Bachelor's Thesis in Computer Science

Rheinisch-Westfälische Technische Hochschule Aachen
Knowledge-Based Systems Group
Prof. Gerhard Lakemeyer, Ph.D.

A CLIPS-Based CBR Framework for Pass Schedule Recommendations in Open-Die Forging

Soomro, Sarah
Date: 07.11.2022

Advisor
Tarik Viehmann, M. Sc.

Supervisors
Prof. Gerhard Lakemeyer, Ph.D.
Prof. i.R. Matthias Jarke, Dr. rer. pol.

Contents

Chapter 1

Introduction

Intelligent systems are widely used today and many application areas benefit from them [Pan15]. Such systems can become very complex, so that their method of operation is no longer transparent. One possible consequence is the reduction of the user's trust in the system [BHvdAW21]. Systems that mimic how humans learn can be beneficial, with Case-Based Reasoning (CBR) being an emulation strategy. It basically mimics human thought processes: A new problem is solved based on previously solved problems. The reasoning is intuitive and naturally understandable and thus CBR applications are generally regarded as being transparent [SWA+21]. A fundamental aspect of developing these is the often non-trivial similarity assessment required to find a suitable solution [MABL20]. Its design may vary from expert to expert, as their subjective perceptions of which previous solution is most beneficial to a new problem often differ. This case is particularly relevant when the domain of the application is not well understood.
In this thesis, we present a web-based CBR framework facilitating the development of CBR applications. Special focus is laid on facilitating user participation in the design of the reasoning process, knowledge exchange between users and usability without programming knowledge. Therefore, the framework provides the user with a selection of predefined similarity measures and the possibilities to customize them using constraints, and to add the user's own customized measure to the selection, so that they can be viewed by other users for comparison. Thus, the framework is primarily concerned with retrieving the most similar cases to a novel problem which corresponds to the Retrieve process of the CBR Cycle process model, explained later.
We consider as use-case the pass schedule generation for Open-Die forgings, where explicit domain knowledge is sparse. Open-Die Forging is a widespread technology used to produce products which are usually large and have high demands in terms of their mechanical properties and reliability [WRRH19]. Starting from a cast work-piece, it is struck between two dies several times to result in an end product with selected geometrical shape and mechanical properties like a minimum of sturdiness and material strength. Therefore, a pass schedule is created for the executing machines. Even small errors of the process subject to the plan can influence the end product, which is not measurable while processing [Rec14]. Managing this process planning task is complex and is often handled by one or a few experts. Problems can arise when an expert leaves the company or is simply not available

at a time. To mitigate this problem, assistance systems that integrate expertise provide knowledge preservation. In this way, for example, a new employee may seek advice from his or her predecessor who has retired. The proposed framework is developed in cooperation with the research group for Bulk Metal Forming of the institute of metal forming (IBF).

The thesis is organized as follows: Chapter 2 explains background knowledge through an introduction to Open-Die Forging, CBR with focus on the similarity assessment, the programming language CLIPS which is used for the CBR processes and the architectural style Representational State Transfer (REST) which we use as a guide. In Chapter 3 the related work is presented. Chapter 4 elaborates the requirements on the framework and the design of the system to address them. Aspects of the implementation of the system are presented in Chapter 5 by highlighting different layers of the architecture and their scope of tasks. Chapter 6 is concerned with evaluation, which is firstly the proof of concept with a presentation of the use of the framework for Open-Die forgings to engineers of the IBF and secondly analyzing the behavior of the system for different number of cases. We then take a critical look at the framework and offer an outlook for future work in Chapter 7. Finally, Chapter 8 provides the conclusions.

Chapter 2

Background

This Chapter presents a concise overview of the topics relevant to this thesis. First, Open-Die Forging is introduced as the domain in which our system is used in. This is followed by an explanation of the CBR methodology, which focuses on the similarity-based retrieval, and the programming language used for the reasoning, C Language Integrated Production System (CLIPS). Finally, we introduce the architecture style REST, which we used as a guideline for the system design.

2.1 Open-Die Forging

Open-Die Forging [Klo17] is a widespread procedure in industry. The products are usually large in size (kilogram to ton range) and mostly single pieces or small series. They are built for a long life and resilience under stress [SRW+10]. In most cases they are near-net shape end products or intermediate products like bar stocks which are further processed. Open-Die Forging is applied, for example, in the production of turbine shafts for the aerospace technology [SRW+10] or crankshafts for shipbuilding [Klo17] which are safety-critical areas.

In this hot forging process, the work-piece is formed repeatedly between dies by applying pressure. The name of the process (Open-Die) comes from the fact that the work-piece is not enclosed by the dies. This allows products to be shaped individually. Figure 2.1 shows on the left an example of a forging process where the work-piece is shaped by means of a hydraulic press and the guidance is carried out by a robot arm. The machines are controlled by trained operators and nowadays it is considered a high-tech process with fine-tuned steps [SRW+10]. Important exemplary parameters included in a schedule are shown on the right side of Figure 2.1. They refer to one bite, which is a single pressing action. The length of the contact surface between the work-piece and the upper die is denoted by s_{B0}. The height and width for the processed section before the bite are respectively called h_0 and b_0 and h_1 and b_1 after it has occurred.

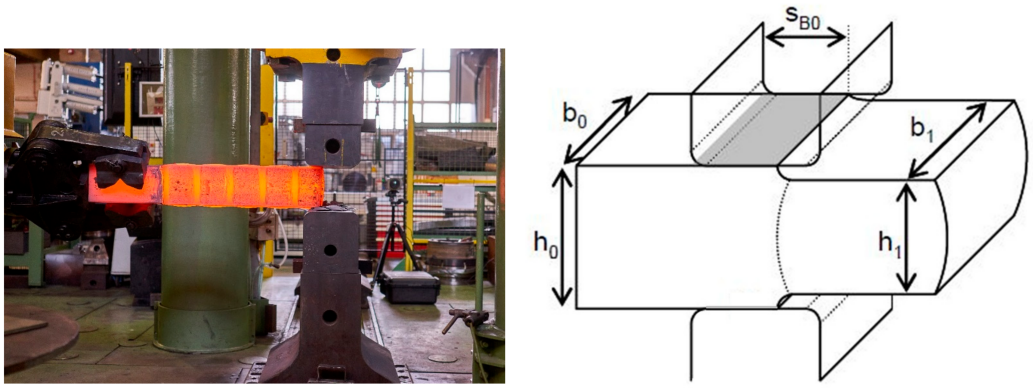

Figure 2.1: A snapshot of an Open-Die Forging Process (left) and a sketch with relevant geometric parameters (right), adopted from [RRG+21].

The task of forging is achieving a certain geometry, but also, to provide certain mechanical properties determined by the underlying microstructure. When a cast ingot is used as a starting piece [Klo17], it does not have a homogeneous structure [Got07], because metals are made up of grains that grow irregularly [EK90]. Likewise, the ingot contains shrinkage cavities and pores caused by gas bubbles [Got07]. Thus, for example, the strength of the metal isn't sufficient, because it is sensitive to fractures. The so-called cogging forging process is suitable for approaching a fine-grained microstructure as homogeneous as possible and thus also the desired mechanical properties. In principle, the height of the work-piece is reduced here iteratively with several passes. A pass in turn includes several bites starting at the beginning and ending at the end of the piece to reduce its height successively.

2.1.1 Simulation Methods

Since the grain structure of the metal cannot be determined during the forging process without damaging the work-piece [EK90], the process is often simulated before it is put into action. One way to obtain a suitable model to make high quality predictions is to use the finite element analysis (FEA). It works with numerical simulations and is relatively time-consuming. Comparable results can be achieved more rapidly using fast models based on semi-empirical data [WRRH19]. With these, the IBF created simulation data on forging processes and provided them to us for the development of the framework.

2.2 Case-Based Reasoning

Case-Based Reasoning (CBR) [RW13] is a reasoning methodology which essentially infers a solution to a new problem by looking at previous experiences, finding the most similar ones and reusing the knowledge about their solutions. It is inspired by cognitive science, which supports that prior experience is used in solving problems

in human thinking [Sch83], [NSM18]. Formally, in CBR an experience is referred to as a *case* and is defined by a triplet $(problem, solution, outcome)$. The first component, *problem*, is the description of the problem and the second component, *solution*, of the solution for it. The third optional component, *outcome*, contains meta-information about the solution like if it was successful or not or how often it was used. The degree of *similarity* between cases is handled by a *similarity assessment component*. Newly occurring cases can be retained in the CB enabling a CBR system to learn. Consistent processes within the methodology were identified by [AP94] who proposed the CBR Cycle with the four phases Retrieve, Reuse, Revise, Retain. Figure 2.2 shows a variant of the cycle with the additional process Problem Formulation suggested by [RW13].

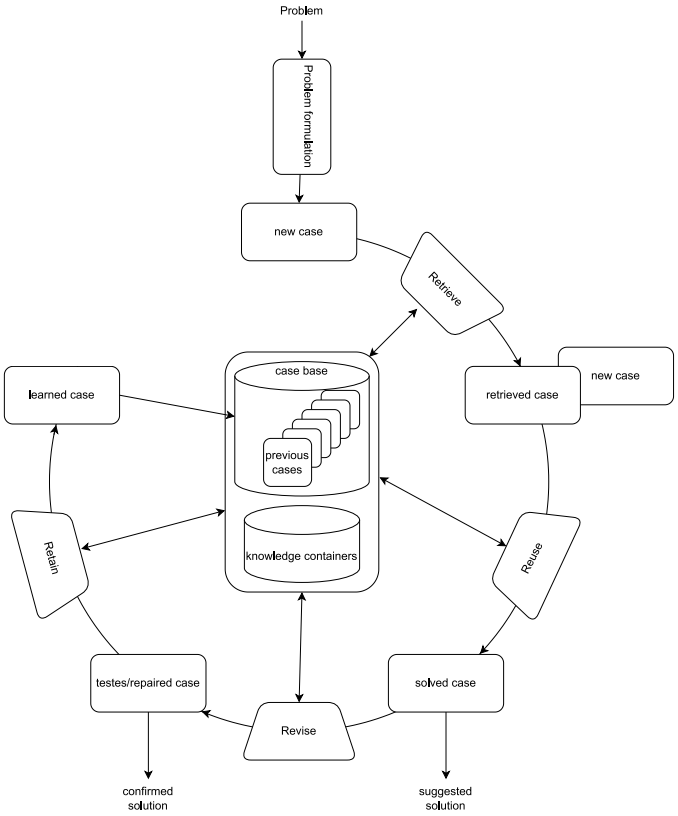

Figure 2.2: Process model for the CBR Cycle, combined from [AP94], [RW13], [Cra17].

The processes are explained in more detail:

- The cycle begins with the problem formulation where the problem to be solved is translated into the representation of a case which is denoted by *new case*. For instance, an engineer is looking for a pass schedule for a bar stock whose target values are queried using a GUI.

- In the subsequent Retrieve process, the n most similar cases to the new case are retrieved from the CB. For instance, for the bar stock with desired final height of one meter, a similar one with final height of 1.1 meters is found.
- In the Reuse process the retrieved case's solution is used to create a suggested solution to the new problem which thus becomes the solved case. However, in contrast to reasoning as defined in formal logics, reasoning in CBR is approximate. It does not infer true conclusions from true assumptions within its case-based deductive system. Additionally, the CB offers partial knowledge since it records single cases which almost certainly do not cover the whole problem space which includes all possible manifestations of the cases. That's why the solution undergoes an adaption step with the help of the adaption knowledge container [RW13]. In the example, additional passes are added to the schedule.
- There is no guarantee that the suggested solution is suitable for the new case. That's why the solution is tested in the Revise process for its applicability. If it is found to not be applicable, then it is repaired. In the example, the schedule is verified using simulation software.
- The system can store the new case with its confirmed solution in the CB in the Retain process. In the example, the schedule for the desired bar stock is saved.

The CBR cycle underlines the cyclic nature of CBR: The learning is incremental and constant. It is also regarded as a machine learning paradigm [AP94] which falls under the category lazy learning since instead of learning a certain model for generalization, this takes place during runtime [Cra17]. Using the recorded knowledge encoded in cases in the CB, CBR does not depend on explicit general knowledge about the domain [AP94]. However, the additional knowledge containers illustrated in the middle of Figure 2.2 may contain domain-specific knowledge. For instance, four knowledge containers are identified by [Ric09]. Besides the CB, there is the vocabulary container, which includes only relevant aspects and all that are necessary to accomplish the task. The third, the similarity container, contains the knowledge with which cases can be adequately compared. The fourth, the adaption container, includes the knowledge necessary to adequately transform a solution [RW13].

The focus of this thesis is the Retrieve phase of the CBR Cycle. We understand a case as a tuple of the structure *(problem, description)*. The inclusion of the other phases and the *outcome* component are subject to future work.

2.2.1 Problem Description and Similarity Assessment

The similarity assessment typically compares cases by their problem description. The fundamental assumption of CBR states that cases with similar problem descriptions have similar solutions as illustrated in Figure 2.3.

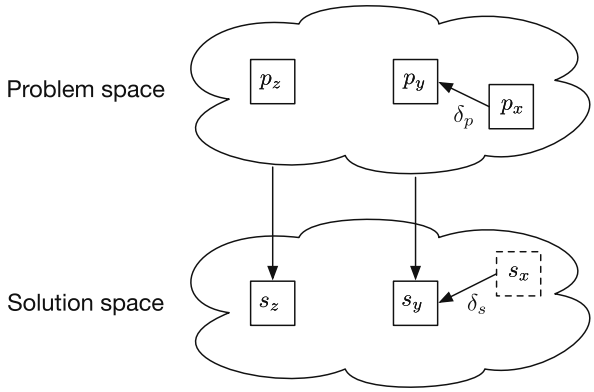

Figure 2.3: Illustration of the fundamental assumption of CBR, adopted from [MABL20].

In the problem space P, the distance δ_p between the problem component p_x of the new case x and that of the recorded case p_y serves as an estimation of the distance δ_s between their solutions s_x and s_y in the solution space S. Since δ_p is the smallest distance in P between p_x and the other points p_y and p_z, it is assumed that the most similar case to x is y [MABL20]. In other words, problem descriptions should be similar if they have the same or a similar solution. Therefore, cases must be adequately represented in a system. Usually, cases are represented using feature-value pairs such that P is the Cartesian product of the different domains of the features F_i. As an example, Table 2.1 provides a snippet of one possible problem description of a bar stock on the left and a snippet of a possible associated pass schedule as the solution of the case on the right.

case 1	
initial cross section	square
initial height	1.5 m
initial width	1.5 m
initial length	5 m
final cross section	square
final height	1 m
final width	1 m
final length	10 m
...	

pass no.	$\frac{h_0}{m}$	$\frac{h_1}{m}$	$\frac{b_0}{m}$	$\frac{b_1}{m}$	$\frac{s_{B0}}{m}$	...
0	1.5	1.5	1.5	1.5	0	...
1	1.5	1.125	1.5	1.656	0.975	...
2	1.656	1.266	1.125	1.266	1.076	...
...						

Table 2.1: Problem Description Snippet (left) and Pass Schedule Snippet (right) of the example case referring to a barstock.

The problem description contains information about the initial work-piece and the desired parameters for the final product while the schedule dictates the parameters like the new height to be reached for each pass respectively and is usually

in the form of a table. Besides the task of case representation, there are also the tasks of finding suitable cases for the initial case base and appropriately storing and indexing them in the system. An organization can be classified into one of the 3 categories flat, structured (e.g., an object-oriented representation) or unstructured (e.g., a text)[RW13]. In the following we assume a flat organisation. Overall, the case representation and organization should enable efficient comparison of cases and storage of new cases [AP94] and this tasks are thus intertwined to the design of the similarity assessment.
Given the set problem space P, the similarity assessment usually establishes a relation R on it, where $R(p_x, p_y, p_z)$ indicates that p_x is at least as similar to p_y as p_x is similar to p_z. Putting the focus on p_x, the same statement can be made with $p_y \leq_{p_x} p_z$ with the induced partial order relation $\leq_{p_x}$, which evaluates the similarity related to p_x. Its most similar case p_u is called the nearest neighbor. The nearest neighbour relation NN indicates that p_u is most similar to p_x by $NN(p_x, p_u)$. It is defined by $NN(p_x, p_u) \Leftrightarrow \forall p_y \in CB : p_y \leq_{p_x} p_u$ with CB the set of known cases. Similarly, the relation NN_k indicating the k most similar cases can be formulated. The assessment can be made either directly with a partial order relation or with a similarity measure sim - a function $P \times P \to [0, \infty)$ which induces a relation. It defines the degree of similarity between cases. For its induced relation $R_{sim}(p_x, p_y, p_z)$ holds that $R_{sim}(p_x, p_y, p_z) \Leftrightarrow sim(p_x, p_y) \geq sim(p_x, p_z)$. Analogously a distance function d with $P \times P \to [0, \infty)$ defines the degree of distance between cases induces a relation R_d with $R_d(p_x, p_y, p_z) \Leftrightarrow d(p_x, p_z) \geq d(p_x, p_y)$. If for functions d and sim holds, that $\forall p_x, p_y, p_z : R_{sim}(p_x, p_y, p_z) \Leftrightarrow R_d(p_x, p_y, p_z)$, they induce the same relation. In general, the concepts are interchangeable since a distance function d can be transformed into a similarity function sim by concatenating it with a bijective, order-inverting mapping like the function f with $f(x) = \frac{1}{x+1}$ such that $sim(x, y) \sim f(d(x, y))$. Depending on the application, one concept is more suitable than the other [RW13]. That means when problems are compared, a score is given. In our context, we use distance functions and thus the lower a score is, the more similar problems are.
In the following, we assume the use of a similarity function sim and depict a case's problem description p_x by $p_x = (p_{x_1}, \ldots, p_{x_n}) \in P$, where the p_{x_i} denote the values of the different features F_i. The function sim typically first compares the values of the cases to the individual features and then determines the overall similarity (local-global principle) from this. Therefore, a function sim_i is described for each feature F_i. There is a variety of possible different global and local functions that come into question. Usually, finding a suitable measure is nontrivial [MABL20]. Traditionally, predefined functions have been used depending on feature scaling, the application domain and more, of which [RW13] gives a comprehensive overview. For instance, they mention the weighted linear sum which also takes the importance of a feature into consideration by assigning it a weight w_i given by Equation 2.1.

$$sim(p_x, p_y) = \sum_{i=1}^{n} w_i \cdot sim_i(p_{x_i}, p_{y_i}) \tag{2.1}$$

If available, domain knowledge can also be incorporated into the construction of the measure. As an example, using the weighted linear sum as the global similarity measure the weights can be set by a domain expert. A recent data-driven approach is that the measure is completely or partially automatically generated. There are various strategies for which [MABL20] gives an overview. Generally, they have in common that they use machine learning algorithms. As an example, using the weighted sum, the weights w_i or the local similarities sim_i may be learned. Another possibility is to use Siamese neural networks (SNN) [Chi21], which usually consist of two or more identical neural networks and a comparison layer allowing them to learn a similarity measure by transforming the inputs into a more suitable vector (embedding them) and comparing them by a predefined distance measure. Figuratively speaking, as an example in a classification task, the case base is decomposed into clusters and the new case is assigned to the cluster with its nearest neighbor. They have also shown that their extended SNN, which learns the measure completely automatically, clusters correctly for the MNIST data set. Also an option is the use of pipelines of transformations as in [AKP+20]. Overall, the similarity measure is use case specific and can vary widely.

2.3 C Language Integrated Production System

Written to define expertise and emulate the reasoning of experts, we believe CLIPS is well suited for our project, as we mimic the reasoning of an expert during the Retrieve phase. This Section explains the basic principles of CLIPS. It was primarily sourced from [Ril21]. The portable and integratable public domain software CLIPS, written in C, was developed by the Software Technology Branch of NASA's Johnson Space Center and the first public release of CLIPS was in 1986 [RCSL87]. The CLIPS language works with forward-chaining with the help of the Rete algorithm [For89]. It includes the implementation of the paradigms rules and procedural programming, where a program can be composed of facts and rules. The following sections explain the portion of CLIPS used in this work.

2.3.1 Basic CLIPS Components and their Mode of Operation

Originally developed as a rule-based expert system tool, CLIPS encompasses all basic components for these [Viz19] which are the knowledge base (KB), consisting of the fact-list and the rule-base (the collection of rules), the inference engine and an User-Interface. Figure 2.4 underlines the interactions between them.

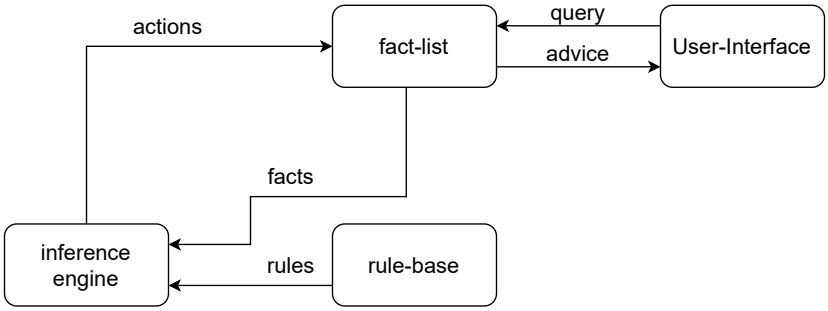

Figure 2.4: Interaction between the basic components of CLIPS.

A fact encodes information about the current state of the system. The fact-list works as the global memory [Gia]. For example, the cases in the CB of a CBR application for Open-Die Forging could be represented as single facts, respectively. For example, *(cbr_case (id 1) (status "inCB") (cbr_h_0 1))* is an unordered fact, where the deftemplate in Figure 2.5 provides the structure of the bundled related information by means of slots.

```
1       (deftemplate cbr_case
2             (slot id)
3             (slot status)
4             (slot h_0))
5
```

Figure 2.5: A deftemplate for a barstock.

Depending on which facts are present in the list, the execution of rules may be triggered. They can be thought of as whenever-then statements which are activated when all patterns of the left-hand side are matched against facts in the fact list. For example, adding a new case through the user-interface may result in performing the right-hand side of the rule in Figure 2.6. This triggers the calculation of the local distance score for the feature *initial height* between it and the cases in the CB based on the absolute value of their height differences if no scores have yet been assigned for this or the total distance (indicated by the value -1.0).

```
(defrule local_measure_function_h_0
    (cbr_case (id ?id) (status "inCB") (h_0 ?h_0))
    (cbr_case (status "new Case") (h_0 ?newh_0))
    ?sim <- (similarity_case (foreign_id ?id)
            (overall_similarity -1.0)
            (h_0 -1.0))
=>
    (bind ?sim_value (abs (- ?h_0 ?newh_0)))
    (modify ?sim (h_0 ?sim_value)))

```

Figure 2.6: CLIPS rule, which sets the similarity of the final height to the absolute value of the difference of heights. 4.1.

Such rules apply fixed actions for a certain state of the world and thus embed heuristic knowledge. As another example, cases with different cross sections could receive immediately a global similarity score of 0. Activated rules are collected in the agenda. Based on their importance, the so-called salience value, the inference engine manages which rule should be performed. If multiple rules with the same salience value happen to be activated, the engine decides according to a conflict resolution strategy. The default is the depth strategy which prefers rules with higher salience and new rules before rules with the same salience. An executed rule, in turn, can manipulate the list. As an example, it could add the most similar retrieved case, which again can be viewed through the user interface. After performing, a rule is taken off from the agenda. The program terminates when no more rule is activated.

2.3.2 Integratability using Subprograms

With an appropriate environment, a CLIPS program can be called as a subroutine in another program written in a different language like Python with use of the clipspy extension module[1] as the API. Procedural code can be defined in user-defined functions (UDFs) and these Python functions are available as CLIPS functions after linking them to the environment.

2.4 Representational State Transfer

Representational State Transfer (REST) is an architectural style for network-based applications conceptualized by [Fie00]. It encompasses constraints to guide the design of apps in the context of hypermedia systems:

- **Client-Server**: The functionality of data storage is handled by servers and that of the user interface by clients. For this purpose, servers provide services that can be requested by clients.

[1] `https://github.com/noxdafox/clipspy`

- **Stateless:** Clients send isolated requests that contain all the information required by the server. Thus, servers do not store information about the state of the application and it is determined by clients.
- **Uniform Interface:** This constraint routes to an unified interface where implementations of services are uncoupled from them. It consists of sub-constraints. The examples used here refer to the HTTP API endpoint that allows clients to read a CBR application, that we implemented.
  - **Identification of Resources:** Namable information is referred to as resources. Each resource involved in the exchange between servers and clients is assigned a Uniform Resource Identifier (URI) for example, *http://localhost:8000/CBRApplications/631b3fadd83b01ab31cfd849*.
  - **Manipulation of Resources through Representations:** Representations of resources are sent between the components to access or communicate them. As an example, the server responses with the queried CBR application, as key-value pairs, in JSON format. The server's own implementation, i.e. a distributed storage of the data in collections of the database, is not relevant for the clients.
  - **Self-Descriptive Messages:** The responses and requests should be self-describing in the sense that another program understands the semantics. For instance, the request also contains the corresponding HTTP method (GET) which corresponds to the standard [Gro99], and the successful response contains the HTTP status code 200 (OK).
- **Layered system:** The system is hierarchically organized into layers. A member of one layer can only see the layer connected to it.

In addition, there are other constraints that do not seem relevant to our project that are omitted for brevity. These can be found in [Fie00]. Overall, the most relevant for us is that the listed constraints lead to an architecture with simplified components that can evolve independently provided that the interface remains the same. In the process, general performance and the scalability of servers is improved and clients are platform-independent. Therefore, we focused on the aforementioned constraints and took them into account during the development of the system. This technically means that our system is not a fully-fledged REST application. According to the Richardson Maturity Model [Ric], which assesses a system's conformance to REST conditions, our system is at level two. In total, there are four levels (from zero to three) and the level correlates with the conformance standard. Our system does not reach level three because it does not use hyperlinks, which are links in a response that point to other possible subsequent requests to clients.

Chapter 3

Related Work

The first prototypes for case-based systems emerged in the 1980s [Sla91]. Since then multiple CBR systems emerged in many different application areas [Smy19]. Given that this thesis is concerned with a CBR framework implemented in CLIPS and tested for systems in the field Open-Die Forging, this Section gives attention to general purpose CBR tools, CBR systems implemented with CLIPS and assistance systems and approaches for the design of pass schedules in Open-Die Forging.

3.1 General Purpose CBR Tools

This Chapter gives an overview of domain independent software that foster the development of CBR systems. Only tools that can be found in recent literature are included here. [BMB+20] mentions myCBR, COLIBRI and the Process-Oriented Case-Based Knowledge Engine (ProCAKE).
COLIBRI[1] is a general platform, which includes among others a CBR framework, its implementation jColibri for Java systems and a toolbox including explainable AI (XAI) methods. The project is envisioned as a collaboration of independent groups sharing their experiences [RGDAGC13][RGGCDA14].
The open-source tool myCBR[2][BA12] for developing structural CBR systems is implemented in Java and encompasses a graphical user-interface in terms of the workbench and a software development kit. [BMJ19] developed a REST API for the tool allowing programming in other languages than Java.
The University of Trier developed ProCAKE[3] - an open-source Java framework for the development of structural and process-oriented CBR systems. XML is used for stability and configurations [BGMZ19].
[AKP+20] introduces the Deep Knowledge Acquisition Framework (DeepKAF) for building natural language processing (NLP) applications in which deep learning techniques and CBR are combined.
Some of the Java frameworks offer their functionalities through powerful GUIs. In contrast to the other frameworks, we focus on the management of expert knowledge and usability for engineers. For this purpose, we offer a web-based frontend and choose Python as the language for the backend. Java has a large user base, but

[1] `https://gaia.fdi.ucm.es/research/colibri/index.php`
[2] `http://mycbr-project.org/`
[3] `http://procake.uni-trier.de/`

Java programming skills are rather unlikely for engineers. Thus, this can become an obstacle for the further development of an application, e.g. the integration of machine learning models for the similarity assessment based on the data in the CB. Overall, the beginner-friendly language Python is a standard for developing such models.

3.2 CBR Systems with CLIPS

[EFC00] developed a framework in CLIPS which enables knowledge engineers to easily integrate a Case-Based Instructional Planner (CBIP) into Intelligent Tutor Systems (ITS). Thereby, the CBIP follows the CBR cycle and cooperates with the conventional planner: If no case can be retrieved the latter is used to generate a plan which is again retained in the CB.
[sSK07] came up with a process planning system which suggests robot welding plans for the assembly of unit blocks in shipbuilding. The sequence planner implemented in CLIPS acts according to the CBR cycle and solves constraint satisfaction problems characterizing the order of welding actions within it.
[Mor00] introduced the CBRMID, a CBR shell which generates a deterioration model for components of a bridge and thereby estimates the condition of it in the future. Bridge management systems suggest with their help which actions are suited for maintenance and guess the costs. The system uses an Inference Engine encoded in CLIPS for the adaption step in the CBR process model.
[BAC97] presented the CBR system CBPlan which generates a process plan composed of machining operations encoded in NC code for a given design of a prismatic part. In more detail cases are represented by a collection of geometrical entities which are composed to build the part, and relations expressing the arrangement between them. CBPlan is implemented in CLIPS integrated in a C++ environment.
[MBd11] developed a CBR prototype which suggests designs for natural gas cogeneration plant design. They integrated it into their previously developed expert system (ES), also written in CLIPS. The fused system follows the CBR cycle, uses general domain knowledge from the ES for the retrieval and adaption step and the ES serves as a fallback in case the retrieved design is questionable.
[Kło02] introduced an AI system which suggests designs for electronic filters. It is a hybrid system since it realizes the two paradigms CBR and neural networks: In the Retrieve phase, Approximate Nearest Neighbour Algorithms or procedures of the neural networks can be used.
Altogether, as a classic tool for expert systems, CLIPS is well suited for knowledge-driven CBR. Since we write the backend in Python and provide interfaces for adding expertise and offer function pool extension, CLIPS knowledge is not necessary for users or programmers who want to extend the framework.

3.3 Assistance Systems and Approaches for Pass Schedule Design in Open-Die Forging

[NB12] developed the CBR system *Manusoft* for pass schedule design. The finite element method was used to revise the cases.
[HM08] presented a CBR system for pass schedule and forging die design. The system is built with the ontology editor *Protégé* and the plugin-version of *myCBR* which assists with similarity assessments.
[BFKT08] introduce the software HFS based on empirical formulae.
The flexible forging cell, developed by [Elb04], consults a knowledge-based system as a module to create schedules incorporating geometry properties.
Besides these there are a variety of different approaches. For instance, [KCYM02] uses a functional neural network, [DS21] a multilayered perceptron network based on back propagation, [RRG+21] combined fast models with double deep Q learning and [WRRH19] combined fast models with an optimization algorithm. Multiple approaches use the FEA [ASBS18] [KPD16]. [RPS03] proposed an expert system, [UMKK17] and [UMKM21] use transformation rules and [YG10] applies empirical formulae.
Overall, the systems are not designed to be customizable. They are based on static knowledge such as empirical forms, strict rules and mathematical models, or on machine learning models that do not appear transparent. In contrast, we build a framework where engineers themselves design the reasoning process.

Chapter 4

System Design

The CBR framework described in this thesis is intended to enable users with little or no programming knowledge to write their own CBR applications. This prototype will follow the CBR cycle with focus on the retrieval process. This is based on a similarity measure that is designed by the users themselves. We believe that the framework can be useful in many application fields.
This Chapter starts with the use case on which the prototype is focused. Based on this, the requirements for the framework are derived. This is followed by a presentation of the functionality of the framework from the user's point of view, which introduces solutions for achieving the requirements. In the following Section, the architecture of the framework is briefly presented. The role of individual components is explained as well as their interrelationships. These are discussed in more detail in Chapter 5.

4.1 Use Case and System Requirements

One possible field of application is Open-Die Forging, which we consider a use case and to which the prototype is oriented. Since the domain of Open-Die Forging is not completely understood and experts create pass schedules based on their subjective accumulated knowledge, a rigid assistance system is not promising. Instead, a personalized assistance system lends itself to institutions in which the design of pass schedules is often concentrated on one or a few experts. Assistance systems adapted by them themselves could be helpful.
Thus, the goal is to provide domain experts (that are not required to have programming skills) with a framework with which they can create their own personalized applications that adequately preserve their knowledge. This results in the following main requirements on the framework:

1. It should produce sufficiently trustworthy applications, so that they are actually used. Generally, people are considered averse to systems they do not understand, and understanding is a prerequisite for acceptance [BHvdAW21]. CBR is generally considered a transparent approach because the reasoning and decisions can be retraced by specifying the CB, the metric, and the most similar cases. Therefore, we believe that a CBR approach, where these can be considered, is well suited to the framework and appropriate for approaching Requirement 1.

2. It should be customizable. One concern of the CBR Approach is that individuals each have a subjective perception of the correct similarity measure, relevant past cases, and their representation [SWA+21]. A deviation can thus lead to a rejection of an application as it is no longer an accurate representation of the expert's conserved knowledge. That is why the possibility of tailoring them should be given.

3. It should be usable without programming knowledge and easily extendible if the necessary programming skills are available.

Further, we address the following secondary requirements:

4. It should offer customizability support. Explanations could help users to use the framework. For example, a brief explanation of the syntax of similarity measures in the framework could be of help.

5. It should provide the possibility that knowledge can be exchanged between users. In a field where experts take different approaches to a problem and where no clear solution is known, it can be interesting for users to see the applications created by colleagues. As an example, they can find an anchor for their own application in a colleague's application.

4.2 System Workflow from Users Perspectives

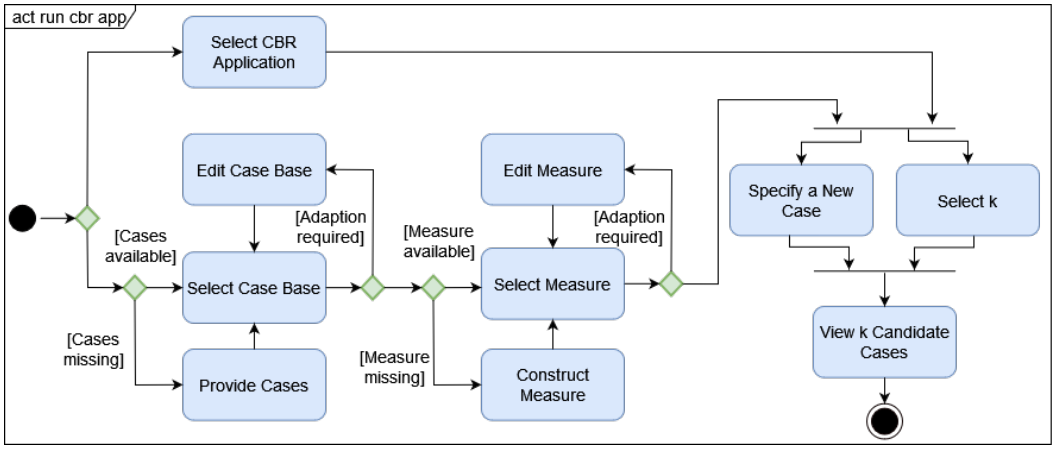

Figure 4.1: Activity diagram covering the User's point of view.

The activity diagram in Figure 4.1 illustrates the basic workflow in the framework based on the actions of the users. All of them are executable via a graphical user interface for Requirement 3 and there is a help function that provides explanations on how to use it for Requirement 4. Users start by either selecting a saved CBR application or creating a new one. The latter includes the definition of a case

base and a similarity measure. Starting with the definition of a case base, they can provide a new collection of cases or select an existing CB. They can adapt it if necessary which includes the addition and removal of cases as well as the deactivation and re-addition of features in favor of Requirement 2. Once this is completed, users define a similarity measure. In order to select one, users should be able to compose the overall similarity from the implemented feature similarities, load and edit measures defined by others also in favor of Requirement 2. For the scope of this thesis we will focus on providing local similarity measures to satisfy the needs for the Open-Die Forging use-case. For example, we provide the absolute difference as a local similarity measure for numerical values and the weighted sum as a global measure as predefined. An interface is available to allow an easy extension of the available local and global similarity measures contributing to the fulfillment of Requirement 3. Further, there is a possibility to formulate constraints with the help of predefined operators to further tailor local measures. For example, the inequality 4.1 states that the final height of the new case and another case must not differ by more than 1%. Cases where this is not the case can be assigned a similarity score of 0. This results in the requirement for users to be able to express their knowledge in inequalities, which we can assume for our use case.

$$\frac{finalHeight\,(CB\,Case)}{finalHeight\,(new\,Case)} \stackrel{!}{=} 1 \pm \Delta 1\% \tag{4.1}$$

Case Bases, Measures and the CBR Applications they form are saveable with annotations explaining users choices to contribute to the fulfillment of requirements 5. With the application set, users specify a new case and k, the number of nearest neighbors to display. The system calculates the most similar cases and presents them in a list sorted in descending order of similarity scores. In the representation, the values of the global as well as the individual local measures can be viewed. At any time the set entities can be viewed for Requirement 1.
After one iteration, another application can be executed, leaving the previous application and its results visible. On the one hand, this allows users to compare their application with those of colleagues (Requirement 5). Also, this functionality can be useful to further develop an own application. As an example, the users might have only a rough idea of a suitable measure but know what good results look like. Then they could vary their first application by trying different constraints for a local measure and can generate an appropriate measure by comparing the results.

4.3 System Software Architecture

The general system architecture is shown in Figure 4.2. The guideline for its design was to create accessibility for users without programming knowledge and comprehensibility for users with such knowledge. Therefore, the system contains a graphical user interface (GUI) and has a modular design. Overall, the framework can be divided into the frontend and the backend with the connection to the database.

Another guiding principle was to increase efficiency by mitigating the data to be transferred between these to keep latency small and to not slow down the system. In the following, the roles of individual packages and components and their interactions are examined.

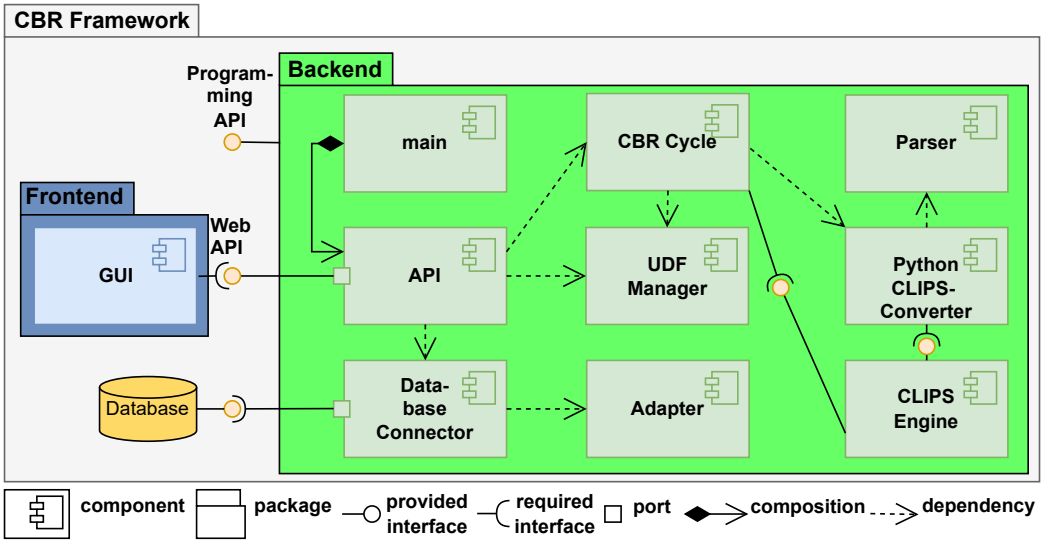

Figure 4.2: System Architecture Component Diagram

4.3.1 Frontend

We opted for a web-based framework where users are not exposed to code to meet Requirement 3. The frontend provides a GUI and serves as a communication interface between users and the framework. This is a react[1] application written in Typescript so that it should be displayable in any browser. The data-fetching library React Query[2] assists with requests e.g. by indicating whether a request is still being executed or has succeeded.

4.3.2 Database

Since there can be large amounts of data in the case base, we need appropriate storage. We decided to use a MongoDB database for the storage of all entities. Since compound storable entities such as a CBR application consisting of a case base and a measure exist, distributed storage lends itself to mitigating data duplication in the database. Further, distribution also helps mitigate data transfer volume, as for Frontend queries of entity sets often a limited view of the data suffices and this can be queried in the database alone.

[1] https://reactjs.org/
[2] https://tanstack.com/query/v4

4.3.3 Backend

Python is chosen as the language for the backend. As a beginner-friendly language, it is a popular choice among people across different domains. It may also facilitate the incorporation of machine learning models, to deduce similarity metrics from the available data within the case base. Python seems to be a popular language for developing these, as the de-facto standard ML libraries (e.g., pytorch[3] , stablebaseline3[4]) are written in Python. On the one hand, we need an intermediary between the frontend and the database that can also accurately prepare data for storage or return. On the other hand, we also need the calculation of the nearest neighbor list, which is handled by the backend. It is composed of a number of components with their own scope of tasks, which we will introduce in more detail.

Programming API

Developers can adapt the systems addresses and/or extend the pool of predefined similarity measures by adding an own programmed function as a user-defined function (UDF) to the UDF Manager. For this purpose, functions are available to them as interfaces.

API

All functionalities of the framework are available via a web API provided by the API component. It was built with the FastAPI[5] Python web framework which takes care of bridging the frontend and backend. Besides providing endpoints, the API component is also responsible for using HTTP standard compliant methods and appropriate status codes.

Adapter

New collections of cases should be able to be uploaded. Oriented to the Matlab test data we got from IBF, it is possible to upload a new collection of cases in an xml and csv file. To be able to store this in the database, the DB Connector contacts the Adapter, which transfers the data into Python types.

Database Connector

The database Connector provides the available operations to retrieve or manipulate entities. It hides their distributed storage and the perparation of data. On the one hand, it includes a database client that is connected to the database using the asynchronous Python driver Motor[6]. On the other hand, it takes care of the

[3] `https://pytorch.org/`
[4] `https://github.com/DLR-RM/stable-baselines3`
[5] `https://fastapi.tiangolo.com/`
[6] `https://motor.readthedocs.io/en/stable/index.html`

composition or deconstruction of entities to store them in the database or to regain them from it.

User Defined Function Manager

The UDF manager is used to specify which externally defined functions should be available for defining a similarity measure. These can be passed either using the Web API via the API component or directly in code via a service (using the programming API).

Python-CLIPS-Converter

The Python Clips Converter is responsible for translating Python expressions and logic into their CLIPS code counterpart and submitting it to an environment.

Parser

The parser acts as a translator between user input and the CLIPS Engine. For example, ordinary users use infix notation for expressions. The parser converts formulae from infix to CLIPS compliant prefix notation so that they can be used within clips rules.

CLIPS Engine

The CLIPS Engine takes care of the Retrieval Phase of the CBR Cycle. By using the declarative language, the possibility is provided that users of the framework can create their own systems with naturally formulated constraints (which are translated into CLIPS rules) without worrying about the control flow [Ril22]. In addition, having a knowledge-based engine is the natural choice when a program explicitly reasons about knowledge of the domain experts.

CBR Cycle

The CBR Cycle module is responsible for the execution of a CBR Cycle. For each cycle a separate CBR Cycle object with its own CLIPS environment is created. CBR App, new Case and k are passed during initialization and are available as attributes afterwards. At initialization, a Cycle object contacts the UDF Manager to link user-defined functions in its environment. An object offers methods for all phases of the CBR Cycle: Retrieve, Reuse, Revise, Retain. Whereby only the first and last are considered in the prototype and the others contain no functions. The reasoning process for the Retrieve Phase takes place in a dynamically determined CLIPS program, the CLIPS engine, that is called as a subroutine. This is constructed, by the Cycle object passing its environment, the CBR application, new case and k - the number of nearest neighbors - to be reported to the PythonCLIPSConverter.

Chapter 5

System Implementation

In this Chapter we take a closer look at the implementation[1] of the open-source CBR framework, which allows users to create and run their own CBR applications via the frontend. The entire framework can be deployed with a Docker Compose file. Backend, frontend and database each run in Docker containers illustrated in Figure 5.1,
where each color corresponds to an individual container. These are built with the

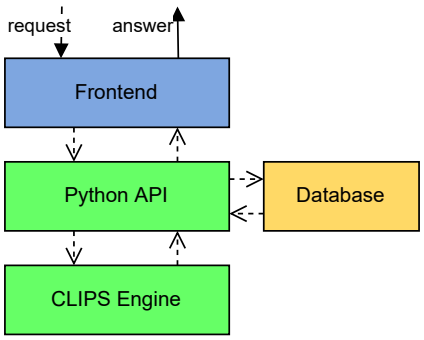

Figure 5.1: Layered Architecture

```
$ docker-compose build
```

command and launched via

```
$ docker-compose up
```

command which aggregates the outputs of the containers. Divided roughly, the system illustrated in Figure 4.2 corresponds to an architecture with four layers, where the CLIPS Engine layer is only present in the calculation of the nearest neighbor list, as shown in Figure 5.1. Requests are passed from the top layer to the bottom layer and sent back again, with each layer performing its processing task. In the following we will highlight the functionality of each layer in more detail.

5.1 Database Layer

The database stores pesistent entities involved, i.e. applications, measures, case bases and user-defined functions in the BSON[2] (basically in key-value pairs) format in collections. As an example, Figure 5.2 shows a snippet of a case in BSON format.

[1] see `https://git.rwth-aachen.de/sarah.soomro/cbrframework/-/tree/main/`
[2] `https://bsonspec.org/spec.html`

```
{{ "_id": {"$oid": "6356d98b292b28cd2d1a753e"},
     "p": {"cbr_h0": 111, ...},
     "s": {"0": {"cbr_ph0": 1, ...},
           ...,
           "means": {"cbr_ph0": 1.8, ...}}}}
```

Figure 5.2: Example Case in BSON Format

It has a unique object ID, a problem description and a solution description consisting of several passes and average values. Partly entities are stored distributed in different collections. For example, the case shown is in the collection with the name *CaseCollection*. This is referenced via a configuration file in the collection for cases collections with an ID, the schemata of the problem and solution description as well as the name for the cases collection depicted in Figure 5.3.

```
{"_id": {"$oid": "635693bad7006bfb2eb990c1"},
  "name": "CasesCollection",
  "p_schema": {"cbr_h0": "n", ...},
  "s_schema": {"cbr_ph0": "n", ...}}
```

Figure 5.3: Configuration entry for the collection of cases named CasesCollection. The "n" in the schemata indicates a numeric domain.

This raw case base can be customized by users regarding their cases and features. The results are recorded in configuration files in an own collection that reference the raw case base by its id and specify ignored cases and features. In this way, multiple tailored Case Bases can be created for the same collection of Cases without having to store duplicate cases. Further, it allows us to mitigate the transfer time between the backend and the database, for example, for the query of all case base configs that we make use of in the frontend. This only queries name and comment to provide the user with a pre-selection. The retrieval of the entries in the case base configuration collection are therefore sufficient.

5.2 CLIPS Engine

The CLIPS engine simulates the retrieval process of the CBR Cycle. In doing so, it assumes distance measures. It contains the different cases as facts and implements the distance calculation in the form of rules. The result of the calculation, the distance scores, are then in turn available as facts. As explained in Chapter 4, the measure is composed according to the local-global principle. The processing of constraints, local measure functions and global measures is handled in this order by specifying salience. As an example, Figure 5.4 shows the translation of constraint

4.1 into two rules that check if the ratio of the final heights is within the allowed limits and assign a global score of 0 otherwise.

```
(defrule local_constraint_h0_0
    (declare (salience 10))
    (cbr_case (id ?id) (status "inCB") (cbr_h0 ?cbr_h0))
    (cbr_case (status "new Case") (cbr_h0 ?newcbr_h0))
    ?sim <- (similarity_case (foreign_id ?id)
                             (overall_similarity -1.0)
                             (cbr_h0 -1.0))
    (test (< (/ ?newcbr_h0 ?cbr_h0) 0.99))
=>
    (modify ?sim (overall_similarity 0.0)))

(defrule local_constraint_h0_1
    ...
    (test (> (/ ?newcbr_h0 ?cbr_h0) 1.01))
=>
    (modify ?sim (overall_similarity 0.0)))
```

Figure 5.4: Translation of the Inequality 4.1 into CLIPS Rules.

Regarding local measures, there are 2 different types: those for numeric types such as the absolute difference for the initial height, which can be translated into the rule 2.6, and those for non-numeric types. The latter correspond to nominal or ordinal data and can be defined by means of a matrix that specifies the pairwise distance of expressions of the domain which is a standard procedure within CBR for these data types [Cra17]. As an example, consider the following matrix for the domain *material* in which the same material is given a distance of 0 and more similar material is given a smaller score.

Distances	42CrMo4	C45	1.4301
42CrMo4	0	0.2	0.6
C45	0.2	0	0
1.4301	0.6	0	0

Table 5.1: Distance Matrix for the Domain Material.

For the calculation, each cell is translated into an unordered fact and the rules filter out the corresponding entry. Further, the global measure weighted Sum (see Equation 2.1) is used by the Engine. The weights are available as facts, are offset per domain against the local scores and again cached in facts, summed up and entered as a global score for the cases by means of rules.

5.3 Python API

For the Python API, we can distinguish between two types of users with different possible actions.
On the one hand, there is the end-user, who manages entities via the frontend. The Python API layer is used for processing such requests. Entities can be read, created, updated or deleted. In most cases, the API component forwards the request to the DB Connector component, which executes it. On success or failure, the API responds to the request accordingly as described in Chapter 5.3.2. Exceptions include uploading a new collection of cases, getting user-defined functions, and requesting the nearest neighbor list, which we highlight below.
On the other hand, there is the actor administrator, who can customize the network distribution and include his own Python functions. We start the explanations from the customizability of the layer through the programming interface and conclude it with the presentation of the HTTP API and the workflow of individual components to the aforementioned exceptions by using this.

5.3.1 Programming API

The code snippet in Figure 5.5 shows the use of the functions provided for the administrator and was written in the *main.py* file.

```
1
2  def f100(x, y):
3      return x+y
4
5
6  udf_manager.add_UDF_callable(f100)
7
8  dbconnector.set_database_name("CBRFramework")
9  dbconnector.set_database_address(
10     "mongodb://CBRFramework-db:27017/CBRFramework")
11 app = CBRAPI(origins="http://cbr-frontend:3000").app
12
13
14 if __name__ == "__main__":
15     uvicorn.run("main:app")
```

Figure 5.5: Customization using predefined functions.

The smallest program created with the framework contains solely the declaration of a CBR API with implicitly set default values and launches it in the main component. These default values give the frontend the address localhost:3000 and the database the address localhost:27017. The administrator can assign his own values and also configure the Cross-Origin Resource Sharing (CORS) middleware between front- and backend by setting the allowed clients, for example. All addresses, i.e. of

frontend, backend and database are also changeable in the dockerized setup. Furthermore, the administrator can register his own functions with the UDF Manager, which are then also available in the frontend.

5.3.2 HTTP API

Client requests are sent to endpoints to retrieve and modify resources. Endpoints indicate these with their paths and specify their function with a HTTP method. As mentioned earlier, we are guided by the REST standard, which dictates a uniform interface. We therefore provide endpoints with the GET, PUT, POST, or DELETE HTTP method with their standard meaning and appropriate HTTP response codes in responses as shown in Table 5.2.

Method	Meaning	Successful Response	Unsuccessful Response
GET	Retrieve Resource	200 OK	404 Not Found
PUT	Update Resource	200 OK	400 Bad Request
POST	Create new Resource	201 Created	400 Bad Request
DELETE	Delete Resource	204 No Content	404 Not Found

Table 5.2: Used HTTP Methods with Meaning and Responses

All endpoints have a similar structure, i.e. for a resource there are mostly the same adapted endpoints. All 4 methods are available, if this was necessary for the functionality of the frontend, e.g. for the CBR applications, where Figure 5.6 shows the existing endpoints.

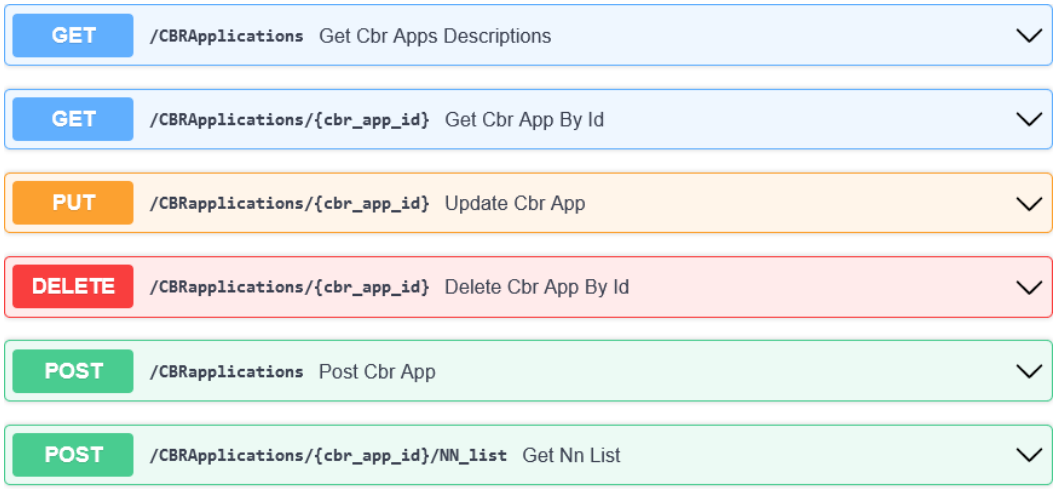

Figure 5.6: Enpoints for the Resource(s) CBR Application(s) displayed via FastAPI's interactive API documentation that utilizes the Swagger UI.

Visible is also the hierarchical structure of the URIs: all CBR applications are

queried via the path */CBRApplications* and one resource of this collection is queried by specifying its ID in the path parameter (*/CBRApplications/cbr_app_id*). The nearest neighbor list as a subcomponent of this application gets the path */CBRApplications/cbr_app_id/NN_list*. The path appended to the address of the backend forms the URI. When a collection of resources is requested, a restricted view of the individual resources is returned. When requesting all CBR applications as an example, their descriptions consisting of name and comment are returned. This is to reduce the traffic between frontend and backend. The overhead is reduced, because in fact the frontend only needs these details. Request and Response bodies use the common JSON format with the exception of the endpoint for uploading a new collection of cases where a byte stream is transferred. FastAPI handles the conversion of types between front- and backend.

5.3.3 Adding a new Collection of Cases

The database can be populated in many ways e.g. via the official MongoDB tools or programmatically via a driver like Motor. However, experience in this area is rather unlikely in our target group. Therefore we offer the upload of a new collection of cases in the form of files tailored to our use case. Two files are expected. The requirements for the files are based on the Matlab test data we received from the IBF and aim to be easy to realize. However, the framework can be easily extended to cope with other file formats as well. The case representation is read from an XML file corresponding to a Matlab struct S and can be generated in Matlab using the command

```
writestruct(S,filename).
```

The raw CB is submitted as a csv file corresponding to a matrix CB and is generated with the Matlab command

```
writematrix(CB).
```

Figure 5.7 shows the interaction of the components involved for a successful upload.

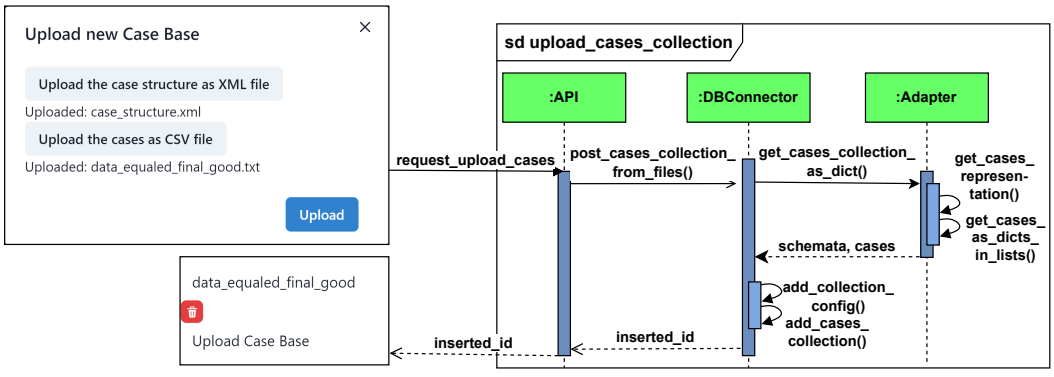

Figure 5.7: Sequence Diagram for Uploading a Collection of Cases

First, the files are uploaded in the frontend and its byte streams are passed to the DB Connector via the API component. This queries the Adapter for the decoded data that has been converted into suitable data structures. Then the Connector creates a configuration file for the raw CB and a separate collection with its cases. When completed, the collection is available for selection in the frontend.

5.3.4 Retrieving User-Defined Functions

There are two types of user-defined functions: those that are added by web clients via the frontend and stored as objects and those that are registered as callables with the UDF Manager by the administrator. For the request of the totality of all functions, Figure 5.8 shows the workflow for a successful request. Thereby the function $f100$ was previously added in the backend as shown in Figure 5.5 and the constant *euler* which corresponds to Euler's number was added via the frontend.

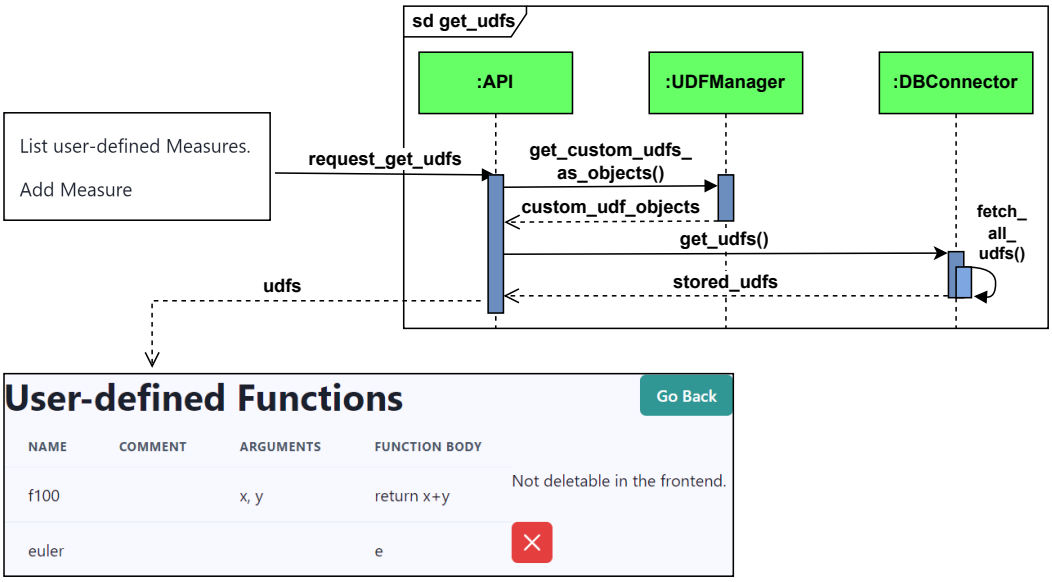

Figure 5.8: Sequence Diagram for the Retrieval of all user-defined functions

The API gets the registered callables as objects from the UDF Manager. This has previously stored them directly as a copy in objects during registration. Further the API gets the stored UDFs from the DBConnector which fetches from the database. Both sets are combined and returned.

5.3.5 Request of a Nearest Neighbors List

The interaction of the components for successfully requesting a nearest neighbors are elaborated in the following. As illustrated in Figure 5.9, the API component is passed the ID of the CBR application, the new case, and k, the number of nearest

neighbors to be determined, as input. It receives the stored app and UDFs from the DB Connector. Thus it has all the information necessary for the Retrieve phase and orders the creation of a new Cycle object, which is initialized with the app, the UDFs, the new case and k.

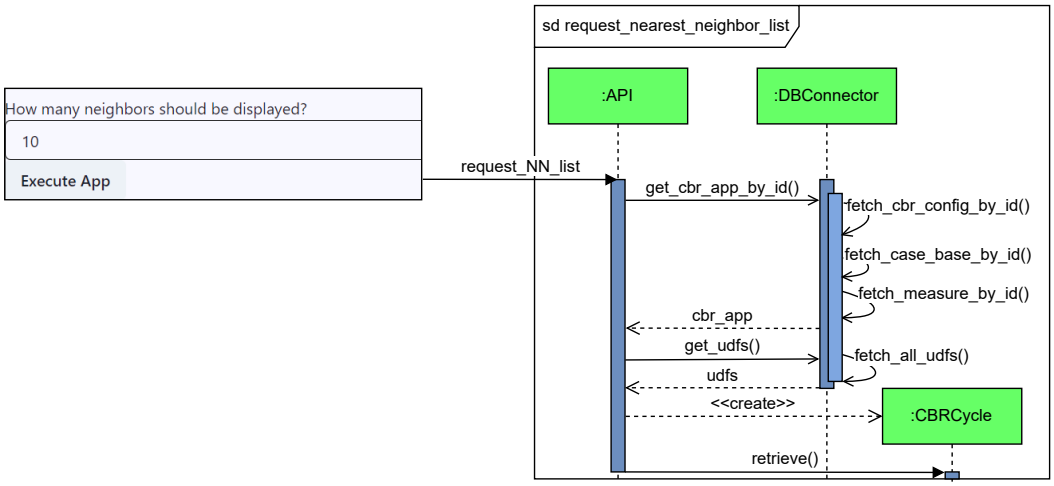

Figure 5.9: Sequence diagram showing the interaction of the components for querying a nearest neighbor list to a CBR application via the HTTP API.

The next step is shown in Figure 5.10. The Cycle object requests the translation of its fields into a CLIPS program that calculates the Retrieve Phase from the PythonCLIPSConverter and passes it its environment in which this is to be registered. Next, the converter instructs the UDF Manager to register all saved UDFs with the environment. After that, the Converter asks the manager for all UDFs in the form of objects, and then passes them to the Parser, so that this can recognize the function symbols and process them according to their precedence. This inserts them into its grammars accordingly. A series of functions are then performed by the Converter, creating templates for the entities involved and inserting them as facts, and inserting the appropriate rules as explained in Section 5.2. CLIPS compliant prefix equivalents to the local measure functions of numeric type and constraints written in infix notation are requested by the parser. Altogether, the Converter creates the CLIPS Engine in this way.

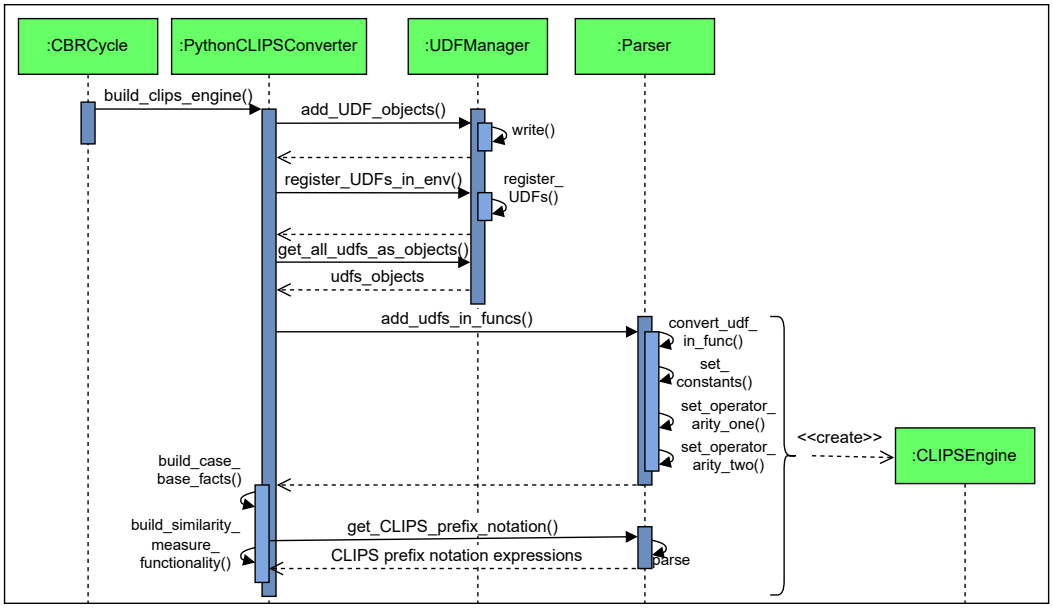

Figure 5.10: Resumed sequence diagram showing the interaction of the components for querying a nearest neighbor list to a CBR application via the HTTP API.

As shown in Figure 5.11, then the Cycle object executes the Engine and asks the Converter for the nearest neighbors list. The Converter translates this from the facts to the distance scores and creates a list sorted in descending order. This is passed on from the Cycle object to the frontend via the API.

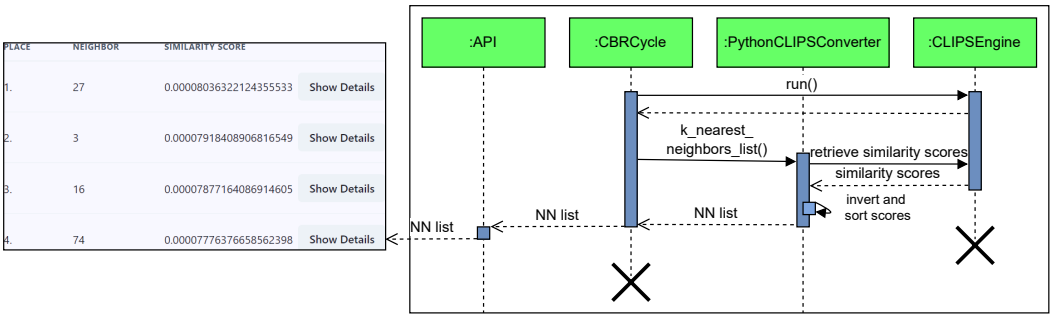

Figure 5.11: Resumed sequence diagram showing the interaction of the components for querying a nearest neighbor list to a CBR application via the HTTP API.

5.4 Frontend

The frontend provides users with a GUI to read and edit entities. It makes use of the HTTP API, provided by the Python API layer. In its design it is aligned with the design goals described in Section 4.2. Since each tab that calls the frontend has its own session storage, multiple applications can be built and their nearest neighbor lists compared. Figure 5.12 shows a snippet of the front end.

Figure 5.12: Layout of the frontend

The basic layout is a menu bar on the left that selects what to manage in the window on the right. Case Base, Measure, New Case and Result sections can be selected at the same time and are displayed one below the other for a good overview. We will start with the explanation of the header and then go into the functionality of individual sections.

Header

The button on the left side of the header opens a menu for the management of the CBR applications. Either users create a new app or they open a modal via which they can load or delete apps as shown in Figure 5.13.

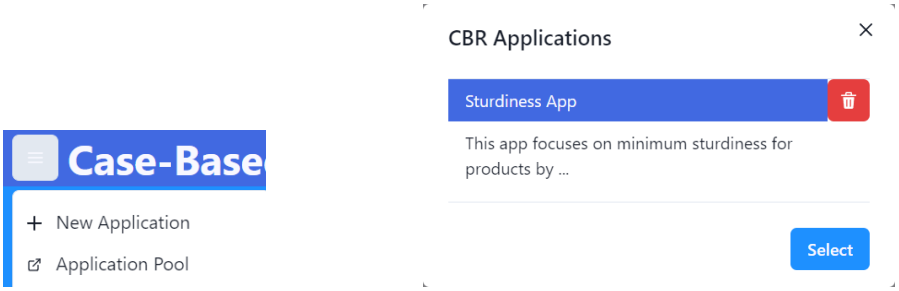

Figure 5.13: CBR Applications Menu (left) and Modal (right)

The question mark button on the right side opens a help drawer that assists e.g. with the formulation of similarity measures by explaining the inductive syntax and giving examples.

Case Base Section

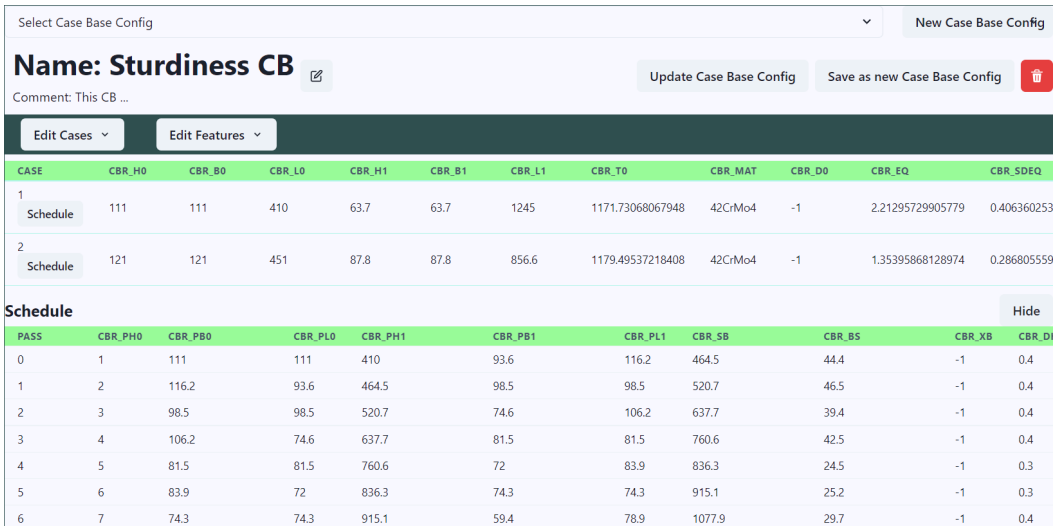

Figure 5.14: Case Base Section

Regarding Case Bases features and cases are editable in the sense that they can be removed and added again. New cases can be added or the name and comment can be changed. The New Case Base Config button can be used to upload new collections of cases from an xml and csv file. For a selected case base, the pass schedule can be called per case and also gives average values of the features as these are a reference point for the evaluation of an entire schedule for engineers.

Measure Section

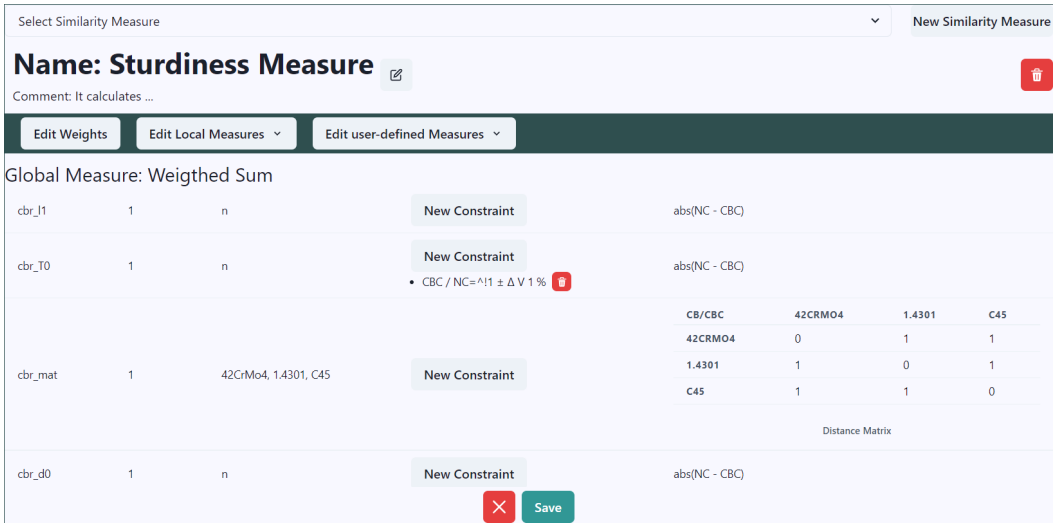

Figure 5.15: Measure Section

In the Measure section, measures can be tailored. Specifically, the weights of the weighted sum can be assigned for the individual features, local functions can be defined and constraints can be formulated. Figure 5.15 shows a snapshot of the section where the constraint $NC \,/\, CBC =^{!} 1 \pm \Delta\, V\, 1\,\%$ has been added and the edit is going to be saved by the green Save button at the bottom. The constraint corresponds to the inequality 4.1. By means of buttons users get assistance in entering the $\pm$ sign and Δ. To reference the values of the feature from the new case and case from the CB, the constants NC and CBC are used respectively, and for the ratio in a feature, the constant V is used. The available UDFs can be managed and expanded using the *edit user-defined Measures* button in the section header. UDFs can be viewed in a list with indication of the name, the comment, the arguments and the function body. They can be added via the frontend and also deleted again. To add a new function, these values must be entered as shown in Figure 5.16. Functions from the Python standard library math[3] are available. An input help is implemented in the form of placeholders for an example.

[3] `https://docs.python.org/3/library/math.html`

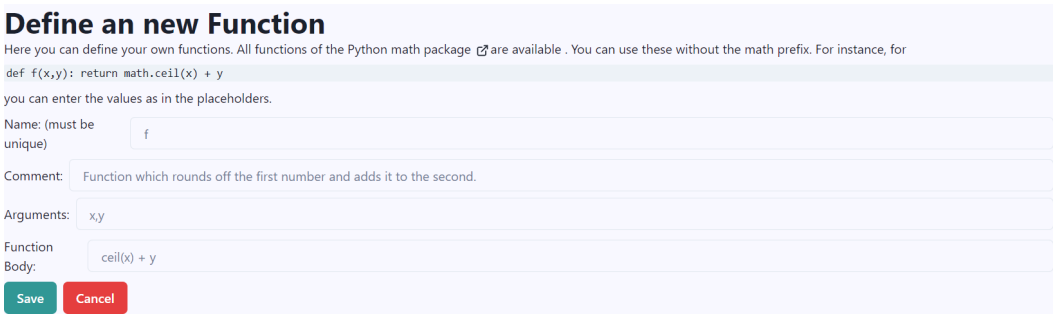

Figure 5.16: Formular for adding a new UDF

New Case Section

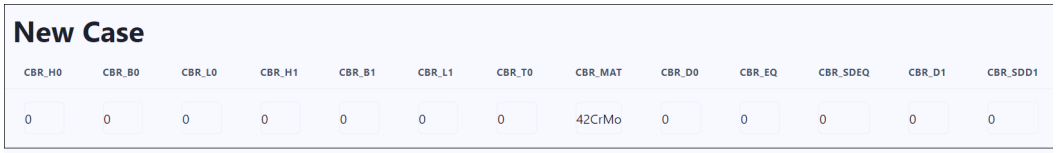

Figure 5.17: New Case Section

In the new case section, users simply enter the values of the new case in a table as depicted in Figure 5.17.

Result Section

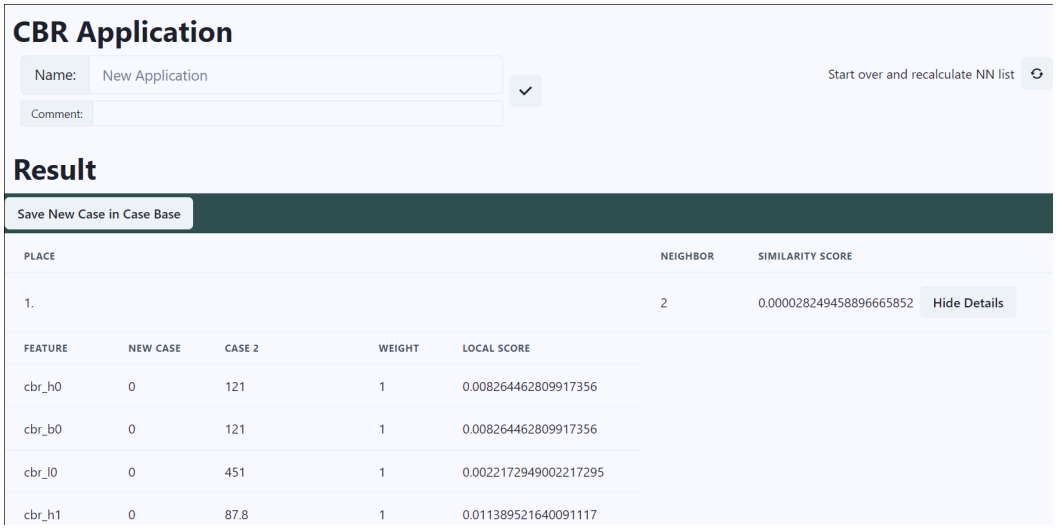

Figure 5.18: Result Section

In the Result Section the created CBR application and the new case can be saved in the CB. Furthermore, the nearest neighbor list is displayed here after specifying the number of neighbors to be displayed. The neighbors are listed in descending order. The local scores are optionally visible, as illustrated in Figure 5.18.

Chapter 6

Evaluation

In the evaluation, we examine the extent to which we have achieved the goals formulated in Chapter 4.1. It consists of two parts. The first step is to prove the concept by applying the framework, where we create CBR applications for Open-Die forgings. In doing so, we demonstrate the usage of the frontend in a presentation to our target audience, engineers at IBF, who provide feedback on the frontend design. Furthermore, we evaluate the efficiency of the Framework in terms of runtime.

6.1 Proof of Concept and Frontend Evaluation

In a presentation including a manual, engineers from IBF were introduced to the frontend and its functions. We filled the case base with the test data provided for Open-Die forgings and therefore created CBR applications for this domain. It was discovered that constraints could be added but not deleted and that the predefined fill of the distance matrix for the feature material corresponded to a similarity matrix. Both bugs have been fixed. It was also noticed that users were not sufficiently informed that distance functions were defined in the frontend. For this, a color-coded sentence was added in the help drawer. In addition, there was the remark that saving a CBR Application in the Result section is rather intuitive and some input fields were too narrow for input. Further, the feedback included requests for additional features that were implemented if it was feasible in terms of time within the scope of the thesis:

- **Pass Schedule Display.** For a recorded case, the respective plan should be available for viewing. Specifically, plans should be viewable in the applications overview menu in the detail view and in the Results and Case Base sections per case. The display was implemented as an example for the Case Base section. Its corresponding react component is reusable for future work.

- **Inclusion of average, maximum and minimum values in pass schedules.** These should be included since they are often used for guidance. The first one was added as an example.

- **Uploading a case from a file.**

- **Compact view of multiple nearest neighbor lists.** The comparison of the different nearest neighbor lists should be enabled within one window, i.e. one tab.

- **Formulation of metachecks.** Checking uploaded data for validity and otherwise triggering an error message in the frontend is beneficial. As an example, this could be in the form of a not to exceed threshold of 100 for values of a numeric feature.

- **Specifications of units of measurement in tables.**

- **Reference or explanation to the Python math library.** UDFs can be created by using the library functions via the frontend, which should be viewable. For this purpose, an icon button was added, which redirects the user to the corresponding entry in the Python documentation.

6.2 Efficiency Assessment

When using the frontend, it is noticeable that the display of the various cases in a case base as well as the nearest neighbor list take a while. In the former case, this is due to the fact that all cases are obtained and displayed simultaneously from the backend. A limited request of cases in the form of paginated requests would remedy this. Then less data is transferred and rendered. As a react component, a table with configurable sections, e.g. with 10 displayed cases per section, is suitable for this. Unfortunately, currently this is not available in the component library Chakra UI[1] we use. This issue also affects the latter case. However, the list to an inquiry of a a small number of neighbors does not occur immediately for the users as well. We therefore focus on the second case and search for bottlenecks.

In our measurements, we drive the framework via the frontend and measure the elapsed time for different numbers of cases in the CB. They were performed on the Windows 10 operating system, using the Firefox browser with 16 GB of RAM and an AMD Ryzen 7 processor. We took the wall-clock time as a measure to consider time elapsed for waiting for requests or sending responses. Since the measurements use wrappers for the functions, for example, the measured times differ a bit from the actual application of the framework.

Overall, we look at the time that elapses between submitting the request for a nearest neighbor list and displaying it in the frontend. We measure separately the time until the frontend receives a response (Request Answered) and has finished processing the list (GUI Response Time). Next we take a look at the backend. After getting the application and the UDFs from the DB Connector, the API component orders a Cycle object and requests the list of nearest neighbors (Retrieve). To do

[1] `https://chakra-ui.com/`

this, the CLIPS Engine is built (BuildEngine), executed (RunEngine) and the list is then read from its facts (RetrieveList). The results are shown in Figure 6.1.

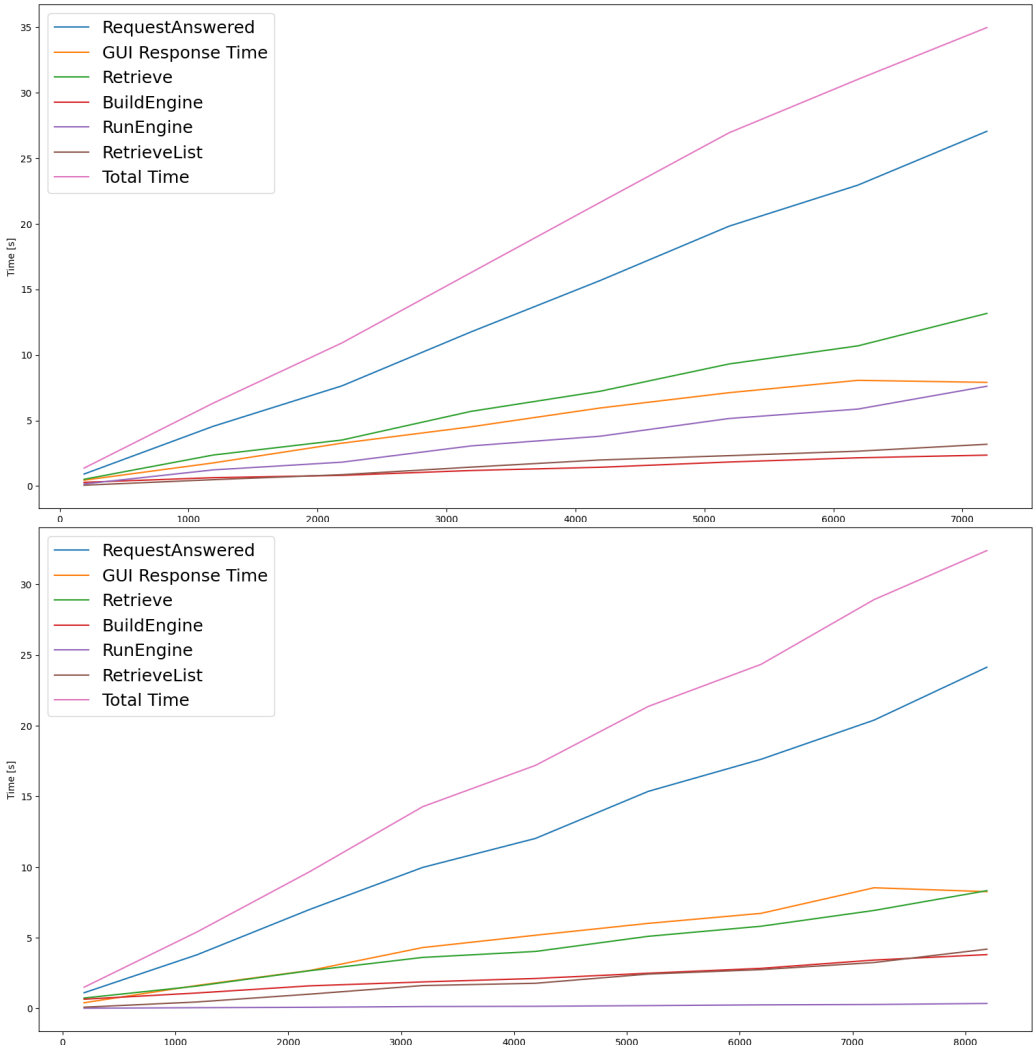

Figure 6.1: Measures elapsed time for increasing the size of the CB without constraints and UDFs (top) and with these (bottom).

It is visible that creating the Nearest Neighbor List after receiving the response takes a lot of time proportionally. The calculation in CLIPS still makes a significant part. Parallelizations of calculations and saving of calculated similarities could be useful.

Chapter 7

Discussion

We gathered feedback from our target audience and tested the framework with regards to its scalability with increased CB. Although performing decently for rather small data sets, where the frontend is the limiting factor, further performance increases in the backend are necessary as well to enable usage in domains with large amounts of available data. We find further optimization potential with regards to performance by caching common calculations and by utilizing multithreading.

In addition, we see opportunities for improvement and potential topics in the future for individual themes, which we describe below.

Frontend We think that a visualization of the scores for a case could be an explanatory aid for users. As an example, [LSG+19] use a distance preserving scatter plot for their CBR system that illustrates the similarity scores between new case and neighbor cases. A visualization of functions like e.g. in the GUI of the Java framework myCBR mentioned before, where functions are underlined with a plot, could also be helpful.

Parsing The parser is currently a minimal implementation and a limiting factor for the user-experience in the frontend as intuitive features, e.g., an exponential to base e cannot be parsed yet. It should be replaced by a more capable implementation in the future.

Testing Occasionally there are unit tests for the framework. Covering the program with further tests could be beneficial for quality assurance and further development.

Adapters The backend has a csv adapter. The acceptance of other file formats in the frontend would increase the user experience. Further adapters can be formulated that make use of the csv adapter as a foundation.

UDFs For security reasons, the Python modules for creating UDFs in the frontend are limited. Currently this is the math library. In order to provide users with the required functions, a function that an administrator can use to determine the imported modules via the UDF Manager could help.

Uniform Interface As an artifact of earlier implementations, the frontend gets the nearest neighbor list via a POST request. This is an exception in the uniformly formulated interface and can be resolved by splitting it into POST and GET requests.

Going one step further, the prototype should be extended to include the functionality of the Reuse and Revise phases of the CBR Cycle. This should happen with the involvement of users, i.e. users should be able to construct both phases. Based on our use case, engineers could formulate adaptation rules for the Reuse phase similar to the constraints in the similarity calculation. For the Revise phase, they could consult an integrated variant of fast models, that simulates forging plans, to assess the quality of a plan.
Glancing at other projects, CBR agents could find application in the multi-agent system developed by [Ker22]. In the system, agents collaboratively develop a production plan. In the considered use case for creating a drive shaft, the IBF agent returns a forging plan. Whether the framework is sufficient for other agents, e.g. the WZL agent (where the executing institute is the Laboratory for Machine Tools and Production Engineering of RWTH Aachen University), responsible for machining, is of significance. In order to obtain a generally usable framework, testing in other disciplines is essential.

Chapter 8

Conclusions

In this work we developed a CBR framework with which domain experts can develop their own CBR applications. CBR refers to the derivation of solutions for new cases based on the solutions of their most similar previously encountered cases. Overall, CBR is characterized by its comprehensibility increasing user trust, since it basically imitates a human way of thinking. By viewing the similarity measure, case base and new case, users can trace the reasoning process step by step.

Thereby we offer the usage in our web-based framework in form of a react based Typescript frontend. It is connected to the Python backend via the web framework FastAPI. The data is stored in a MongoDB database, which is accessed by the backend with the driver Motor. The reasoning process is executed in a CLIPS subprogram. The Retrieve phase of the CBR cycle is the focus of this work, in which measures are composed according to the local-global principle and enriched by constraints from users. Several features assist the user when defining the similarity, such as the possibility to add simple user-defined Python functions via the frontend and more complex ones such as ML models in the backend or to restore settings from other users or previously defined measures. The opportunity to share knowledge among colleagues is given by insight in each other's applications.

Altogether, special attention is given to accessibility of creating CBR applications for experts who do not require to be skilled in programming. We deployed the CBR framework in the context of Open-Die Forging. We considered it a use-case as it is a not fully understood domain with sparse data and is guided by expert knowledge when creating forging schedules.

In the evaluation, we found that the GUI response time significantly affects the runtime.

In summary, this thesis developed a CBR framework with focus on usability when designing similarity measures. It provides a baseline that covers the Retrieve phase and which should be extended in future works to cover a full CBR cycle.

References

[AKP+20] K. Amin, S. Kapetanakis, N. Polatidis, K. Althoff, and A. Dengel. DeepKAF: A Heterogeneous CBR & Deep Learning Approach for NLP Prototyping. In *2020 International Conference on INnovations in Intelligent SysTems and Applications (INISTA)*, Novi Sad, Serbia, 2020. pp. 1–7.

[AP94] A. Aamodt and E. Plaza. Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, 7(1):39–59, 1994.

[ASBS18] A. Abedian, B. Shirani Bidabadi, and R. Shateri. Numerical and experimental study of open die forging process design for producing heavy valves. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 12(1):49–61, February 2018.

[BA12] K. Bach and K. Althoff. Developing Case-Based Reasoning Applications Using myCBR 3. In *Case-Based Reasoning Research and Development (ICCBR 2012)*, Lyon, France, 2012. pp. 17–31.

[BAC97] D. Ben-Arieh and M. Chopra. A case-based NC code generating system for prismatic parts. *International Journal of Production Research*, 35(7):1925–1944, 1997.

[BFKT08] D. Bombač, M. Fazarinc, G. Kugler, and R. Turk. Tool for programmed open-die forging–case study. *RMZ–Materials and Geoenvironment*, 55(1):19–29, 2008.

[BGMZ19] R. Bergmann, L. Grumbach, L. Malburg, and C. Zeyen. ProCAKE: A Process-Oriented Case-Based Reasoning Framework. In *The 27th International Conference on Case-Based Reasoning (ICCBR 2019), Workshop Proceedings*, Otzenhausen, Germany, 2019. pp. 156–161.

[BHvdAW21] K. Bauer, O. Hinz, W. van der Aalst, and C. Weinhardt. Expl(AI)n It to Me – Explainable AI and Information Systems Research. *Business & Information Systems Engineering*, 63(2):79–82, April 2021.

[BMB+20] R. Bergmann, M. Minor, K. Bach, K-D Althoff, and H. Muñoz-Avila. Fallbasiertes Schließen. In G. Görz, U. Schmid, and T. Braun, editors, *Handbuch der Künstlichen Intelligenz*. De Gruyter Oldenbourg, 2020.

[BMJ19] K. Bach, B. M. Mathisen, and A. Jaiswal. Demonstrating the myCBR Rest API. In *The 27th International Conference on Case-Based Reasoning (ICCBR 2019), Workshop Proceedings*, 2019, pp. 144–155.

[Chi21] D. Chicco. Siamese neural networks: An overview. In Cartwright H., editor, *Artificial Neural Networks. Methods in Molecular Biology*. Humana, New York, NY, 2021.

[Cra17] S. Craw. Case-Based Reasoning. In C. Sammut and G. I. Webb, editors, *Encyclopedia of Machine Learning and Data Mining*. Springer US, Boston, MA, 2017.

[DS21] A. Di Schino. Open die forging process simulation: a simplified industrial approach based on artificial neural network. *AIMS Materials Science*, 8(5):685–697, 2021.

[EFC00] J. A. Elorriaga and I. Fernández-Castro. Using Case-Based Reasoning in Instructional Planning. Towards a Hybrid Self-improving Instructional Planner. *International Journal of Artificial Intelligence in Education (IJAIED)*, 11:416–449, 2000.

[EK90] H. Engel and C. A. Kestner. *Metallfachkunde 1: Grundlagen*. Vieweg+Teubner Verlag Wiesbaden, second edition, 1990.

[Elb04] M. A. Elbadan. *Adaptive Scheduling for an Incremental Flexible-Forging Cell*. Dissertation, McMaster University, Hamilton, Canada, 2004. `http://hdl.handle.net/11375/6276`.

[Fie00] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Dissertation, University of California, Irvine, 2000.

[For89] C. L. Forgy. Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In J. Mylopolous and M. Brodie, editors, *Readings in Artificial Intelligence and Databases*. M. Kaufmann, San Francisco (CA), 1989.

[Gia] J. C. Giarratano. *CLIPS User's Guide*. `https://www.clipsrules.net/`, 6.40th edition.

[Got07] G. Gottstein. *Physikalische Grundlagen der Materialkunde*. Springer Berlin, Heidelberg, third edition, 2007.

[Gro99] Network Working Group. *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2616. `https://www.rfc-editor.org/rfc/rfc2616`, 1999.

[HM08] C. Ho and J. Mathew. Case-Based Reasoning System for Forging Process Design. In *2008 3rd International Conference on Innovative Computing Information and Control*, Dalian, China, 2008. pp. 45–45.

[KCYM02] P.H Kim, M.S Chun, J.J Yi, and Y.H Moon. Pass schedule algorithms for hot open die forging. *Journal of Materials Processing Technology*, 130-131:516–523, 2002.

[Ker22] Maximilian Kerpen. Multi-Agent System for Production Processes in the Automotive Industry. Bachelor's thesis, Rheinisch-Westfalische Technische Hochschule Aachen, Germany, 2022.

[Kło02] M. Kłosowski. Application of Case Base Reasoning to Hybrid Expert System for Electronic Filter Design. *Bulletin of the Polish Academy of Sciences. Technical Sciences*, 50(1):37–43, 2002.

[Klo17] F. Klocke. *Fertigungsverfahren 4 - Umformen*. Springer Vieweg, Berlin, Heidelberg, 6. edition, 2017.

[KPD16] J. Kaur, B. S. Pabla, and S. S. Dhami. A Review on Field Areas of Research in Forging Process using FEA. *International Journal of Engineering Research & Technology (IJERT)*, 5(1):12, 2016.

[LSG+19] Jean-Baptiste Lamy, Boomadevi Sekar, Gilles Guezennec, Jacques Bouaud, and Brigitte Séroussi. Explainable artificial intelligence for breast cancer: A visual case-based reasoning approach. *Artificial Intelligence in Medicine*, 94:42–53, 2019.

[MABL20] B. M. Mathisen, A. Aamodt, K. Bach, and H. Langseth. Learning similarity measures from data. *Progress in Artificial Intelligence*, 9(2):129–143, June 2020.

[MBd11] J. A. Matelli, E. Bazzo, and J. C. da Silva. Development of a case-based reasoning prototype for cogeneration plant design. *Applied Energy*, 88(9):3030–3041, 2011.

[Mor00] G. Morcous. *Case-based Reasoning for Modeling Bridge Deterioration*. Dissertation, Concordia University, Montreal Quebec, Canada, 2000.

[NB12] C. Numthong and S. Butdee. The Knowledge Based System for Forging Process Design based on Case-Based Reasoning and Finite Element Method. *Applied Science and Engineering Progress*, 5(2):45–54, 2012.

[NSM18] R. M. Nosofsky, C. A. Sanders, and M. A. McDaniel. A Formal Psychological Model of Classification Applied to Natural-Science

Category Learning. *Current Directions in Psychological Science*, 27(2):129–135, 2018.

[Pan15] A. Pannu. Artificial Intelligence and its Application in Different Areas. *International Journal of Engineering and Innovative Technology (IJEIT)*, 4(10):79–84, 2015.

[RCSL87] G. Riley, C. Culbert, R. T. Savely, and F. Lopez. CLIPS: An expert system tool for delivery and training. In *NASA. Marshall Space Flight Center, 3rd Conference on Artificial Intelligence for Space Applications, Part 1*, 1987, pp. 53–57.

[Rec14] D. Recker. *Entwicklung von schnellen Prozessmodellen und Optimierungsmöglichkeiten für das Freiformschmieden.* Dissertation, RWTH Aachen University, Shaker Verlag, Aachen, 2014.

[RGDAGC13] J. A. Recio-García, B. Díaz-Agudo, and P. A. González-Calero. The COLIBRI Open Platform for the Reproducibility of CBR Applications. In *Case-Based Reasoning Research and Development (ICCBR 2013)*, Saratoga Springs, NY, USA, 2013. pp. 255–269.

[RGGCDA14] J. A. Recio-García, P. A. González-Calero, and B. Díaz-Agudo. jcolibri2: A framework for building Case-based reasoning systems. *Science of Computer Programming*, 79:126–145, 2014.

[Ric] L. Richardson. Justice Will Take Us Millions Of Intricate Moves. Conference Presentation. QCon, San Francisco, CA, United States. November 19-21, 2008. `https://www.crummy.com/writing/speaking/2008-QCon/act3.html`.

[Ric09] M. M. Richter. The search for knowledge, contexts, and Case-Based Reasoning. *Engineering Applications of Artificial Intelligence*, 22(1):3–9, 2009.

[Ril21] G. Riley. *CLIPS Reference Manual: Basic Programming Guide*, volume 1. `https://www.clipsrules.net/`, 6.40th edition, 2021.

[Ril22] G. Riley. *Adventures in Rule-Based Programming: A CLIPS Tutorial.* Secret Society Software, LLC, 2022. ISBN-13:979-8985783919.

[RPS03] R. Ravi, YVRK Prasad, and VVS Sarma. Development of expert systems for the design of a hot-forging process based on material workability. *Journal of Materials Engineering and Performance*, 12(6):646–652, 2003.

[RRG+21] N. Reinisch, F. Rudolph, S. Günther, D. Bailly, and G. Hirt. Successful Pass Schedule Design in Open-Die Forging Using Double Deep Q-Learning. *Processes*, 9(7):1084, 2021.

[RW13] M. M. Richter and R. O. Weber. *Case-Based Reasoning: A Textbook*. Springer Berlin Heidelberg, 2013.

[Sch83] R. C. Schank. *Dynamic memory: A theory of reminding and learning in computers and people*. Cambridge University Press, 1983.

[Sla91] S. Slade. Case-based reasoning: A research paradigm. *AI magazine*, 12(1):42–42, 1991.

[Smy19] B. Smyth. A Tale of Two Communities: An Analysis of Three Decades of Case-Based Reasoning Research. In *Case-Based Reasoning Research and Development (ICCBR 2019)*, Otzenhausen, Germany, 2019. pp. 343–357.

[SRW+10] S. Sheikhi, R. Rech, F-J Wahlers, D. Bokelmann, and C-D Wuppermann. Fortschritte beim Freiformschmieden in den letzten 25 Jahren. *Stahl und Eisen: Zeitschrift für die Herstellung und Verarbeitung von Eisen und Stahl*, 130(1):22–38, 2010.

[sSK07] Y. seo, D. Sheen, and T. Kim. Block assembly planning in shipbuilding using case-based reasoning. *Expert Systems with Applications*, 32(1):245–253, 2007.

[SWA+21] J. M. Schoenborn, R. O. Weber, D. W. Aha, J. Cassens, and K. Althoff. Explainable Case-based Reasoning: A Survey. In *AAAI-21 Workshop Proceedings*, 2021.

[UMKK17] M. Umeda, Y. Mure, K. Katamine, and K. Kawahigashi. General Step Reduction Method for Knowledge-Based Process Planning of Non-Axisymmetrical Forged Products. Cambridge, United Kingdom, 2017. pp. 448–453.

[UMKM21] M. Umeda, Y. Mure, K. Katamine, and K. Matsunaga. General Step Reduction and Enlargement Method for Knowledge-Based Process Planning of Totally Non-axisymmetric Forged Products with Blanking and Punching. virtual event, 2021. Springer International Publishing, pp. 1161–1171.

[Viz19] P. Vizureanu. Introductory Chapter: Enhanced Expert System - A Long-Life Solution. In P. Vizureanu, editor, *Enhanced Expert Systems*. IntechOpen, London, 2019.

[WRRH19] M. Wolfgarten, D. Rosenstock, F. Rudolph, and G. Hirt. New approach for the optimization of pass-schedules in open-die forging. *International Journal of Material Forming*, 12(6):973–983, 2019.

[YG10] C. Yan and F. Gao. Simulation of programmed incremental open-die forging processes based on spread coefficient and geometric transformation. *Proceedings of the Institution of Mechanical Engineers, Part E: Journal of Process Mechanical Engineering*, 224(2):129–135, 2010.