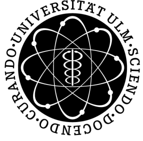

ulm university universität **u**ulm

Universität Ulm | 89069 Ulm | Germany

Faculty of Engineering, Computer Science and Psychology
Databases and Information Systems Department

Design and Implementation of a Generic Framework for Rule-based Automated User Role Management

Master's thesis at Universität Ulm

Submitted by:
Sandro Eiler
sandro.eiler@uni-ulm.de

Reviewer:
Prof. Dr. Manfred Reichert
Prof. Dr. Rüdiger Pryss

Supervisor:
Robin Kraft

2020

Version from April 17, 2020

© 2020 Sandro Eiler

This work is licensed under the Creative Commons. Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/de/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Composition: PDF-LaTeX 2_ε

Abstract

Many systems that host groups of users, like social media platforms, provide solutions or tools for users and groups of users. Within these groups, there can exist roles, which again might be connected to permissions. The role model itself can vary from system to system. Some are hierarchical, some have flat hierarchies, some are related to the permissions, others with notifications. Decoupling the role model implementation from the rest of the system has multiple advantages, such as achieving automatic role assignment more easily and being able to change a role model on the fly.

Within this master thesis, challenges of automated role assignment are examined and necessary elements for how a development tool can help are distilled. The main contribution is the tool *karmantra*, which allows to integrate arbitrary role models in software projects and lets developers extend or alter them. The tool is kept in a very generic way to be expandable easily. With this tool, a step is taken towards decoupled and transparent role systems, that can not only serve the needs of common commercial platform needs.

Acknowledgment

This master thesis would not exist in this way without following wonderful people: Robin helped me as supervisor with his calm and prudent willingness to adapt to all needs that came up, not only for this master thesis. Erce supported me tirelessly with his programming experience, friendship and talks. And the lovely and idealistic *Karrot* group, allowed me not only to find my thesis subject. Through this group I learned a lot about the connection between democracy and computer science, group structures and utopistic living communities. Warm hugs to you Nick, Tilmann, Janina and Bruno!

My thanks also go to my closest friends, my flat mates and my family who supported me throughout the years. Without all that love, the things I do would not have the same meaning and higher purpose.

As this master thesis is written, the world is hit by the Corona Virus. I hope that we can make the best out of it and build a better society where humans stand together regardless of nationality and wealth.

Contents

1 Introduction **1**

1.1 Problem statement . . . 2

1.2 Objective . . . 3

1.3 Structure of the thesis . . . 4

2 Fundamentals **5**

2.1 Related work . . . 5

2.2 Definitions . . . 7

2.3 Rule-based group role models . . . 8

3 Requirements **13**

3.1 Functional requirements . . . 13

3.2 Non-functional requirements . . . 15

4 Design **17**

4.1 DevOps . . . 17

4.2 Application specification . . . 18

4.3 Data model . . . 22

4.4 Task model . . . 26

5 Implementation **29**

5.1 Development process . . . 29

5.2 Project structure . . . 31

5.3 Implementation components . . . 33

5.4 Addressing functional requirements . . . 40

6 Evaluation **41**

6.1 Testing . . . 41

6.2 Fulfillment of non-functional requirements . . . 43

6.3 Theoretical application of concept . . . 44

7 Conclusion **47**
7.1 Summary 47
7.2 Discussion 48
7.3 Outlook 49

1 Introduction

In social life, acting within groups has countless advantages over acting alone. No matter if we connect through work, a sports club, a political party, scientific collaborations or on social media platforms. We can observe that participating within groups always entails having roles. But groups are not only a matter of social relations. In computer systems, a user may be part of an administrator's or moderator's group for example. In this case, a role does not necessarily represent a social circumstance anymore, but becomes a matter of read and write privileges. Or to be more general, roles can be used to map to permissions within a system. Roles can be obvious like *employees* and *bosses* or *standard users* and *administrators*. But there can also be "hidden" roles, which a system may not or does not want to represent. Considering the social component, we can give examples like "the one that brings fun to the group" or "the one that is the driving force". On the other hand, roles that lead to permissions in a computer system can influence the social structures. Humans connect to others within groups because they are social beings. Today we can not only find humans within groups, like for example, a therapy dog. If we think groups more technically, members can not only be living entities. To satisfy the user's need of acting in groups and the developer's need of supplying software with different roles and permissions, innumerable tools, frameworks and systems exist, trying to provide a best solution.

How good a software solution is (for the users or the supplier) may depend on how interaction between groups members is made possible. As described, group members have roles. In software systems, roles are often assigned by administrative users. The assignment of roles is based on rules. Rules can be explicit and comprehensible. For a rule, being explicit means, that roles are applied on the basis of objective criteria.

Being comprehensible means, that a rule is understandable by group members and the assignment process is transparent to the software users. Conversely, rules that are not comprehensible and explicit can exist if they are kept in an administrator's mind who then performs role assignments. In a more technical way the aim of achieving explicit and comprehensible rules can be missed if rule implementation is nested and hidden in code or implemented in a complex way.

As we will examine, there are endless possibilities of role models that can be applied to systems. Some use cases demand a flexible approach that allows different role assignment models for different groups. Automating role assignments in a rule-based way can lead to clearer and more comprehensive assignment processes. This in turn can lead to positive effects in the sense that participation for improving a system is easier. We will see that through a rule-based automated user role management, group structures can be reflected more in a needed, natural and flexible way.

We will analyze the international online platform *Karrot*, which allows people from all over the world to connect in groups for saving food, and see how its needs for an automated rule-based role evaluation can be satisfied with this approach.

1.1 Problem statement

The development of a role model leads to various potential challenges:

- The role model may change over time and has to be re-implemented.
- The role model may not reflect the actual roles and needs within groups.
- Having a diverse user base raises the need of providing various role models for different groups. Reasons like the lack of development resources or the developer's will can prevent respecting such needs.
- A role model may be hidden or dispersed in code and therefore hard to learn by other developers and others with interest of understanding the role assignment. It might be striven to make a role model in a software comprehensible even for users without much technical background to allow a more holistic participation.

- Software tools focus on specific target groups only, such as enterprises or municipal stakeholders.
- Democratic group processes can be lived in arbitrary environments, like organizations and enterprises. The evaluation of member's roles can be an intended objective for such processes. Role assignment is rarely thought as a result of such democratic processes and therefore few software approaches include this premise.

Considering all these challenges can be difficult in a development process. A developer tool that supports building wanted role models in a way that it is decoupled from the rest of the system and that can be changed easily afterwards, might be helpful. For this it's important to figure out a meta model that can fit all needs of systems with rule-based role assignment. Additionally a generic tool has to have a high degree of module interchangeability, since arbitrary environments, operating systems and programming languages have to be supported. With this master thesis these challenges are addressed.

1.2 Objective

Having stated problems of implementing rule-based role assignment, this thesis strives to address these. The main contribution is the design of a role assignment solution for developers which is meant to be kept as generic as possible. This includes an implementation of the emerging concepts in form of a tool, called *karmantra*. First, an analysis and design is done of what is needed to fulfill arbitrary developer needs. This includes figuring out a meta model for role models and all its preconditions. The design also includes defining interchangeable implementation layers for *karmantra* to ensure, arbitrary developing environments can be supported in the long run. An important question for this project is, how decoupling of role evaluation can be implemented, especially in arbitrary systems. The tool *karmantra* is supposed to support a developer with generating and managing a decoupled role model, allowing an rule-based automated role evaluation.

Referring to the stated challenges, the following main objectives for the tool *karmantra* are going to be achieved:

- Automated rule-base role models can easily be generated and modified.
- The role model can be generated in a decoupled, comprehensible and clean way.
- The tool allows to integrate different programming languages and environments like e.g., *python* and *php*.

1.3 Structure of the thesis

In Chapter 2, basic concepts and terms are specified. Therefor, related work in this field is introduced. In particular, concepts which are needed to meet the thesis' objectives, are examined. Important examples for rule-based role models will be given. In Chapter 3, both functional and non-functional requirements are collected and described. In Chapter 4, the concepts and designs for the implemented framework are presented. The actual implementation solution is shown in Chapter 5, where the overall development process and the important implementation details for all framework modules can be found. In Chapter 6, the outcome is evaluated through defining tests and a theoretical application of concept for the online platform *Karrot*. Lastly the thesis approach and outcome are discussed to be able to give an outlook for future work.

2 Fundamentals

This chapter covers a retrospection on related scientific literature, presents basic concepts for role-based access control and specifies relevant terms of this master thesis.

2.1 Related work

Role assignment evolved with research on permission assignment that originates from security administration needs. Jin et al. describe [1] the first steps towards today's approaches with mentioning three influential access control models, namely the Discretionary Access Control (DAC) [2], the Mandatory Access Control (MAC) [3] and the Role-Based Access Control (RBAC) [4, 5].

2.1.1 Role-based access control

While the first strategies have been evolved from the 1970s [5], RBAC has become dominant later [1]. Enterprises' and other medium to large organizations' needs for security management [6] are addressed in previous papers on an Enterprise RBAC model (ERBAC) [7, 8]. Some drawbacks of only using roles have been presented by e.g., Kern et al. [6]: In bigger organizations, a large number of roles is needed if roles only contain explicit authorizations, which led to defining more generic roles as in [7]. Second, despite having advantages with using roles concerning permissions, the work of assigning roles to users is still a factor to be reduced. Efforts to automate this process are e.g., described in [8].

When it comes to using a role model, different approaches have been described. Epstein describes a top-down approach, called decomposition and a bottom-up approach, called aggregation. Decomposition means, that roles are decomposed into permissions, whereas aggregation means that permissions are aggregated into roles [9, 10].

2.1.2 Rule support for RBAC

Another approach, called Attribute-Based Access Control (ABAC) tries to combine the advantages of the three mentioned access control models [11] and takes the approach of assigning attributes to users on which permission granting can take place. Attributes can be anything like identities or roles for example. Al-Kahtani and Sandhu define a model, called Rule-Based RBAC (RB-RBAC) and define a simple extendable language for defining rules [11]. The language is solely descriptive and does not provide a mechanism to compare provided user attributes to the attributes required by a rule. In [11], triggers are mentioned as mechanism for initiation of a role evaluation task. In [6] it is stated, that a high level of automation can be achieved using rules for role models.

2.1.3 Seniority levels

With the model of Al-Kahtani and Sandhu, seniority levels of rules are introduced. This means that a value, which is used by a rule, can dominate another value. Seniority levels have to be predefined to be used by the role evaluation mechanism. If a user's attributes meet the conditions for a rule A and A's conditions are a subset of a rule B, we say that rule A is senior to rule B. Seniority levels, which can be described as rule hierarchies [6], can conflict with role hierarchies.

2.1.4 Provisioning

The presented approaches do not only focus on the role models and the related role evaluation. An important field is also the assignment of roles to users, which can be a lot of work [8, 6]. Kern and Walhorn propose a method called rule-based provisioning

and state that this allows a high level of automation. The expected effects are reduced administration costs and a high security level.

2.2 Definitions

In literature some definitions vary. To avoid confusion, the terms below are explained to have a distinct understanding of how the master thesis' contribution has to be understood.

karmantra

The developed tool, being the contribution of this master thesis, is called *karmantra*. *karmantra* originates from **Kar**rotish **Man**aged **T**rustbased **R**ole **A**ssignment. It was inspired by the online platform *Karrot*. The original idea was to develop a tool for trust based role assignment. This concept became more generalized within this master thesis and is not limited to trust anymore.

context

A context defines the environment in which a role model is being used. An example could be, having *python3.7* as a context. Contexts can inherit from others. For example the *Django* framework can be a context inheriting from a *python* context. *karmantra* uses a context to deploy a role model customized for the target system.

user / DevUser

A user is person that can have memberships in system's groups. A person using *karmantra* to generate and alter a role model is called DevUser.

member

A member is a system's entity and part of a group. A member can be a human user or another entity.

role

A role is a status for members of a group. Usually roles are mapped to permissions.

rule

A rule is a condition that has to be met to have a role as a group member. A role

can have multiple rules. A rule can imply the need of computing a result instead of only comparing values against each other.

trigger
Triggers are a mechanism to initiate the role evaluation process. A trigger can be fired for one or more rules.

2.3 Rule-based group role models

There are infinite possibilities for rule-based role models, depending on the system using roles. This section gives an idea of how role models can look like. Every mentioned model consists of roles that a user can obtain within a group. Every role has one or more rules, defining the conditions of being entitled to have the role as a user.

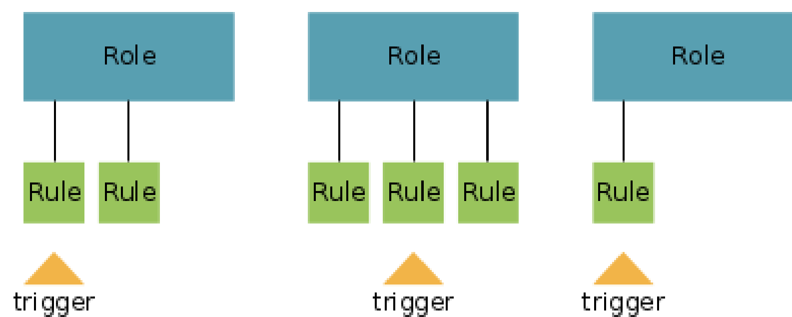

Figure 2.1: An example model (with different roles and rules) allows automating through triggers.

Figure 2.1 shows an example model that allows automating through triggers. These triggers form the interface between tho role model instance and the rest of the system.

Advantages of this approach of having roles, rules and triggers are, that:

- Administrators are not necessary.
- Power (in form of permissions) is set by the model and not by a privileged user. Thus power is potentially less likely to be abused.

- Through the decoupling of role evaluation from the system's other modules, documentation and transparency of the systems behavior can be improved.
- The real world can be reflected more naturally through models that allow self-managed role assignment instead of having rigid role assignment.

2.3.1 Hierarchical vs non-hierarchical

In Figure 2.1 we have seen a non-hierarchical role model. Since we don't want to focus on specific types of role models, also hierarchical or mixed models are examined. Following figure shows a mixed model with partial hierarchy:

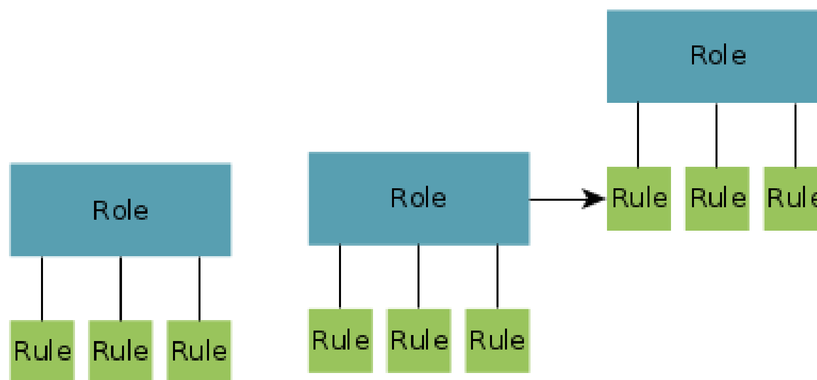

Figure 2.2: An example model with two of three roles being within a hierarchy.

While hierarchical models are easy to implement if the task of role assignment is left to administrative users, they have the disadvantage of providing ground for misuse and rigid distribution of roles.

Since administration permissions can lead to subjective role assignment, hierarchical role models do not necessarily reflect power structures and the possibilities to get roles may depend on the persons having administrative roles. Of course there are endless scenarios for environments, where hierarchical role models are unavoidable. But this kind of model may seem to be the easier way for some developers to be implemented.

Role models, without being strictly hierarchical, have further possibilities like representing democratic decision processes.

2.3.2 Modeling aside from groups

Besides well known models that come from roles and rules, there are further fields of application.

Reward systems

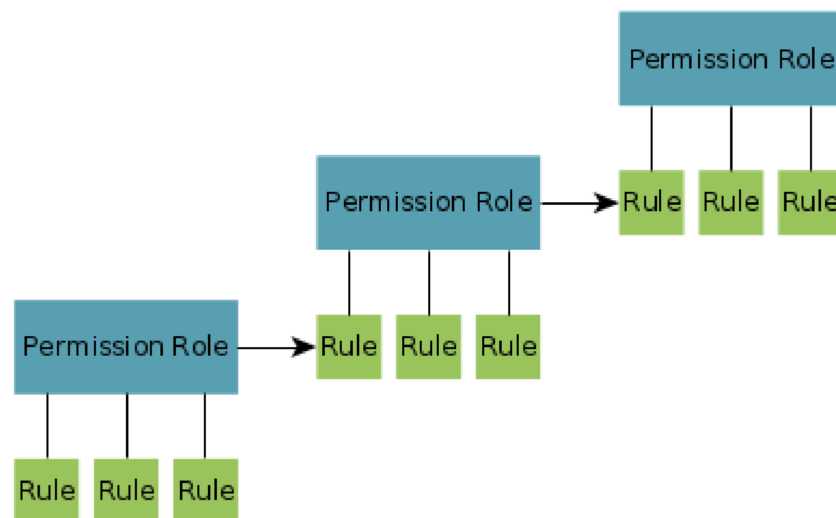

Figure 2.3: An example model for a reward system.

Figure 2.3 shows, how a reward system's model can look like. It could e.g., be implemented with a points system. In this case users can only get higher roles if the respective rules apply. It is conceivable that roles can be lost also.

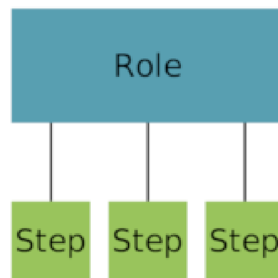

Figure 2.4: An example role for a process-driven role assignment.

Including processes

Sometimes a role is not depending on attributes only, but on complex (possibly democratic) processes or work flows. In this case a role's rule can represent one step of a process of gaining (or losing) a role. Figure 2.4 shows how rules can be used to represent process steps.

Notifications

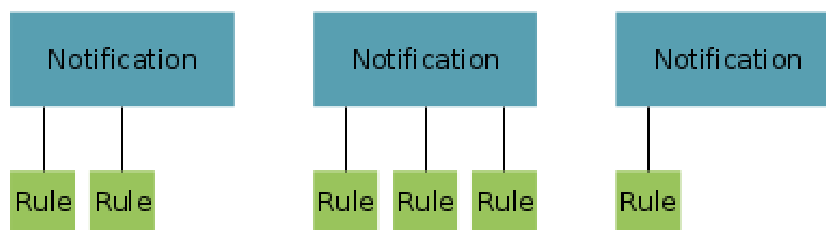

Figure 2.5: An example model where roles represent notifications.

Figure 2.5 shows a completely different scenario. Here the use case is the regulation of sending notifications. For this we replace the roles with notifications. A notification can have one or more rules for being sent. This still perfectly fits to our environment of users and groups.

3

Requirements

The following collected requirements were collected by interviewing developers from different platforms and tie in the related work from above.

3.1 Functional requirements

The following functional requirements build the verifiable basis for the evaluation of Chapter 4 (design) and Chapter 5 (implementation).

3.1.1 System integrability

The developer has to have the possibility to integrate the roles into the preferred location. Adding new roles to the system must not be limited to a specific development environment.

Components:

API

Let's the DevUser control the tool in a generic way.

Framework Configurability

Let's the DevUser define this tool's behavior (e.g., to override existing roles).

3.1.2 Project management

The DevUser must have the possibility to specify a path, where the project shall be deployed. A project must be producible and removable. Dealing with multiple projects has to be possible.

Components:

Listing projects
Allows registering projects for a quicker access.

Project creation
Allows creating a new role model in a specified location.

Project removal
Allows removing a role model in a specified location.

Model structure
The model structure can also be changed manually in an easy way.

3.1.3 Role modeling

The DevUser must be able to add and remove roles. The DevUser must be able to add and remove rules. The DevUser must be able to add and remove triggers. The DevUser must be able to map triggers to role's rules.

Components:

Define abstract role class
All roles can inherit from the abstract class.

Define abstract rule class
All rules can inherit from the abstract class.

Trigger mapping

Allows that triggers are mapped to the related role's rules.

Role evaluation mechanism

The role model is eventually usable for users of groups.

3.1.4 Command line interface

Besides an API, a command line interface must give the possibility to execute all tasks as described in 3.1.1, 3.1.2 and 3.1.3.

3.2 Non-functional requirements

With the section of non-functional requirements, criteria are specified to be used for evaluating the approach's design implementation.

Documentation

A documentation is required for a better understanding of *karmantra* and its usage. It must explain basic design concepts and how to integrate an outcoming role model within a project. Accordingly, the target group is *DevUser*s.

Re-Usability Of Code

It's required to re-use existing ("external reuse") and own ("internal reuse") components for code and design where possible and useful. The aim is to save time and resources, to reduce redundancy and to take advantage of the fact that software quality of external components can be high if it ran through a software development process with adequate testing resources.

Robustness

karmantra has to tolerate erroneous input and to cope with errors during execution. A hard crash without finishing the process gracefully should be avoidable. Guidance for resolving problems should be provided where possible.

Portability

The tool must be usable in different development environments. Also for the applied role module and its role evaluation mechanisms, portability should be possible.

Open Source

The developed concepts and code are licensed with an open source approach. Permission to re-use the code, at least in a non-commercial way, has to be guaranteed.

Low Usability Complexity

karmantra has to be implemented in a comprehensible way. This affects the CLI as well as the rest of the API. It is required that a low complexity for usability is implemented so that both a guided walkthrough as well as automated approaches are feasible for DevUsers.

4

Design

Within this chapter, requirements (system integrability, project management, role modeling, command line interface) from Chapter 3 and principles which are presented in Chapter 2, are used to develop a design for tools and frameworks. This design intends to enable role model creation as well as a role evaluation mechanism. The design is used to specify an architecture whose implementation is described in Chapter 5.

4.1 DevOps

Noticing the way tools like *karmantra* are intended to be used, this work can be described as a *DevOps* contribution. DevOps' name originates from *software development* (Dev) and *information-technology operations* (Ops). While there is no unique definition of DevOps in literature [12, 13], it can be described as method to address "the challenge of what is often described as a gap between development and operations personnel" [13]. To understand how this description leads to some design decisions, it is useful to have a look at some specifications of DevOps [12]:

- enabling communication between Development and Operations Team
- providing a development method / software delivery technique
- enabling an automated deployment / continuous integration / quality assurance
- connecting development to execution by encompassing people, processes and technologies

The following aspects are incorporated for the design and implementation of *karmantra*: Automation, improving interaction of development and deployment and providing better connection possibilities for people. The following section makes clear, how DevOps objectives are realized for role model creation and role assignment.

4.2 Application specification

In this section, flows are defined for the application of role assignment. Remembering Chapter 2, it is obvious that we have to differentiate between role provisioning and role-based assignment of permissions. Assuming that mapping roles to permissions is not a complex work, we focus on provisioning only. A rule-based provisioning, as in 2.1.4, can be split into two domains, which will be treated separately:

1. Role and rule *modeling*
2. rule-based role evaluation

The mentioned advantage of a high level of automation can be optimized with this differentiation between modeling and evaluation. In the following sections flows are described for the DevUser, the target system and the tools, that are used for role modeling and role evaluation. These flows make clear, how the two provisioning domains *role modeling* and *role evaluations* are applied.

4.2.1 Modeling

Creating, modifying or removing a role model includes the steps *role model deployment* (including the integration into the target system) and *rule implementation* for the DevUser which is abstractly demonstrated in Figure 4.1.

Initially, a DevUser wants to create a role model. The modeling tool receives all necessary information on roles, rules, role evaluation triggers and how these are connected to build the role model. The latter is then deployed to the DevUser's target system. The remaining task for the DevUser is to implement the following components:

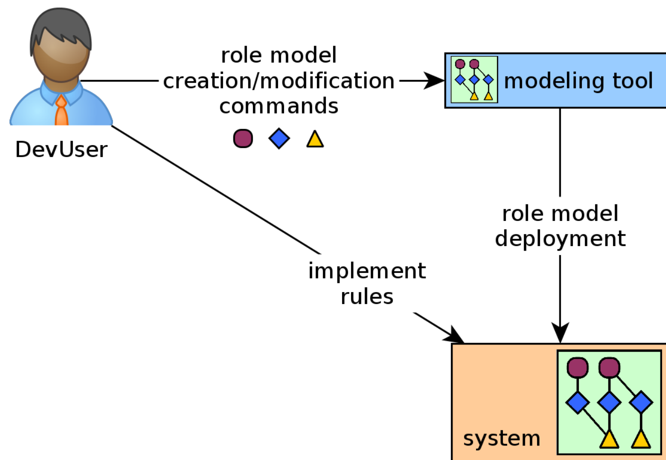

Figure 4.1: Interaction between DevUser, modeling tool and target system. The green box represents the role model, while its content represents connected rules, roles and triggers. In our case, the blue box represents *karmantra*.

wrapper [mandatory]

The wrapper ensures that arbitrary group and user objects can be used with the role evaluation framework (described in 4.2.2). The DevUser has to specify how the role evaluation framework can extract the needed information in the required way.

rules [mandatory]

The DevUser has to implement the rule functions which are needed to check if the rules apply for a user. A rule can be, for example, that the user has a specific attribute value or that a system event occurred.

module globals [mandatory]

To check if a rule applies might require access to resources of the embedding system.

trigger firing [mandatory]

Wherever triggers have to be fired, the role evaluation tool has to be indicated as trigger receiver.

result format [optional]

If the default result format does not fit the DevUser's needs, it can be replaced.

role's evaluation behaviour [optional]

If a role's behavior has to be adjusted, its inherited functions can be overwritten. A DevUser might like to order rules that are about to be checked in a specific order.

The components which are mentioned earlier in this chapter, namely the *role model* and the *role evaluation mechanism* are tied together as importable module and deployed into the target system. The DevUser of course has to import this module in the correct locations.

To minimize the DevUser's costs of time and implementation complexity, all components within the importable framework can easily be prepared with templates, that just have to be filled with missing code. It is indispensable that the DevUser has to contribute some code. Because the modeling tool has to deal with arbitrary systems, it is not possible to generate code for all existing possibilities of target systems. The next section 4.2.2 describes how the role evaluation import module has to work.

4.2.2 Role evaluation

Before describing (in section *Mechanism*) how the role evaluation mechanisms are designed, basic assumptions are given (in section *Role model elements*) on how a role model is defined.

Role model elements

Role models such as described in 2.3 can be described with the three element types *role*, *rule* and *trigger*. Between different element types, edges can be defined. Following, an edge is also called "connection". The relation of these elements can be described through following rules:

- a role can have multiple rules
- a rule can be used by multiple roles
- one rule can have one trigger

- a trigger can be mapped to multiple rules

The question remains whether we can model relations between roles. E.g., hierarchical roles have to be used. With the definitions above we can also describe dependencies between roles indirectly: The dependency of one role to another can be defined as a rule.

Mechanism

To explain the role evaluation mechanism, the role evaluation import module's overall behavior is described from the target system's perspective first. Second, the internal view is explained.

The role evaluation mechanism is started whenever a trigger in the system is fired. Figure 4.2 shows the flow of a role evaluation mechanism.

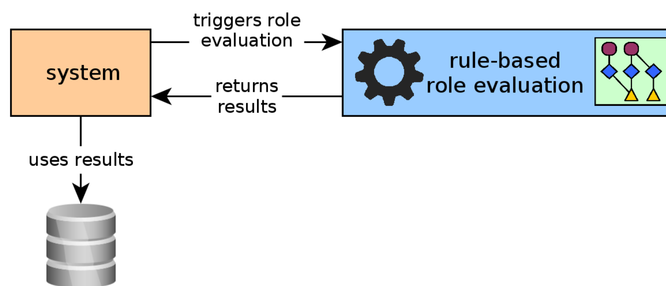

Figure 4.2: Interaction between the system and the role evaluation tool, using the previously defined role model.

The role evaluation framework provides an interface for firing the triggers. The module will expect

- a trigger identifier
- the affected user
- and the user's group as scope

for the role evaluation. Stating the group is necessary because a user's roles can vary for different groups. Smaller systems may not need to have multiple groups and thereby

can keep all users within one group. *karmantra* provides the option to select all users from a group for the evaluation process if no user is specified.

Following steps are performed in a role evaluation process:

1. A trigger is fired with information of user and group.
2. All rules connected to the trigger are collected.
3. For all roles that are connected to a rule from Step 2, an evaluation process is run for every user, stated in step 1.
4. An evaluation process checks whether or not all rules apply for a user.
5. The collected evaluation results for the evaluations are collected and returned to the system.

This design earmarks that all of a role's rules are checked, even if only one of its rules is connected to a trigger. The reason is that there may exist rules without connected triggers and that a system might want to inform users or the system provider exactly about which rules recently do or do not apply.

In the following sections it will become more clear, how all these principles and definitions can be included into a tool's architecture.

4.3 Data model

Referring to the objectives (automation, improving interaction of development and deployment and providing better connection possibilities for people) that are derived from our DevOps considerations (in 4.1), an approach with multiple layers is designed as shown in 4.3.1. All following descriptions are implemented for *karmantra*, which is explained in detail within Chapter 5.

4.3.1 Implementation layers

Figure 4.3 shows the design of a tool that complies with the explanations from Section 4.2. In Section 4.3.1 (modeling tool), *karmantra* as CLI tool and importable tool for role

modeling tasks is presented. In Section 4.3.1 (role model deployment components), *karmantra*'s deployed role evaluation approach is presented.

Modeling tool

Following components are strictly separated from each other to allow a high degree of interchangeability. These layers are presented as blue boxes in Figure 4.3:

CLI
Makes the *karmantra* python module available for the command line.

Core
Acts as proxy for tasks and conveys tasks to lower layers. Described in detail in Section 4.4.

Modeler
Provides all functions for executing modeling tasks and for role model deployment.

Role Model Import Module
Static files and templates for the deployment of the role model import module (explained below) depending on the DevUser's stated context.

The boxes *Configuration*, *Logging* and *Helper* (white) represent classes, that do not implement functional requirements directly and adapt to the layer's needs. They can be seen as support classes. The main layers (blue) *CLI*, *Core*, *Modeler* and *Role Model Import Module* are more likely to experience an evolution or even a replacement. The command line interface is not part of the importable python module which contains all functional behavior. Figure 4.4 shows the class diagram for *karmantra*. It indicates the base classes *Core*, *Configuration*, *Helper*, *Logger* and *Modeler*. The edges show in which classes instances of other classes are used. The indicated classes *Backup_Error* and *Config_Error* are used for a more specific exception handling.

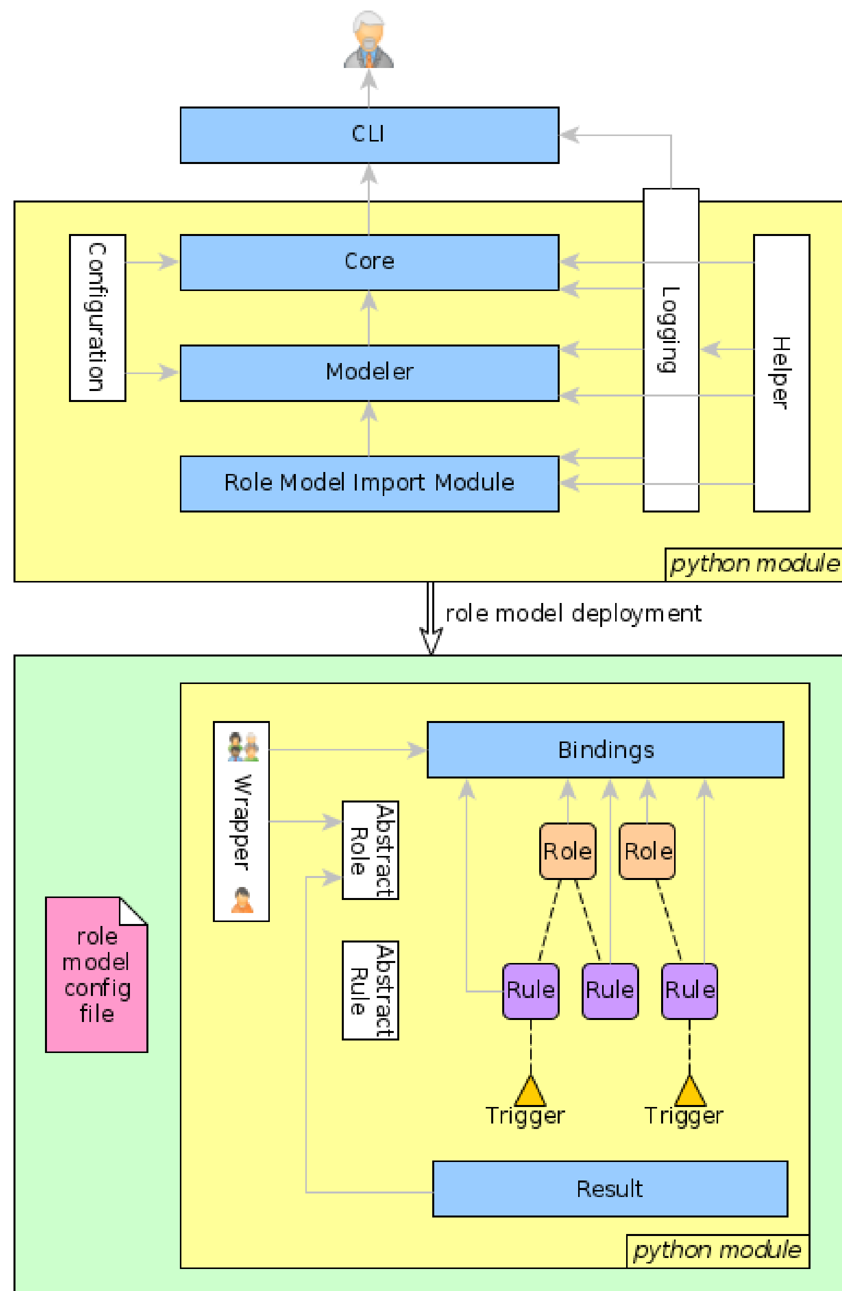

Figure 4.3: *karmantra*'s implementation layers (blue) with supporting and needed components (white). Upper box: CLI and the modeling tool. Lower box: Deployed role model with role evaluation mechanism.

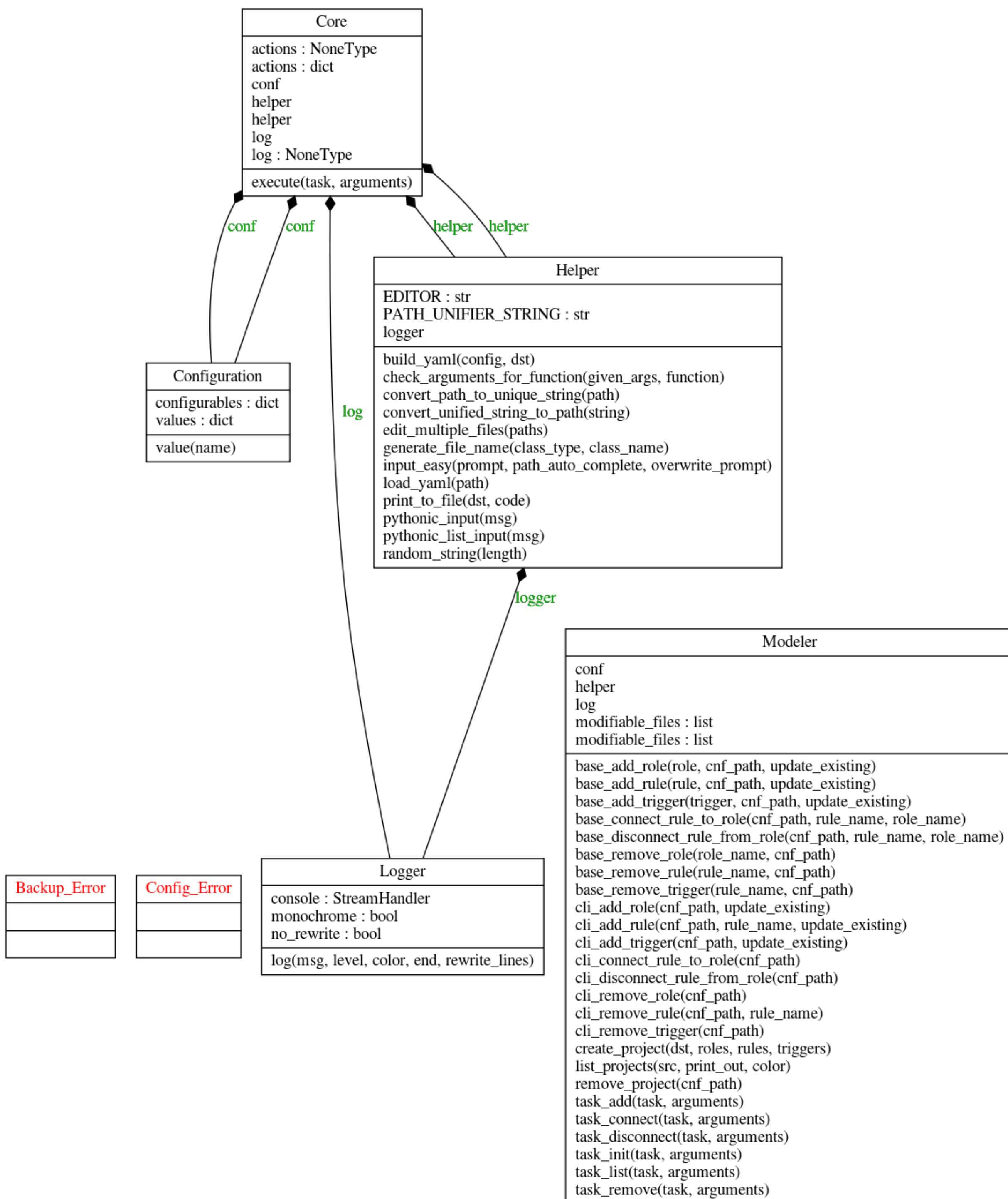

Figure 4.4: UML class diagram for *karmantra*'s module.

Role model deployment components

The deployment of a role model and the role evaluation mechanism depends on the DevUser's specifications. Still, the structure remains the same. As shown in Figure 4.3, the deployment consists of a "role mode configuration file" and an importable python module, containing the role model and the role evaluation mechanism.

The *role model configuration file* allows *karmantra* to read the current role model and how it is deployed together with the role evaluation mechanism, including the DevUser's implementations. The latter are provided as importable python module. The above described *Bindings* and *Results* components are interchangeable as the other layers. The role model itself is kept modular. Roles and rules inherit from abstract classes to ensure a defined behavior.

Having described a data model for rule-based role evaluation in general and for *karmantra*, the following chapter explains in detail, how tasks are designed.

4.4 Task model

karmantra is a tool allowing tasks to be commissioned by the DevUser via the CLI or the python module. The layer approach from Section 4.3.1 led to a task model that implies a hierarchical task. The component *Core* presents the root of the task execution. This approach has the advantages of being useful intuitively and being arbitrary in a way that tasks can be extended easily.

For example, executing the task "add role to model" via *karmantra* can be done with following command:

```
1          ./start\_cli add role --monochrome
```

Whenever a task like "add role" is received by a layer (in this case *Core*), it is responsible to verify the correctness of the first section. If core did not know anything about the task section "add", its responsibility would be to interrupt the process. If the task section "add" is valid, the tail is sent to the layer that is now responsible. In this case the

section "role" belongs to the *Modeling* layer. Arguments that are provided with a task may be forwarded to a tasks subsection. The transferred arguments can control the task execution's behavior. The advantage of this task model is that a high degree of automation can be achieved.

4.4.1 Core tasks

Task roots that are received and processed by the core component form core tasks. Possibly a CoreTask has no meaning for itself and acts as proxy. Current **core tasks** are "add", "connect", "disconnect", "remove", "list" and "init".

4.4.2 Modeling tasks

The following tasks are executed by the modeling layer.

add role|rule|trigger: adds elements to role model

remove role|rule|trigger: removes elements from role model

remove project: removes a complete *karmantra* deployment

list projects: lists existing *karmantra* deployments

init: creates a new role model and deploys it

connect: connects rules and roles

disconnect: disconnects rules from roles

Having developed a design for rule-based role evaluation, this contribution is extended by the practical implementation of *karmantra*, which is described in Chapter 5.

5 Implementation

This chapter shows, how the design of rule-based role evaluation is implemented for the tool *karmantra*. While Chapter 4 demonstrated a general design that can be used in many different ways, a practical implementation is presented here.

5.1 Development process

Conventions are necessary to prevent frustration and a waste of resources when it comes to collaboration. Also for a single developer it is important to define clear framework conditions for the development to keep a consistent code quality. This chapter meets these needs by defining defining conventions and explaining implementation decisions.

5.1.1 Licensing

The license used for this master thesis' contribution is version three of the GNU General Public License (GPLv3). To keep this work open-source is the main reason.

5.1.2 Contribution workflow

Languages

As programming language, python3 is used. For the generated code representing a role model and its evaluation mechanisms, *Jinja2* [14] is used as templating language. For configuration files, the *YAML* [15] standard is chosen.

Coding conventions

Coding conventions are adapted to *flake8* and *black*. flake8 is a linting tool performing static analysis of source code which verifies *pep8*. pep8 [16] is a style guide with many conventions.

black is an opinionated tool that helps formatting code. On its website the providers state: "Black makes code review faster by producing the smallest diffs possible. Blackened code looks the same regardless of the project you're reading. Formatting becomes transparent after a while and you can focus on the content instead." [17]

Development environment

The operating system used for implementing and testing, was *Ubuntu 19.10*. For package management, pip [18] and the open source system *Conda* is used. As version control system, GIT was selected.

Contribution workflow

The following steps are recommended for contributing to the project and represent the development workflow.

1. Getting to know coding conventions, license and the documentation.
2. Work on changes in assigned branch.
3. Test the results.
4. Commit changes. Black and flake8 possibly interfere and changes have to be applied before commitment is possible.
5. Merge to the master branch.
6. Update documentation.

5.1.3 Documentation

The documentation is written with the markup language *Markdown* and deployed with the tool and platform *readthedocs* [19].

5.2 Project structure

In the root directory for development, there is a folder for the source code (*src*), a folder for the documentation (*docs*) and files related to the development workflow (such as hooks for git). Figure 5.1 shows the directory structure for *karmantra*. In 4.3.1 more about the background of this structure has been explained.

Within the *src* folder, the code for running *karmantra* can be found in folder *main*. Test implementations can be found in folder *tests*.

5.2.1 main

main includes a starter file for the command line interface, a configuration file where *karmantra*'s behavior can be adjusted and the directory *karmantra* which contains the code with the required functionality. The configuration file is useful for automating purposes where a developer wants to avoid command line arguments.

5.2.2 main/karmantra

This directory contains *karmantra*'s main functionality. Namely following components are represented:

- configuration: Default behavior and mechanism for defining a specific behavior
- logger: Supports debugging and is used by CLI to print to console
- helper: Modules that are usable by multiple components and not specific enough to be included in other modules

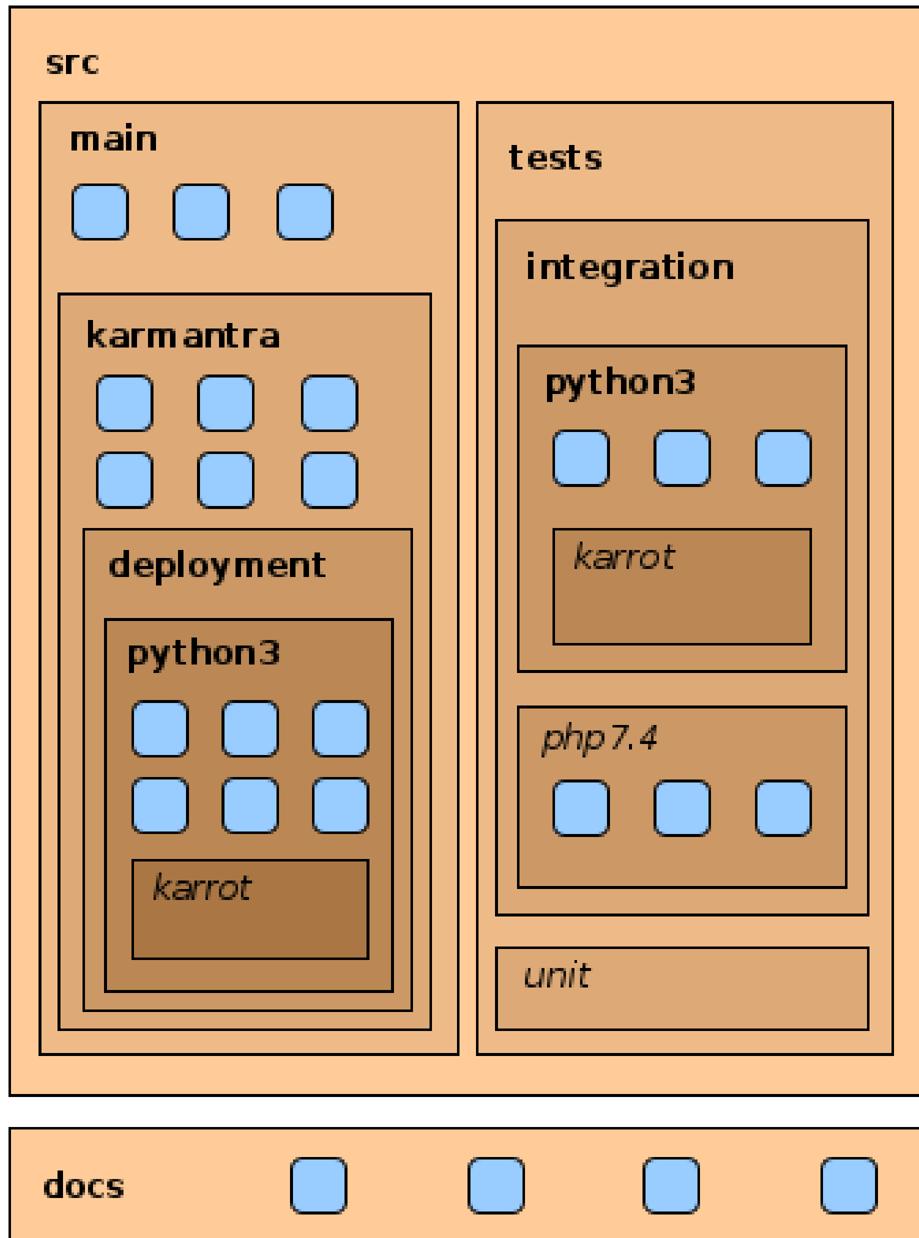

Figure 5.1: Provided abstract data structure. Blue boxes are files. Other boxes are directories. Directories with italic description are for possible extensions.

- core: Maps tasks to the execution of module's functionalities
- modeler: All modules being responsible for a project dealing with role models
- style: A special helper for the appearance of console outputs.
- deployment: a directory, explained below

5.2.3 main/karmantra/deployment

The deployment directory's structure contains one folder for every context. In this case a context of python3 is specified. More specific contexts that are sub-contexts to python3 will be placed within the python3 folder. In Figure 5.1, such a sub-context would be *Karrot*. In every context's folder the template files for the target systems role model deployment are kept. This is described in detail in 5.3.5.

5.2.4 tests

Within the *tests* directory there is one folder for every type of testing. In our case there is a folder for integration testing.

5.2.5 tests/*<test_type>*/*<context>*

Within every folder in *tests*, following structure is intended: As described in the *deployment* folder section above, one folder per context is created. Sub-contexts are represented deeper in the structure respectively. The context folders contain the test implementations, which are described in 6.1.

5.3 Implementation components

In the following sections, implementation components parts are described. The sections, which explain the implemented modules, are kept short and simple, which means that only important ones are described, because of their number.

5.3.1 Modeler

This component is represented by the class *Modeler* and its functions.

With the class functions, modules for following scopes are implemented:

- deployment of static files
- deployment of deployment configuration file and files from templates (bindings, rules, roles, wrapper)
- enquiry (to DevUser) for data needed for a task (e.g., a file path)
- query of DevUser's input and conversion into role model items (roles/rules/triggers)
- update of the deployment configuration
- checking the health of a role model and its deployment (validating paths, attributes and connections)
- creation / removal of a role model deployment
- execution of a modeling task (as described in 4.4) in a base and a CLI variant
- core tasks (as described in 4.4)

Requirements fulfillment

The components within this section are connected to the requirements *System Integrability*, *Project Management* and *Role Modeling*.

Safe modifications

Modifying and overwriting in the DevUser's file system comes with some high risk of failures. A task, which, for example, alters a role model, might fail and result in data being written to the file system only partially. This can lead to an un-usable role model with its role evaluation mechanism.

To prevent losing time and other resources for fixing broken components, a safety wrapper is provided for all tasks, which may alter something in the file system. Basically the wrapper provides a backup and restore mechanism that works in the background. Before a wrapped task is executed, the complete deployment destination is backed up. If a task fails in any way so that it is detectable for python, the deployment destination is cleared up completely and the backed up data is restored.

There are multiple conceivable use cases that include using multiple modifying tasks in a row. To deal with these cases, multiple tasks can be wrapped together.

The involved modules are implemented as following private functions:

- _backup
- _restore
- _safe_tasks_wrapper
- _safe_task_wrapper

5.3.2 Helper

The *Helper* class provides functions that can be used across *karmantra*'s components.

Following module implementations satisfy, inter alia, following needs:

- simple file system path completion for DevUser input
- input validation
- validation of arguments for a function
- support for the YAML format
- providing external editor support

Requirements fulfillment

The component *Helper* provides modules for all components that support functional requirements.

5.3.3 Configuration

The class *Configuration* implements another important layer that allows adjustability for DevUsers and an easy extensibility for developers who want to improve *karmantra* itself.

An instance of the *Configuration* class can be used to define a determined behavior for *karmantra*. This may include how task execution behaves (e.g., whether existing roles may be overwritten without prompt). It is also configurable how the CLI behaves (e.g., the level of verbosity). A *Configuration* object provides default values that can be changed.

Configuration Attributes

There are two types of configuration attributes:

configurable These values can be changed through command line arguments or the *karmantra.ini* configuration file.

preset These values are predefined and can be changed by developers who want to modify *karmantra* in general. For example, if a context has been added to *karmantra*, it has to be indicated in the configuration as well.

The following tables list the used configuration attributes.

Attributes, settable by DevUsers:

name	**description**
version	Selects a specific *karmantra* version.
context	Sets the context for deployment.
log_level	Defines the verbosity as well as the file log level.
deployment_folder_name	Defines the name of the deployed module.
deployment_config_file	Defines the file name of the deployment configuration.
interactive	Attribte set by CLI to control if *karmantra* is interactive.
prevent_file_editor	Prevents opening an external file editor after deployment.
monochrome	Flag that effects printing to stdout without colors.
update_existing	Allows to overwrite existing elements like roles and rules.
configuration_path	The path to *karmantra*'s configuration file *karmantra.ini*.

Preset Attributes:

name	description
base_folder_name	Folder name of deployed role model module.
templates_folder_name	The name of the directory containing templates.
static_files_folder_name	The name of the directory containing static files.
context_paths	A dictionary, containing all context's paths.
templates	A dictionary with template's file names.
static_files	A dictionary with the static file's names.
help_texts	A dictionary containing filling text.
deployment_config_attributes	The role model deployment's default values.

Initialization

To provide an automatable and easy configurability, two ways for defining configurable attributes are provided: Calling *karmantra* with appropriate command line arguments and defining behavior in a configuration file. The format is set by python's standard library *configparser*.

To allow flexibility and to prevent ambiguity, a hierarchy for configuration setting is defined: A configuration object can be initialized with default values without specifying any attribute. With the object initialization all attributes from the file *karmantra.ini* are looked up first and are used to overwrite the default values. Secondly, all existing parameters given to the initialization function overwrite configuration values.

Some configuration values depend on others, so that these are generated in the end of an initialization process.

5.3.4 Command Line Interface

The command line interface (CLI) is built with the aim of being simple and fast to use. It was designed in a way, command line users would expect it to be, compared to other standard tools on the command line. The CLI has a starter that uses python's standard

library *argparse* for parsing arguments. Figure 5.2 shows the help text, indicating possible arguments.

These arguments can be clustered to "task arguments" and "parameter arguments". A task is represented as concatenated strings. The first string is processed by the *core* that has been described above. The later strings are processed by the subsequent components. Parameter arguments can be used as parameters for task's functions (e.g., a path of a needed file) and can be specified for determining *karmantra*'s behavior (like the degree of verbosity).

```
❯ ./start_cli --help
usage: start_cli [-h] [--cnf-path CNF_PATH] [--dst DST] [--role_name ROLE_NAME]
                 [--rule_name RULE_NAME] [--src SRC] [--version VERSION]
                 [--context {karrot}] [--monochrome] [--update_existing]
                 [--log_level {debug,info,warning,error}]
                 [--deployment_folder_name DEPLOYMENT_FOLDER_NAME]
                 [--deployment_config_file DEPLOYMENT_CONFIG_FILE] [--prevent_file_editor]
                 [--configuration_path CONFIGURATION_PATH]
                 TASK [TASK ...]

karmantra - Managed Rule-Based Role Assignment

positional arguments:
  TASK                  task for karmantra

optional arguments:
  -h, --help            show this help message and exit
  --cnf-path CNF_PATH   your projects config file deployed by karmantra
  --dst DST             the location to deploy karmantra's module to
  --role_name ROLE_NAME
  --rule_name RULE_NAME
  --src SRC             the yaml source file containing a model for roles rules and triggers
  --version VERSION     karmantra version
  --context {karrot}    deployment context
  --monochrome          prevents colors
  --update_existing     sets overwriting policy for existing elements
  --log_level {debug,info,warning,error}
                        log level
  --deployment_folder_name DEPLOYMENT_FOLDER_NAME
                        name for deployed module (default: karmantra)
  --deployment_config_file DEPLOYMENT_CONFIG_FILE
                        name for deployed module config file (default: config.yml)
  --prevent_file_editor
                        if set, opens editor for created files that have to be edited
  --configuration_path CONFIGURATION_PATH
                        path to karmantra.ini, having calling parameters, and a list of
                        projects
```

Figure 5.2: The current arguments that can be passed to *karmantra* via the CLI.

The command line interface uses the configuration setting "interactive" to provide tasks with the information on whether or not to get function parameters via user input. There's a limited input validation for command line arguments. Namely paths are checked for correctness before continuing with the program execution.

Requirements fulfillment

The component *Command Line Interface* fulfills the requirement of the same name.

5.3.5 Role model deployment components

The role model deployment consists of two parts, as it was shown in Figure 4.3. First there is the deployment configuration file. This file has the YAML format and contains all important information on how the role model is deployed. It allows *karmantra* to backup and modify the role model easily. Second, there is the role model provided in the form of a python module. It can be imported to the system by the DevUser.

The role model import module contains the components that are described in 4.2.2:

wrapper.py Wraps the system's group and user objects for *karmantra*.

binding.py Provides the API for the system and connects roles, rules and triggers.

result.py A class providing role evaluation result objects.

module_globals.py A location where the DevUser can make resources available to the module.

role.py, rule.py Abstract classes for roles and rules.

role_*<name>*.py The respective role class inheriting from the abstract one.

rule_*<name>*.py The respective rule class inheriting from the abstract one.

5.4 Addressing functional requirements

The following list shows, how the implementations are used to address the functional requirements from 3.1:

requirement	**fulfillment**
System Integrability	Layer Approach (4.3.1), Modeler (5.3.1), Configuration (5.3.3)
Project Management	Modeler (5.3.1)
Role Modeling	Modeler (5.3.1), Role Model Deployment Components (5.3.5)
Command Line Interface	CLI (5.3.4)

The classes *Style* and *Helper* (5.3.2) are helper classes that are used by most of the other components.

6 Evaluation

In this chapter the test procedure for *karmantra* is presented, the fulfillment of non-functional requirements is discussed and a theoretical application of the developed concepts is explained using the platform *Karrot*.

6.1 Testing

To evaluate a correct behavior of *karmantra*, integration test cases are written with python's standard test libraries. The following list explains the tested cases and which functions they are mapped to. The tests cover the functional requirements as addressed in Section 5.4 have all run successfully.

6.1.1 Modeling

For testing *karmantra*'s modeling full set of functions, tests are applied for tasks with the scope of core tasks (as described in Section 4.4.1). This is done by importing *karmantra* as module. With every test case, a task is submitted to the module to be executed. To prepare and clean up tests, python's provided *unittest* functions *setUp* and *tearDown* are used.

- Creating a new project for modeling roles:
  test_create_new_project
- Delete an existing project:
  test_project_deletion

- List existing projects:
  test_project_list_projects_existing
  test_project_list_projects_empty
- Add a role to project:
  test_project_add_role
  test_add_role_failing
  test_add_role_overwriting
- Remove a role from project:
  test_remove_role
- Add a rule to project:
  test_add_rule
- Add a trigger to project:
  test_add_trigger
- Connect a rule to a role:
  test_connect_rule_to_role
- Remove a rule from project:
  test_remove_rule_failing
- Remove a trigger from project:
  test_remove_trigger_and_rule
- Disconnect a rule from a role:
  test_disconnect_rule_from_role

6.1.2 Model usage

Having implemented tests for the correct behavior of *karmantra*'s modeling options, the functionality of the outcoming model is the subsequent important field for testing.

To test the functionality of the importable role model, a test class with example users within a group has been created. To simulate a realistic use case, the user and group objects lack attributes necessary for *karmantra*'s evaluation process. Having implemented

such a test set, the wrappers for users and groups were adapted to demonstrate the usability of arbitrary systems. In a second step the role evaluation itself is tested both for the case of not being applicable for a user and for being applicable for a user.

6.2 Fulfillment of non-functional requirements

The fulfillment of non-functional requirements from 3.2 is verifiable with Chapter 4 mostly:

Documentation

A documentation of design and implementation is provided with this master thesis. Additionally, a deployable *readthedocs* documentation with descriptions and tutorials is provided.

Re-Usability Of Code

The speraration of layers (4.3.1), helper classes 5.3 and models allowed a very modular approach. The use of standard library solutions was mentioned and performed.

Robustness

karmantra tolerates erroneous input e.g., with the input validation for command line arguments 5.3.4 but also with modeling functions. With using python's exception handling, a most unwanted hard crashes can be prevented.

Portability

Portability is ensured through implementation layers (4.3.1) and the use of templates for different contexts (5.2.3).

Open Source

As described in 5.1.1, *karmantra* is licensed under the *GNU General Public License, version 3*.

Low Usability Complexity

The careful design of tasks (4.4) and command line arguments ensures that DevUsers can profit from low usability complexity. For the command line interface,

a walkthrough is usable, while using *karmantra* as python module comes with clearly defined and easily understandable tasks.

6.3 Theoretical application of concept

To show that *karmantra* is applicable, the online platform *Karrot* is analyzed below. On this basis it will be apparent how *karmantra* can be included into a working system. *Karrot* has been chosen, because it fits the idea of automated rule-based role assignment, it shows another use case among many business solutions and because it is assumed to be more complex to include *karmantra* into an existing system than starting from scratch. The exchange with and help from the developers of *Karrot* did not only support the following theoretical application of concept, but also the design chapter of this master thesis.

6.3.1 Karrot

Karrot is a platform, connecting people all over the world to save food from being thrown away. Its motivation comes from the fact that about one billion people have to hunger while twelve billion people can be fed with today's possibilities. Foodsaving groups intend to raise awareness for production and consumption of food through saving food. *Karrot* provides the possibility for groups to organize via its platform. Its idea is to use automated role assignment without interfering with group's decisions on role assignment.

6.3.2 Current role implementation

Querying roles

Assuming, *Karrot* wants to find out if a user has the role *editor*, the function *is_editor(user)* is called. *is_editor(user)* is a member of a group instance. The function itself calls the function *is_member_with_role(user,rolename)* from the class *GroupMembership*. Figure 6.1 visualizes the process.

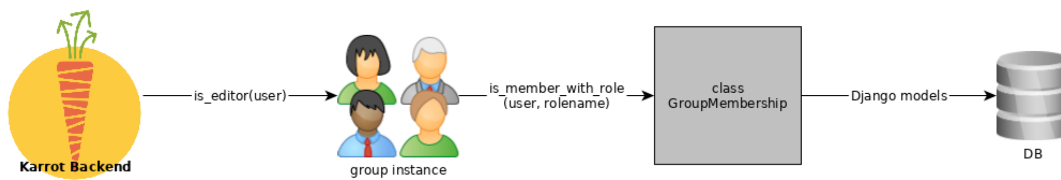

Figure 6.1: *Karrot* queries roles through the Django framework.

Updating roles

At the moment there is only one trigger for the change of roles: Giving *trust carrots*. Whenever a *trust carrot* is given by a user A to a user B, *Karrot* will check if the change will affect user B's *editor* role status. If so, the new role will be made persistent. Figure 6.2 represents the process in an abstract way.

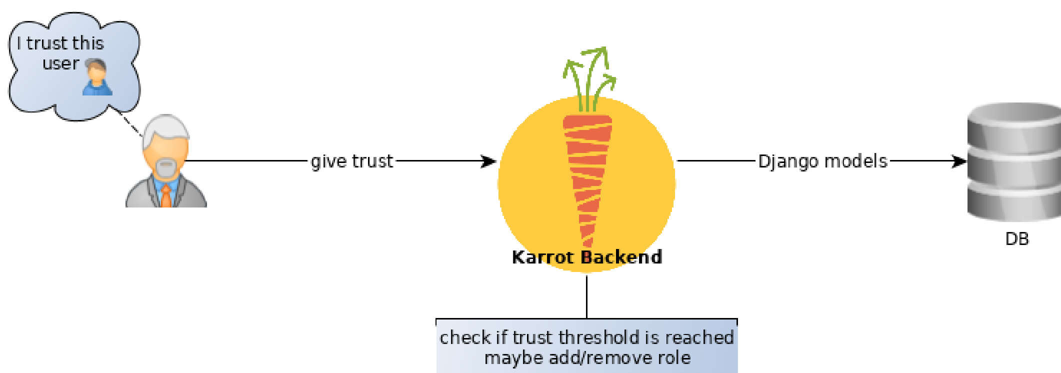

Figure 6.2: *Karrot* queries roles through the Django framework.

Proposal for implementation

Following the intentions and requirements of *Karrot*, a decoupling of the role evaluation process can be achieved with *karmantra*, as visualized in Figure 6.3.

Figure 6.3 represents all steps that are taken within a role assignment process:

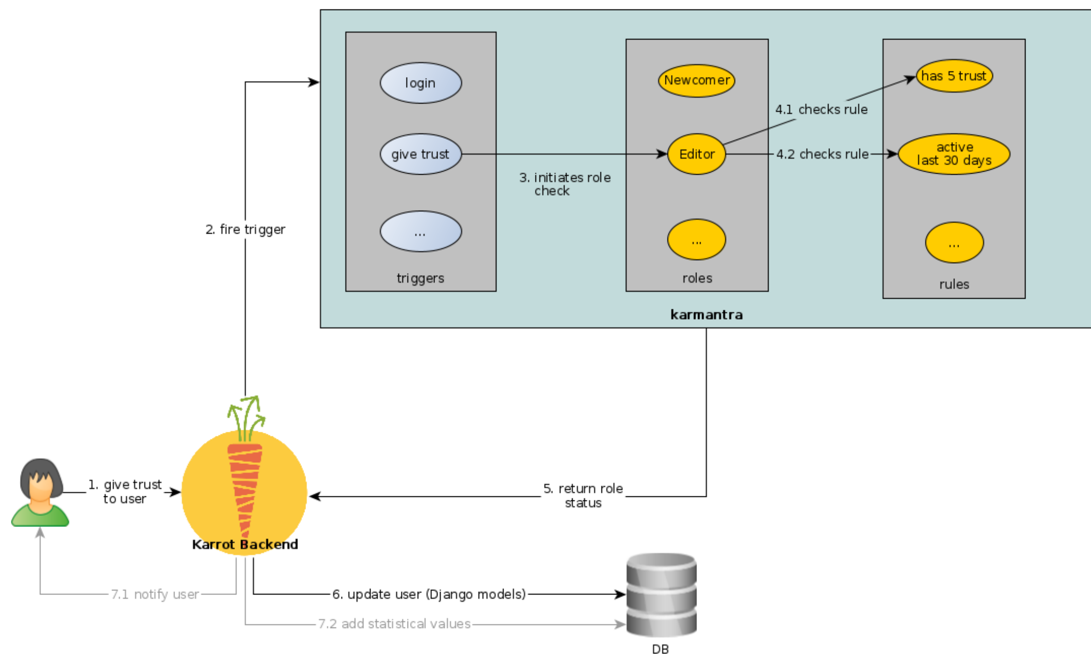

Figure 6.3: Rule-based role evaluation can be implemented for *Karrot*, using *karmantra*.

1. A user action or system task takes place.
2. The system uses the action from 1. as a trigger for *karmantra*'s role evaluation framework. Information like the user and group are passed to the framework.
3. Knowing which triggers map to which rules, an evaluation process is initiated for the associated roles.
4. Every role that has to be checked, examines whether its rules apply.
5. The results are given back to the system.
6. The system can use the results e.g., to update the user's attributes in the database.
7. Additionally the system can use the result for e.g., user notification (7.1) or statistics (7.2).

Knowing the key functions (see 6.3.2) for the recent role evaluation mechanism, it is possible to connect this approach to *Karrot*.

7 Conclusion

Within this chapter, a summary is given for the approaches and contributions, given in this master thesis. Last of all, the implementation is discussed and an outlook for future work is given.

7.1 Summary

Within this master thesis, the specified challenges for software development led to the design and implementation of the framework *karmantra*. It is independent from operating systems, can be adapted to other programming languages and has the functionality of generating arbitrary models for automated rule-based role evaluation. As shown, the requirements for diverse role models can be complex. With the defined premises it was possible to develop a meta model that allows automation in a generically usable way. With this, we have seen that it is possible to build a very modular tool that is adaptable for future advancements.

Within this thesis, general problems that can occur when dealing with role models were introduced first in Chapter 1. Also the motivation to enable automated role assignment through generic tools was explained. In Chapter 2, related work was referenced, important definitions were made and rule-based role models were presented. In Chapter 3, requirements were collected. In Chapter 4, a software tool design was developed that should enable both the creation of role models and a role evaluation mechanism. The realization of *karmantra* according to the principles in the *Design* chapter was explained in chapter 5, which also presented how the requirements were addressed. Finally in

Chapter 6 the evaluation of *karmantra* by integration tests was presented, the fulfillment of non-functional requirements was checked and a theoretical application of *karmantra* for the platform *Karrot* was explained.

7.2 Discussion

Accomplishing the implementation of a generic tool like *karmantra*, we can have a look at the non-functional requirements and ask if it is possible to do better. If we take the requirement of comprehensibility as an example, we can argue that it is still hard or impossible for many people to understand the developer's implementation of role models. Focusing on the links between rules, roles and triggers, this contribution provides enough clarity for non-technical users already through human-readable and easy to read configuration files. Since the developers have to implement the rule's behaviors, we could look at how to make these implementations more transparent. The developer's rule implementations fully depend on the role model embedding system. This does not allow to build a generic tool anymore. Instead the approach would have to be to build a full service system with an interface to allow communication between the developer's system and the role model system. That role model system might need to have full control over the user database as well, which results in new challenges of assigning responsibilities to different services.

The reason of generating a generic tool, which does not allow having a rule creation service affects other developer needs as well. For example a notification system that informs members about role changes has to be implemented by the developer. Fortunately this has been respected in the design of *karmantra*, so that it is made easy to include cases like this one.

We have seen that the tool *karmantra* can provide a platform independent tool that can even allow including templates for other programming languages. If we look at the development process, it can be questioned if there are use cases where having *karmantra* as a python tool fits all needs. Despite justifying the decision of using python,

it should not be too challenging to transfer the concepts within this thesis to implement *karmantra* with other programming languages.

Improvement can come from extending *karmantra* by the possibility to define, how rules are connected. Modeling of arbitrary rules could become more comfortable, if e.g., some rules can be connected through a disjunction instead of a conjunction. For now this option is manually feasible by overriding a role's evaluation function.

7.3 Outlook

Having done a contribution in form of a design and implementation for a tool to build more flexible, automated and comprehensible rule-based role models, there is still a way to go to better software regarding the defined requirements. As mentioned, more and more programming languages and contexts like python's *Django* can be included. We can see this as an ongoing process, adapting *karmantra* to the changing needs of developers. It is even thinkable to offer loadable context modules in future, where the needed respective context can be specified by the developer to keep the core of *karmantra* slim.

Coming to a point where more and more features are available, other testing methods like unit testing will become valuable and important as an addition to integration tests. Since this thesis' approach has not been tested in a long term for a developer's implementation routine yet, this is something useful in future to get more insights for possible improvements.

Concerning software development there is still a broad field for research when it comes to systems that seek to provide role evaluation on the base of commonly agreed-upon rules. This does not necessarily touch the subject of role assignment only, but becomes clear especially here. Not only social organizations can profit from the results.

Having given this outlook and knowing that software can not provide solutions for all problems, the master thesis shall be concluded with the hope that software can and will be used to build tools that serve people and their goals for a better organisation, exchange and cohabitation on earth.

Bibliography

[1] Jin, X., Krishnan, R., Sandhu, R.: A unified attribute-based access control model covering dac, mac and rbac. In: IFIP Annual Conference on Data and Applications Security and Privacy, Springer (2012) 41–55

[2] Sandhu, R.S., Samarati, P.: Access control: principle and practice. IEEE communications magazine **32** (1994) 40–48

[3] Sandhu, R.S.: Lattice-based access control models. Computer **26** (1993) 9–19

[4] Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed nist standard for role-based access control. ACM Transactions on Information and System Security (TISSEC) **4** (2001) 224–274

[5] Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. Computer **29** (1996) 38–47

[6] Kern, A., Walhorn, C.: Rule support for role-based access control. In: Proceedings of the tenth ACM symposium on Access control models and technologies. (2005) 130–138

[7] Kern, A.: Advanced features for enterprise-wide role-based access control. In: 18th Annual Computer Security Applications Conference, 2002. Proceedings. (2002) 333–342

[8] Kern, A., Kuhlmann, M., Schaad, A., Moffett, J.: Observations on the role life-cycle in the context of enterprise security management. In: Proceedings of the seventh ACM symposium on Access control models and technologies. (2002) 43–51

[9] Epstein, P., Sandhu, R.: Engineering of role/permission assignments. In: Seventeenth Annual Computer Security Applications Conference, IEEE (2001) 127–136

[10] Epstein, P.A., Sandhu, R.: Engineering of role/permission assignments. George Mason University (2002)

[11] Al-Kahtani, M.A., Sandhu, R.: A model for attribute-based user-role assignment. In: 18th Annual Computer Security Applications Conference, 2002. Proceedings., IEEE (2002) 353–362

[12] Rütz, M.: Devops: A systematic literature review. (2019)

[13] Smeds, J., Nybom, K., Porres, I.: Devops: a definition and perceived adoption impediments. In: International Conference on Agile Software Development, Springer (2015) 166–177

[14] : Jinja2. (https://jinja.palletsprojects.com/) Accessed: 2020-04-02.

[15] : Yaml. (https://yaml.org/) Accessed: 2020-04-02.

[16] : Pep8. (https://www.python.org/dev/peps/pep-0008/) Accessed: 2020-04-02.

[17] : black. (https://black.readthedocs.io/en/stable/) Accessed: 2020-04-02.

[18] : pip. (https://pypi.org/project/pip/) Accessed: 2020-04-02.

[19] : readthedocs. (https://readthedocs.org/) Accessed: 2020-04-02.

List of Figures

2.1 An example model (with different roles and rules) allows automating through triggers. 8
2.2 An example model with two of three roles being within a hierarchy. 9
2.3 An example model for a reward system. 10
2.4 An example role for a process-driven role assignment. 11
2.5 An example model where roles represent notifications. 11

4.1 Interaction between DevUser, modeling tool and target system. The green box represents the role model, while its content represents connected rules, roles and triggers. In our case, the blue box represents *karmantra*. 19
4.2 Interaction between the system and the role evaluation tool, using the previously defined role model. 21
4.3 *karmantra*'s implementation layers (blue) with supporting and needed components (white). Upper box: CLI and the modeling tool. Lower box: Deployed role model with role evaluation mechanism. 24
4.4 UML class diagram for *karmantra*'s module. 25

5.1 Provided abstract data structure. Blue boxes are files. Other boxes are directories. Directories with italic description are for possible extensions. . 32
5.2 The current arguments that can be passed to *karmantra* via the CLI. . . . 38

6.1 *Karrot* queries roles through the Django framework. 45
6.2 *Karrot* queries roles through the Django framework. 45
6.3 Rule-based role evaluation can be implemented for *Karrot*, using *karmantra*. 46

Name: Sandro Eiler　　　　　　Matriculation number: 751972

Honesty disclaimer

I hereby affirm that I wrote this thesis independently and that I did not use any other sources or tools than the ones specified.

Ulm, 17.04.2020

Sandro Eiler