to get further inspiration for best practices. In addition, I have fortified my understanding through personal exploration of JetBrains MPS, particularly focusing on projectional editing. Furthermore, an essential aspect of this objective-setting process was the engagement in discussions with the client to validate the resulting objectives.

The objectives that the developed DSL should meet are as follows:

- **Centralization of Information:** It should enable the association of vital information from multiple configuration files into a single document, streamlining the management and deployment processes.

- **Efficiency Enhancement:** The DSL should significantly improve the efficiency of creating configuration files.
- **User-Friendliness:** The DSL should be user-friendly, ensuring that individuals with varying levels of expertise can effectively utilize it to generate configurations.
- **Future-Proofing:** The DSL must be designed with flexibility in mind, allowing it to be adaptable to potential changes in the technical stack or platform.

4.3 Design and Development

The development of the domain-specific language unfolded through a series of iterative phases. Initially, the goal was to construct the language as closely as possible to the Cloud Foundry manifest[1], with a strong focus on ensuring user-friendliness. To achieve this objective, both the Cloud Foundry Manifest metamodel[2], illustrated in Figure 5.1, and the example files provided by the system architect were leveraged. This initial phase was accomplished by a continuous exchange of insights and experiences with other MPS users, complemented by a process of trial and error, which collectively contributed to the creation of the first version of the artifact. During this phase the following components of the language were built:

- **Structure**: Defines the building blocks and relationships of the language.
- **Editor**: Controls the visual and interactive representation.
- **Intentions**: Offers quick-fixes and code modification suggestions.
- **Typesystem**: Enforces type rules for language constructs.
- **Constraints**: Define additional rules and conditions on language elements.
- **Behaviour**: Provide common operation on nodes.

In the subsequent phase, an extension designed to facilitate the generation of actual files for the manifest was introduced. A dedicated generator language was added to streamline the generation of output files for the manifest, ensuring that the system stays adaptable to future changes.

With a focus on the objectives of centralizing information and enhancing efficiency, the language was further enhanced to accommodate additional configurations, specifically the domains "Pipeline-CD" and "Moneysaver". Consequently, the concept of "generation-targets", as illustrated in Figure 4.1, was introduced to address a limitation imposed by MPS, wherein only one generator could be active at a given time. It's important to

[1] https://docs.cloudfoundry.org/devguide/deploy-apps/manifest.html
[2] The metamodel was made available through the contribution of Gabriel Morais.

emphasize that each generation target requires a distinct language extension built upon the foundational ConF Gen base language. These extension languages exclusively contain the generator logic, while the structural aspects mentioned earlier are implemented within the core language. With this iteration of the development cycle the language was able to efficiently construct different outputs files based on a single model, called the *SystemModel*, where every configuration aspect was stored.

However, due to the aforementioned limitation concerning the active status of a single generator, a mechanism to enable the selection of a specific generation target through the utilization of the intentions menu or within the tools-tab was needed. This was a limiting factor by MPS but consequently lead to improvements regarding the user-friendliness aspect by providing clarity in the user's selection of output.

Furthermore, it is essential to note another technical limitation: due to existing bugs in the YAML plugin[3], all generators currently produce XML files instead of the intended YAML format. External tools and converters can be used to transform the XML output to the desired YAML output.

To generate files aligned with the selected target, users need to execute the "Preview Generated Text" action while the active generator is in use. For each specific generation target, it's crucial to provide all the required values as defined by the DSL. This ensures that the resulting XML file is valid, as omitting values may lead to incomplete XML tags. Despite the system's general attempt to minimize erroneous or empty fields, there remains the possibility of such occurrences. In the event that users identify mistakes, incorrect values, or missing information, they have the flexibility to amend missing values and repeat the regeneration process as many times as necessary to correct any issues. This iterative approach empowers users to fine-tune and perfect their output according to their requirements.

To view, test, and make practical use of the generated files, it's essential to use a XML to YAML converter. This tool effectively converts valid XML files into YAML format, which is compatible with Cloud Foundry deployment requirements.

For an in-depth exploration of the DSL, including its usage and operational principles, please refer to Section 5.2.

4.4 Demonstration

In this step, the practical application of the DSL will be presented:

1. **Defining a model:** The process begins with the user defining a SystemModel within the ConF Gen language. In this model, various configurations, settings, and information are stored, which correspond to the desired cloud infrastructure or

[3]https://plugins.jetbrains.com/plugin/16835-dataformats

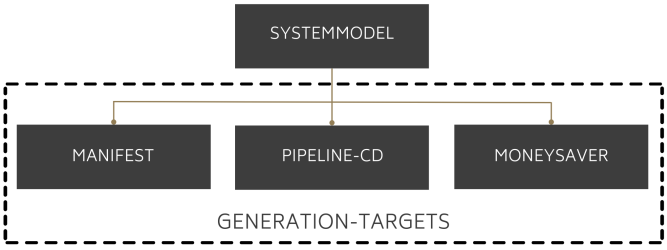

Figure 4.1: Generation-Targets

deployment setup. This model acts as a structured representation of the user's specifications.

2. **Changing the target generator:** Within the DSL, users can select a specific "generation target". This selection corresponds to the type of output they wish to create. For example, if they want to generate a manifest output, they would employ the intention mechanism "Switch Generation Target to XML-Manifest".
3. **Generating:** Once the model is defined, and the target generator is chosen, the user initiates the generation process within the DSL. The DSL's generator language then takes the model and uses its configuration rules to create the output in XML format. This XML output contains all the relevant configuration from the SystemModel for the specific generation target.
4. **Transforming the output XML into YAML:** Given that the DSL only generates output in XML, users can employ external tools and converters to transform the generated XML into YAML.
5. **Using the files in the Cloud Foundry platform:** Finally, the generated YAML files, which now contains the configuration information in a format suitable for the Cloud Foundry platform, can be used.

For a detailed walk-through on to how use and operate the DSL, please refer to Section 5.1 and 5.3. Furthermore, there are examples showcasing various configurations and the according output files of the DSL.

4.5 Evaluation

The evaluation was conducted as follows:

- **Comparative analysis with user-provided files**: One essential aspect of evaluating the effectiveness of the DSL is to compare the output files generated by the DSL with the files provided by the system architect. This comparison helps ensure that the DSL accurately captures the user's requirements and successfully translates them into functional output files.

- **User-driven IDE exploration**: To measure the user-friendliness of the DSL, the user-expert is provided with access to the IDE equipped with the DSL. This hands-on approach allows the user to explore the DSL's features, experiment with different configurations, and assess the ease with which they can interact with the DSL to define models and select generation targets.
- **User-expert feedback on interface usability and file generation**: Feedback from the user-expert is a critical component of the evaluation process. The user-expert's insights and observations on the simplicity of using the DSL interface and the process of generating various files are invaluable. The feedback provides real-world insights into the user-friendliness and practicality of the DSL. Additionally, it helps identify any areas for improvement or refinement in the DSL's design and functionality. It's worth noting that there is no formal written evaluation process in place. Instead, this feedback was gathered during weekly meetings with the user-expert, where insights and experiences were exchanged to guarantee that all requirements were met. These regular interactions allowed for a dynamic and ongoing assessment of the DSL's performance and usability. This collaborative approach ensures that any issues or areas for enhancement could be addressed promptly, making the DSL a more effective and user-friendly tool for its intended purposes.

4.6 Communication

The DSL is comprehensively documented, ensuring that users have access to clear and informative resources for both understanding the language and effectively employing it. This documentation encompasses written materials, live demos and trainings, and practical demonstrations to support users with the DSL.

Detailed written documentation was created that covers every aspect of the DSL, from its core concepts to advanced functionalities. This written documentation serves as a comprehensive reference, guiding users through the DSL's features, best practices, and use cases. To enhance the learning experience, live demonstrations of the DSL in action were prepared. These live demos provide users with the opportunity to witness how the DSL is employed, facilitating a deeper understanding of its capabilities. For hands-on learning, live training sessions were conducted, allowing users to interact with the DSL. In addition to live demos and training, recorded demonstrations were provided. The DSL repository is accessible at the following web address: https://github.com/UQAR-TUW/alex_mps_yml_templating

CHAPTER 5

The ConF Gen DSL

With the aim of streamlining Cloud Foundry deployments, the Configuration File Generation (ConF Gen) DSL is introduced. Tailored specifically for Cloud Foundry deployments, it revolves around the SystemModel, a central model responsible for generating diverse outputs. Through the utilization of ConF Gen's integrated generation targets, including "Manifest", "Pipeline-CD", and "Moneysaver", the efficiency of creating Cloud Foundry configurations will be enhanced and vital information of diverse configuration files will be centralized. The following sections provide a detailed explanation of the usage and the operational principles.

5.1 General use

The user can input the required configuration values. The intention dropdown menu helps in specifying attributes or keys effectively. By using the keyboard shortcut (Mac: "Option + Return", Windows/Linux: "Alt + Return"), a menu displaying current interactions is revealed. Most of the intentions are context-specific, meaning they relate to the current concept in the SystemModel or its underlying concepts. Certain intentions are consistently displayed regardless of context.

Another feature is the suggestions menu. By pressing "Control + Space" on Mac or "Ctrl + Space" on Windows or Linux, a list of potential values is presented. This feature is particularly beneficial for enumerations or references where pre-existing values are available.

The tool tab in the IDE toolbar allows the user to modify the SystemModel's generation targets: Manifest, Pipeline-CD and Moneysaver. Additionally, this can be accomplished at any point using the intentions menu, as these intentions are always accessible.

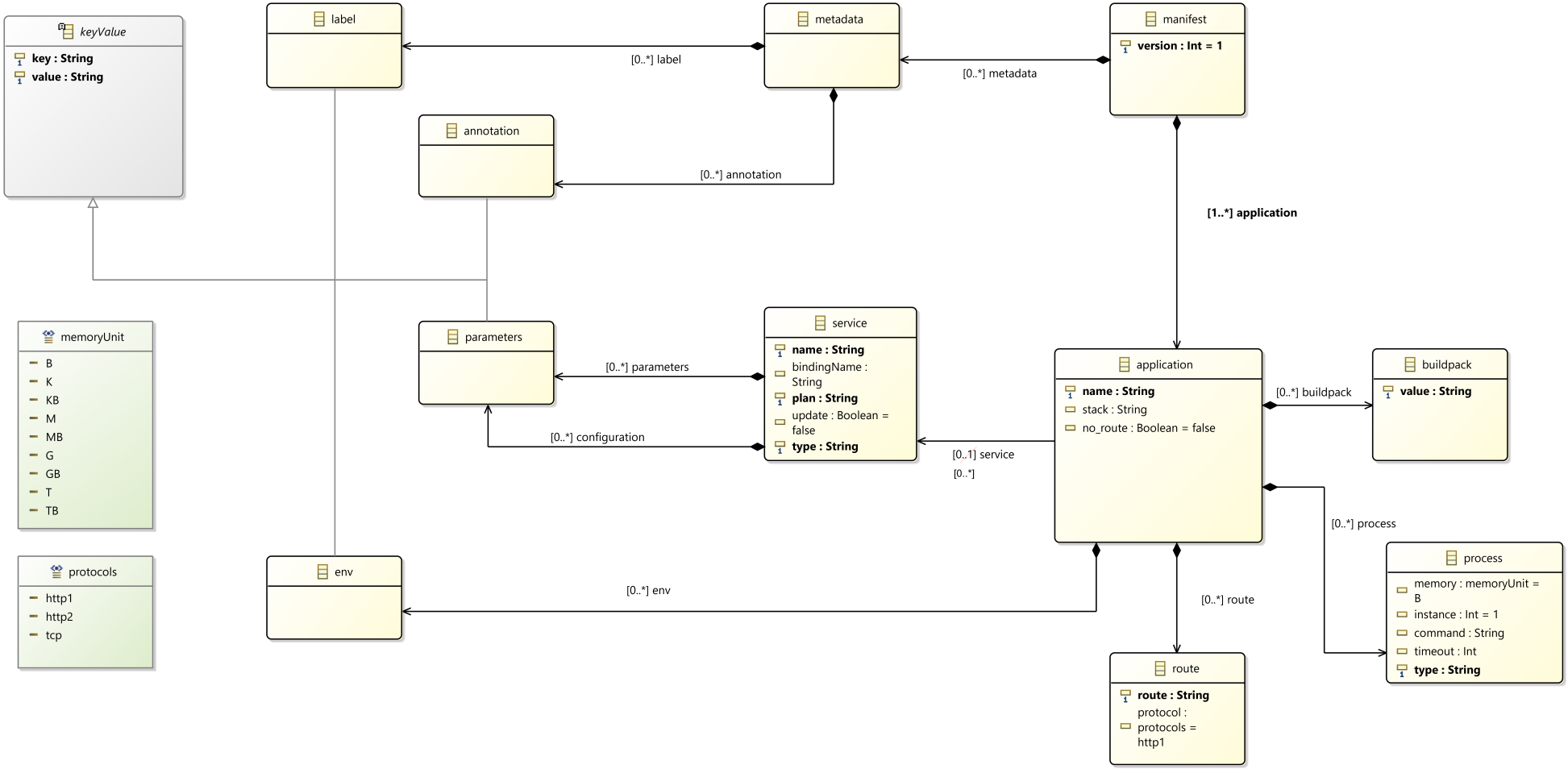

Figure 5.1: Metamodel Cloud Foundry Manifest

5.2 The SystemModel

At its core, the SystemModel unifies system requirements. It allows users to describe individual applications or systems, defining various attributes and configurations. This comprehensive view is valuable when orchestrating complex systems where multiple applications interact, offering a global and comprehensive perspective on the system without the need to explore individual manifests. This capability simplifies the system architecture, making it more accessible and manageable.

Moreover, the real value of the SystemModel becomes evident when it comes to manifest generation. It can automatically generate individual manifests for each application. This automation significantly reduces the manual, time-consuming tasks, saving users time and effort while ensuring a more streamlined and error-free deployment process. The SystemModel is at the core of simplifying complex system management and making it more efficient.

During the initial developmental phase, I leveraged both the Cloud Foundry Manifest metamodel, as illustrated in Figure 5.1, and the example files provided by the system architect. The Cloud Foundry Manifest metamodel describes the structure and relationships of elements within a Cloud Foundry manifest, which is used to define the configuration of applications and services to be deployed on the Cloud Foundry platform. Here is a breakdown of the elements in this metamodel:

- **Manifest (root element):** It specifies the overall manifest, and it can have one or more applications associated with it.
- **Application**: Each application is characterized by a *name*, *stack*, including whether it has a *no_route* property. An application can have zero or more *buildpacks*, *processes*, *routes*, and *environments (env)* associated with it.
- **Buildpack**: Describes the buildpack used for the application, identified by a value.
- **Process**: Represents the processes associated with an application. It defines properties such as *instance*, *command*, *timeout*, *type* and the enumeration *memory*.
- **Route**: Describes the routing information for the application, including the *route* itself and the *protocol* as an enumeration.
- **Service**: Represents the services that the application relies on, with properties like *name*, *bindingName*, *plan*, *update*, *type*, and other service-related details like *label*, *annotation*, *parameters*, and *env*.
- **Enumerations**: This section specifies two enumerations: *memoryUnit* and *protocols*.

- **Compositions**: These associations define how the elements in the manifest are related. For instance, a manifest can have multiple applications, and each application can have multiple buildpacks, processes, routes, and environment variables. Similarly, a manifest can be associated with metadata, and services can have parameters, annotations, and labels.
- **Direct Associations**: This section specifies a direct association between applications and services, indicating that applications can be associated with zero or more services.

A SystemModel encompasses multiple child elements, each possessing its own set of attributes and potentially containing additional nested child elements. Among these elements, the three primary core concepts that form the backbone of the SystemModel are:

- Application
- System
- Service

Upon the creation of a new SystemModel root node, the user is prompted to input values for the attributes and populate the child elements with their respective configurations. This process enables the customization and specification of the SystemModel according to the user's requirements and desired system behavior.

In the subsequent chapters, a detailed explanation will be provided on how to effectively utilize and harness the power of these three core concepts within the SystemModel. This comprehensive understanding will empower users to accurately describe their system, configure its components, and generate the corresponding configuration files necessary for the successful deployment and operation of their system.

5.2.1 Modelling the System Concept

The SystemModel contains the *system* concept, which represents overall information. The attributes *organization*, *team*, and *contact* need to be specified by the user as string values. The element *components* (referred to as the *referencedApplications* concept) is a list of applications referenced within the SystemModel. It cannot be set without its corresponding counterpart and should be created or removed using the intentions menu.

The *RunningRules*, defined under the *system* concept, apply globally to every *application* defined in the SystemModel, without considering application-specific *RunningRules*. The *start* and *stop* entries for a trigger should not be interpreted as typical start-stop or from-until values. They are entirely independent and incorporate a 30-minute timeframe during the generation step. For the user, a general guideline is to define a time in the

format "HH:MM AM/PM" for each entry and consider the trigger's end time to be 30 minutes after the defined value. For additional details regarding the timeframe, please refer to the Section 5.3.2.

Moreover, the *RunningRules* can include day restrictions that limit the applicability of rules to specific days. These restrictions can only be set using the intentions menu to prevent user interference. If all days are selected, the entry will be removed as this configuration would be redundant.

In the context of the global *RunningRules* concept, there is an additional feature that allows the definition of spaces where the rules will be applied. Example spaces which are commonly found in a system setup can include values such as development, staging, and production. Each space can have its own set of day restrictions and start-stop values, providing a higher level of configuration flexibility.

By specifying day restrictions for a particular space, the *RunningRules* will only be applicable on the specified days within that space. This allows for fine-grained control over when the rules should be enforced, allowing for specific requirements or workflows associated with each space.

Similarly, start-stop values can be assigned to spaces, enabling the definition of time intervals during which the *RunningRules* should be active. These timeframes should be viewed as intervals for the activation or deactivation of triggers, during which the triggers can be either initiated or terminated. This feature allows for greater customization by specifying different time periods for each space.

However, it's crucial to remember that application-specific *RunningRules* always take precedence over the globally defined ones. This means that if a specific application has its own *RunningRules* configured, those rules will replace the global rules. The same principle applies to spaces as well. If a particular application within a specific space has its own set of rules defined, those rules will be prioritized over the *global RunningRules* for that specific application and space combination.

This hierarchical prioritization ensures that the system remains flexible and adaptable, allowing for the customization of *RunningRules* at both the global and application-specific levels, as well as within different spaces.

5.2.2 Modelling the Service Concept

The fundamental idea of the root concept *service* possesses the capability to function either independently as an individual node or as an integral part of a SystemModel. This means that a *service* can function as a standalone one or can be bound to an application. Similar to the application to component binding in the *system* concept, a new *service* binding should be added using the intentions menu. This ensures automatic linking or removal of the binding.

The attributes assigned to a *service* include the *type*, *name*, *command*, *plan*, and an *update_service* boolean. Additionally, by utilizing the intentions menu, users can define

a configuration for the *service* using a key-value list, allowing for customization and fine-tuning of its settings. Furthermore, it is possible to include additional tags that provide supplementary information related to the service.

By leveraging these attributes and options through the intentions menu, users can effectively configure and manage services within the SystemModel.

5.2.3 Modelling the Application Concept

An application resides within a SystemModel and serves as a mandatory concept with a cardinality ranging from 1 to n. The *name* attribute of the application must be defined as a string and is automatically linked as a reference to the components list within the *system* concept. The user is required to provide a valid repository URL as a string, beginning with either "http://" or "https://", along with a *stack* string and a *no_route* boolean. The intention functionality enables the user to set several child concepts specific to the application, including:

- Annotation
- Buildpack
- Environment
- Label
- Process
- Route
- RunningRules

To ensure seamless integration and removal of new *applications* with the components list within the *system* concept, it is crucial for users to leverage the intentions menu. This menu serves as a hub for creating and removing *applications*, offering a range of options for accurately linking and unlinking them with the *system's* components. Furthermore, the intentions associated with *applications* provide users with a versatile toolkit to add *metadata*, including *annotations*, *labels*, and other configurations, thereby enhancing the overall functionality and customization of the *applications*.

Within an *application*, a dedicated list exists for *buildpacks*, allowing users to add specific *buildpacks* tailored to their requirements. Additionally, a key-value list is available to manage *environment* entries. It is important to note that this list primarily defines the global environment and does not directly encompass environment variables.

By explicitly specifying the desired *route* and *protocol*, users can define multiple *routes* for their *applications*. The protocol options, namely "http1", "http2", and "tcp", can be selected from a predefined enumeration. It is worth mentioning that the management of

route configurations, including their addition or removal, is exclusively facilitated through the intentions menu, ensuring a structured and controlled process.

Metadata, a crucial aspect of application configuration, is divided into two distinct fields: *Annotations* and *Labels*. Both fields enable users to define key-value entries that accommodate user-specific configurations. Furthermore, a *Process* is defined by its *Type*, *Command*, *Instances*, and *Memory*. The *memory* allocation for a process is specified in units such as "B", "KB", "MB", "GB", or "TB". In order to improve readability and minimize unit errors, the intentions menu offers a quickfix-feature that automatically converts values to higher units if applied by the user. This not only eliminates the need for large numbers but also enhances the overall usability of the application configuration process.

Additionally, each *application* possesses *RunningRules* that are unique to that particular *application*. While these rules are based on the principles of globally defined *RunningRules*, they carry higher priority and are exclusively applicable to the specific *application* to which they are assigned. Similarly to global *RunningRules*, it is important to note that the *start* and *stop* entries associated with a trigger should not be misconstrued as conventional start-stop or from-until values. Instead, they operate as independent entries, incorporating a 30-minute timeframe during the generation step. To ensure consistent and accurate configuration, users are obliged to define time entries in the format of "HH:MM AM/PM" for each entry. Errors are identified for values that do not adhere to the specified format, using a regular expression for verification. Moreover, it is crucial to consider that the trigger's end time is set 30 minutes after the explicitly defined value, ensuring a standardized timeframe for effective trigger management. An example can be found in the generation-target Section 5.3.2.

In the manner of globally defined *RunningRules*, an additional feature allows users to define *spaces* where these rules will be applied. In cases where a globally defined *space* exists, the application-specific *RunningRules* can override the default configurations, enabling fine-grained customization and control. This overwrite functionality extends to the start/stop entries and day restrictions, providing users with the flexibility to tailor the *RunningRules* to the unique requirements of each *application*. By specifying day restrictions within a particular space, users can limit the applicability of *RunningRules* to specific days within that *space*, ensuring a more nuanced and granular approach to the management of *RunningRules*.

Similarly, start-stop values can be assigned to *spaces*, empowering users to define distinct time intervals during which the *RunningRules* should be active. This advanced feature facilitates comprehensive customization by allowing different time periods to be specified for each *space* and *application*. These timeframes should be viewed as intervals for the activation or deactivation of triggers, during which the triggers can be either initiated or terminated.

Furthermore, it is possible to establish a connection between an *application* and a new or pre-existing *service* by designating the *service* as a reference. This process mirrors the

practice of binding an *application* to the components list within the *system*, except in this case, a list containing references to *services* is utilized. It is important to note that the configuration of these *services* cannot be altered within this context, as their settings and specifications are encapsulated within the *service* concept itself. Bear in mind that *service* can live inside a SystemModel or separate as standalone root elements.

5.3 Usage & Operation of the DSL

The ConF Gen language[1] offers three generation targets: Manifest, Pipeline-CD, and Moneysaver. These targets can be created based on the user-provided information. However, due to existing bugs in the YAML plugin, all generators currently produce XML files instead of YAML files. Nevertheless, it is feasible to implement a solution where the generator can directly generate YAML files, if desired.

In this solution, only one generator can be active at a time. Therefore, the user needs to select the desired generator either through the intentions-menu or in the tools-tab. To generate the files according to the selected target, the "Preview Generated Text" action must be triggered while the active generator is in use.

For each specific generation target, it is important to provide all the required values. This ensures that the resulting XML file is valid, as missing values may result in incomplete XML tags. Despite the generator's general attempt to minimize incorrect or empty fields, it is still possible for them to occur. If any mistakes, incorrect values, or missing information are identified by the user, missing values can be added and the regeneration process can be repeated as many times as necessary to correct them.

5.3.1 Generation Target: Manifest

Manifests are crucial when deploying applications on platforms like Cloud Foundry (CF). They specify how each application should work, including the resources and settings it needs, ensuring an enhanced deployment process. When you use the "Manifest" generation target in the DSL, it automatically creates manifest files, making the deployment process easier.

As indicated in Figure 5.2 the fundamental concept known as SystemModel serves as the root node in the generation tree, encompassing all other concepts. Most of the crucial information used throughout the system originates from the Application concept. However, there is one specific piece of data from the System concept that comes into play: the organization name. This organization name is utilized to assign a name, in the form of a prefix, to the output file. In order to initiate the generation process in Jetbrains MPS, a root node and a "root_mapping_rule", is required. This root_mapping_rule serves as the starting point for generating the desired output. It is the anchor from which the entire generation process begins to unfold.

[1] https://github.com/UQAR-TUW/alex_mps_yml_templating

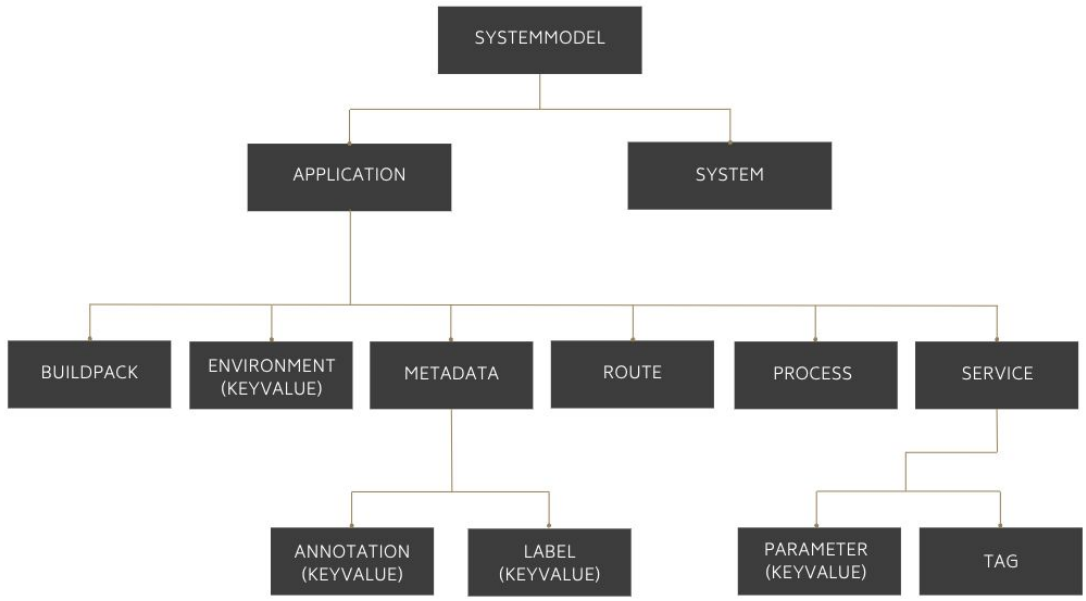

Figure 5.2: Concepts of the generation target: manifest

Another key concept in the generation of manifests, alongside the SystemModel, is the *Application*. It incorporates various other concepts that define additional configurations for each specified application. From a generation perspective, each *application* is encapsulated within an *application* XML tag nested inside the top-level *applications* XML tag. Since the solution allows for multiple *applications* to be defined and configured, it is logical to establish a parent tag and iterate over each *application* element with its respective configuration values. To maintain a clean and concise SystemModel mapping file while handling application-specific generation, a "Copy_SRC_LIST_MACRO" is employed, separating the application-specific generation into its own reduce file.

The reduce file, in conjunction with reduction rules and a designated concept, plays a crucial role in generating the desired output language or code. It serves as a dedicated space where the content of a concept can be processed and transformed.

Within an *Application* XML tag, attributes such as name, stack, and no_route are defined. For attributes that can have multiple values, an additional loop is implemented to iterate over each value. Once again, the "Copy_SRC_LIST_MACRO" is utilized to achieve this behavior.

Essentially, every remaining concept (*Buildpack*, *Environment*, *Metadata*, *Process*, *Service*) within the *application* reduce file follows this processing approach. The list macro iterates through the entries and generates the corresponding output for each entry encountered.

To address specific requirements, such as handling empty values and preventing the appearance of empty parent XML tags, a template macro called "IF" is employed. This

macro serves as a conditional mechanism, allowing the generator to check for empty values within certain concepts, namely *environment*, *metadata*, *process*, and *service*. In cases where these concepts contain multiple entries, they are wrapped within a parent XML tag to facilitate proper organization and containment. Consequently, if there are no entries within these concepts, the parent wrapper is not displayed, thereby preventing the creation of empty parent XML tags. The "IF" macro plays a vital role in ensuring consistency and optimal structure within the generated output.

However, it is worth noting that there exists one inconsistency: Specifically, the *metadata* parent XML tag is displayed if any of its child elements, namely *Annotations* or *Labels*, are not empty.

At a deeper level of iteration, we explore into the value level XML tags for concepts such as *Environment*, *Annotation*, and *Label*. These concepts follow a key-value pair structure, and therefore share a single reduce file named KeyValue. This reduce file generates XML layouts in the format <key> value </key> with the corresponding values as specific to each concept, considering *Environment*, *Annotation*, and *Label*.

On the other hand, the *Buildpack* and *Tag* concepts have their own dedicated reduction files. They both utilize the same XML tags for their values, namely <buildpacks> [value] </buildpacks> and <tags> [value] </tags>, but with distinct specified values. Notably, these concepts do not require a wrapping XML tag, as all entries are specified with the *buildpacks* or *tags* key and their respective values. This design choice is made possible due to the XML-to-YAML parser grouping *buildpacks* within an application or tags within a service configuration. In contrast, other concepts cannot employ this approach as they either have predefined keys or consist of multiple entries that oblige a wrapping XML tag.

Moving forward, the *route* concept employs a <route> wrapping tag to group together the entries for name and protocol. Given the desired output format, it is necessary to utilize a wrapping XML tag in this case. Routes should be generated as individual route entries with their respective values. The protocol tag is only displayed if it is specified. Similar to earlier cases, the node macro "IF" is utilized to control the visibility of the protocol tag.

Next, we encounter the *Process* concept, which includes essential attributes such as type, command, instance, and memory. Again, due to the specific output format requirements, the only viable solution is to employ a wrapping XML tag. A distinction lies in the memory attribute, which combines an integer "Amount" with the "MemoryUnit" enumeration. While it combines two values, it does not differ after generation in terms of the final output from other values.

Lastly, the *Service* concept adheres to the same structure and techniques as before. It employs a wrapping XML tag with fixed keys and user-specified generated values. Additionally, it incorporates the node macro "IF" to determine whether a wrapping XML tag <parameters> should be displayed, based on whether the service is configured or

not. Tags are displayed using the "Copy_SRC_LIST_MACRO" method, without the need for a wrapping tag as mentioned earlier.

The illustrated example in Listings 5.1 and 5.2 provides an illustration of a generation process involving two *applications* and one *service*. In this scenario, one of the *applications*, referred to as app1, is fully customized with all possible configurations, while the other application, app2, only has the required attributes of repository, stack, and no_route defined. Additionally, the *service* is fully configured and bound to app1.

The generator generates a parent XML tag, <applications>, within which the two *applications* and their corresponding configuration values are generated. In this case, there are two <application> XML tags representing the two apps. Starting with app1, all attributes are defined within their respective XML tags and grouped accordingly. Following this, a <services> XML tag is introduced, where the bound services are positioned. Here, the single bound service with all its configurations can be found within app2. Lastly, app2 is presented with only the minimum required attributes.

```
system:
 organization: manifest_example
 team: team
 contact: example@domain.at

 applications:
 - name: app1
   repository: https://github.com/repository
   stack: stack
   no_route: false
   buildpacks:
   - buildpack
   environment:
      key1: val1
   routes:
   - route: route1
     protocol: http1
   metadata:
      annotations:
        anno1: val1
      labels:
        label1: val1
   processes:
   - type: process
     command: command
     instances: 1
     memory: 128MB
   services:
   - service1
 - name: app2
   repository: https://github.com/repository
   stack: stack
   no_route: false

 services:
```

```
- type: type
  name: service1
  command: command
  plan: plan
  update_service: false
  configuration:
     key: val
  tags:
     tag1
```

Listing 5.1: A SystemModel as input for the generation of Manifest configuration files

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no"?>
<applications>
  <application>
    <name>app1</name>
    <stack>stack</stack>
    <no_route>false</no_route>
    <buildpacks>buildpack</buildpacks>
    <environment>
      <key1>val1</key1>
    </environment>
    <routes>
      <route>
        <name>route1</name>
        <protocol>http1</protocol>
      </route>
    </routes>
    <metadata>
      <annotations>
        <anno1>val1</anno1>
      </annotations>
      <labels>
        <label1>val1</label1>
      </labels>
    </metadata>
    <processes>
      <process>
        <type>process</type>
        <command>command</command>
        <instances>1</instances>
        <memory>128MB</memory>
      </process>
    </processes>
    <services>
      <service>
        <type>type</type>
        <name>service1</name>
        <plan>plan</plan>
        <parameters>
          <key>val</key>
        </parameters>
        <tags>tag1</tags>
      </service>
```

```
43      </services>
44    </application>
45    <application>
46      <name>app2</name>
47      <stack>stack</stack>
48      <no_route>false</no_route>
49    </application>
50  </applications>
```

Listing 5.2: Manifest Generation: Output

5.3.2 Generation Target: Moneysaver

The "Moneysaver" generation target is another integral element of the DSL. It generates an additional output based on the SystemModel when the "Moneysaver" generation is activated. The term "Moneysaver" is used as a descriptive name for the generation target to emphasize its primary function, which is to optimize resource allocation and utilization within a Cloud Foundry environment.

According to Figure 5.3, the primary concept called SystemModel serves as the foundational node in the generation tree, encompassing all other concepts. Vital information is derived from the *Application* and *System* concepts. Generation is occasionally performed using both concepts simultaneously, as attributes such as *RunningRules* and *GlobalRunningRules* define the output, with the application-specific *RunningRules* taking precedence. A comprehensive explanation will be provided.

The name of the organization is used to assign a prefix to the output file. To initiate the generation process, a root node and a "root_mapping_rule" are necessary. The root_mapping_rule acts as the starting point for generating the desired output. It serves as the anchor from which the entire generation process unfolds.

Starting with the fundamental concept *System*, the attribute organization is utilized to designate the output file's name as a prefix. Additionally, the key focus of this section is the generation of *RunningRules*, specifically *GlobalRunningRules*. These *RunningRules* generate resources that globally define triggers for every application. They can be more specific if a *Space* with additional *(Global-)RunningRules* is defined. Multiple *spaces* can be defined. Each *RunningRule* contains a start and stop value, along with a day restriction.

On the other hand, application-specific *RunningRules* are defined within the *Application* concept. The configurations of these *RunningRules* are identical to the aforementioned *GlobalRunningRules* but at the application level. They hold higher priority since the *application* does not rely on the triggers defined globally; instead, it utilizes its own defined rules. During the generation step, the generator compares and checks both concepts for rules and utilizes them accordingly.

At the outset of the generated file, fixed values for attributes such as name, type, and source are provided within an <resource_types> XML tag. Subsequently, a section

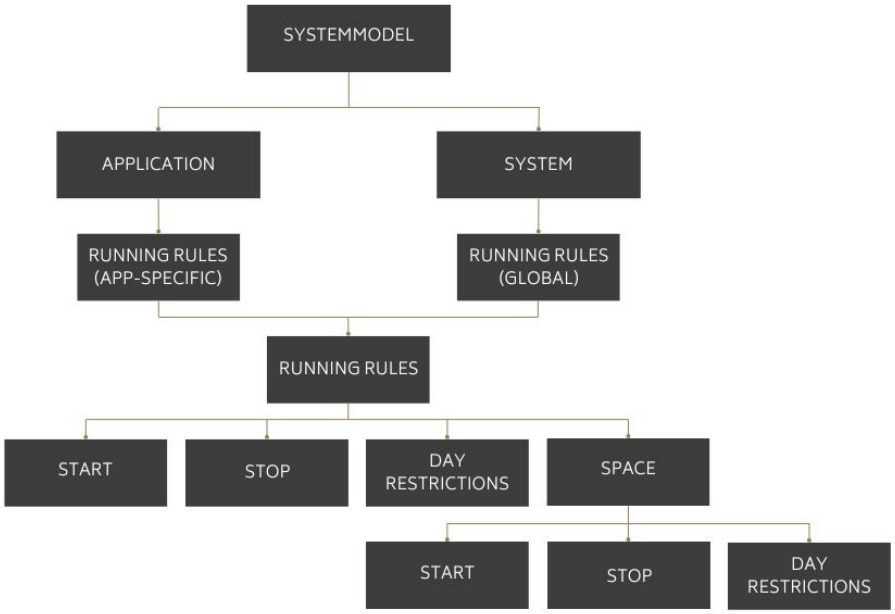

Figure 5.3: Concepts of the generation target: Moneysaver

is generated for triggers. Each trigger is defined within a <resources> XML tag and contains two <resource> XML tags, one for the start trigger and the other for the stop trigger. The <source> tag includes predetermined values for <interval> and <location>. The remaining entries, namely <start>, <stop>, and <days>, are mapped using the values specified within the currently iterated *space*. Both triggers are essentially identical except for the name and the values of start/stop/days. The name within the <name> tag is generated as follows: "trigger-[space]-start" for the start trigger or "trigger-[space]-stop" for the stop trigger. At this stage, the start-stop values can be considered as typical start-stop values since they are now divided into their specific triggers, each containing a 30-minute time slot for triggering the start or stop actions. To establish a 30-minute execution window, a function takes the start time in the format "HH:MM AM/PM" and adds 30 minutes to calculate the stop time. Thus, this function is called twice for each trigger: firstly, to compute the stop time for the start trigger, and secondly, to calculate the stop time for the stop trigger. This generation process is achieved by combining an "IF" and a "COPY_SRCL" node macro that iterates over application-specific and globally defined *spaces*. Consequently, every *space* defined in the SystemModel is generated into a trigger containing the specified values. For the sake of readability, the mapping exists in a separate reduce file called "reduce_Space". To examine and iterate over each *space* defined in the SystemModel, it employs a combination of an "IF" statement and a "COPY_SRCL" node macro.

Following the trigger generation, a brief generated section maps the *System* concepts into a <resources> XML tag. It shares the same attributes as the previously mentioned

triggers, except with different fixed values for name, type, and source. Notably, the <source> tag includes the attributes api, organization, and skip_cert_check, where only the organization is mapped to its corresponding value as stated within the *system* concept. A “COPY_SRC” node macro is used to generate this section.

As mentioned earlier keep in mind that during the generation process, both the *system* and *application* concepts are utilized to determine the output. It checks whether the *application* has its own *RunningRules* defined, and if not, the globally defined *GlobalRunningRules* take precedence.

The subsequent section is responsible for generating <jobs> XML tags. Once again, a combination of node macros is utilized, particularly an "IF" statement and two "LOOP" node macros. Firstly, it checks for the existence of the *RunningRules* using the IF macro, and then it iterates over each *application* and subsequently over each *space* within that *application*. For every *space* defined in either the *application-specific RunningRules* or the *GlobalRunningRules*, two job tags are generated: one for the start trigger and another for the stop trigger. Each job tag includes attributes such as <name>, <public>, <serial>, and <plan>. The name within the <name> tag is generated as follows: "start-apps-[space]-time" for the start job or "stop-apps-[space]-time" for the stop job.

To ensure proper grouping when converting from XML to YAML, two <plan> XML tags are used. The first plan tag contains a <get> tag with the value "trigger-start" or "trigger-stop". The second plan tag includes additional attributes with fixed values based on the trigger type. It consists of a <put> tag with a fixed value and a <params> tag containing attributes such as <commands>, <space>, and <app_name>. The command tag holds either the value "start" or "stop" depending on the trigger. The *space* tag contains the space name, while the app_name is a list of apps associated with that *space*.

Regarding *GlobalRunningRules*, the app_name list is generated using a combination of "LOOP" and "IF" node macros. These macros iterate over each *application* and check for the presence of application-specific *RunningRules*. If they are not defined, the globally defined rules take precedence, and the *application* is added to the app-list for that *space*. In the case of an *application* with *RunningRules* defined, the app-list for that *space* will contain only one entry, making it unnecessary to implement the node macros.

The illustration presented in Listing 5.3 and 5.4 demonstrates the generation process involving three *applications* and three *spaces*. In this scenario, there are two *spaces* defined globally, along with an application-specific *space* within app1. As a result, a total of six triggers (three start triggers and three stop triggers) will be generated. Following the trigger section, there is a brief segment responsible for generating the paas-resource, which primarily consists of fixed values except for the organizational name. Subsequently, there are six <jobs> tags structured as described earlier. Among these jobs, two of them correspond to app-specific start and stop actions, containing tags with values associated with the application-specific *RunningRules*. The app-list in these jobs consists of a single entry: app1.

On the other hand, the remaining four jobs relate to globally defined *GlobalRunningRules*, with each job representing a combination of start and stop triggers. These jobs include tags with values aligned with the globally defined *GlobalRunningRules* and their respective *spaces*. In this case, the app-list comprises two entries: app2 and app3.

```
system:
 organization: moneysaver_example
 team: team1
 contact: example@domain.at
 global_running_rules:
  triggers:
    start: 08:00 AM
    stop: 08:30 PM
    days: Monday
    space:
    - name: globalspace1
    - name: globalspace2
      start: 09:00 AM
      stop: 09:30 PM
      days: Monday Tuesday Wednesday

 applications:
 - name: app1
   repository: https://github.com/repository
   stack: stack
   no_route: false
   running_rules:
     triggers:
       start: 10:00 AM
       stop: 10:30 PM
       space:
       - name: appspace1
 - name: app2
   repository: https://github.com/repository
   stack: stack
   no_route: false
 - name: app3
   repository: https://github.com/repository
   stack: stack
   no_route: false
```

Listing 5.3: A SystemModel as input for the generation of Moneysaver configuration files

```
<?xml version = "1.0" encoding = "UTF-8" standalone = "no"?>
<moneysaver>
  <resource_types>
    <name>cf-cli-resource</name>
    <type>registry-image</type>
    <source>
      <repository>nulldriver/cf-cli-resource</repository>
      <tag>latest</tag>
    </source>
  </resource_types>
```

```
<resources>
  <resource>
    <name>trigger-globalspace1-start</name>
    <type>time</type>
    <source>
      <interval>5m</interval>
      <start>09:00 AM</start>
      <stop>09:30 AM</stop>
      <days>
        <day>Monday</day>
        <day>Tuesday</day>
        <day>Wednesday</day>
      </days>
      <location>America/Montreal</location>
    </source>
  </resource>
  <resource>
    <name>trigger-globalspace1-stop</name>
    <type>time</type>
    <source>
      <interval>5m</interval>
      <start>09:30 PM</start>
      <stop>10:00 PM</stop>
      <days>
        <day>Monday</day>
        <day>Tuesday</day>
        <day>Wednesday</day>
      </days>
      <location>America/Montreal</location>
    </source>
  </resource>
</resources>
<resources>
  <resource>
    <name>trigger-globalspace2-start</name>
    <type>time</type>
    <source>
      <interval>5m</interval>
      <start>08:00 AM</start>
      <stop>08:30 AM</stop>
      <days>
        <day>Monday</day>
      </days>
      <location>America/Montreal</location>
    </source>
  </resource>
  <resource>
    <name>trigger-globalspace2-stop</name>
    <type>time</type>
    <source>
      <interval>5m</interval>
      <start>08:30 PM</start>
      <stop>09:00 PM</stop>
```

```
64        <days>
65          <day>Monday</day>
66        </days>
67        <location>America/Montreal</location>
68      </source>
69    </resource>
70  </resources>
71  <resources>
72    <resource>
73      <name>trigger-appspace1-start</name>
74      <type>time</type>
75      <source>
76        <interval>5m</interval>
77        <start>10:00 AM</start>
78        <stop>10:30 AM</stop>
79        <location>America/Montreal</location>
80      </source>
81    </resource>
82    <resource>
83      <name>trigger-appspace1-stop</name>
84      <type>time</type>
85      <source>
86        <interval>5m</interval>
87        <start>10:30 PM</start>
88        <stop>11:00 PM</stop>
89        <location>America/Montreal</location>
90      </source>
91    </resource>
92  </resources>
93  <resources>
94    <resource>
95      <name>paas-resource</name>
96      <type>cf</type>
97      <source>
98        <api>https://api.run.pivotal.io</api>
99        <organization>org1</organization>
100       <skip_cert_check>false</skip_cert_check>
101     </source>
102   </resource>
103 </resources>
104 <jobs>
105   <name>start-apps-appspace1-time</name>
106   <public>true</public>
107   <serial>true</serial>
108   <plan>
109     <get>trigger-start</get>
110   </plan>
111   <plan>
112     <put>paas-resource</put>
113     <params>
114       <commands>
115         <command>start</command>
116       </commands>
```

```
117         <space>appspace1</space>
118         <app_name>
119           <app>app1</app>
120         </app_name>
121       </params>
122     </plan>
123   </jobs>
124   <jobs>
125     <name>stop-apps-appspace1-time</name>
126     <public>true</public>
127     <serial>true</serial>
128     <plan>
129       <get>trigger-stop</get>
130     </plan>
131     <plan>
132       <put>paas-resource</put>
133       <params>
134         <commands>
135           <command>start</command>
136         </commands>
137         <space>appspace1</space>
138         <app_name>
139           <app>app1</app>
140         </app_name>
141       </params>
142     </plan>
143   </jobs>
144   <jobs>
145     <name>start-apps-globalspace1-time</name>
146     <public>true</public>
147     <serial>true</serial>
148     <plan>
149       <get>trigger-start</get>
150     </plan>
151     <plan>
152       <put>paas-resource</put>
153       <params>
154         <commands>
155           <command>start</command>
156         </commands>
157         <space>globalspace1</space>
158         <app_name>
159           <app>app2</app>
160           <app>app3</app>
161         </app_name>
162       </params>
163     </plan>
164   </jobs>
165   <jobs>
166     <name>start-apps-globalspace2-time</name>
167     <public>true</public>
168     <serial>true</serial>
169     <plan>
```

```
      <get>trigger-start</get>
    </plan>
    <plan>
      <put>paas-resource</put>
      <params>
        <commands>
          <command>start</command>
        </commands>
        <space>globalspace2</space>
        <app_name>
          <app>app2</app>
          <app>app3</app>
        </app_name>
      </params>
    </plan>
  </jobs>
  <jobs>
    <name>stop-apps-globalspace1-time</name>
    <public>true</public>
    <serial>true</serial>
    <plan>
      <get>trigger-stop</get>
    </plan>
    <plan>
      <put>paas-resource</put>
      <params>
        <commands>
          <command>stop</command>
        </commands>
        <space>globalspace1</space>
        <app_name>
          <app>app2</app>
          <app>app3</app>
        </app_name>
      </params>
    </plan>
  </jobs>
  <jobs>
    <name>stop-apps-globalspace2-time</name>
    <public>true</public>
    <serial>true</serial>
    <plan>
      <get>trigger-stop</get>
    </plan>
    <plan>
      <put>paas-resource</put>
      <params>
        <commands>
          <command>stop</command>
        </commands>
        <space>globalspace2</space>
        <app_name>
          <app>app2</app>
```

```
223             <app>app3</app>
224           </app_name>
225         </params>
226       </plan>
227     </jobs>
228 </moneysaver>
```

Listing 5.4: Moneysaver Generation: Output

5.3.3 Generation Target: Pipeline-CD

Configuration files for continuous deployment pipelines, which serve as a fundamental component in DevOps practices, are generated through the use of the "Pipeline-CD" generation target.

The distinctive feature of the "Pipeline-CD" generation target's generator sets it apart from other generators. This particular generator has the capability of producing two output files. The first file is a parameter file that contains attributes related to the *System* and *Application* concept. Specifically, it includes the attributes organization and contact from the *system* concept, as well as the attributes *name* and *repository* from the *application* concept as illustrated in Figure 5.4. These attributes are enclosed within a <parameters> XML tag to adhere to the XML format.

On the other hand, the second file generated by this generator is a "service-creation" file, which is structured with an <jobs> XML tag as the primary wrapping tag. Inside

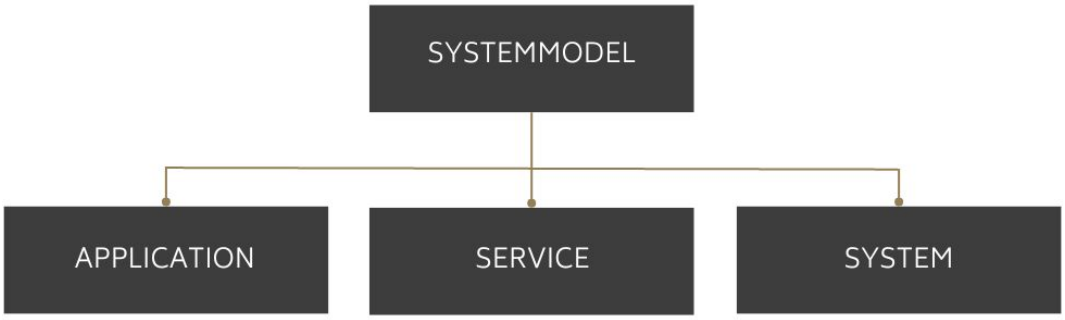

Figure 5.4: Concepts of the generation target:Pipeline-CD

this tag, there is a <name> tag with a fixed value, and a <plan> tag for the main output. Each service is produced within a <try> XML tag. This method of generation, previously used and adopted here, proves suitable for the required functionality. A parser that converts XML to YAML would combine multiple <try> tags together, as desired.

The following section is created utilizing the *service* concept. A <try> tag consists of a <put> XML tag with a fixed value, and a <params> XML tag that contains attributes relating to the currently iterated *service*. These attributes include <command>, <service>, <plan>, <service_instance>, <configuration>, <tags>, <wait>, and <update_service>. While mainly the associated value is used for the output, the command-tag differs in terms of a prefix: A “create-“ prefix is added to the given value. The <configuration> tag further contains key-value pairs in the format <key> [value] </key>, while the <tags> tag contains tags in the format <tag> [tag-value] </tag>. It is important to note that both the <configuration> and <tags> XML tags are generated only if they have defined values; otherwise, they will not be included in the output. This condition is achieved using an "IF" node-macro. Furthermore, the output of these tags is processed by a separate "reduce_file" and iterated through a "COPY_SRCL" node-macro. Another node-macro, known as the "LOOP" macro, is used to iterate through the defined services, each iteration producing a <try> tag.

The provided listings, specifically 5.5, 5.6, and 5.7, demonstrate an illustrative example of a generation process involving two *services*. These *services* are referred to as "service1" and "service2." In this context, "service2" is fully customized with configurations and tags, whereas "service1" contains only the required attributes, namely type, name, command, plan, and update_services. Both *services* are associated with an *application*.

The generation process results in the creation of two distinct files: a parameter file and a service-creation file. During the current stage of development, the parameter file always utilizes the values from the first *application*. Other values relate to the specific *service* and *system* concept.

The second file, the service-creation file, is generated based on the *services* bound to the *application*. In this particular scenario, the service-creation file includes a wrapping plan-tag that contains two try-tags. Each try-tag corresponds to either "service1" or "service2" and encapsulates their respective values in the previously described format.

```
1  system:
2   organization: pipeline_example
3   team: team1
4   contact: example@domain.at
5
6  applications:
7  - name: app1
8    repository: https://github.com/repository
9    stack: stack
10   no_route: false
11   services:
12   - service1
13   - service2
```

```
14
15 services:
16 - type: type1
17   name: service1
18   command: command1
19   plan: plan1
20   update_service: false
21 - type: type2
22   name: service2
23   command: command2
24   plan: plan2
25   update_service: false
26   configuration:
27    key: val
28   tags:
29    tag1
```

Listing 5.5: A SystemModel as input for the generation of Pipeline-CD configuration files

```
1 <?xml version = "1.0" encoding = "UTF-8" standalone = "no"?>
2 <parameters>
3   <org>pipeline_example</org>
4   <app_name>app1</app_name>
5   <git-url>https://github.com/repository</git-url>
6   <contact-mail>example@domain.at</contact-mail>
7 </parameters>
```

Listing 5.6: Pipeline-CD Generation Output: Parameter file

```
1 <?xml version = "1.0" encoding = "UTF-8" standalone = "no"?>
2 <jobs>
3   <name>create-services</name>
4   <plan>
5     <try>
6       <put>cf-cli</put>
7       <params>
8         <command>create-service1</command>
9         <service>type1</service>
10        <plan>plan1</plan>
11        <service_instance>service1</service_instance>
12        <wait>true</wait>
13        <update_service>true</update_service>
14      </params>
15    </try>
16    <try>
17      <put>cf-cli</put>
18      <params>
19        <command>create-service2</command>
20        <service>type2</service>
21        <plan>plan2</plan>
22        <service_instance>service2</service_instance>
23        <configuration>
24          <key>val</key>
25        </configuration>
```

```
26          <tags>
27            <tag>tag1</tag>
28          </tags>
29          <wait>true</wait>
30          <update_service>true</update_service>
31        </params>
32      </try>
33    </plan>
34 </jobs>
```

Listing 5.7: Pipeline-CD Generation Output: Service-Creation file

5.3.4 Standalone IDE

Jetbrains MPS possesses the capability to construct language plugins for MPS itself, create language plugins for IntelliJ, or even develop a Standalone Integrated Development Environment (IDE) tailored for specific languages [mpsa]. In consideration of the demonstration objectives, I have chosen the Standalone IDE option due to its streamlined focus. Standalone IDEs offer an efficient means of publishing domain-specific languages to end users, enabling them to utilize these languages within an IDE environment enriched with comprehensive support, including refactorings and code analysis, thoughtfully crafted by language designers [mpsb]. This dedicated IDE ensures the removal of unnecessary language-design-related functionalities and irrelevant languages, thus optimizing the user experience.

The project includes a standalone solution that can generate snapshots for Windows, MacOS, and Linux. To build the IDE, users only need to adjust the base directory path and the MPS path specified within the CloudFoundryDistribution file under “base directory” and "Macros – Folder – mps_home". To initiate the build process, users can run the "CloudFoundry" and "CloudFoundryDistribution" solutions. As a result, MPS will create a folder structure in the directory: build/artifacts/CloudFoundryDistribution, where you can find the compiled IDEs.

CHAPTER 6

Discussion & Limitations

In this section, an exposition of the solution will be provided, including its limitations, encompassing the shortcomings, challenges, and other issues that emerged throughout the research process.

The presence of existing bugs within the YAML plugin[1], which leads to the unintended generation of XML files rather than the desired YAML format, presents a technical challenge. A YAML format is the prefered choice due to its human-readability, clear structural representation, as its adoption in Cloud Foundry deployments. A potential workaround, which has been employed for cross-verification of generated output, involves using a converter[2] to transform XML into YAML. This limitation can be addressed by investing effort into resolving the bugs within the YAML plugin and enhancing the generators to support the updated version of the plugin. Despite this limitation, it demonstrates the adaptability of the solution to accommodate future improvements to existing constraints. For more detailed information on this issue, please refer to the "YAML-Generator" branch in the GitHub repository[3].

The requirement of having only one active generator at any given time is an unwanted design choice. While it ensures clarity in the user's selection of output, MPS restricts concurrent activation of multiple generators. To address this, the intentions menu and the tools tab have been introduced as mechanisms to switch the generation target.

Although the generators validate whether users have provided all required values, there remain potential scenarios where invalid output may occur. This limitation is a consequence of having a complex generator with numerous MPS macros, particularly nested loops due to structure concepts enclosed into each other. It also raises questions about potential enhancements to the generator's robustness and error-handling capabilities.

[1] https://plugins.jetbrains.com/plugin/16835-dataformats
[2] https://jsonformatter.org/xml-to-yaml
[3] https://github.com/UQAR-TUW/alex_mps_yml_templating/tree/YAML-Generator

JetBrains MPS can be effectively utilized to generate outputs and abstract DSLs. The ConF Gen DSL, as demonstrated, underscores the viability of this approach. Templates offer a structured and user-friendly means to specify desired output formats, while DSLs provide a higher-level abstraction for defining requirements. The system's ability to produce tailored outputs for various targets (Manifest, Pipeline-CD, and Moneysaver) highlights its capacity to abstract DSL concepts and generate outputs in a user-centric manner.

Although JetBrains MPS excels in constructing DSLs, the goal delved into the potential of handling various outputs derived from a single model. The aim was to achieve this functionality while streamlining the user's input effort. Exploring the capabilities of JetBrains MPS in handling different outputs with minimal effort is an essential aspect of this research. It is crucial to understand that there exists a high level of complexity, which prevent non MPS experts from adapting and customizing these outputs. To illustrate this, a scenario where a shift to a different cloud platform or a change of tools is required, can be considered. This change in context underscores the clear need for an expert to investigate the adaptation and extension aspects.

In the course of this research, I have developed a domain-specific language tailored for generating configuration files specific to Cloud Foundry deployments. As detailed in earlier chapters, the ConF Gen DSL encompasses a complex structure that enables the generation of multiple outputs from a single SystemModel. However, an obvious limitation appears in the context of adapting these outputs to various other cloud platforms, such as AWS or Google Cloud, when using JetBrains MPS. This limitation underscores the critical role of an MPS expert in customizing and expanding the outputs. This extends beyond the present generation targets (Manifest, Pipeline-CD, and Moneysaver), which are specifically designed for Cloud Foundry and tools such as the cloud provider Tamzu and the Concourse orchestration tool.

Despite the current limitation of generating XML files due to issues with the YAML plugin, the possibility of extensions suggests that it can be adapted to support direct YAML file generation, thus strengthening its versatility. While challenges and limitations exist, including the need for users to provide all required values and the constraint of a single active generator, the overall findings support the affirmative answer to the research question. The solution's iterative refinement and user empowerment underscore its potential to enhance output generation and DSL abstraction within the JetBrains MPS environment.

CHAPTER 7

Conclusion

A solution for enhancing cloud foundry deployment configuration was presented, demonstrating its strengths, and unveiling its limitations, and challenges that emerged throughout the research process.

The DSL was developed with a clearly defined set of user objectives. It successfully achieves the centralization of information by consolidating data from various configuration files into a single SystemModel, enhances efficiency through the generation of multiple outputs, ensures user-friendliness through a continuous exchange of insights and experiences, and provides future-proofing by remaining adaptable to potential changes.

At its core, the SystemModel unifies system requirements. It allows users to describe individual applications or systems, defining various attributes and configurations. This comprehensive view is valuable when orchestrating complex systems where multiple applications interact, offering comprehensive perspective on the system without the need to explore individual manifests. This capability simplifies the system architecture, making it more accessible and manageable. JetBrains MPS can be used to to generate outputs and abstract DSLs. The ConF Gen DSL serves as a verification to the feasibility of this approach.

The solution was manually tested and evaluated using the integrated Run-Processes of MPS. To expand its utility and assess its effectiveness, integrating this solution with continuous integration pipelines of popular cloud platforms like AWS or Google Cloud is a viable opportunity for future work. Templates and DSLs tailored for specific cloud providers could facilitate seamless configuration and deployment.

The system's potential for supporting direct YAML file generation is a promising area for future research. Although it currently generates XML files due to issues with the YAML plugin, the extensibility of the system implies the feasibility of adapting it to facilitate direct YAML file generation. This prospect opens the door to addressing current limitations and enhancing the overall functionality of the system.

In conclusion, this work represents a significant progress toward simplifying output generation and DSL abstraction using templates in JetBrains MPS, with a demonstrated capacity to address constraints and adapt to future improvements in the field.

Listings

List of Figures

Bibliography

[BCCP21] Antonio Bucchiarone, Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. *Domain-Specific Languages in Practice*. Springer International Publishing, 2021.

[BCW17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-driven Software Engineering in Practice, Second Edition*. Springer International Publishing AG, 2nd edition, 2017.

[Bet13] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt, 2013.

[Fowa] Martin Fowler. A language workbench in action - mps. https://martinfowler.com/articles/mpsAgree.html. [Accessed: 30-09-2023].

[Fowb] Martin Fowler. Language workbenches: The killer-app for domain specific languages? https://martinfowler.com/articles/languageWorkbench.html. [Accessed: 30-09-2023].

[Hat] Red Hat. What is cloud foundry? https://www.redhat.com/en/topics/application-modernization/what-is-cloud-foundry? [Accessed: 27-09-2023].

[Joh22] Daniel Johansson. Model-driven development for microservices: A domain-specific modeling language for kubernetes, 2022.

[MA20] Gabriel Morais and Mehdi Adda. Omsac-ontology of microservices architecture concepts. In *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 0293–0301. IEEE, 2020.

[Man11] Ashwin Kumar Manjunatha. A domain specific language based approach for developing complex cloud computing applications, 2011.

[MBA21] Gabriel Morais, Dominik Bork, and Mehdi Adda. Towards an ontology-driven approach to model and analyze microservices architectures. In *Proceedings of the 13th International Conference on Management of Digital EcoSystems*, MEDES '21, page 79–86, New York, NY, USA, 2021. Association for Computing Machinery.

[mpsa] *Jetbrains MPS - Build Language.* Available at https://www.jetbrains.com/help/mps/build-language.html [Accessed: 30-09-2023].

[mpsb] *Jetbrains MPS - Building Standalone IDEs for your language.* Available at https://www.jetbrains.com/help/mps/building-standalone-ides-for-your-languages.html [Accessed: 30-09-2023].

[She01] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, pages 2–44, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[SŢCS13] Dustin Steiner, Cătălina Ţurlea, Cristian Culea, and Stephan Selinger. Model-driven development of cloud-connected mobile applications using dsls with xtext. In Roberto Moreno-Díaz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *Computer Aided Systems Theory - EUROCAST 2013*, pages 409–416, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[VKS+19] Markus Völter, Bernd Kolb, Tamas Szabo, Daniel Ratiu, and Arie Deursen. Lessons learned from developing mbeddr: a case study in language engineering with mps. *Software & Systems Modeling*, 18, 02 2019.

[Voe] Markus Voelter. Dsl best practices. https://www.voelter.de/data/pub/DSLBestPractices-2011Update.pdf. [Accessed: 30-09-2023].

[Wie14] Roel J. Wieringa. *Design Science Methodology for Information Systems and Software Engineering.* Springer Berlin, Heidelberg, 2014.

[Win17] Duncan C. E. Winn. *Cloud Foundry: The Definitive Guide.* O'Reilly Media, 2017.