Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Formal specification of the semantics of grasping using the Web Ontology Language (OWL) and integration into a robot programming GUI

Benjamin Aaron Degenhart

Computational Science and Engineering
(International Master's Program)

Technische Universität München

Master's Thesis

Formal specification of the semantics of grasping using the Web Ontology Language (OWL) and integration into a robot programming GUI

Author:	Benjamin Aaron Degenhart
Examiner:	Prof. Dr.-Ing. habil. Alois Christian Knoll
Assistant advisor:	Alexander Perzylo, fortiss GmbH
Submission Date:	March 15, 2019

I hereby declare that this thesis is entirely the result of my own work except where other-wise indicated. I have only used the resources given in the list of references.

March 15, 2019 Benjamin Aaron Degenhart

Acknowledgments

I want to thank my advisor Alexander Perzylo from fortiss for his competent, helpful and friendly guidance throughout my entire Master's Thesis process.
Also I want to thank Stefan Profanter and Ingmar Kessler, both from fortiss as well, for offering their expertise on certain topics.
Futhermore, thanks to Anabele-Linda Pardi for proofreading and giving general advice.

Abstract

In this thesis the semantics of grasping get developed first conceptually, then modelled ontologically and finally implemented in fortiss' Robot Instruction Framework. The advantage of doing this on a symbolic level compared to classical grasp planning approaches is that it saves a lot of computational effort and allows semantically meaningful reasoning about grasp modes. By logically guaranteeing incompabilities, the remaining search space for subsymbolic methods shrinks considerably. The basic idea is to compare the grasping capabilities of a gripper with the geometric conditions of the object to grasp, in order to find matching candidates - the applicable grasp modes. Basic sanity checks like values within ranges are performed and feasiblity scores per grasp mode allow ranking the resulting list of matches.

Legend

If not stated otherwise, all figures and screenshots were created by me.

Variable names or file extension are displayed as `such`.

Bold and *italic* will occasionally be used to highlight keywords.

In the digital version of this document, URLs in footnotes and references to other sections of the document are active links.

Contents

Acknowledgements vii

Abstract ix

1 Introduction 1
- 1.1 Motivation and Goal 1
- 1.2 Scope Of Tasks And Structure Of This Thesis 2

2 Related Work 3

3 Theory 5
- 3.1 Theoretical Basis 5
- 3.2 What Are Ontologies 5
- 3.3 Inference Enriches The Ontology And Yields Insights 6
- 3.4 Existing Ontologies From fortiss 7
  - 3.4.1 OntoBREP: Reasoning About CAD Data 7
  - 3.4.2 Semantic Process Description 9

4 Developing The Semantics Of Grasping 11
- 4.1 Starting With A Taxonomy 11
- 4.2 The Concept To Find Valid Grasp Modes 11
- 4.3 What The Gripper Offers: Grasping Capabilities 12
- 4.4 What The Object Offers: Geometric Conditions 13
- 4.5 Getting Applicable Grasp Modes 14

5 Implementation 19
- 5.1 Part 1: Work On The Knowledge Base 19
  - 5.1.1 Additions To The Existing Ontologies 21
  - 5.1.2 SPARQL Queries Working With The Modelled Ontologies 25
  - 5.1.3 GraphDB Plugin To Compute Overlapping Area 34
- 5.2 Part 2: Integration Into The Existing Software 37
  - 5.2.1 fortiss' Robot Instruction Framework 37
  - 5.2.2 Parsing OntoBREP In The GUI 38
  - 5.2.3 Two New GUI Elements For The Semantics Of Grasping 39

6 Discussion, Conclusion And Future Work 49

Appendix 55

Bibliography 63

1 Introduction

The Robotics division at fortiss has been involved in the project SMErobotics[1] from 2012 to 2016 together with 24 other institutions, companies and universities. It is funded by the European Unions Seventh Framework Program under grant agreement 287787[2], under the full title of “the European Robotics Initiative for Strengthening the Competitiveness of SMEs in Manufacturing by integrating aspects of cognitive systems”.

The main motivation is to make the use of robots more affordable for small and medium-sized enterprises (SMEs), see the project website or this article [13] for more information, including research on the considerable economic potential of this technology and how this approach can substantially reduce the level of robotics expertise required. A big part of fortiss’ contribution is to work on a semantic description language (ontology) for robot-based industrial automation. This research continues throughout other projects with multiple industry- and research partners, currently in the project Data Backbone[3] (from Jan 2018 to Mar 2020) which is about “the data infrastructure for continuous production without system interruptions” and is funded by the Bavarian Ministry of Economic Affairs and Media, Energy and Technology with project support by Zentrum Digitalisierung.Bayern. Among other proof of concept implementations, fortiss developed a software with a web-based graphical user interface (GUI) that allows programming robots via an intuitive interface that uses semantic reasoning in the background. The Software can either simulate an assembly process or control a real robot to do so, after a user created the process. Details of its implementation can be found in section 5.2.1. The focus is on allowing non-experts access to robot programming by means of intuitive interfaces.

1.1 Motivation and Goal

The aspect of this software system that I worked with in the context of this thesis, is the grasp planning. In the demo-workflows (using the above mentioned GUI) that were developed so far, all of the robot grasping actions were hardcoded. This was because more central parts of the system were prioritized when it came to developing semantics and reasoning capabilities for them. Hardcoded in this case means that predefined numerical values for the grasping poses (relative coordinates and angles) are prepared for all possible assembly-tasks of objects within a small demo-set of objects that exist in both physical form as well as in virtual form in a workcell that an assembly process gets deployed to.

[1] http://www.smerobotics.org/
[2] https://cordis.europa.eu/project/rcn/101283/factsheet/en
[3] https://www.fortiss.org/en/research/projects/data-backbone/

However, all the spatial, geometrical (see section 3.4.1) and procedural information (see section 3.4.2) required to define grasping procedures already resides in the ontology. Therefore, it makes perfect sense to keep expanding the "vocabulary" that this semantic description language can "speak". One way to achieve this is by developing the semantics of grasping.

Therefore is the goal of this thesis, to specify the semantics of grasping, to build them into fortiss' ontology and to develop an addition to the GUI in the form of a grasp planning tool that integrates into the existing assembly-process workflow. As a result of this work no more hardcoded values will be necessary and a generic way to handle grasp planning is enabled.

This work builds on the research fortiss has done over the past few years in collaboration with Prof. Knoll from the Technical University of Munich about using ontologies in robot-supported assembly.

1.2 Scope Of Tasks And Structure Of This Thesis

My task was to develop a formal specification of the semantics of grasping using the Web Ontology Language (OWL) and as proof of concept, integrate that into fortiss' robot programming GUI.

The next chapter 2 will give some insights into related work regarding grasping in the context of cognitive robotics. Then, chapter 3 will introduce the theoretical building blocks necessary for understanding the next part. That is, in chapter 4, the development of the semantics of grasping derived from a grasping taxonomy of industrial robots. In the first part of chapter 5, these semantics will be expressed through ontological modelling to augment the existing ontologies. The second part of chapter 5 is then to integrate the semantics of grasping into the existing software system at fortiss and to develop new elements for the workflow in the GUI that allow users to make use of the new functionality.

The repeating pattern throughout the chapters will be to look at what the gripper has to offer, what the object has to offer, how the two come together to form actionable grasp modes and then to check this for valid values and optionally rank them by feasibility. This pattern stays the same throughout the phases of conceptual explanation, ontological modelling, required queries and implementation.

2 Related Work

In the field of human grasping, there exists extensive research. Probably the most notable one is the Grasp Taxonomy by Feix et al., 2015 [6]. It had the goal to determine the largest set of different grasp types that can be found in literature. Overall, 33 different grasp types were identified and arranged into the "GRASP" taxonomy. A comparison sheet can be found on the project website[1]. They were building on their own work from 2009 [7] and, among others, they were exploring Napier's work from 1956 [11]. Which was an early attempt to classify human grasps by their needs for precision or power. Prof. Kragic, who was also involved in the GRASP taxonomy, worked together with others on task-based robot grasp planning: [20]. They developed a probabilistic framework for the representation and modelling of robot-grasping tasks. This is the contrary direction though that is being taken in this thesis - the driving principle here is to cover as much as possible on a purely symbolic level by means of ontological reasoning.
Regarding ontologies for robotics, the "IEEE Standard Ontology for Robotics and Automation" by Schlenoff et al., 2012 [18] is a crucial contribution to the field. It came out of a working group who's goal it was, "to develop a standard ontology and associated methodology for knowledge representation and reasoning in robotics and automation, together with the representation of concepts in an initial set of application domains". [8] then builds on this Core Ontology for Robotics and Automation (CORA) and introduces a set of ontologies that complement it with notions such as industrial design and positioning. Furthermore it contained some updates to CORA aiming to improve the ontologically representations of autonomy and of robot parts.
Another notable development is "cloud robotics", an emerging field in the intersection of robotics and cloud computing. It gives robots access to greater processing power and storage capacity than they could host themselves. Goldberg and Kehoe published a survey of related work in 2013 [9]. Beetz et al. introduced openEASE, "a web-based knowledge service providing robot and human activity data"[2] in 2016 [2]. Later they have shown use cases for it, namely [3] talks about using cloud robotics to provide a computationally-expensive "mental simulation service" that robotic agents can use to predict consequences of their actions right before the execution of everyday manipulation tasks. Later, in [4] they develop the "openEASE robot interface" and a framework for robots to reason on a central cloud application that has encyclopedic knowledge in the form of ontologies as well as execution logs from other robots. Their use case was an exchange of knowledge between different environments regarding the opening trajectories of a fridge using episodic memories.
The "Robotics and Semantic Systems group"[3] around Prof. Malec at Lund University is working on applying semantic techniques in robotics and automation, in order to facilitate

[1] http://grasp.xief.net/
[2] http://www.open-ease.org/
[3] http://rss.cs.lth.se/

high-level interfacing and for improving human-robot interaction. They call the combination of semantic technologies with industrial automation "semantic systems". In [12] Malec et al. present "a knowledge integration framework for robotics" with the goal to to represent, store, adapt, and distribute knowledge across engineering platforms. They make use of the Automation Markup Language AutomationML[4] as well as RDF triples and SPARQL endpoints. In [21] Malec et al. present an approach to support semantic capture during kinesthetic teaching (also called "manual guidance") of collaborative industrial robots. The system captures the users utterances, or typings, during the task demonstration. In that way, the user can create skills and reference systems from scratch by extending the vocabulary of actions and objects right as they are used in context. This approach lets users bootstrap system knowledge and is a promising answer to the fact that most industrial robot systems being deployed today, contain no domain knowledge and as such can't aid their operators in setup and defining use cases. In terms of robots learning from demonstrations, Argall et al. [1] did a comprehensive survey in 2009 of robot Learning from Demonstration (LfD).
Another take on the topic of teaching industrial robots is explained in [17], coming from the chair of Cognitive Systems, led by Prof. Cheng at the Technical University of Munich. Among other things, they work on a novel semantic-based method to obtain general recognition models through teaching by demonstration. Their latest journal article [5] on the topic is done together with Prof. Beetz who's work is cited above. The term "purposive learning" is used to describe the notion of robots to learn from observation. The semantic-based techniques they developed, focus on extracting meaningful intentions from human behaviors and secondly on the ability to transfer past experiences into new domains.

[4]https://www.automationml.org/

3 Theory

3.1 Theoretical Basis

Building systems that allow the programming of robots with much less expertise requires shifting considerable parts of this expertise towards the system. Things that are obvious enough to be automatically derived should not bother the operator. Anomalies, errors and the need to handle uncertainties are inevitable, yet many categories can be accounted for automatically if the system has an "understanding" of what is going on.
To achieve this level of understanding, a programming paradigm is necessary in which common-sense knowledge as well as the robotics- and domain-specific knowledge is modelled explicitly using semantics via ontologies. In this way the system can understand, interpret and reason about itself. The semantic description language developed by fortiss is based on the Web Ontology Language[1] (OWL), which is a natural choice, as many ontologies have already been developed as OWL-ontologies. The QUDT ontology[2] for instance standardizes data types and units and is one of many generic ontolgoies about common-sense knowledge. Even though there exist basic ontologies for the robotics domain [18], the requirements of SMEs and their specific domains are not adequately considered. Therefore fortiss chose to augment base ontologies with domain-specific knowledge in order to create cognitive robotic systems that are specialized for a domain.

3.2 What Are Ontologies

The term ontology is used in philosophy to describe the study of being:

> *"It studies concepts that directly relate to being, in particular becoming, existence, reality, as well as the basic categories of being and their relations. [...] ontology often deals with questions concerning what entities exist or may be said to exist and how such entities may be grouped, related within a hierarchy, and subdivided according to similarities and differences."*[3]

It's meaning in computer- and information science is similar:

> *"An ontology encompasses a representation, formal naming, and definition of the categories, properties, and relations between the concepts, data, and entities that substantiate one, many, or all domains. Every field creates ontologies to limit complexity*

[1] https://www.w3.org/TR/owl2-primer/
[2] http://www.qudt.org/
[3] https://en.wikipedia.org/wiki/Ontology

> *and organize information into data and knowledge. As new ontologies are made, their use hopefully improves problem solving within that domain."*[4]

There are many terms to get familiar with when working with ontologies. Firstly, it is a general concept and ontologies could just as well be written with pen and paper. However, once a formal language is used to describe an ontology such as OWL 2[5] in our case, it becomes very useful. OWL is an extension of RDFS, which in itself builds on top of RDF, the Resource Description Framework. RDF is organized in triples, each RDF triple has a subject, a predicate and an object. The predicate is also called property sometimes and connects the subject and the object. When modelling an ontology, one creates classes within a hierarchy of classes. `Thing` is the root-class of all others. From a class, arbitrarily many individuals can be instantiated. These classes need to have concrete values for object- and data properties that their parent class restricted them to. Object properties link to other individuals whereas data properties link to literals like a string or a double value. Properties are also organized in a hierarchy and can have characters like Symmetric or Functional. They can also be declared as the inverse of other properties. An example from the OntoBREP ontology (more in section 3.4.1) is the object property `represents` that is the inverse of `representedBy`. So when a face is representedBy a wire, automatically, without explicitly stating so, the wire represents the face. And this relation can be actively used when doing inference, via SPARQL queries (more in section 5.1.2) for instance.

Ontologies can be perfectly stored in simple text-files, usually with the extension `.owl`. However, in most use cases, they need to be "alive" in a production environment. A database needs to store them and answer to queries.
In general it can be said that the more knowledge is present in a cyber-physical system (CPS) in the form of ontologies, the more synergies arise from the the possibility of automatically linking and combining knowledge sources through a language based on a logical formalism. Among multiple use cases, this can be utilized for developing intuitive interfaces for end-users who can be relieved of having to know all technical and programmatic details and rely on using the semantic language tailored to the task. The complexity behind simple instructions is being handled by the reasoning system in connection to the knowledge base. Instructions are being multiplied out to be used by the concrete hardware, all variables get parameterised and additional statements get automatically inferred if applicable.

3.3 Inference Enriches The Ontology And Yields Insights

Because OWL 2 is based on a logical formalism, inferring becomes possible.

> *"Inference is the derivation of new knowledge from existing knowledge and axioms. In an RDF database, such as GraphDB, inference is used for deducing further knowledge based on existing RDF data and a formal set of inference rules."*[6]

[4]https://en.wikipedia.org/wiki/Ontology_(information_science)
[5]https://www.w3.org/TR/owl2-overview/
[6]http://graphdb.ontotext.com/free/devhub/inference.html

In the context of this work, the default ruleset `RDFS-Plus (Optimized)` was used. An option would have been to extend the `OWL2-RL`-one with custom rules in a `.pie`-file.
A reasoner is the tool used for inference. In Protégé, see section 5.1, "Pellet" and "Hermit" are usually used whereas GraphDB, see section 5.1, has its own.
Reasoners are good at "understanding" transitive properties across large graphs of connections. The simplest example of a transitive property is this: if $a > b$ and $b > c$ then it can be inferred that $a > c$.

Categories Of Computation: Symbolic vs. Subsymbolic

As will be seen later, the distinction between symbolic and subsymbolic levels is crucial in categorizing types of computations. It means drawing a line between logical, symbolic operations that require no arithmethic and all other mathematical operations where numerical values are involved. When working with ontologies this distinction nicely describes everything that can happen "cleanly" by reasoning in the ontology in comparison to more computationally intensive operations involving "number crunching". The classical grasp planning for instance involves random sampling of millions of options to find useful ones. In contrast to that, a reasoning setup like described in this thesis, can find a useful set of options with very little computational effort in comparison. However, even then there will be a last step of finding the concrete pose for a gripper that has to be computed subsymbolically. So there is a balance to be found between these two domains.

3.4 Existing Ontologies From fortiss

fortiss has developed many ontologies so far for various projects and purposes. Two will be highlighted in the following as my work builds on them.

3.4.1 OntoBREP: Reasoning About CAD Data

A prerequisite to reason about geometrical objects is to describe spatial information semantically. Perzylo et al., 2015 [15] present an approach for leveraging Computer-aided design (CAD) data to a semantic level by using OWL to define boundary representations (BREP) of objects. In this way, each object is defined as a compound of topological entities which are represented by geometric entities. This ontology, OntoBREP, is available online[7]. The advantage over polygon-based geometry models is that the exact mathematical representation are stored. From these representations, approximations (like meshing the geometry using triangulation) can be generated at the level of detail most suitable to an application's requirements. In [15] it is stated that:

> *This separation of represented data and use-case dependent calculations is as important as the separation of data and the software that was used to generate it. These paradigms allow flexible sharing and re-use of information.*

[7] https://github.com/OntoBREP

From the perspective of Product Process Resource (PPR)-based modelling, OntoBREP can be seen as an "enabling technology" that serves both the product-side (both for direct description of CAD objects as well as basis for further object models) as well as the resource-side by describing the specifics of resources like the fingers of a gripper involved in a grasping capability for instance.

Figure 3.1 shows an overview of the BREP structure as modelled in the OntoBREP-ontology. Listing 1 in the appendix shows excerpts of a `cube.owl` file in OntoBREP format (which is expressed in XML). How to use this format to retrieve information and infer insights will become clearer in chapter 4.

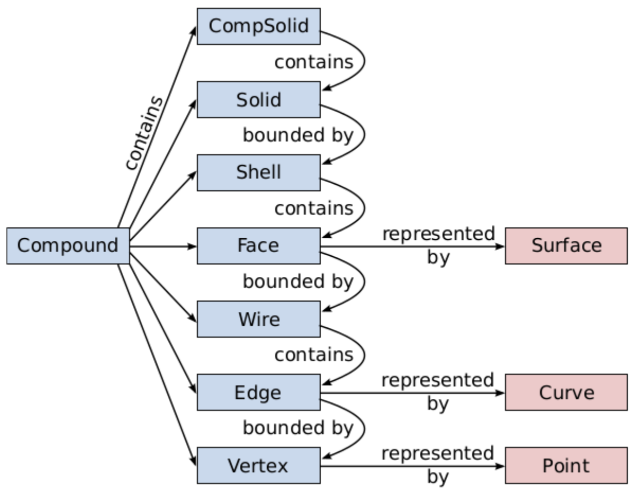
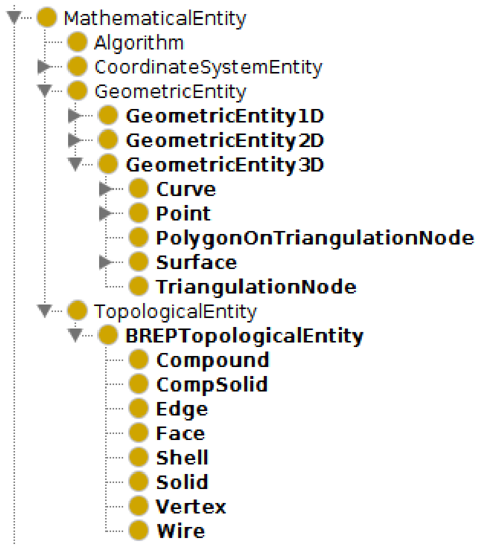


Figure taken from [15]
Topological entities in blue, geometric entities in red



- MathematicalEntity
  - Algorithm
  - CoordinateSystemEntity
  - GeometricEntity
    - **GeometricEntity1D**
    - **GeometricEntity2D**
    - **GeometricEntity3D**
      - **Curve**
      - **Point**
      - **PolygonOnTriangulationNode**
      - **Surface**
      - **TriangulationNode**
  - TopologicalEntity
    - **BREPTopologicalEntity**
      - **Compound**
      - **CompSolid**
      - **Edge**
      - **Face**
      - **Shell**
      - **Solid**
      - **Vertex**
      - **Wire**

Class hierarchy in Protégé

Figure 3.1

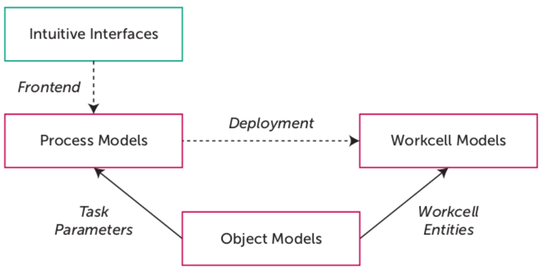


Figure taken from [14]: overview of semantic models and interrelations

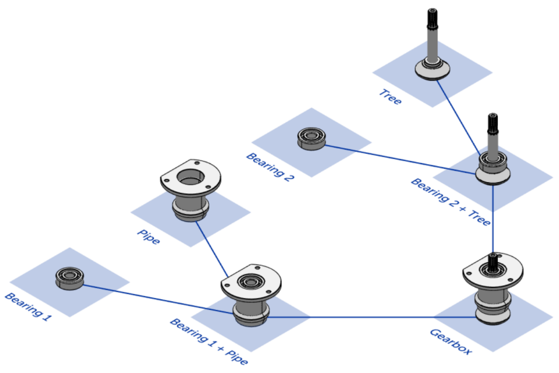


Figure taken from [14]: industrial use case of assembling a gearbox in three steps

Figure 3.2

3.4.2 Semantic Process Description

Perzylo et al., 2016 [14] introduced a novel robot programming paradigm, namely semantic process descriptions for small lot production. The programming happens at object level, abstracted away from raw coordinates and basic commands. Tasks get only parameterized, and possibly automatically augmented with further inferred tasks, when being deployed on a concrete workcell. Constraints between geometric entities as well as individual assembly steps get captured on an ontological level. This reduces the human expertise required to define production processes dramatically. Statistics on estimated time-savings by using this method instead of traditional programming can be found in [13].
Figure 3.2 shows two illustrations taken from [14]. The workflow for the user in the frontend will be showcased in section 5.2.1.1.

4 Developing The Semantics Of Grasping

In this chapter, the development of the semantics of grasping will be explored from its starting point as taxonomy to the conceptual building blocks necessary to semantically fully describe grasp modes that a gripper can perform on an object.

4.1 Starting With A Taxonomy

The difference between taxonomy and ontology is, that taxonomies are more similar to enriched classifications and mainly serve as information to humans. An ontology however has more information about the behavior of the entities and the relationships between them. Furthermore it can be processed by machines to do operations like checking for logical consistency using a semantic reasoner and run inference for drawing automatic conclusions.

To get an idea about gripper types on the market, their grasp modes and relevant parameters, research into industrial grasping was conducted via the product specifications of various manufacturers, see figure 4.1. The purpose was not to condense this into a conclusive and comprehensive taxonomy, but merely to inform a meaningful approach to developing a grasping ontology in the next step.

4.2 The Concept To Find Valid Grasp Modes

From this taxonomy-overview, the overall concept took form. It become clear that the gripper and the object both have to "offer" something to end up with a list of options that are ideally even ranked by usefulness. The solution is to semantically enrich both parties in such a way, that matches can be found between them on a symbolic level. From looking at the grasping taxonomy, it becomes clear that a gripper can be described as having capabilities when it comes to grasping. A parallel gripper for instance has the ability to grab an object by approaching it with its two fingers in parallel from outside. However, depening on the shape of the fingers, the same gripper could potentially also grab objects from inside if there is space to insert the closed fingers and open them until they clamp the object from inside. This gripper can then be said to offer two "grasping capabilities". But how does the gripper know, if one or more of its capabilities can be used on a given object? That's where the offer from the object comes in. To be properly grasped in parallel from outside, the object ideally has two parallel planar faces that the gripper can close in on. These "features" from the object can be called "geometric conditions". An object can have arbitrarily many of them. These two concepts can now be brought together by a simple "requires" relation. A grasping capability from the gripper requires a certain geometric

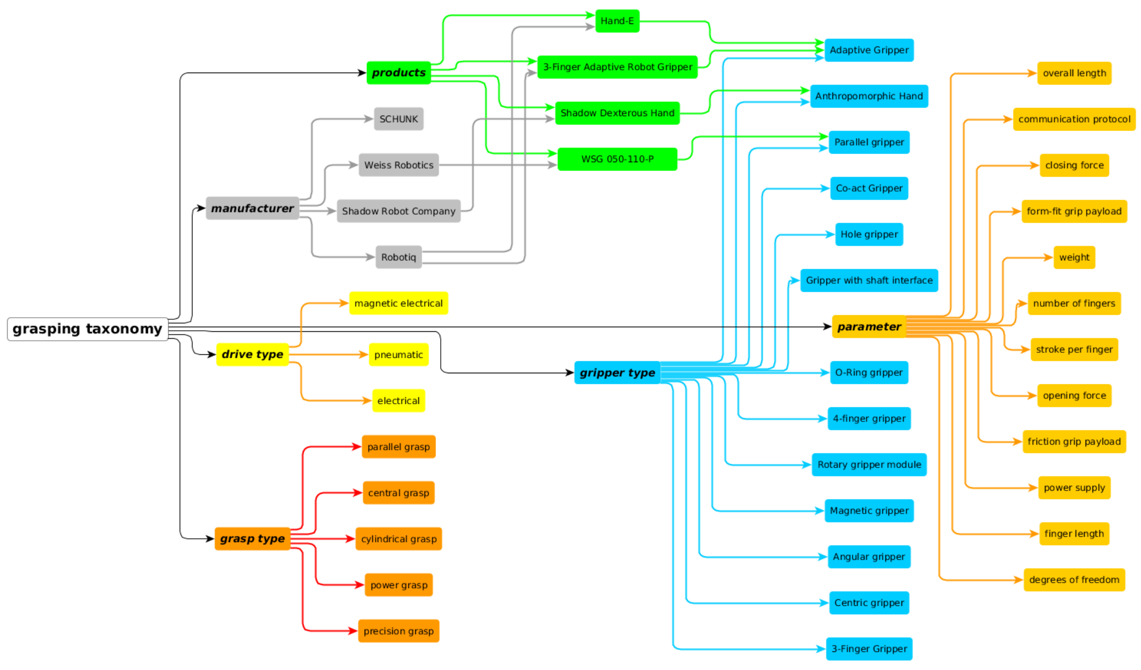


Figure 4.1: Sketch of a robotic grasping taxonomy

condition to be present in the object, otherwise it can not be performed.

This is the concept: given a gripper annotated with its grasping capabilities (i.e. parallel from outside) and given an object that is annotated with it's geometric conditions (i.e. two parallel planes facing outwards), matches can be deduced where each represent a possible grasp mode. Note that throughout this thesis, both grasping capability as well as grasp mode will be used. The first is meant more as the passive capability of the gripper whereas the second is the result of a match with an objects geometric condition and can be actively acted upon.

Figure 4.2 illustrates this paradigmatically. The conceptual elements will be discussed in more detail throughout this chapter. It's implementation will be presented in the following chapter 5.

4.3 What The Gripper Offers: Grasping Capabilities

We decided to narrow down the development of grasping semantics in the context of this thesis to a small set of 5 meaningful grasp types and parameters. This can be taken as basis for extending it further, see section 6. The parameters were restricted to those directly relating to a grasp mode. General ones like the maximum payload were ignored. The following five grasp modes were chosen, including the relevant parameters from the gripper's side.

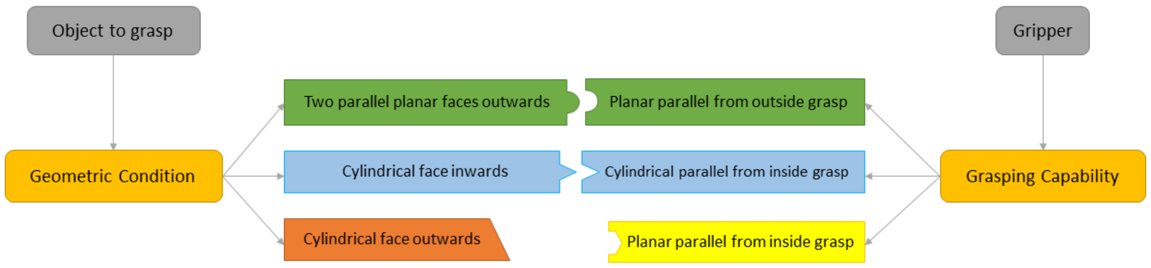

Object to grasp
Gripper
Two parallel planar faces outwards
Planar parallel from outside grasp
Geometric Condition
Cylindrical face inwards
Cylindrical parallel from inside grasp
Grasping Capability
Cylindrical face outwards
Planar parallel from inside grasp

Figure 4.2: Examples of matches

- **Planar parallel from inside grasp**
  → the range fitting between the planar outer surfaces of the grippers fingers
- **Planar parallel from outside grasp**
  → the range fitting between the planar inner surfaces of the grippers fingers
- **Cylindrical parallel from inside grasp**
  → the range the grippers cylindrical halves can get in and then span up
- **V-shaped notch from outside grasp**
  → the range of radii this grippers notch-halves can close in on
- **Planar surface vacuum suction grasp**
  → the surface area of the suction cup when engaged

With “V-shaped notch” is meant that it’s not ellipses cut out on the grippers finger that would only allow a form-fitting grasp. Instead it is two V-shaped surfaces that allow a range of cylinder radii to fit in. The gripper used in the implementation part, 3D-printed by fortiss, has this feature, that’s why it is included here. More about that in section 5.1.1.

Figure 4.3 shows three of these grasp modes in action. The screenshots are taken in simulation-mode in the GUI-frontend of fortiss’ Robot Instruction Framework, which will be presented in more detail in section 5.2.1.
Now that grasp modes are defined in this way, a strategy is needed about how to assess the object that is to be grasped. How can the system filter for valid grasp modes when confronted with a specific object? Our answer to this is to annotate objects with their “Geometric Conditions”.

4.4 What The Object Offers: Geometric Conditions

Geometric conditions can be thought of as features that an object exhibits. When humans are planning grasping actions, we scan the object for features that we can utilize. Mostly without giving it much conscious thought. The handle of a cup for instance calls for a certain way to hold the cup. Just like the cylindrical outwards facing surface of a glas calls for a certain way.

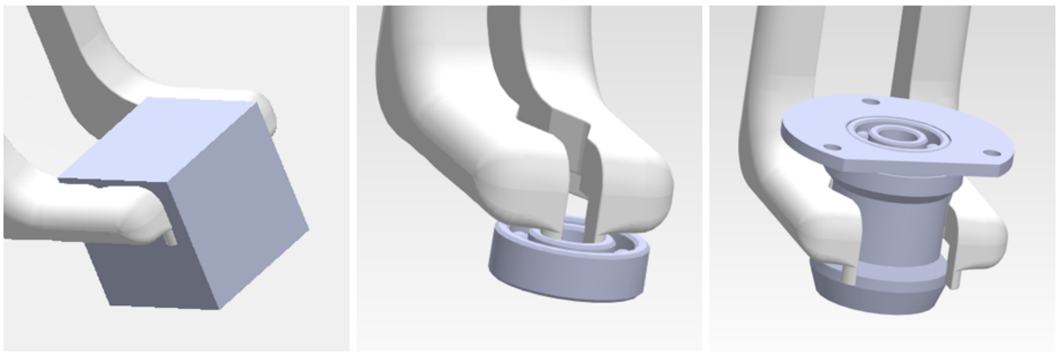


Figure 4.3: Examples of grasp modes

In the following list, the grasp modes as shown above in section 4.3 are extended by the types of geometric conditions and their parameters that objects need to possess to be matched with the respective grasping capability.

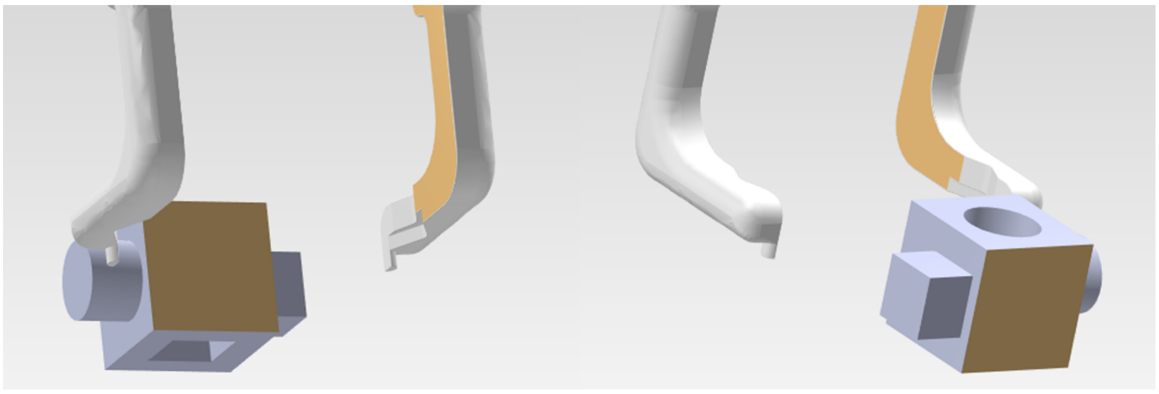
- Planar parallel from inside/outside grasp
  → **Two parallel planar faces inwards/outwards**: Two planar parallel faces must be present that are both oriented either inwards or outwards. The orthogonal *distance* between them is required for comparison with the grippers range. Also the *overlapping area* of the two planes (upon projection into the same plane) must be at least > 0 for a proper grasp to be applied.
- Cylindrical parallel from inside grasp / V-shaped notch from outside grasp
  → **Cylindrical face inwards/outwards**: An inward or outward facing cylindrical face must be present. Its *radius* is required for comparison to the grippers range.
- Planar surface vacuum suction grasp
  → **Planar face outwards**: A planar surface must be present. Its *surface area* must be big enough to place the grippers suction cup.

Another conceptual element that was introduced alongside the annotation of geometric conditions is the ability to flag faces as `noGrasp`. This concept means that the faces are not suitable for grasping. To stick with the analogy of human grasping, some faces of objects should better not be touched; like the hot body of a pan. Or maybe some fresh paint that is drying during the assembly.

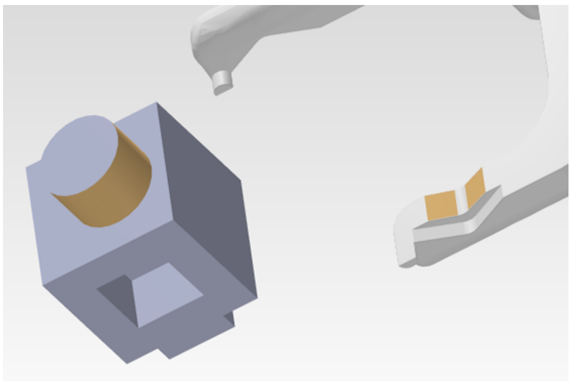
4.5 Getting Applicable Grasp Modes

Each grasping capability requires a specific geometric condition to be applicable. For more complicated abilities in the future, multiple required conditions will likely be necessary.

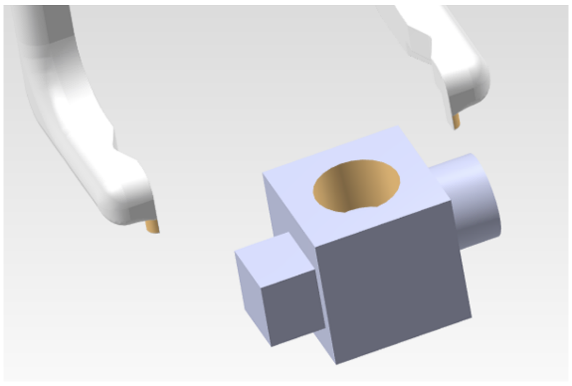
Figure 4.4 shows three matches. The screenshots are taken in the grasp planner, a new step added to the workflow in the GUI, described in detail in section 5.2.3.2.



Planar parallel from outside grasp matched with its required geometric condition of *two parallel planar faces outwards* (2 screenshots to show all 4 involved faces)



V-shaped notch from outside grasp matched with *Cylindrical face outwards*



Cylindrical parallel from inside grasp matched with *Cylindrical face inwards*

Figure 4.4: Examples of matches

If a match is found based on the geometric condition, a value check must be passed before counting as suitable match.

Value Checking To Filter Out Invalid Matches

As discussed in the previous sections 4.3 and 4.4, a grasping capability might have a range of valid values that it requires values of geometric condition to be in. All parallel grasp modes for instance have a minimum and a maximum defined by the gripper span. The distance between parallel planes or the radius of the cylindrical face to be matched with, has to fit inside this range. This first check is of boolean nature, matches that have `false` in any of the value checks, are disregarded going forward. The second round of checks however, is of subsymbolic nature.

Feasibility Scores To Rank Matches

The matches which passed the value check can now be inspected as to how good of a candidate they are. An object that fits right into the middle of a grippers span might be scoring highest, whereas one that requires an almost maximum opening scores lower. As another example, two parallel planes that have a high overlapping area are preferred over a pair with a low overlap as it would result in a weaker grasp. Both the reasons for scoring high or low as well as the numerical method to compute these scores will vary depending on the use case. One could think of a metric related to how much time a grasp mode costs in relation to an object cooling off in an assembly process, or a metric capturing how well the coefficients of friction of the surfaces participating in a grasp mode fit together. A basic metric would also be assigning lower scores as the object's mass comes close to the grippers maximum payload.
The modelling of this conceptual part must therefore be kept as generic as possible.

Is A Grasp Mode Actually Doable - Subsymbolic Postprocessing

This part of the concept was not achieved in the implementation part of this thesis.

After passing the value check 4.5 and getting a feasiblity score 4.5 assigned, a match consists of a theoretically possible grasp mode that the gripper can conduct on the object. However, orientation of objects within the workcell (the object might be resting on the surface one is trying to grasp it with), "things in the way" and factors like gravity or friction have not been taken into account yet and might very well render this grasp mode unpractical or even completely useless for a real execution.

Figure 4.5 shows a *Planar parallel from outside grasp* that matched correctly with its required geometric condition of *two parallel planar faces outwards*. However, this grasp mode would be impossible to execute.

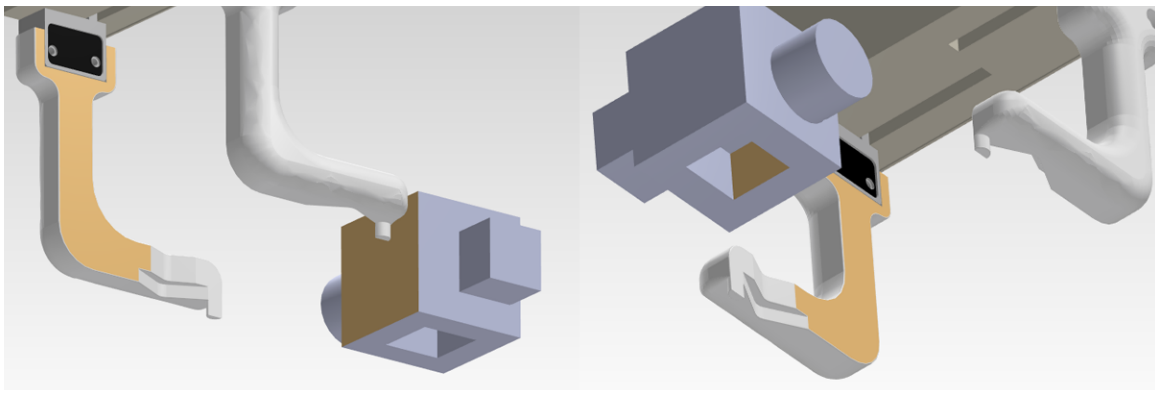


Figure 4.5: Example of a logically valid match that is physically impossible to execute (2 screenshots to show all 4 involved faces)

For a human these types of "mismatches" might be very obvious, but some of the spatial possibilities that make things unpractical are hard or impossible to capture on a logical level. It requires, ordered by increasing computational workload, subsymbolic postprocessing like constraint solving, collision checking and potentially even physics simulation. Only then can the list of matches be narrowed down to the physically possible ones.

5 Implementation

This chapter explains in detail the newly implemented developed semantics of grasping as presented in the previous chapter 4. The first part, section 5.1 covers the work done on the knowledge base, meaning the ontological embodiment of the developed concepts - plus the SPARQL queries operating on this ontology. The second part 5.2 is about integrating this into the frontend and develop two new GUI elements that users can use to plan grasp modes.

A focus of the frontend-implementations was, to keep things at a lightweight level in a sense that all the "intelligence" should stem from the knowledge base and SPARQL-queries that operate on it. The guiding principle here is to separate knowledge from program code as consequently as possible.

The approach to additions to the knowledge base is more generous. Here we encourage storing more then absolutely necessary. The reason being that it might well be useful in other reasoning situations later on and storing triples is comparatively "cheap". Section 5.1.2.1 covers some examples of this purposeful redundancy. GraphDB itself also follows this philosophy by immediately materializing inferred triples:

> *GraphDB supports inference out of the box and provides updates to inferred facts automatically. Facts change all the time and the amount of resources it would take to manually manage updates or rerun the inferencing process would be overwhelming without this capability. This results in improved query speed, data availability and accurate analysis.*[1]

5.1 Part 1: Work On The Knowledge Base

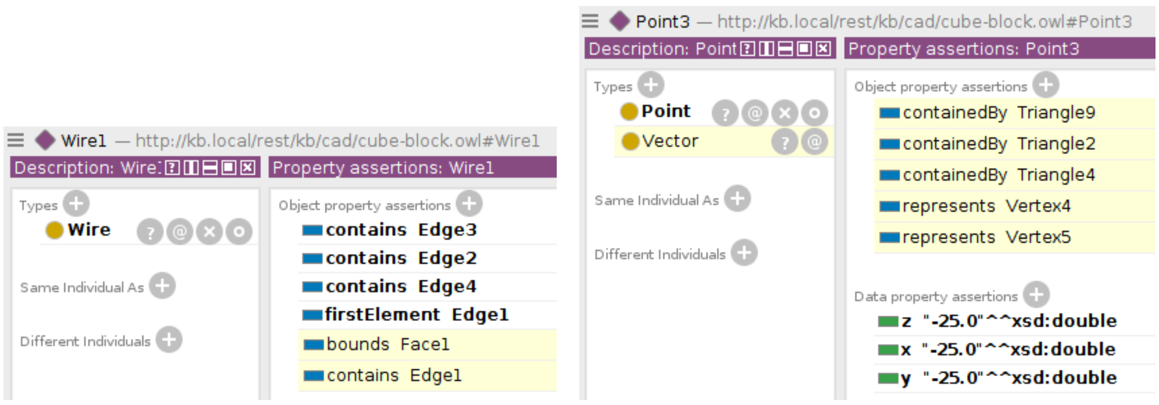
fortiss follows the Product Process Resource (PPR) modell and as such, their core ontologies cover these three areas. As tools for modelling, Protégé and GraphDB were used.

Modelling tool: Protégé

Protégé[2] was used as the tool for modelling. As an example for its various useful editing- and development features, figure 5.1 shows inferred object properties, in yellow background color, that show up once a reasoner is started, Pellet in this case.

[1] http://graphdb.ontotext.com/free/devhub/inference.html
[2] https://protege.stanford.edu/

Figure 5.1 (a) shows an individual of the `Wire` class. *Face1* is `boundedBy` *Wire1* - and because `bounds` is marked as inverse of `boundedBy`, this relation shows up here without being explicitly modelled. *Edge1* shows up twice because `firstElement` is a child property of `contains`. If one would ask for all edges contained by this wire (e.g. with a SPARQL expression as such: `Wire1 cad:contains ?edge .`), *Edge1* would therefore also show up in the results. However, only if the triple store hosting these triples, supports inference. Figure 5.1 (b) shows an individual of the `Point` class. This one contains no explicitly modelled object properties at all, just the three coordinates as data properties. However, since this point is referenced from triangles via `contains` and vertices via `representedBy`, these links show up inferred via their inverse object properties `containedBy` and `represents`. A query on a triple store that supports inference, could utilize these relations by querying "upwards starting from the point", instead of finding the way towards it by its parents.


Wire1 — http://kb.local/rest/kb/cad/cube-block.owl#Wire1

Description: Wire1 | Property assertions: Wire1

Types: Wire

Same Individual As

Different Individuals

Object property assertions
- contains Edge3
- contains Edge2
- contains Edge4
- firstElement Edge1
- bounds Face1
- contains Edge1

Point3 — http://kb.local/rest/kb/cad/cube-block.owl#Point3

Description: Point3 | Property assertions: Point3

Types: Point, Vector

Same Individual As

Different Individuals

Object property assertions
- containedBy Triangle9
- containedBy Triangle2
- containedBy Triangle4
- represents Vertex4
- represents Vertex5

Data property assertions
- z "-25.0"^^xsd:double
- x "-25.0"^^xsd:double
- y "-25.0"^^xsd:double

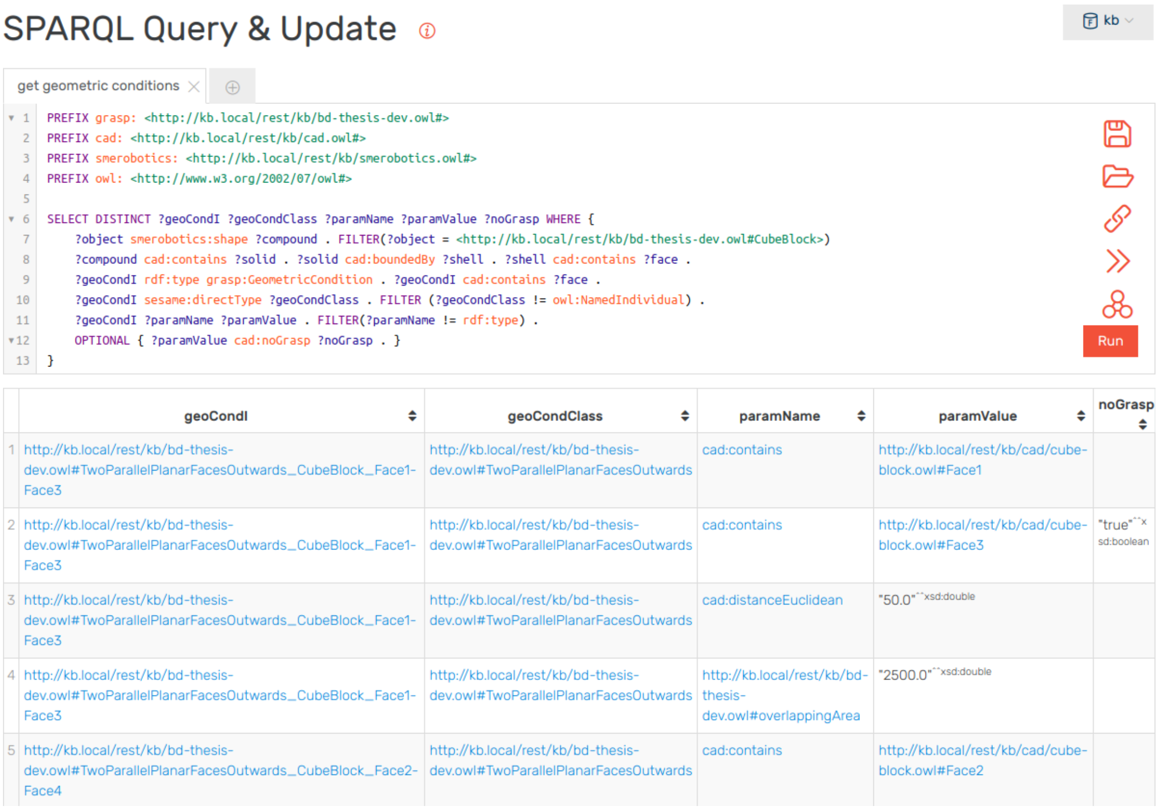
(a) A `Wire` individual (b) A `Point` individual

Figure 5.1: Reasoner inferring object properites in Protégé

Graph Database: GraphDB

GraphDB[3] is a semantic graph database, also called RDF triplestore. Since it is used in fortiss' robot instruction framework (see section 5.2.1), it made sense to use it all along for developing and testing SPARQL queries before embedding them in the frontend. Protégé also supports SPARQL queries, however, there are occasional differences in how the respective reasoning-engines interpret patterns. Figure 5.2 shows an exemplary query to get the geometric condition individuals attached to the object `CubeBlock`. This query can also be found in the appendix 5.
We had to use GraphDB version 8.8.0 or higher because only since then math functions are natively supported. More about that in section 5.1.2.
Figure 5.3 shows another feature of GraphDB's Workbench, the "Visual graph". It allows expanding nodes manually from a starting node. This can be useful to get an overview of how nodes are connected and if certain links have been established. In this example one can see parts of the OntoBREP graph of the object *CubeBlock*. Furthermore, a geometric

[3] `http://graphdb.ontotext.com/`



```
PREFIX grasp: <http://kb.local/rest/kb/bd-thesis-dev.owl#>
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>

SELECT DISTINCT ?geoCondI ?geoCondClass ?paramName ?paramValue ?noGrasp WHERE {
    ?object smerobotics:shape ?compound . FILTER(?object = <http://kb.local/rest/kb/bd-thesis-dev.owl#CubeBlock>)
    ?compound cad:contains ?solid . ?solid cad:boundedBy ?shell . ?shell cad:contains ?face .
    ?geoCondI rdf:type grasp:GeometricCondition . ?geoCondI cad:contains ?face .
    ?geoCondI sesame:directType ?geoCondClass . FILTER (?geoCondClass != owl:NamedIndividual) .
    ?geoCondI ?paramName ?paramValue . FILTER(?paramName != rdf:type) .
    OPTIONAL { ?paramValue cad:noGrasp ?noGrasp . }
}
```

	geoCondI	geoCondClass	paramName	paramValue	noGrasp
1	http://kb.local/rest/kb/bd-thesis-dev.owl#TwoParallelPlanarFacesOutwards_CubeBlock_Face1-Face3	http://kb.local/rest/kb/bd-thesis-dev.owl#TwoParallelPlanarFacesOutwards	cad:contains	http://kb.local/rest/kb/cad/cube-block.owl#Face1	
2	http://kb.local/rest/kb/bd-thesis-dev.owl#TwoParallelPlanarFacesOutwards_CubeBlock_Face1-Face3	http://kb.local/rest/kb/bd-thesis-dev.owl#TwoParallelPlanarFacesOutwards	cad:contains	http://kb.local/rest/kb/cad/cube-block.owl#Face3	"true"^^xsd:boolean
3	http://kb.local/rest/kb/bd-thesis-dev.owl#TwoParallelPlanarFacesOutwards_CubeBlock_Face1-Face3	http://kb.local/rest/kb/bd-thesis-dev.owl#TwoParallelPlanarFacesOutwards	cad:distanceEuclidean	"50.0"^^xsd:double	
4	http://kb.local/rest/kb/bd-thesis-dev.owl#TwoParallelPlanarFacesOutwards_CubeBlock_Face1-Face3	http://kb.local/rest/kb/bd-thesis-dev.owl#TwoParallelPlanarFacesOutwards	http://kb.local/rest/kb/bd-thesis-dev.owl#overlappingArea	"2500.0"^^xsd:double	
5	http://kb.local/rest/kb/bd-thesis-dev.owl#TwoParallelPlanarFacesOutwards_CubeBlock_Face2-Face4	http://kb.local/rest/kb/bd-thesis-dev.owl#TwoParallelPlanarFacesOutwards	cad:contains	http://kb.local/rest/kb/cad/cube-block.owl#Face2	

Figure 5.2: Screenshot of the SPARQL-tab in Workbench, GraphDB's web-based administration tool. Showing 5 out of 12 rows in the results-table. 12 because a cube has 3 pairs of planar parallel outward faces, each of which is represented by a geometric condition individual that has 4 parameters attached.

condition individual in green can be seen, linking two planar faces together that are parallel to each other and facing outwards. This example will be discussed further in section 5.1.2.1.

5.1.1 Additions To The Existing Ontologies

The implementation work is bundled in one OWL-file, even so some elements would better be located with the existing ontologies. However, during development this encapsulated one-file approach was useful as proof-of-concept. Later on, once decided where the pieces should go, they will likely disseminate into a new ontology dedicated to grasping and into existing ones.
This is a quick run-through of the modelling-concept before it is presented in more detail throughout the remainder of this chapter. Each `GraspingCapability` has the following:

- the object property `requiresGeometricCondition` links to a geometric condition class
- it `contains` one or more `FaceLists` which group together faces of a finger that are

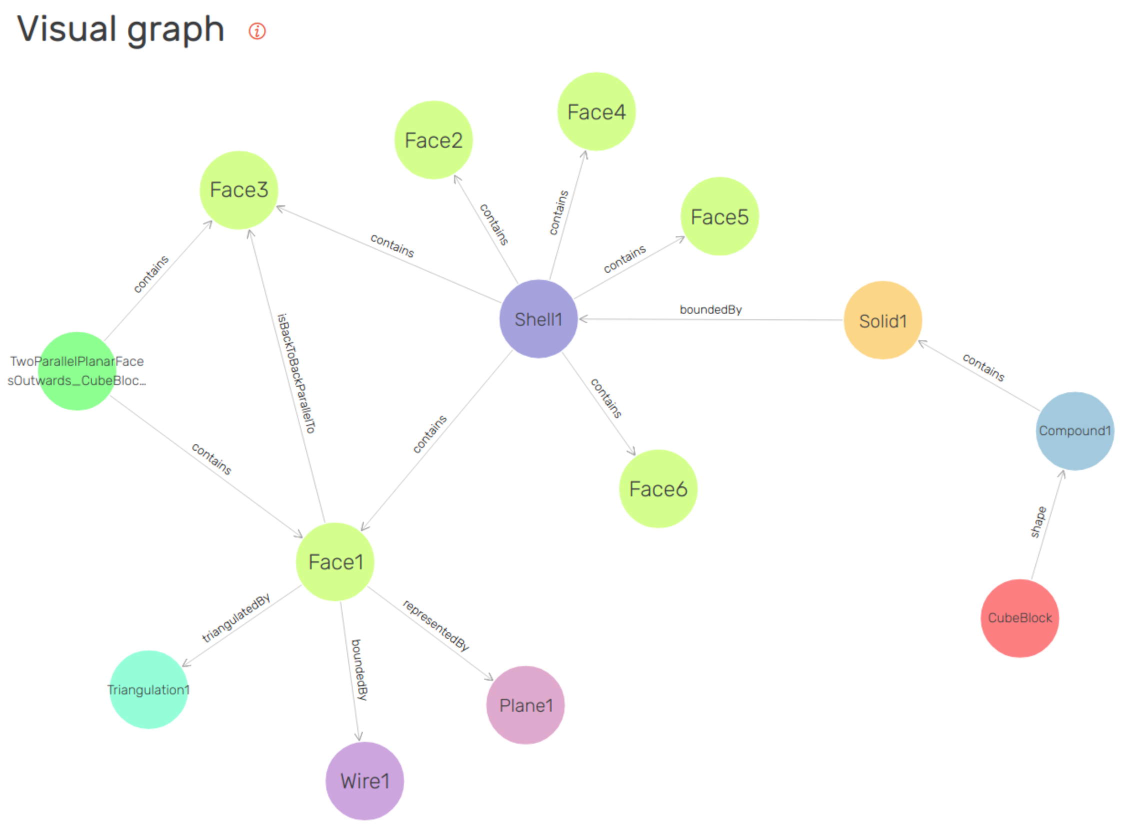


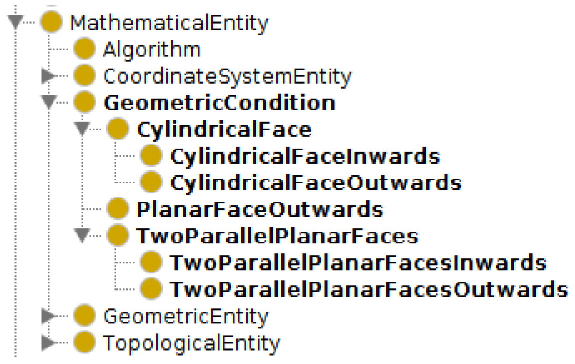
Figure 5.3: Screenshot of the Visual graph feature in GraphDB's Workbench

involved in a particular grasp

- an optional link `needsValueCheck` to members of the class `ValueCheck` - if present, this demands data property values of a geometric condition to be in a specific range for a match to be valid, e.g. if an object fits between the grippers finger
- a `ValueCheck` might further contain a link to a `FeasibilityCheck` that assigns a score to how well a value fits into a range, based on the given evaluation method

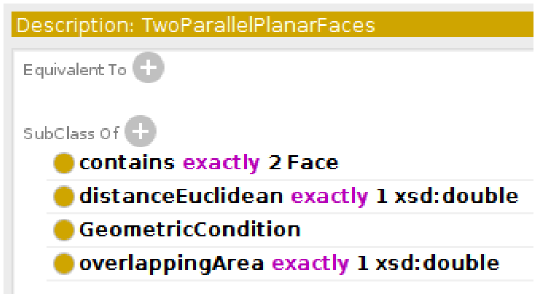
5.1.1.1 Modelling Geometric Conditions

Figure 5.4 shows the addition of `GeometricCondition` to the overall class hierarchy. Note that the example of an individual would normally not appear materialized on modelling level like that. It would be inserted within the graph database later on via queries from the frontend. It was added only to be shown here for explanatory purposes.
Note that having the `radius` as a property of the geometric condition individual seems redundant because that value is already attached to the `CylindricalSurface` that represents *Face8* in that case. One way to lift this information up to the level of the geometric condition for more convenient reasoning is via a property chain, see 3.3. However, in this case redundant information is acceptable as other geometric conditions have properties



```
MathematicalEntity
    Algorithm
    CoordinateSystemEntity
    GeometricCondition
        CylindricalFace
            CylindricalFaceInwards
            CylindricalFaceOutwards
        PlanarFaceOutwards
        TwoParallelPlanarFaces
            TwoParallelPlanarFacesInwards
            TwoParallelPlanarFacesOutwards
    GeometricEntity
    TopologicalEntity
```

Positioning in the class hierarchy

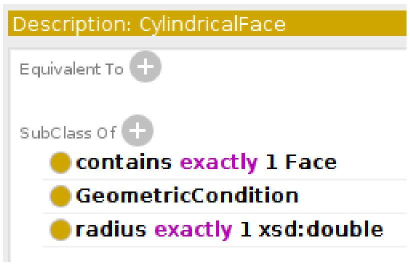


Description: TwoParallelPlanarFaces

Equivalent To

SubClass Of
- contains exactly 2 Face
- distanceEuclidean exactly 1 xsd:double
- GeometricCondition
- overlappingArea exactly 1 xsd:double

The TwoParallelPlanarFaces class

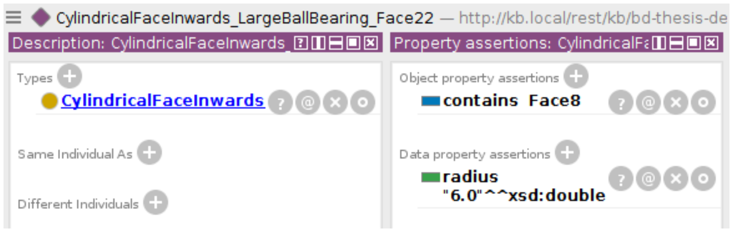


Description: CylindricalFace

Equivalent To

SubClass Of
- contains exactly 1 Face
- GeometricCondition
- radius exactly 1 xsd:double

The CylindricalFace class



CylindricalFaceInwards_LargeBallBearing_Face22 — http://kb.local/rest/kb/bd-thesis-de

Description: CylindricalFaceInwards_

Types
- CylindricalFaceInwards

Same Individual As

Different Individuals

Property assertions: CylindricalF

Object property assertions
- contains Face8

Data property assertions
- radius "6.0"^^xsd:double

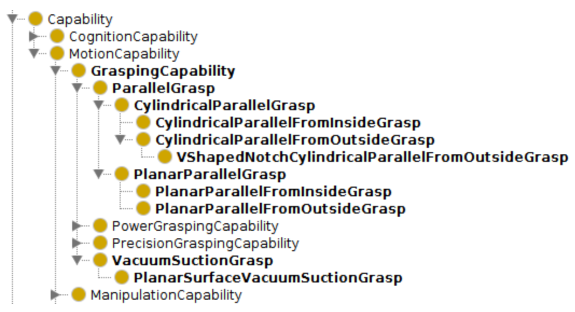
An individual of the CylindricalFaceInwards class

Figure 5.4: Ontological modelling of geometric conditions in Protégé

that are not directly copied from their faces, like distanceEuclidean between two parallel planes. And as "primitive shape matching" should be supported in the future (see 6), it could be the case that the radius is not retrievable as simple as looking two more steps further down - the surface could be very complicated and the radius is the one of the primitive cylindrical shape that was matched into it.

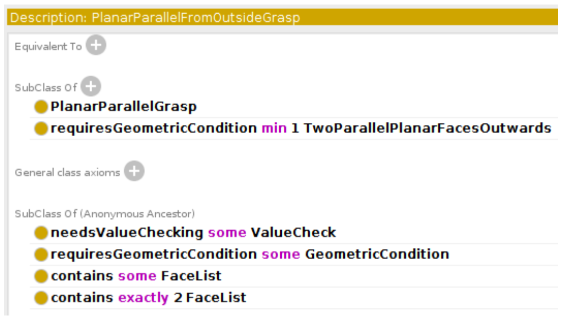
5.1.1.2 Modelling Grasping Capabilities

Figure 5.5 shows the addition of new GraspingCapabilitys to the existing ones. Unlike geometric condition individuals, grasping capability individuals get defined manually before importing the OWL-files into the graph database. Eventually this is something the manufacturer of a gripper should provide alongside the overall specifications.
"SubClass Of (Anonymous Ancestor)" in Protégé are subclass-statements this class inherits from its parents. *contains some FaceList* for instance is the general statement inherited from GraspingCapability whereas *contains exactly* 2 *FaceList* is the concretization inherited from ParallelGrasp.
FaceList bundles faces that are involved in a grasp mode from one finger. The "V-shaped notch from outside grasp" for instance has 4 faces involved in total; 2 on each finger - as can be seen in figure 4.4. Each FaceList-individual keeps a link to it's finger via the object property containedBy. The ValueCheck concept will be presented in the next section.
Note that the grasping capability individual does not contain a requiresGeometricCondition-link. That is because this value stays the same for all individuals of that class



```
Capability
  CognitionCapability
  MotionCapability
    GraspingCapability
      ParallelGrasp
        CylindricalParallelGrasp
          CylindricalParallelFromInsideGrasp
          CylindricalParallelFromOutsideGrasp
            VShapedNotchCylindricalParallelFromOutsideGrasp
        PlanarParallelGrasp
          PlanarParallelFromInsideGrasp
          PlanarParallelFromOutsideGrasp
      PowerGraspingCapability
      PrecisionGraspingCapability
      VacuumSuctionGrasp
        PlanarSurfaceVacuumSuctionGrasp
    ManipulationCapability
```

Positioning in the class hierarchy



Description: PlanarParallelFromOutsideGrasp

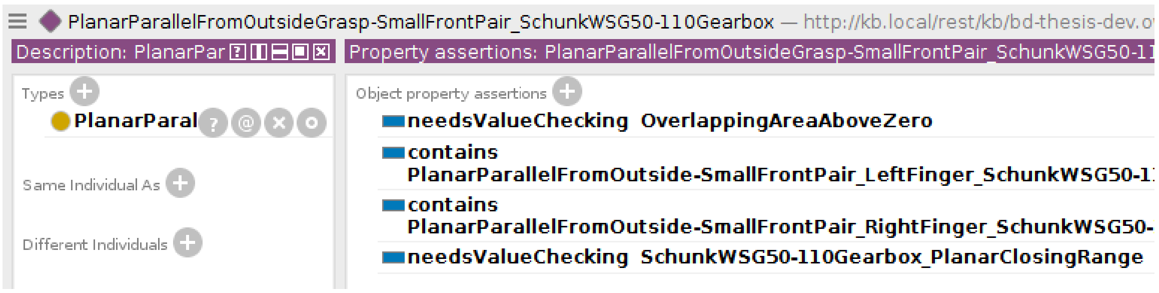
Equivalent To

SubClass Of
- PlanarParallelGrasp
- requiresGeometricCondition min 1 TwoParallelPlanarFacesOutwards

General class axioms

SubClass Of (Anonymous Ancestor)
- needsValueChecking some ValueCheck
- requiresGeometricCondition some GeometricCondition
- contains some FaceList
- contains exactly 2 FaceList

The `PlanarParallelFromOutsideGrasp` class



PlanarParallelFromOutsideGrasp-SmallFrontPair_SchunkWSG50-110Gearbox — http://kb.local/rest/kb/bd-thesis-dev.o

Description: PlanarPar

Types
- PlanarParal

Same Individual As

Different Individuals

Property assertions: PlanarParallelFromOutsideGrasp-SmallFrontPair_SchunkWSG50-11

Object property assertions
- needsValueChecking OverlappingAreaAboveZero
- contains PlanarParallelFromOutside-SmallFrontPair_LeftFinger_SchunkWSG50-1
- contains PlanarParallelFromOutside-SmallFrontPair_RightFinger_SchunkWSG50-
- needsValueChecking SchunkWSG50-110Gearbox_PlanarClosingRange

An individual of the `PlanarParallelFromOutsideGrasp` class

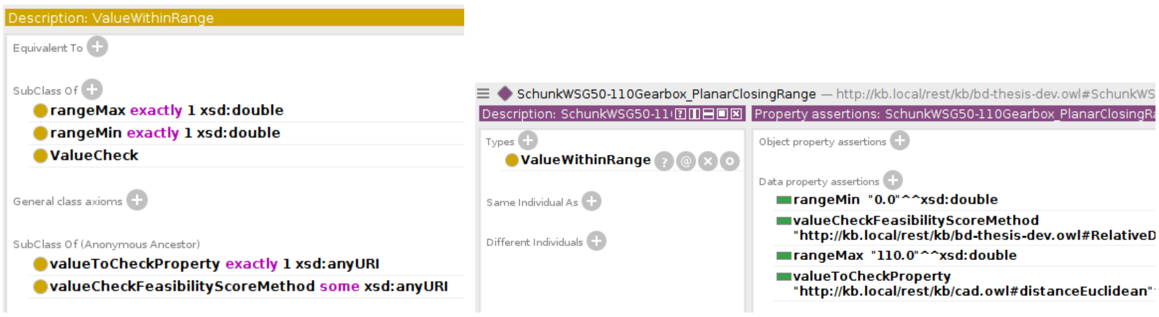
Figure 5.5: Ontological modelling of grasping capabilities in Protégé

and therefore doesn't have to be copied to its individuals. The SPARQL query making the matches will retrieve this info from the individual's class rather than the individual itself.

5.1.1.3 Modelling Value- And Feasibility Checks

As can be seen on the previous figure 5.5, a grasping capability might have an object property `needsValueChecking` that points to an individual of the class `ValueCheck` from which, for now, only the subclass `ValueWithinRange` exists.

Figure 5.6 shows both this class as well as an individual of it. The individual shown is attached to the grasping capability *Planar parallel from outside grasp* and defines the range the grippers finger can have between them. The `valueToCheckProperty` points in this case to the data property `distanceEuclidean` that the required geometric condition of *Two parallel planar faces outwards* possesses. If this value is within the range, the value check passes with *true*.
The ValueWithinRange-individual in figure 5.6 also points to a `FeasibilityCheck` via the data property `valueCheckFeasibilityScoreMethod`.
In this case it's the `RelativeDifferenceToMeanFeasibilityCheck`. Figure 5.7 shows both its class and it's individual containing an actual SPARQL query as string. Notice the `$REPLACE$`-string in the query - the frontend will replace this with pairs of geometric condition individuals and their value within range individuals before sending it to the graph database. The returning table contains said pairs, ordered by their feasiblity score as com-



Description: ValueWithinRange

Equivalent To

SubClass Of
- rangeMax exactly 1 xsd:double
- rangeMin exactly 1 xsd:double
- ValueCheck

General class axioms

SubClass Of (Anonymous Ancestor)
- valueToCheckProperty exactly 1 xsd:anyURI
- valueCheckFeasibilityScoreMethod some xsd:anyURI

SchunkWSG50-110Gearbox_PlanarClosingRange — http://kb.local/rest/kb/bd-thesis-dev.owl#SchunkWS

Description: SchunkWSG50-11

Types
- ValueWithinRange

Same Individual As

Different Individuals

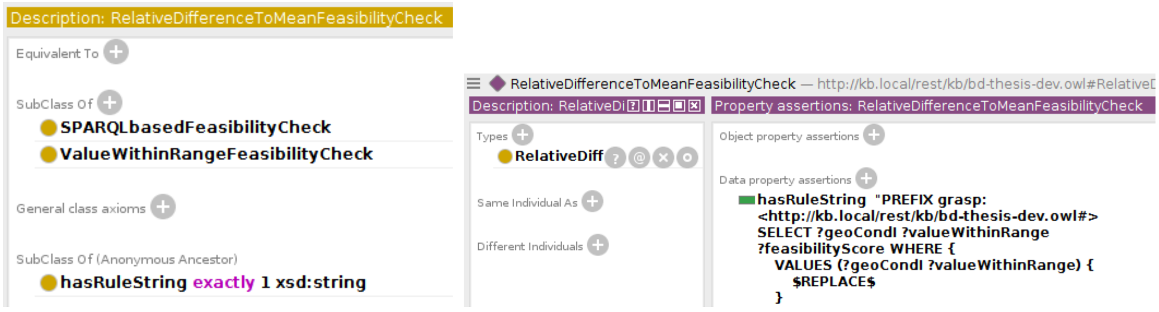
Property assertions: SchunkWSG50-110Gearbox_PlanarClosingR

Object property assertions

Data property assertions
- rangeMin "0.0"^^xsd:double
- valueCheckFeasibilityScoreMethod "http://kb.local/rest/kb/bd-thesis-dev.owl#RelativeD
- rangeMax "110.0"^^xsd:double
- valueToCheckProperty "http://kb.local/rest/kb/cad.owl#distanceEuclidean"

Figure 5.6: Modelling of value check

puted by the method used. This allows a ranking of the matches found for the gripper and the object to grasp. The full query can be found at 5.11.



Description: RelativeDifferenceToMeanFeasibilityCheck

Equivalent To

SubClass Of
- SPARQLbasedFeasibilityCheck
- ValueWithinRangeFeasibilityCheck

General class axioms

SubClass Of (Anonymous Ancestor)
- hasRuleString exactly 1 xsd:string

RelativeDifferenceToMeanFeasibilityCheck — http://kb.local/rest/kb/bd-thesis-dev.owl#RelativeD

Description: RelativeDi

Types
- RelativeDiff

Same Individual As

Different Individuals

Property assertions: RelativeDifferenceToMeanFeasibilityCheck

Object property assertions

Data property assertions

```
hasRuleString "PREFIX grasp:
<http://kb.local/rest/kb/bd-thesis-dev.owl#>
SELECT ?geoCondI ?valueWithinRange
?feasibilityScore WHERE {
  VALUES (?geoCondI ?valueWithinRange) {
    $REPLACE$
  }
```

Figure 5.7: Modelling of feasibility score

When further developing this grasping-ontology, it will make sense to replace `anyURI` from both `valueToCheckProperty` and `valueCheckFeasibilityScoreMethod` with concrete classes that have yet to be modelled.

5.1.2 SPARQL Queries Working With The Modelled Ontologies

The Modelling described above needs to be queried to retrieve triples or insert new ones. This is done in GraphDB using the SPARQL 1.1 Query Language[4]. We do make ourselves dependent on Ontotext's GraphDB[5] as semantic graph database because of the use of the property `sesame:directType` and the use of standard math functions[6] (such as `f:pow(2,3)`) provided by Ontotext. Such dependencies on a specific product are undesirable, but we consider them acceptable for the time being.
Note that all geometry-related queries expect the CAD data to be present in the OntoBREP format as explained in section 3.4.1.

In the following, important parts from the queries regarding the annotation of geometric

[4]https://www.w3.org/TR/sparql11-query/
[5]http://graphdb.ontotext.com/
[6]http://graphdb.ontotext.com/documentation/standard/using-math-functions-with-sparql.html

conditions, matching them with grasping capabilities, checking for values within ranges and assessing feasibility scores will be highlighted.

5.1.2.1 Annotate Objects with their Geometric Conditions

As mentioned at the beginning of this chapter, the geometric conditions, and with some more effort also the grasping capabilities, wouldn't necessarily have to be stored explicitly anyhwere. It would be possible to derive them on the fly by huge queries that include everything required. However, it makes sense to store these explicitly before looking for matches. To annotate a gripper with its grasping capabilities for instance could be seen as the task of the manufacturer. And having the geometric conditions of an object available might be useful later on for other purposes as it enriches the available knowledge on CAD objects. When identifying the geometric condition of parallel planar faces for instance, it is necessary to find normal vectors that are in opposite direction towards each other. This "byproduct" of the computation can be stored as well as it might be useful for other queries in the future.

As described in section 4.4, faces can be annotated as `noGrasp` to mark them unsuitable for grasping. Since this marking is done before annotating geometric conditions, we could have decided not to create geometric condition individuals if at least one of the involved faces has a `noGrasp` flag. Yet, also here we choose to insert the geometric conditions anyways and leave the task of filtering the noGrasp-ones out for the matching process later on. The reason being that we might want to make a GUI addition in the future where users can see invalid grasp modes to identify ways to make them valid for instance. The query to mark faces as `noGrasp` is one line per face, see figure 5.1.

```
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
INSERT {
    <cube.owl#Face1> cad:noGrasp "true"^^xsd:boolean .
    <cube.owl#Face2> cad:noGrasp "true"^^xsd:boolean .
} WHERE {}
```

Listing 5.1: Flag faces as noGrasp (URI shortened)

As explained in section 4.3, we have 5 grasping capabilities with a required geometric condition each. Inward or outward facing surfaces can be distinguished within one query, leaving us with three queries needed to annotate 5 different geometric conditions.

The query to annotate planar parallel faces inwards/outwards will be discussed as a walk-through in the following (and can be found in full at listing 2). The queries to annotate cylindrical face inwards/outwards and the one to annotate planar faces outwards can be found in the appendix at 3 and 4.

Walk-trough of the query to insert TwoParallelPlanarFaces individuals

First, prefixes are being declared at the top of the listing 5.2. They serve as convenience to make the query less visually complex and at the same time show the involved ontologies at one central place of the query, similar to includes at the top of a C++ class for instance. By having declared the `PREFIX cad` for example, we can address the object property `boundedBy` from the CAD ontology in the following manner: `cad:boundedBy` instead of having to use its full uri <http://kb.local/rest/kb/cad.owl#boundedBy> every time.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX f: <http://www.ontotext.com/sparql/functions/>
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
PREFIX grasp: <http://kb.local/rest/kb/bd-thesis-dev.owl#>
```

Listing 5.2: TwoParallelPlanarFaces-query start: prefixes

Next, with listing 5.3 we look at how to select pairs of faces within an object which are represented by planes and prepare their normal vectors for mathematical comparison. The following query starts from an `?object`, which means all objects at once based on the pattern restricting what connections define an object. Running the query on one specific object would mean replacing `?object` with an uri like <http://kb.local/rest/kb/bd-thesis-dev.owl#CubeBlock>. Within each object, all pairs of faces get selected. Through filtering out by comparing the uri-strings, reverse-order duplicates are avoided: when we have *A*-*B* we do not want *B*-*A*, because they are the same.
Once it is ensured that the face-representations are both of type `Plane` (a planar surface), we retrieve the $x/y/z$ components of their respective normal vectors.

```
# INSERT {}
WHERE {
    ?object smerobotics:shape ?compound .
    ?compound cad:contains ?solid .
    ?solid cad:boundedBy ?shell .
    ?shell cad:contains ?face1, ?face2 .
    FILTER(STR(?face1) < STR(?face2)) .
    ?face1 cad:representedBy ?plane1 .
    ?plane1 sesame:directType cad:Plane .
    ?face2 cad:representedBy ?plane2 .
    ?plane2 sesame:directType cad:Plane .
    ?plane1 cad:directionNormal ?p1dN .
    ?plane2 cad:directionNormal ?p2dN .
    ?p1dN cad:x ?p1x ; cad:y ?p1y ; cad:z ?p1z .
    ?p2dN cad:x ?p2x ; cad:y ?p2y ; cad:z ?p2z .
```

Listing 5.3: TwoParallelPlanarFaces-query continuation: select pairs of planes and their normal vectors

At this point we have not ensured that the planes are parallel to each other in 3D. If they are, their normal vectors share the same direction. However, we are only interested in

the cases where the normal vectors point towards each other or looking into opposite directions. The first means we have a planar parallel inwards situation, whereas the second indicates an outwards situation. Therefore we determine the angle between the first vector and the negation of the second vector. If they are indeed in exactly opposite direction, this angle yields 0 (it would be 180 without the negation of the 2nd vector).
The cosine of an angle between two vectors is equal to their dot product divided by the product of their magnitude:

$$\cos\alpha = \frac{\vec{a}\cdot\vec{b}}{\|\vec{a}\|\cdot\|\vec{b}\|} \tag{5.1}$$

Since we are dealing with numerical values, the angle will only be exactly 0.0 in edge-cases, therefore an suitable range should be used instead, see listing 5.4.

```
BIND(?p1x * -?p2x + ?p1y * -?p2y + ?p1z * -?p2z AS ?dotProduct) .
BIND(f:sqrt(f:pow(?p1x, 2) + f:pow(?p1y, 2) + f:pow(?p1z, 2)) AS ?p1dNmag) .
BIND(f:sqrt(f:pow(?p2x, 2) + f:pow(?p2y, 2) + f:pow(?p2z, 2)) AS ?p2dNmag) .
BIND(f:acos(?dotProduct / (?p1dNmag * ?p1dNmag)) AS ?angle) .
BIND(IF(?angle < 0, -?angle, ?angle) as ?angle) .
FILTER (?angle < f:toRadians(1)) .
```

Listing 5.4: TwoParallelPlanarFaces-query continuation: filter out pairs of planes that are not parallel to each other facing both inwards or both outwards

Now the geometric conditions as such are established; all face-pairs not filtered out yet, are inwards our outwards facing parallel planes. What is left to figure out if they are inward or outward oriented and the two parameters: the distance between the planes and their overlapping area when projected into the same plane. Also, since a new individual will be created, a unique URI must be assembled for it.
The distance between the parallel planes corresponds with the length of a perpendicular from one plane to the other. For that we create a line equation from a point from one plane (since every `Plane` has a `cad:position` already, this can be utilized) in the direction of its normal vector. By inserting this line into the scalar equation of the other plane we get the intersection point and thus can compute the distance between the two plane-intersection points.

To get a scalar equation of *Plane1* in the form $ax + by + cz = d$ based on its point-normal form, only the d-coefficient is missing since a, b and c are equivalent with the components of the normal vector. We can get d by using a known point (denoted with subscript 0) in the plane as such: $a(x-x_0)+b(y-y_0)+c(z-z_0) = 0$ and rearranging it to: $d = ax_0+by_0+cz_0$. Called `dCoefficient` in the query 5.5.

Next, the line-equation is defined from a point in *Plane2* in the direction of the normal vector of *Plane1*. Using the variable names as in the query, the equation becomes:

$$l : \vec{x} = \begin{pmatrix} p2posX \\ p2posY \\ p2posZ \end{pmatrix} + \lambda \begin{pmatrix} p1dNX \\ p1dNY \\ p1dNZ \end{pmatrix} \tag{5.2}$$

This line equation inserted into the scalar equation of *Plane1* and multiplied out can be grouped into all scalar parts and all parts tied to λ, called `scalarPart` and `lambdaPart` in the query. Solving the equation for λ (`?lambda`) gives the factor by which to multiply *Plane1*'s normal vector `?p1dN` to get the intersection point with *Plane1* when placing it at `?p2pos`, denoted as `?iX/?iY/?iZ`. The `?distanceBtwnPlanes` between the planes is then equal to the euclidean distance between `p2pos` and this computed intersection point.

```
?p1 cad:position ?p1pos . ?p2 cad:position ?p2pos .
?p1pos cad:x ?p1posX ; cad:y ?p1posY ; cad:z ?p1posZ .
?p2pos cad:x ?p2posX ; cad:y ?p2posY ; cad:z ?p2posZ .
BIND(?p1posX * ?p1dNX + ?p1posY * ?p1dNY + ?p1posZ * ?p1dNZ AS ?dCoefficient) .
BIND(?p2posX * ?p1dNX + ?p2posY * ?p1dNY + ?p2posZ * ?p1dNZ AS ?scalarPart) .
BIND(f:pow(?p1dNX, 2) + f:pow(?p1dNY, 2) + f:pow(?p1dNZ, 2) AS ?lambdaPart) .
BIND((?dCoefficient - ?scalarPart) / ?lambdaPart AS ?lambda)
BIND(?p2posX + ?lambda * ?p1dNX AS ?iX) .
BIND(?p2posY + ?lambda * ?p1dNY AS ?iY) .
BIND(?p2posZ + ?lambda * ?p1dNZ AS ?iZ) .
BIND(f:sqrt(f:pow(?p2posX - ?iX, 2) + f:pow(?p2posY - ?iY, 2) + f:pow(?p2posZ - ?iZ, 2))
↪ AS ?distanceBtwnPlanes) .
```

Listing 5.5: TwoParallelPlanarFaces-query continuation: distance between the planes

A useful side effect of having calculated λ, is that it also indicates if the planes are facing each other or have their backs towards each other. Listing 5.6 uses this to distinguish between the inward or outward facing class and to prepare `?parallelismType` to be used as a new object property between the two faces. The other parameter needed is the overlapping area of the face pairs. This however, would be unnecessarily complicated to compute using SPARQL and consequently a Java-based plugin was developed for this purpose. It is called via a predefined predicate-keyword and assigns the computed value to `?overlappingArea`. More details about the workings of this plugin can be found in section 5.1.3. Next, a unique URI for these new geometric condition individuals must be constructed. `?newGeoCondInstanceIri` is a concatenation of its class, the name of the object and the name of the involved faces.

```
BIND(IF(?lambda > 0, grasp:isBackToBackParallelTo, grasp:isFrontToFrontParallelTo) AS
↪ ?parallelismType).
BIND(IF(?lambda > 0, grasp:TwoParallelPlanarFacesOutwards,
↪ grasp:TwoParallelPlanarFacesInwards) AS ?geoCondClassIri) .

?newGeoCondInstanceIri <http://www.fortiss.org/kb/computeOverlappingAreaOfPolygons> (?f1
↪ ?f2) , ?overlappingArea .

BIND(IRI(CONCAT(STR(?geoCondClassIri), "_", STRAFTER(STR(?object), "#") , "_",
↪ STRAFTER(str(?f1), "#"), "-", STRAFTER(str(?f2), "#"))) AS ?newGeoCondInstanceIri) .
```

Listing 5.6: TwoParallelPlanarFaces-query continuation: inwards or outwards, overlapping area and building new URIs

Finally, it is time to insert the newly computed knowledge as triples into the graph database. Listing 5.7 shows the *INSERT* block of this query. Note that it sits above the *WHERE* block that started in listing 5.3. These insertion patterns tie the two normal vectors together via `isOppositeDirectionOf` and the two faces via `isBackToBackParallelTo` or `isFrontToFrontParallelTo`. Furthermore, it creates the new geometric condition individual and ties the two faces as well as the two parameters to it.

```
INSERT {
    ?p1dN grasp:isOppositeDirectionOf ?p2dN.
    ?f1 ?parallelismType ?f2 .
    ?newGeoCondInstanceIri rdf:type owl:NamedIndividual
      ; rdf:type ?geoCondClassIri
      ; cad:contains ?f1 , ?f2
      ; cad:distanceEuclidean ?distanceBtwnPlanes
      ; grasp:overlappingArea ?overlappingArea .
}
```

Listing 5.7: TwoParallelPlanarFaces-query continuation: insert new triples

5.1.2.2 Get Info About The Gripper And Its Grasping Capabilities

As the workflow in the two query-responsible new JavaScript-classes `Annotator` and `GraspPlanner` in the Frontend will show later in section 5.2.3, info about the gripper will be retrieved before matching grasping capabilities with geometric conditions to get actionable grasp modes. Query 11 in the appendix fetches the spatial orientation of the grippers parts to be displayed correctly in the graps planner. This is however rather specific to fortiss' demo-setup and will not be discussed in detail.

Listing 12 in the appendix shows the SPARQL-query used to retrieve parameters of the gripper as a whole (which at this point is only title and description, but might encompass values like maximum payload in the future) plus its annotated grasping capabilities with their involved gripper-faces and the required geometric condition for each.
A separate query for getting these infos from the gripper would not have been necessary. The matching query, explained in the next section, could have retrieved this info along with the geometric condition infos. However, that would have expanded that query substantially and since it is potentially the most time consuming one, it gives a smoother frontend-experience for the user to see the grasping capabilities appear first and then after a while to see them getting these categories filled with matches.

5.1.2.3 Matching With Value Checks

The full matching-query can be found in the appendix at 6. The frontend only has to fill in the URIs for the object to grasp and for the gripper.

In most, if not all match-cases of geometric condition and grasping capability, it needs to be checked if values are within suitable ranges. The concept of `ValueCheck` is kept generic enough to accommodate a wide range of possible checks. ValueCheck-individuals have a `rangeMin` and a `rangeMax` value and two data properties that contain URIs. `valueToCheckProperty` points to the data property in the geometric condition whose value must be between min and max. And `valueCheckFeasibilityScoreMethod` optionally points to members of the class `FeasibilityCheck`. The idea with feasibility checks is, that after the matches with all values within the respective ranges are established, a second query can be run to assess how well that value scores given the method. As proof of concept for this thesis for instance, the `RelativeDifferenceToMeanFeasibilityCheck` was implemented to give advantage to cylinder-radii and parallel-plane-distances that lie right in the middle of a grippers span for a particular grasping capability.

Listing 5.8 presents the query after the first half in which grasping capabilities of the gripper were matched with the geometric conditions of the object. If the `geoCondParamValue` currently investigated has a `cad:noGrasp` object property tied to it, its value gets recorded in `?noGraspFace`. Since this property is only used for faces, it is implicitly clear, that the `geoCondParamValue` in that moment is a face, bound to the geometric condition via `cad:contains`. SPARQL queries are like a pattern-mask where everything in the graph database sticks that fit into this pattern - a fundamentally different logic compared to simply looping through something and distinguishing items one by one.
The next block handles the value checking logic. It is completely optional as we do not require grasping capability to demand value checks for matches with their desired geometric conditions. If the grasping capability individual does indeed point to an individual of the class `ValueCheck` via the object property `grasp:needsValueChecking`, the necessary values get extracted. Namely *min* and *max* of the range that the grasping capability demands the `?valueToCheckProperty` of the geometric condition to be in. Also the `valueCheckFeasibilityScoreMethod` gets added to the results to be used in a secondary round of queries, more on that in the next section. Since the `?valueToCheckProperty` is modelled as data property of type `xsd:anyURI`, it must first be casted to an active IRI - only then the query-pattern can narrow the results, using the `FILTER()`-notation, down to the geometric condition's property that needs assessment. Once it is ensured we are looking at the right property, the value must be within *min* and *max*. If it does, `?valuePassesCheck` becomes `true`, otherwise `false`. The last step is to give a potential `noGrasp` flag a chance to turn the value check to `false`. If no value is bound to `?valuePassesCheck` (meaning no `grasp:needsValueChecking` for this particular parameter), the value check defaults to `true`.

```
OPTIONAL {
    ?geoCondParamValue cad:noGrasp ?noGraspFace .
}
OPTIONAL {
    ?graspCapI grasp:needsValueChecking ?valueCheckingType .
    ?valueCheckingType grasp:rangeMin ?rangeMin
      ; grasp:rangeMax ?rangeMax
      ; grasp:valueToCheckProperty ?valueToCheckPropertyStr
      ; grasp:valueCheckFeasibilityScoreMethod ?valueCheckFeasibilityScoreMethod .
    BIND(IRI(?valueToCheckPropertyStr) AS ?valueToCheckProperty) .
    FILTER(?geoCondParamName = ?valueToCheckProperty) .
    BIND(IF(?geoCondParamValue >= ?rangeMin && ?geoCondParamValue <= ?rangeMax, true,
    ↪  false) AS ?valuePassesCheck) .
}
BIND(IF(EXISTS{?geoCondParamValue cad:noGrasp ?noGraspFace}, false,
↪  IF(BOUND(?valuePassesCheck), ?valuePassesCheck, true)) AS ?valuePassesCheck) .
```

Listing 5.8: Matching-query 2nd half (see fully query at 6): value checking

Next, the results of running the matching query for a simple cube and the gripper *SchunkWSG50-110Gearbox* (used by fortiss in their demo-setup) is presented. Listing 5.9 shows snippets of the "Raw Response" in *JSON* format that also a GET-request via GraphDB's REST API would yield, more about that in section 5.2.2.
The JSON-response format contains two top-level sections: the header containing the column titles as they would appear on top of the results rendered as table, and the results, which is an array of bindings to the parameter names of the header. Note that a binding must not have values for all header variables. Non-values are an option and are rendered as empty cells in the table format.
The first binding shows one of the 24 resulting rows. Just 7 out of 10 columns have values. That is because it is a face which is not subject to a range-check. It is attached via `cad:contains` to the geometric condition individual and is in this case flagged as `noGrasp`, this renders `?valuePassesCheck` as false. The second binding has values in 9 out of 10 columns, the missing one is due to no value in the noGraspFace-column as this row is not dealing with a face but with the parameter `cad:distanceEuclidean`. And since its value of 50 is well within the `ValueWithinRange`-individual "SchunkWSG50-110Gearbox_PlanarClosingRange" (it has a range of $0 - 110$, which is not neccessary to include in the results because a previous query already transported this info to the frontend, see 5.1.2.2), it passes the value check with `true`. Said individual also points to a feasibility scoring method that will come to use in the next step, initiated by the frontend upon receiving this very results.

```
{
  "head": {
    "vars": ["graspCapI", "geoCondI", "matchingGeoCond", "geoCondParamName",
      "geoCondParamValue", "noGraspFace", "valueCheckingType",
      "valueToCheckProperty", "valuePassesCheck",
      "valueCheckFeasibilityScoreMethod"]},
  "results": {
    "bindings": [{
        "valuePassesCheck": {"datatype": "#boolean",
          "value": "false" },
        "matchingGeoCond": { "type": "uri",
          "value": "#TwoParallelPlanarFacesOutwards" },
        "graspCapI": { "type": "uri",
          "value": "#PlanarParallelFromOutsideGrasp-LargeBackPair" },
        "geoCondI": { "type": "uri",
          "value": "#TwoParallelPlanarFacesOutwards_CubeBlock_Face2-Face4" },
        "geoCondParamValue": { "type": "uri",
          "value": "http://kb.local/rest/kb/cad/cube.owl#Face4" },
        "geoCondParamName": { "type": "uri",
          "value": "http://kb.local/rest/kb/cad.owl#contains" },
        "noGraspFace": { "datatype": "#boolean",
          "value": "true" }
      },{
        "valueToCheckProperty": { "type": "uri",
          "value": "http://kb.local/rest/kb/cad.owl#distanceEuclidean" },
        "valuePassesCheck": { "datatype": "#boolean",
          "value": "true" },
        "matchingGeoCond": { "type": "uri",
          "value": "#TwoParallelPlanarFacesOutwards" },
        "valueCheckFeasibilityScoreMethod": { "datatype": "#anyURI",
          "value": "#RelativeDifferenceToMeanFeasibilityCheck" },
        "graspCapI": { "type": "uri",
          "value": "#PlanarParallelFromOutsideGrasp-LargeBackPair" },
        "valueCheckingType": { "type": "uri",
          "value": "#SchunkWSG50-110Gearbox_PlanarClosingRange" },
        "geoCondI": { "type": "uri",
          "value": "#TwoParallelPlanarFacesOutwards_CubeBlock_Face1-Face3" },
        "geoCondParamValue": { "datatype": "#double",
          "value": "50.0" },
        "geoCondParamName": { "type": "uri",
          "value": "http://kb.local/rest/kb/cad.owl#distanceEuclidean" }
      }]}}
```

Listing 5.9: Partial results from a matching query (edited for better readability)

Note that matches do not get stored in the knowledge base at this point. They "live" only in the results table and in how the frontend displays them. Storing these would require modelling them in the ontology and deciding for a place to store them: with the main repository, in the process-ontologies that get created for each process, or somewhere else. The advantage would be a very desirable reduction of code-complexity in the frontend as the matches are being constructed there. Furthermore it would save inference-time when the same gripper asks for grasp modes with the same object again, they would already be stored.

5.1.2.4 Compute Feasibility Scores

Once the matching-results (see listing 5.9) reach the frontend, a second round of queries are prepared to cover the subsymbolic calculations regarding feasiblity scores. First, a single query gets dispatched to fetch the rule-strings (in itself SPARQL queries with sections to be replaced), combining all feasiblity methods that appeared in the matching-results, see 5.10. At this point only `RelativeDifferenceToMeanFeasibilityCheck` is implemented as proof of concept though.

```
PREFIX grasp: <http://kb.local/rest/kb/bd-thesis-dev.owl#>
SELECT * WHERE {
    VALUES (?method) {
        (grasp:RelativeDifferenceToMeanFeasibilityCheck) (grasp:SomeOtherFeasiblityCheck)
    }
    ?method grasp:hasRuleString ?ruleString .
}
```

Listing 5.10: Query to fetch the SPARQL queries for the feasiblity scoring methods

Once those queries are received, the frontend can replace $\$REPLACE\$$ with those cases that were indicated as candidates for feasibility scoring. For each a pair of the geometric condition individual in question as well as the value within range individual, see listing 5.11. One result from the previous json-results in listing 5.9 for instance, would yield this pair for insertion at the replace-position:

```
(<#TwoParallelPlanarFacesOutwards_CubeBlock_Face1-Face3>
<#SchunkWSG50-110Gearbox_PlanarClosingRange>)
```

```
PREFIX grasp: <http://kb.local/rest/kb/bd-thesis-dev.owl#>
SELECT ?geoCondI ?valueWithinRange ?feasibilityScore WHERE {
    VALUES (?geoCondI ?valueWithinRange) {
        $REPLACE$
    }
    ?valueWithinRange grasp:valueToCheckProperty ?valueToCheckStr .
    BIND(IRI(?valueToCheckStr) AS ?valueToCheck) .
    ?geoCondI ?valueToCheck ?value .
    ?valueWithinRange grasp:rangeMin ?rangeMin
      ; grasp:rangeMax ?rangeMax .
    BIND((?rangeMin + ?rangeMax) / 2 AS ?mean) .
    BIND(abs(?value - ?mean) AS ?absDiffToMean) .
    BIND(?absDiffToMean / ?mean AS ?relDiffToMean) .
    BIND(1 - ?relDiffToMean AS ?feasibilityScore) .
} ORDER BY DESC(?feasibilityScore)
```

Listing 5.11: SPARQL query hosted as string in the data property `hasRuleString`

In this case the math for calculating the relative difference to the mean is fairly simple and could easily have happend in the frontend as well. However, the guiding principle is to keep as much knowledge as possible independent of code, and therefore, it also makes sense to store algorithms in the knowledge base. Future feasibility scoring algorithms might also well be more complicated or call a java-based plugin.

Another feasibility score that would make sense to implement next for instance, is the overlapping area of not only the parallel planes with each other, but also how well they overlap with the surfaces of the grippers finger. This would clearly be something to solve via a plugin rather than a lengthy and complicated SPARQL query.

5.1.3 GraphDB Plugin To Compute Overlapping Area

GraphDB offers a Java-based Plugin API[7] to allow extending the functionality of the graph database. This was used to compute the overlapping area of planar parallel faces when projected into the same plane. This value must at least greater than zero for a proper grasp to be applied as mentioned in section 4.4.

A GraphDB Plugin extends the class `PluginBase` that is provided by Ontotext. Furthermore, the interfaces `PatternInterpreter` or `ListPatternInterpreter` need to be

[7]http://graphdb.ontotext.com/documentation/standard/plug-in-api.html

implemented - the first to interpret simple triple patterns whereas the second can read multiple inputs. In the case of the `PolygonAreaPlugin` developed in the context of this thesis, both interfaces were used to achieve the syntax seen in listing 5.6, namely the "passing" of the two faces as "method variables" and then assigning the computed value to `?overlappingArea`. This was the line in question from listing 5.6 where it is used to calculate the overlapping area of two parallel planes and record that value in the respective geometric condition individual:

```
?geoCondI <#computeOverlappingAreaOfPolygons> (?f1 ?f2) , ?overlappingArea .
```

The comma is just a SPARQL-shortcut for reusing the subject and predicate for a new triple statement with the object that follows. Next I present what happens in the Plugin upon being called from a SPARQL query like shown in the line above. The URI `www.fortiss.org/kb/computeOverlappingAreaOfPolygons` was freely choosen as predicate to trigger the plugin. Since the call with the two faces happens before the one to assign `overlappingArea`, nothing else happens in that first step but storing the two faces in a HashMap with the URI of the geometric condition individual (`geoCondI`) as key:

```
facesOfGeoCondIndividuals.put(entities.get(subject).stringValue(), objects);
```

Upon receiving the second call, the `facesOfGeoCondIndividuals` HashMap is used to re-associate these two faces with the geometric condition individual. By using the same math as was used for computing the distance between parallel planes (see listing 5.5), two perpendicular points on the planes are identified and then used as origins in each planes 2D-representation. These 2D-polygons can then be fed to the `intersection()` method of the JTS Topology Suite[8] and the result of that consequently to the `getArea()` method to get the value we want to be assigned to `?overlappingArea`. More explanations in the comments of listing 5.12. Note that the predicate URI got translated to it's internal representation as `long` value by the variable name `predicateId`.

```
@Override
public StatementIterator interpret(long subject, long predicate, long object, long
↪  context, Statements stmts, Entities entities, RequestContext rc) {
  if (predicate != predicateId) {
     return null;
  }
  if (entities.get(subject) == null ||
   !facesOfGeoCondIndividuals.containsKey(entities.get(subject).stringValue())){
     // -1 indicates the computation has failed
    return createLiteral(entities, subject, predicate, -1);
  }
  try {
    long[] faceIDs =
      facesOfGeoCondIndividuals.get(entities.get(subject).stringValue());
    // iriToId: a Map<String, Long> mapping IRIs to long IDs the plugin uses
    Face face1 = new Face(getLogger(), stmts, entities, iriToId, faceIDs[0]);
    Face face2 = new Face(getLogger(), stmts, entities, iriToId, faceIDs[1]);
    face2.computePlaneEquation();

    // The idea is to project the geom. center point of face1 via it's normal
```

[8] https://projects.eclipse.org/projects/locationtech.jts

```java
    // vector to face2. The point where it hits face2 plus the point where it
    // started from are then treated as origins on the respective planes in 2D.
    Point face1GeomCenter = face1.computeGeometricCenter();
    double scalarPart = face2.planeEqScalar - (face2.planeEqX1 * face1GeomCenter.x +
    ↪  face2.planeEqX2 * face1GeomCenter.y + face2.planeEqX3 * face1GeomCenter.z);
    double lambdaPart = (face2.planeEqX1 * face1.dirNormal.x + face2.planeEqX2 *
    ↪  face1.dirNormal.y + face2.planeEqX3 * face1.dirNormal.z);
    double lambda = scalarPart / lambdaPart;
    Point pointWhereNormalLineFromFace1GeomCenterHitsFace2 =
    ↪  Point.pointPlusVector(face1GeomCenter, Vector.scalarMultiply(face1.dirNormal,
    ↪  lambda));

    // creates points2D list with the respective points as origins
    face1.compute2Dpoints(face1GeomCenter);
    face2.compute2Dpoints(pointWhereNormalLineFromFace1GeomCenterHitsFace2);

    // from the JTS Topology Suite, package com.vividsolutions.jts.geom
    // use the points2D list to build a polygon
    Polygon poly1 = face1.get2dPolygon();
    Polygon poly2 = face2.get2dPolygon();
    Geometry intersection = poly1.intersection(poly2);

    return createLiteral(entities, subject, predicate, intersection.getArea());
  } catch (Exception e) {
    getLogger().info(e.getMessage());
    return createLiteral(entities, subject, predicate, -1);
  }
}
```

Listing 5.12: `interpret()` method of the GraphDB plugin to calculate overlapping area of parallel planes

Before `compute2Dpoints()` computes the 2D-representation of the plane, the 3D-points on the plane already got collected in the `Face()` constructor by iterating over the edges contained by the face's wire and collecting the points representing their bounding vertices, following the OntoBREP structure (3.4.1) of faces.
What happens in `compute2Dpoints()` then, is that for each of these 3D-points a linear equation system of the following form needs to get solved for `point2D`:

$$\begin{bmatrix} dirX.x & dirX.y & dirX.z \\ dirY.x & dirY.y & dirY.z \\ dirZ.x & dirZ.y & dirZ.z \end{bmatrix} * \begin{bmatrix} point2D.x1 \\ point2D.x2 \\ point2D.x3 \end{bmatrix} = \begin{bmatrix} origin2DToPoint3D.x \\ origin2DToPoint3D.y \\ origin2DToPoint3D.z \end{bmatrix} \tag{5.3}$$

Where `dirX`, `dirY` and `dirZ` are the normalized three direction vectors that are part of defining every `Plane` according to the OntoBREP format. `origin2DToPoint3D` is the vector from the previously computed origin (for plane 1 its geometric center and for plane 2 its intersection with the perpendicular from the geometric center of plane 1) to the point in question on the plane. These 3D-vectors from the artificial origin to each point are the coordinates in the 2D system we are interested in. After solving the linear system, two components of `point2D` will be non-zero. These are the components of the point in 2D that we can use to construct the 2D-polygon representing this 3D-plane from the perspective of an observer floating directly above it, whatever orientation it has in 3D-space. Java-listing 5.13 shows this part in `compute2Dpoints()`, using the linear algebra tools from Apache Commons Math[9].

```java
for (Point point3D : points3D) {
    Vector origin2DToPoint3D = Vector.fromTo(origin2D, point3D);
    RealMatrix coefficients = new Array2DRowRealMatrix(new double[][] {
      {dirXnorm.x, dirXnorm.y, dirXnorm.z},
      {dirYnorm.x, dirYnorm.y, dirYnorm.z},
      {dirZnorm.x, dirZnorm.y, dirZnorm.z}}, false);
    DecompositionSolver solver = new LUDecomposition(coefficients).getSolver();
    RealVector constants = new ArrayRealVector(new double[]
      {origin2DToPoint3D.x, origin2DToPoint3D.y, origin2DToPoint3D.z}, false);
    RealVector solution = solver.solve(constants);
    // ...
}
```

Listing 5.13: Solving a linear system for each point in `compute2Dpoints()`

5.2 Part 2: Integration Into The Existing Software

This part presents how the SPARQL queries from part 1 of this implementation chapter found their way into the frontend. First though, a short introduction to the Robot Instruction Framework fortiss has been developing.

5.2.1 fortiss' Robot Instruction Framework

As mentioned before, fortiss developed an intuitive robot programming interface, a "Robot Instruction Framework". The underlying academic work is spread over various papers. Profanter et al. [16] for instance describes a user study conducted to analyze the preferred input modalities for task-based robot programming. Kraft and Rickert [10] compare a previous version of the GUI to the current one. The redesign was driven by applying user experience (UX) paradigms to human-robot-interaction.
Behind a browser-based GUI, the middleware ties together various subsystems written in different programming languages. All configuration steps the user makes, are directly represented by entries in a GraphDB database. The user can conveniently configure assembly-processes. The workflow is simple and intuitive: the user places objects on a workspace to then create assembly-tasks for combinations of them. For each task, constraints are added until the desired position is exactly defined. The overall assembly-process then consists of the chronological execution of those tasks. This can be done entirely virtually for development purposes or it gets wired to a real robot where a perception system detects objects on the table.

Figure 5.8 shows a simplified overview of the software components. All parts communicate via a middleware, however, the new additions, drawn in blue, do not use the middleware. The SPARQL queries from the frontend directly land in the graph database via GET and POST requests - asynchronously by using Ajax-calls.

[9] http://commons.apache.org/proper/commons-math/userguide/linear.html

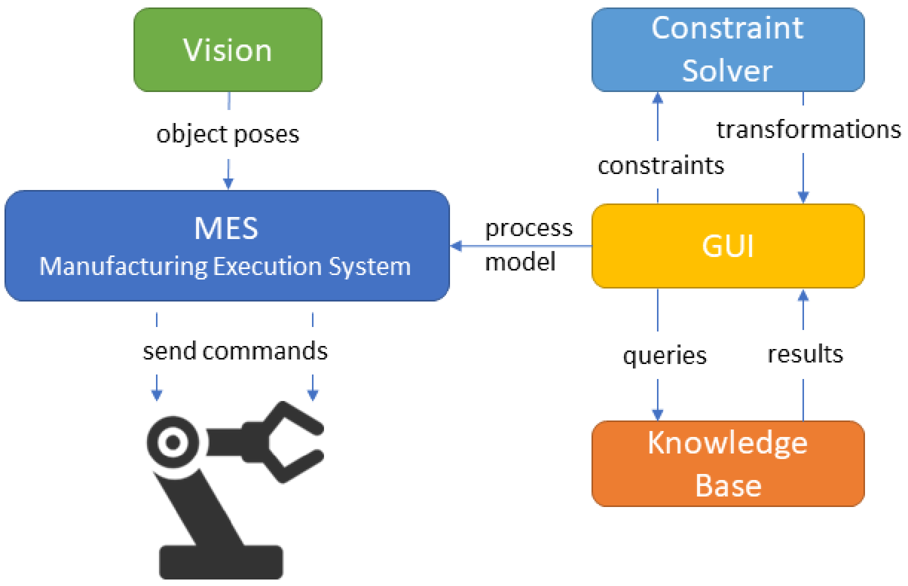


Figure 5.8: Schematic overview of the software components

The proof of concept for the semantics of grasping explained throughout chapter 4, was, to implement it in this software.

5.2.1.1 Workflow In The GUI

Figure 5.9 shows the workflow in the robotic workcell GUI. Outlined in blue are the two conceptual new elements that got added, more about both in section 5.2.3:

- the **Annotator** that lets users mark noGrasp-faces and inserts geometric conditions
- the **Grasp planner** that shows actionable grasp modes based on value-checked matches of grasping capabilities with geometric conditions, ranked by feasibility score

5.2.2 Parsing OntoBREP In The GUI

To further the direct communication between frontend and graph database it made sense to get rid of a json-format that was used for parsing CAD data in the GUI so far. Back in 2016 fortiss developed a converter from *.step* files to *.json*, however, that is no longer maintained and is not seen as the path forward anymore. Instead, that converter is being restructured to convert to *.owl*-files following the OntoBREP format. This follows the philosophy of concentrating all knowledge in ontologies and shedding the need for auxiliary code-based steps or formats as much as possible. All the info required exist in the OntoBREP format, it makes perfect sense to only use this as source for rendering 3D objects. Alongside it's use for inferring geometric conditions.
In the context of this thesis, the class `BrepOwlLoader` in the TypeScript file `brep.owl-loader.ts` was developed to achieve this. Given an URI of the object to fetch, it will dispatch four SPARQL queries, they can be found in the appendix in section 6.

- query 7 collects URIs of all faces, wires, edges, vertices and points of the object
- query 8 fetches the triangulations of faces, each row in the results correlates with one triangle and gives its three point-URIs
- query 9 now collects all points with their coordinates that are appearing in triangles - because of the way triangulation works, this necessarily also includes all points that represent vertices
- query 10 finally fetches the surface-info about all faces of the object; what type is the surface and what are it's direction vectors

Upon receiving the results for these queries, a `BrepObject` is assembled containing edges, faces and vertices. From these, a `THREE.Object3D` is generated that can be viewed and be interacted with tools the JavaScript library Three.js provides, heavily augmented with custom functionality throughout multiple TypeScript classes.

The plan is to expand the use of this new `BrepOwlLoader` to also parse the CAD data of the workcell and the robot arm, ideally eventually the entire software can handle CAD data in OntoBREP format. This would greatly simplify the conversion-pipeline of handling new objects in the system.

5.2.3 Two New GUI Elements For The Semantics Of Grasping

From all of the theoretical- and implementation-work described so far, the user only gets to see changes in the GUI-workflow. Flowchart 5.9 highlighted the two places across the workflow were new elements were introduced. The purpose of these additions is to support the user in specifying grasp modes in the assembly-process. Grasp mode candidates are being semantically generated using the queries and mechanisms described. These get presented to the user as ranked options to choose from.

5.2.3.1 Part 1: Annotator

The addition of the annotator is outlined on the left side of the flowchart 5.9. It has two tasks: giving the user the opportunity to flag faces as noGrasp and to send the query to automatically infer geometric conditions. Since both of these tasks relate to objects alone, a natural place to locate this functionality is the “Upload New Part” tab next to the default tab of “Select Existing Part” that slides open on the right side of the GUI when creating a new project for instance. Since annotating conditions has to happen only once per object, why not do it when the object enters the pool of objects.
When the annotator opens, the first task is to select noGrasp-faces, if applicable, as shown in figure 5.10. A click on the face directly on the object adds it to the selection, a 2nd click on that face or clicking on its name in the selection removes it from the selection.
The click on “Apply no-grasp annotations” (it is saying “Skip no-grasp annotations” if no face is selected) sends off the query to insert 5.1-flags to the selected faces. Then it posts three queries to inspect the object for the five different geometric conditions currently supported as explained in section 5.1.2.1: *Two parallel planar faces inwards/outwards*, *Cylindrical*

face inwards/outwards and *Planar face outwards*.

Once these four queries are completed, another one gets send to fetch the geometric conditions that were just inserted, see listing 5. These get displayed, sorted by category as shown in figure 5.11. As it can get a lot of results per category depending on the object, a category collapses if it has more then 4 entries. One of them is grayed out in the screenshot because it contains a noGrasp-face.
In the future one might want to add an option here to manually remove inferred conditions that are actually not suitable. Or even manually insert missing ones, although a modelling-mechanism would have to be developed to support that kind of flexibility.
To showcase parts of the TypeScript code, 5.14 shows the nested queries being set off by clicking on the Apply/Skip no-grasp annotation button as described above.

```
this.annotatorListToolbar.onApplyNoGraspFaceSelection.subscribe(
  (faceIris: string[]) => {
  this.annotator.runMarkNoGraspFacesQuery(faceIris).then(() => {
    this.annotator.runInferGeometricConditionsQueries().then(() => {
      this.annotator.getGeometricConditionsQuery().then(
        (geoConds: GeometricCondition[]) => {
        this.annotatorListToolbar.listGeoConds(geoConds);
});});});});
```

Listing 5.14: Snippet from the `Annotator3DComponent`-class

To give insight into one of these methods, TypeScript-listing 5.15 shows the method that runs the three queries to infer geometric condition individuals. The queries are not shown, they can be found in the appendix at 6

```
public runInferGeometricConditionsQueries(): Promise<void> {
  return new Promise<void>((resolve, reject) => {
    let twoParallelPlanarFacesQuery = ""; // ...
    let cylindricalQuery = ""; // ...
    let planarFacesQuery = ""; // ...
    let promises = [];
    promises.push(this.runQuery(twoParallelPlanarFacesQuery));
    promises.push(this.runQuery(cylindricalQuery));
    promises.push(this.runQuery(planarFacesQuery));
    Promise.all(promises).then(() => {
      resolve();
    }).catch(() => {
      reject();
});});}
private runQuery(query: string): Promise<void> {
  return new Promise<void>((resolve, reject) => {
    $.ajax({
      url: 'http://localhost:7200/repositories/kb/statements?update=',
      type: 'POST', dataType: 'json', data: query,
      contentType: 'application/sparql-update',
      complete: function (msg) {
        if (msg.status === 204) {
          resolve();
        } else {
          reject();
}}});});}
```

Listing 5.15: `runInferGeometricConditionsQueries()`-method in the `Annotator`-class

5.2.3.2 Part 2: Grasp Planner

The addition of the grasp planner is outlined on the lower part of the flowchart 5.9. When a user is done defining tasks of the assembly process, the click on "Simulate & Run" opens a modal window showing the tasks as squares horizontally instead of the vertical list from the previous screen. The only option at this point is to "Choose a workcell". Upon doing so, the process gets deployed on the selected workcell. This entails a range of actions:

- the ontological description of the workcell get's loaded
- the controller for the robot arm gets added, both the ontological description as well as its CAD data (which in the future will only be parsed from the ontology, see section 5.2.2)
- the perception systems and the object detectors get activated, if none is connected to the middleware (in case of pure virtual development for instance), a dummy-perception serves as fallback
- then the process gets loaded into the workcell and initialized, meaning all values that did not need to be specified up to know, need concrete values fitting to the workcell environment
- as part of initializing the process, new tasks might be inferred in the knowledge base - examples of inferred task are "Move robot home" at the end of a process or a "Grasp and Release" task for potentially every object if the perception system is not precise enough (after a grasp and release a fuzzy location of an object is known precisely, namely in the middle of the gripper span as it closed on the object)
- as the frontend gets notice of the updated process, newly inferred tasks get inserted as squares with a green frame into the task list

Figure 5.12 shows the tasklist after a workcell was chosen and the process got deployed on it. The grasp-icons open the grasp planner on a specific object. Since in most assembly tasks two or more (in case of assembling assemblies) objects are involved, choosing which one to plan a grasp for, also counts as decision to have the robot move this object instead of the other one in an assembly step. Currently the solution to distinguish the objects is via a mouse-over tooltip as seen in the screenshot - other solutions to this user interface challenge will likely be considered in the future.
Figure 5.13 shows the grasp planner after having pressed a gripper icon in figure 5.12. By default it opens with the matches sorted by category, with the grasping capabilities being the categories. Note that it deals with a different object then the ones seen previously. Namely one used in development, constructed to be simple but still exhibit all 5 currently supported geometric conditions.

At the stage as shown in figure 5.13, all 5 (currently - each new category of feasiblity scoring method added in the future will add one query) queries that the grasp planner is equipped with, have already been sent to the graph database and have returned with results. Namely the following, shown with their TypeScript method name.

- `getGripperDataFromWorkcell()` fetches the CAD data of the gripper to be displayed in 3D, see listing 11, more info in section 5.1.2.2
- `getGrippersGraspingCapabilities()` gets the grippers parameters and its grasping capabilities, including which of its faces are involved, see listing 12, more info in section 5.1.2.2
- `inferMatches()` compares the grippers grasping capabilities with the objects geometric conditions, the matches that pass the value checks are actionable grasp modes, see listing 6
- `fetchFeasibilityScoreMethodSPARQLqueries()` collects all feasibility scoring methods mentioned in value checks of the previous results and fetches their rule-string, which in itself is a SPARQL query were certain contents needs replacing before dispatching it, see listing 5.10
- run the queries fetched in the method above after inserting custom info at the places marked for replacing, example in listing 5.11

Figure 5.14 shows the grasp planner with the matches sorted by feasibility score. The idea being that in the future automatically the highest one will be used by default if the user does not choose to select a grasp mode by hand.
Note that matches who didn't pass any of their value checks are not displayed. This can however be realized in the future as a feature for operators to investigate why a certain grasp mode might not show up. Those invalid matches are already included in the result table, there is just a user interface for it missing in the frontend.

In terms of TypeScript code, listing 5.16 shows the method `inferMatches()` of the class `GraspPlanner`. The query-string itself is left out, it can be found in listing 6. What happens here is looping through the results table and sorting things to the respective matches. Notice the use of the classes `Match`, `GeometricCondition` and `ValueCheck`. These were created to assist the sorting of data throughout the frontend. Moreover, the map `self.grippersGraspCaps` makes use of the class `GraspingCapability`.

```
public inferMatches(): Promise<void> {
    let query = ""; // ...
    return new Promise<void>((resolve, reject) => {
      let self = this;
      $.getJSON(GraspPlanner.repositoryURL, {query: query, infer: true})
        .done(function(response) {
          let rows = response.results.bindings;
          if (rows.length === 0) { reject(); return; }
          let allMatches = new Map<string, Match>();
          for (let i = 0; i < rows.length; ++i) {
            let row = rows[i];
            let matchKey = row.graspCapI.value + '+' + row.geoCondI.value;
            if (!allMatches.has(matchKey)) {
              let match = new Match(row.graspCapI.value, row.geoCondI.value);
              match.graspCap = self.grippersGraspCaps.get(row.graspCapI.value);
              match.geoCond = new GeometricCondition(row.geoCondI.value);
              allMatches.set(matchKey, match);
            }
            let match = allMatches.get(matchKey);
            if (row.geoCondParamName.value === 'http://kb.local/rest/kb/cad.owl#contains')
            ↪  {
              match.geoCond.addFace(row.geoCondParamValue.value, row.noGraspFace ?
              ↪  row.noGraspFace.value === 'true' : false);
            } else {
              match.geoCond.addParam(row.geoCondParamName.value,
              ↪  row.geoCondParamValue.value);
            }
            if (row.valueCheckingType) {
              let valueCheck = new ValueCheck(row.valueCheckingType.value,
              ↪  row.valueToCheckProperty.value, row.valuePassesCheck.value === 'true',
              ↪  row.valueCheckFeasibilityScoreMethod.value);
              if (!valueCheck.valuePassesCheck) {
                match.hasInvalidValue = true;
              }
              match.valueChecks.push(valueCheck);
            }
          }
          allMatches.forEach((match: Match, key: string) => {
            if (!match.hasInvalidValue) { self.matches.set(key, match); }
          });
          Msg.success(self.matches.size + " valid matches ("
            + (allMatches.size - self.matches.size) + " invalid)");
          resolve();
        })
        .fail(function () {
          reject();
});});}
```

Listing 5.16

To give an insight into the workings of AngularJS in the HTML of the frontend, listing 5.17 shows the part in `graspmodelist.toolbar.component.html` that is responsible for rendering the matches on the right side of the grasp planner when viewed by feasibility score. `matchesRanked` is an array of matches, sorted by decreasing feasiblity score.

```
<!-- By feasibility score -->
<div class="group clearfix layer-1" [ngClass]="{ 'expand': byCategory === false}">
  <div class="header" (click)="toggleViewMode()">
    <span class="name">By feasibility score</span>
    <span class="arrow"><i class="fa-angle-down"></i></span>
  </div>
  <div class="group-content">
    <div class="parts">
      <div *ngFor="let match of matchesRanked" class="part" [ngClass]="{'selectedGeoCond':
      ↪  match===selectedMatch}">
        <span class="hint geoCondSpan">
          <span class="smallFont feasibilityScore">
            score: {{match.feasibilityScoreAverage | number : '1.2-2'}}
          </span><br>
          <span *ngFor="let face of match.geoCond.faces">
            {{face.iri.split('#')[1]}}
          </span><br>
          <span class="smallFont" *ngFor="let param of match.geoCond.parameters">
            {{param.name.split('#')[1]}}: {{param.value | number : '1.2-2'}}<br/>
          </span>
</span></div></div></div></div>
```

Listing 5.17: HTML-part responsible for rendering the matches sorted by feasiblity score (edited for better readability)

The next step would be to get from a grasp mode to concrete poses for the gripper to approach during execution. This however, was unfortunately not achieved in the timeframe of this thesis. It will involve consulting the constraint solver and the collision checker to see if a grasp mode is physically possible.

Also, as mentioned at the end of section 5.1.2.3, currently matches as standalone individuals are not modelled ontologically yet. They live only in result tables and frontend-display. It is certainly worth considering permitting them a triple store presence of their own. The fact that pseudo-IDs for matches became necessary for handling them in the frontend-code, as seen in code snippet 5.16 indicates this further.

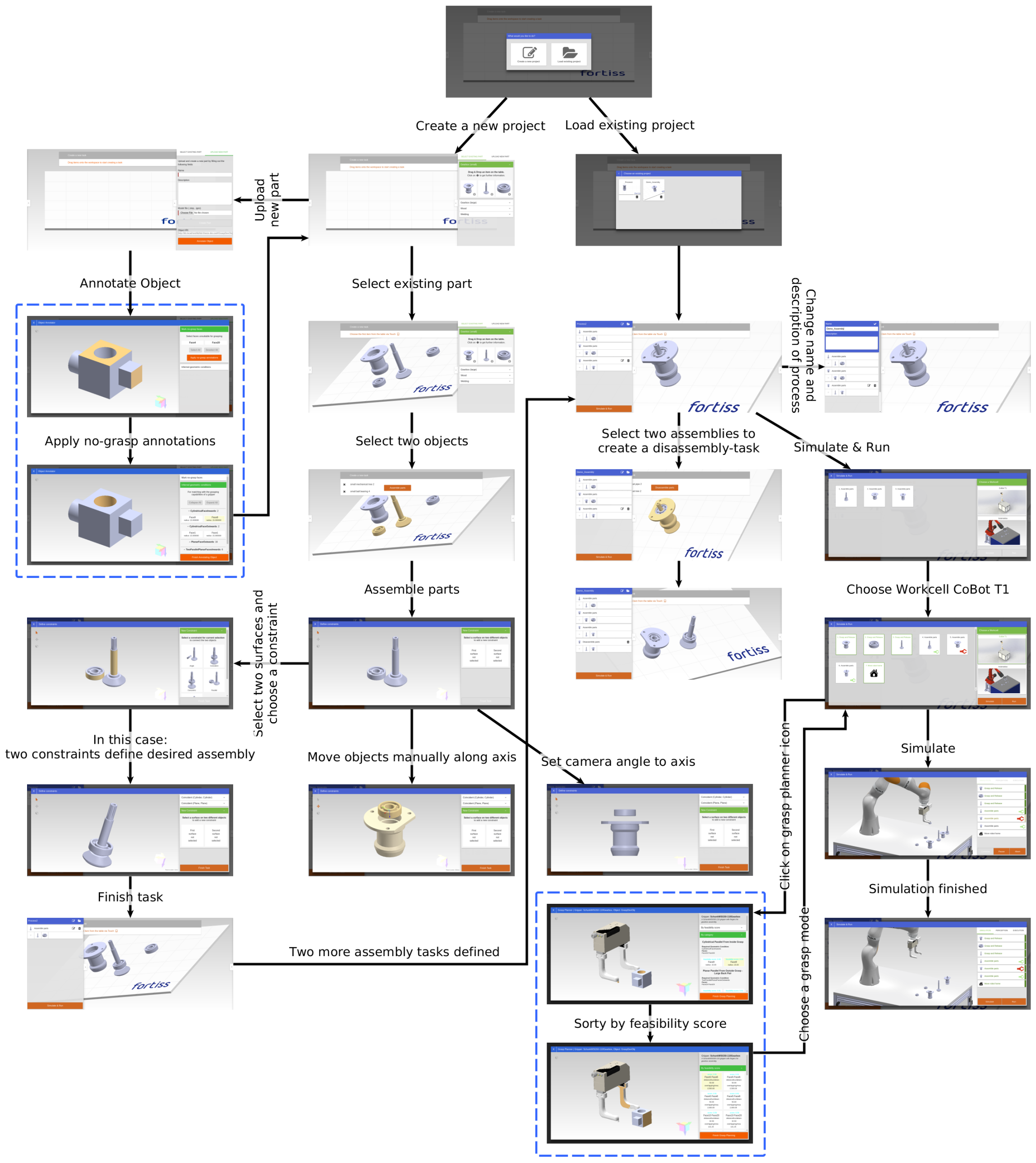
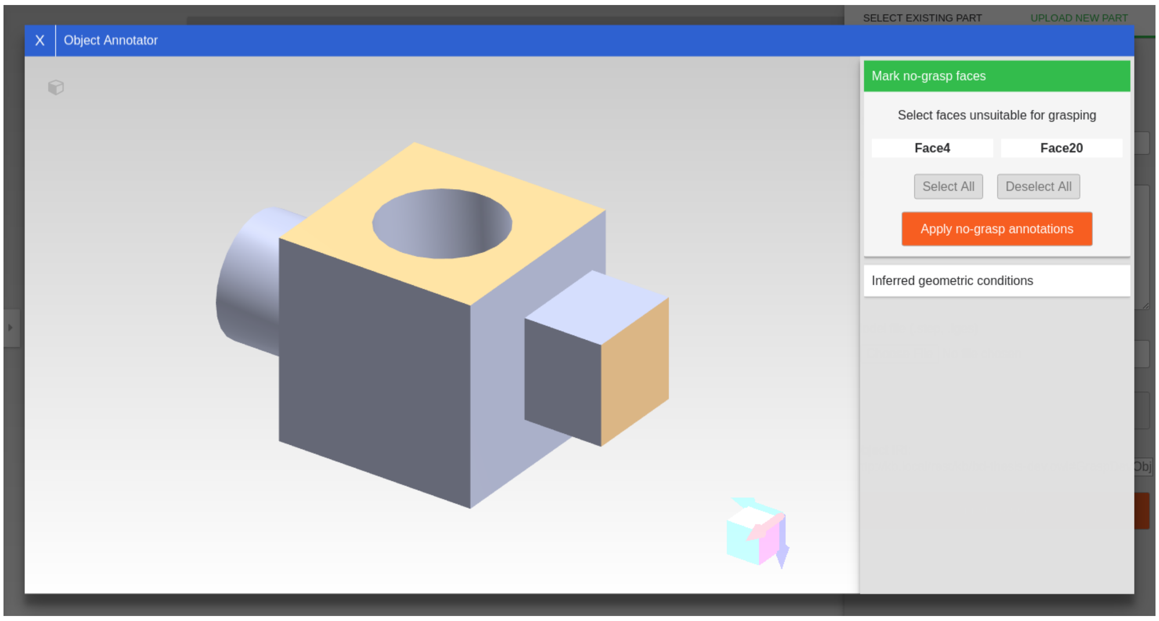

Create a new project
Load existing project
Upload new part
Annotate Object
Apply no-grasp annotations
Select existing part
Select two objects
Assemble parts
Select two surfaces and choose a constraint
In this case: two constraints define desired assembly
Finish task
Move objects manually along axis
Set camera angle to axis
Two more assembly tasks defined
Change name and description of process
Select two assemblies to create a disassembly-task
Simulate & Run
Choose Workcell CoBot T1
Simulate
Simulation finished
Click on grasp planner icon
Choose a grasp mode
Sorty by feasibility score

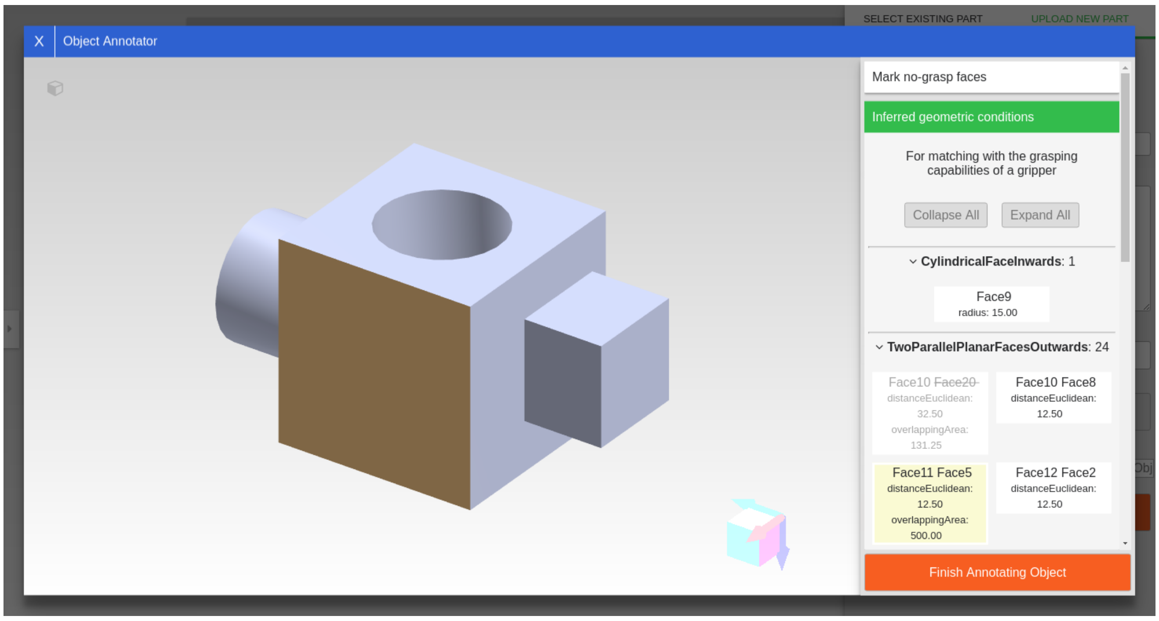
Figure 5.9: Workflow in the GUI



Select faces unsuitable for grasping

Face4	Face20

Figure 5.10: First step in the Annotator: mark noGrasp-faces



For matching with the grasping capabilities of a gripper

CylindricalFaceInwards: 1

Face9
radius: 15.00

TwoParallelPlanarFacesOutwards: 24

Face10 ~~Face20~~
distanceEuclidean: 32.50
overlappingArea: 131.25

Face10 Face8
distanceEuclidean: 12.50

Face11 Face5
distanceEuclidean: 12.50
overlappingArea: 500.00

Face12 Face2
distanceEuclidean: 12.50

Figure 5.11: Second step in the Annotator: list inferred geometric conditions

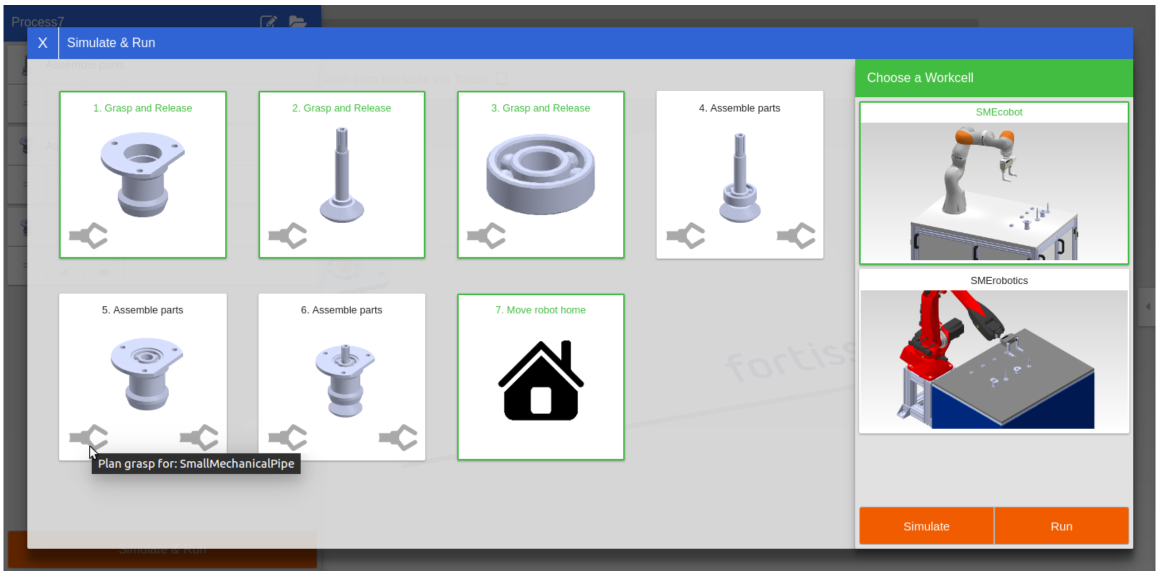


Figure 5.12: Task list augmented with inferred tasks after deploying the process on a work-cell and icons to open the grasp planner

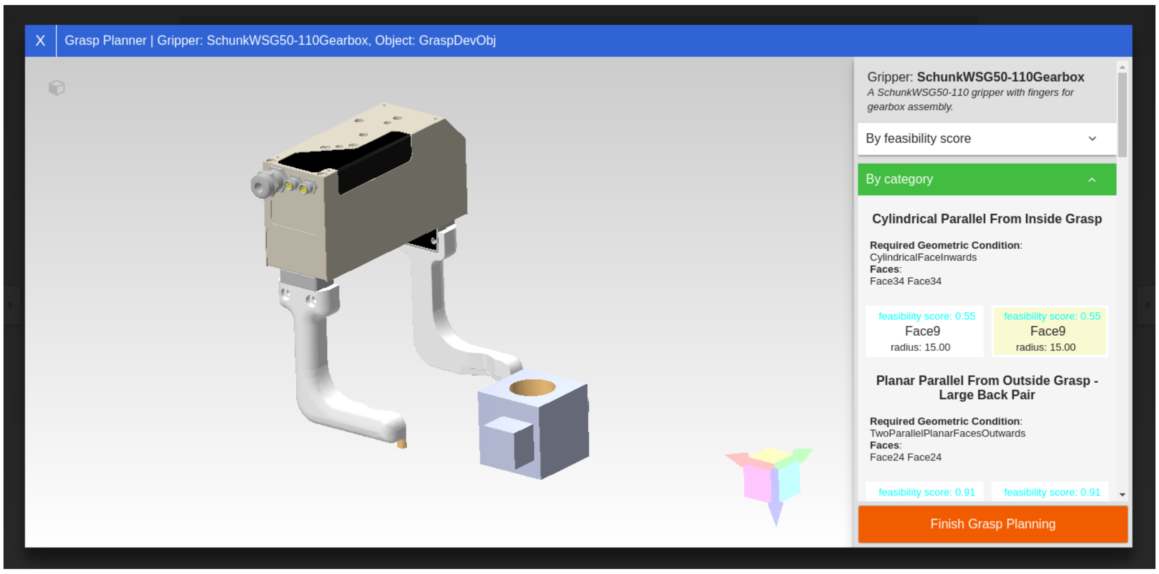
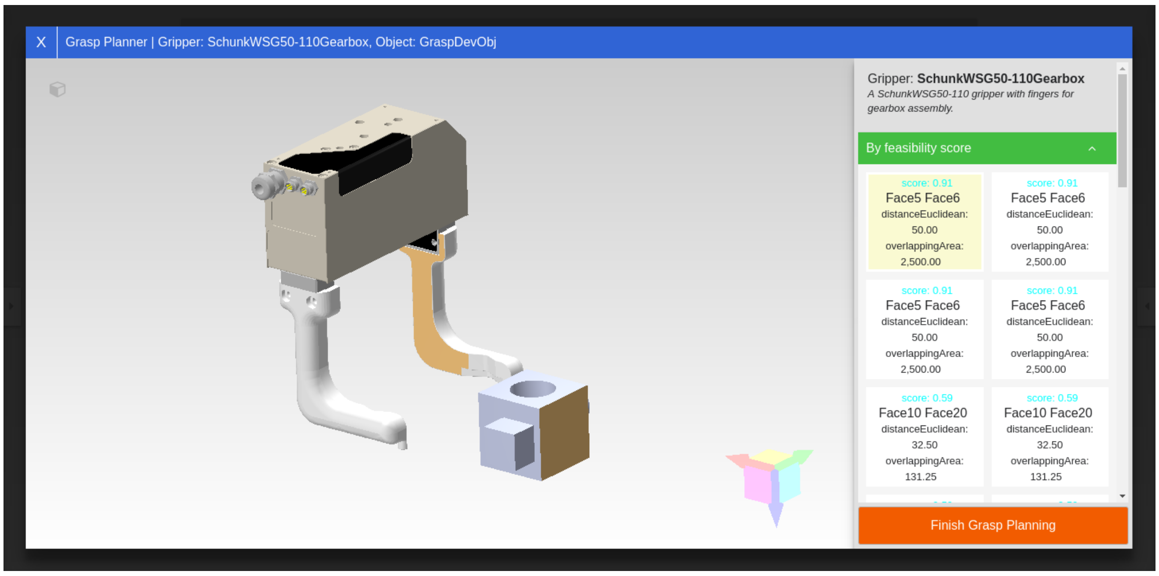

Grasp Planner | Gripper: SchunkWSG50-110Gearbox, Object: GraspDevObj
Gripper: SchunkWSG50-110Gearbox
A SchunkWSG50-110 gripper with fingers for gearbox assembly.
By feasibility score
By category
Cylindrical Parallel From Inside Grasp
Required Geometric Condition: CylindricalFaceInwards
Faces: Face34 Face34
feasibility score: 0.55
Face9
radius: 15.00
feasibility score: 0.55
Face9
radius: 15.00
Planar Parallel From Outside Grasp - Large Back Pair
Required Geometric Condition: TwoParallelPlanarFacesOutwards
Faces: Face24 Face24
feasibility score: 0.91
feasibility score: 0.91
Finish Grasp Planning

Figure 5.13: Grasp Planner, one match selected, sorted by category



Grasp Planner | Gripper: SchunkWSG50-110Gearbox, Object: GraspDevObj

Gripper: **SchunkWSG50-110Gearbox**
A SchunkWSG50-110 gripper with fingers for gearbox assembly.

By feasibility score

score: 0.91 Face5 Face6 distanceEuclidean: 50.00 overlappingArea: 2,500.00	score: 0.91 Face5 Face6 distanceEuclidean: 50.00 overlappingArea: 2,500.00
score: 0.91 Face5 Face6 distanceEuclidean: 50.00 overlappingArea: 2,500.00	score: 0.91 Face5 Face6 distanceEuclidean: 50.00 overlappingArea: 2,500.00
score: 0.59 Face10 Face20 distanceEuclidean: 32.50 overlappingArea: 131.25	score: 0.59 Face10 Face20 distanceEuclidean: 32.50 overlappingArea: 131.25

Figure 5.14: Grasp Planner, one match selected, sorted by feasibility score

6 Discussion, Conclusion And Future Work

Discussion

The result of the conceptual and implementation work done in this thesis is a foray into the semantics of grasping in an industrial context. The concept, the modelling and the implementation, all of it will likely undergo further evolution, yet the work in this thesis represents a significant contribution.

Before, there was no semantic concept dealing with grasping. In fortiss' software, the grasp modes were hardcoded for a small demo-set, meaning exact numerical poses for every object and combination of objects was deposited. This is not a sustainable strategy when opening up to other scenarios and further objects. The assembly-combinations that would have to be covered by manual pose-definitions would increase exponentially. In contrast to this, now it is possible to infer grasp modes based on the one-time annotation of a grippers grasping capabilities and a one-time annotation of an objects geometric conditions.

The effects on a users relation to such a software is also worth considering. Now, there is a process to witness an object getting enriched by making its features explicit, even though unexpected things may come to light. Also the process of going through grasp modes could be enjoyable as one might be comparing the options to how one might grab this object with their own hands.

These being the potentially positive effects of it, the argument can also be made that this adds to the overall time required to define assembly processes. Both the object annotation as well as choosing a grasp mode is something that requires active user input - unless it becomes further automatized that is. However, things like marking faces unsuitable for grasping and annotating the grasping capabilities of a gripper can also be assets that can be shared with others who work with similar setups. It could be included in standards how companies offer their products enriched by ontological descriptions of its specifications and features.

As a tangible example of the added value, one might think of an engineer that has the most experience with robotic grasping in the context of the companys specific assembly-requirements. As valuable as such skillsets are, it creates a strong dependency on that person. Painfully large parts of knowledge might go missing, at least temporarily, when such key-persons retire or leave a company. Even a proper handover can't replace decades worth of intuition grown in a particular production context. The approach of ontological domain modelling seeks to precisely address this. The knowledge migrates from the "heads of experts" into the ontologies. And as such, can continously be refined and shared with others in the same field. Standards can emerge that might further the developments in adjacent fields etc.

Conclusion

All in all, the formal specification of the semantics of grasping using the Web Ontology Language (OWL) and its integration into fortiss' robot programming GUI were successful in most parts. Although the final step from a valid grasp mode to the required poses is missing and would have rounded the development off in a nice way as we could see a robot moving around objects and grasping them the way the new concept "told them to" instead of using hardcoded values.
The concept of matching a grippers grasping capabilities with an objects geometric condition while considering value ranges and rank them by feasiblity, is a novel one as such. It emerged from the immersion in the grasping taxonomy research and the thought-process on how to model grasp modes as generic and convenient as possible.

Future Work

The trend in industry is to dilute conservative multitier models where each level can only talk to its neighbouring ones. Instead at times we might want to puncture through from a top level to sensor data on a very low level. The guiding principle of this work, to separate the knowledge base and the code as much as possible is playing into that. Furthermore, by shifting maximally much "intelligence" from the code to the knowledge base, we become more independent on any concrete implementations and can quickly change, translate or adapt the implementations.

Another trend is "Zero Programming", the "automation of the automation" or also called the knowledge based digital engineering. Having reasoning tools that can on a symbolic level guarantee incompabilities, is a crucial puzzle piece in this way forward as it can drastically limit the search spaces that subsymbolic methods have to comb through and can give users semantically meaningful insights.

Full Stack Of Grasp Mode Validation

As stated in the conclusion, the full stack of grasp mode validation was not explored for this thesis. Here is an overview of the different levels, ordered from symbolic to more and more subsymbolic:

Symbolic level: On the symbolic level, matching grasp modes can be found as explained throughout this thesis.

Constraint solver: However, to reach concrete poses for the gripper, the constraint solver must perform arithmetic operations that take the geometric restrictions and finds concrete manifestations that satisfy all constraints. Basically enough constraints must be defined to narrow down the degrees of freedom to zero.

Collision Checker: The next level of vetting a grasp mode for its actual usefulness is collision checking. A collision checker answers the question if one volume is ingested by another, or not. If this passes without problems, there is still one further level.

Physics simulation: Namely considering forces like friction and gravity that occur in reality. Only a physics simulation can answer seamingly simple questions, like, will the assembly the gripper wants to grasp, fall apart the moment it is lifted? Think of trying to grab a ball bearing on a mechanical tree in order to lift this assembly. The effect will be to take the bearing off the tree and not move both.

One could draw parallels to human grasping. We intuitively assess the geometric features of objects in order to derive grasping strategies. This constitutes the logical reasoning part. Once we touch it, further subsymbolic optimizations are being done by small repositioning and applying pressure based on friction, weight and the centre of mass.

Ideas For Improvements

There are many meaningful threads to continue in various directions from this work. Both in terms of furthering the concept, the modelling and the implementation.

The very obvious one is to complete the workflow-pipeline and take the step(s) from grasp mode to concrete gripper pose.
Also high on the priority list I would suggest to model a match-class in the ontology and create individuals upon running the matching query. A decision where to store these would have to be made.
Using OntoBREP for all CAD-usages across the back- and frontend will bring clarity to the formats required by the system and standardize a CAD-format that is maximally enriched by being able to hold both symbolic as well as approximated representations.
Develop more classes of geometric conditions and grasping capabilities beyond the 5 basic ones chosen for this thesis. Also, the modelling needs to be adjusted to support grasping capabilities in demanding multiple geomtric conditions, not just one as currently the case. For instance the capped V-shaped notch from outside grasp (the lower one of the two notches on the gripper fingers as seen on the gripper-screenshots) actually needs an outward facing cylinder *plus* a restriction on the cylinder in one vertical direction.
Value checks must also support other types than ranges (e.g. must be higher then or must be unequal to) and more feasiblity scoring methods must be developed based on demand. Both SPARQL- and Plugin-based.
The computation of overlapping area must be extended to the gripper finger and the objects face or faces to asses their feasibility. Currently it only calculates the overlapping area of two parallel planes upon projection in the same plane.
The user interface regarding how to open the grasp planner should be given some more thought. With the two icons currently it is a bit cumbersome to plan a grasp on the desired object and it is also not clear what that means in terms of picking order during assembly.
Support the automatic selection of the grasp mode with the highest score by default. Confidence could be indicated by coloring the grasp planner icon.

In the annotator screen the user might want to have the option to manually remove geometric conditions that were automatically inferred. Maybe even allowing the insertion of own custom ones. Even so that might be not trivial to model.
Just like the grasp planner should be default select the best option, the annotator should also do its job silently and without a mandatory intervention by the user. The marking of faces as noGrasp is optional in any case.
Finding a convenient way to also give the user insight into invalid grasp modes if desired. Including the reasons for failing value checks gives the user a chance to investigate issues with the object or the gripper.
Supporting properties of the gripper as a whole, like a maximum payload, would become important at some point. These might then have to be matched with properties of geometric conditions or with the objects properties as a whole, like its mass.
It would be great to be able to annotate grasping capabilities also more or less automatically like geometric conditions. Or at least update them upon changes. When the geometry of a finger changes for instance or the gripper span changes, this should cause immediate updates in the grasping capabilities too.
It might be worth considering to merge/mix/augment OntoBREP with "GeoSPARQL - A Geographic Query Language for RDF Data"[1] for convenient access to complex geometrical functions.
Other graph databases might have to offer more useful functionality. Stardog[2] for instance seems to have better support for propert chains and custom rules. It is not available for free though.
Eventually the grasp planner might also have to consider the target location where to place the object. There might be physical constraints there making match found on the logical level impossible to execute. For instance, don't grasb the ball bearing from outside if it needs to go into the pipe - the gripper finger would be in the way.
Along that line but thinking even further ahead - considering the next tasks ahead could bring great benefits. Which grasp mode is best suited to manoeuver the gripper already in a pose that is useful for the next assembly task. Which grasp mode on the other hand makes certain things impossible in the next task.
The concept of "primitive shape matching" (e.g. as described in Somani et al. [19]) could help with organic objects with potentially too many surface features to "reasonable reason" about them. Subsymbolic algorithms would match the best fitting primitive objects into the organic shape and these would then be used for further reasoning about the object, like matching with grasping capabilities.

[1] https://www.opengeospatial.org/standards/geosparql
[2] https://www.stardog.com/

Appendix

```xml
<!-- Establishing the link to the cube's namespace -->

<owl:NamedIndividual rdf:about="dev.owl#Cube">
  <rdf:type rdf:resource="dev.owl#Cube"/>
  <smerobotics:shape rdf:resource="cube.owl#Compound1"/>
</owl:NamedIndividual>

<!-- Compound1 contains Solid1 contains Shell1 contains 6 faces.
The first one: -->

<owl:NamedIndividual rdf:about="cube.owl#Face1">
  <rdf:type rdf:resource="cad.owl#Face"/>
  <cad:boundedBy rdf:resource="cube.owl#Wire1"/>
  <cad:representedBy rdf:resource="cube.owl#Plane1"/>
  <cad:triangulatedBy rdf:resource="cube.owl#Triangulation1"/>
  <cad:hasForwardOrientation>true</cad:hasForwardOrientation>
</owl:NamedIndividual>

<!-- Plane1 has a position and vectors for x, y and z direction -->
<!-- Triangulation1 contains two Triangles. The first one: -->

<owl:NamedIndividual rdf:about="cube.owl#Triangle1">
  <rdf:type rdf:resource="cad.owl#Triangle"/>
  <cad:firstNode rdf:resource="cube.owl#Point4"/>
  <cad:secondNode rdf:resource="cube.owl#Point2"/>
  <cad:thirdNode rdf:resource="cube.owl#Point1"/>
</owl:NamedIndividual>

<!-- Wire1 contains 4 edges. The first one: -->

<owl:NamedIndividual rdf:about="cube.owl#Edge1">
  <rdf:type rdf:resource="cad.owl#Edge"/>
  <cad:adjacentEdge rdf:resource="cube.owl#Edge2"/>
  <cad:boundedBy rdf:resource="cube.owl#Vertex1"/>
  <cad:boundedBy rdf:resource="cube.owl#Vertex2"/>
  <cad:representedBy rdf:resource="cube.owl#Line1"/>
  <cad:hasForwardOrientation>false</cad:hasForwardOrientation>
</owl:NamedIndividual>

<!-- Vertices are represented by Points that contain the coordinates. -->
<!-- Line1 has a direction vector and a position. -->
```

Listing 1: Excerpts of a cube in OntoBREP format (URIs are displayed shortened)

SPARQL queries

SPARQL queries regarding geometric conditions

```sparql
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX f: <http://www.ontotext.com/sparql/functions/>
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
PREFIX grasp: <http://kb.local/rest/kb/bd-thesis-dev.owl#>
INSERT {
    ?p1dirN grasp:isOppositeDirectionOf ?p2dirN.
    ?f1 ?parallelismType ?f2 .
    ?newGeoCondInstanceIri rdf:type owl:NamedIndividual
      ; rdf:type ?geoCondClassIri
      ; cad:contains ?f1 , ?f2
      ; cad:distanceEuclidean ?distanceBtwnPlanes
      ; grasp:overlappingArea ?overlappingArea .
}
WHERE {
    <http://kb.local/rest/kb/bd-thesis-dev.owl#CubeBlock> smerobotics:shape ?compound .
    ?object smerobotics:shape ?compound .
    ?compound cad:contains ?solid .
    ?solid cad:boundedBy ?shell .
    ?shell cad:contains ?f1, ?f2 .
    FILTER(STR(?f1) < STR(?f2)) .
    ?f1 cad:representedBy ?p1 . ?p1 sesame:directType cad:Plane .
    ?f2 cad:representedBy ?p2 . ?p2 sesame:directType cad:Plane .
    ?p1 cad:directionNormal ?p1dirN .
    ?p2 cad:directionNormal ?p2dirN .
    ?p1dirN cad:x ?p1dirNX ; cad:y ?p1dirNY ; cad:z ?p1dirNZ .
    ?p2dirN cad:x ?p2dirNX ; cad:y ?p2dirNY ; cad:z ?p2dirNZ .
    BIND(?p1dirNX * -?p2dirNX + ?p1dirNY * -?p2dirNY + ?p1dirNZ * -?p2dirNZ AS
    ↪  ?dotProduct) .
    BIND(f:sqrt(f:pow(?p1dirNX, 2) + f:pow(?p1dirNY, 2) + f:pow(?p1dirNZ, 2)) AS
    ↪  ?p1dirNlength) .
    BIND(f:sqrt(f:pow(?p2dirNX, 2) + f:pow(?p2dirNY, 2) + f:pow(?p2dirNZ, 2)) AS
    ↪  ?p2dirNlength) .
    BIND(f:acos(?dotProduct / (?p1dirNlength * ?p2dirNlength)) AS ?angleBtwnDirNs) .
    BIND(IF(?angleBtwnDirNs < 0, -?angleBtwnDirNs, ?angleBtwnDirNs) as ?angleBtwnDirNs) .
    FILTER (?angleBtwnDirNs < f:toRadians(1)) .
    ?p1 cad:position ?p1pos .
    ?p2 cad:position ?p2pos .
    ?p1pos cad:x ?p1posX ; cad:y ?p1posY ; cad:z ?p1posZ .
    ?p2pos cad:x ?p2posX ; cad:y ?p2posY ; cad:z ?p2posZ .
    BIND(?p1posX * ?p1dirNX + ?p1posY * ?p1dirNY + ?p1posZ * ?p1dirNZ AS
    ↪  ?p1coordFormRightSideValue) .
    BIND(?p2posX * ?p1dirNX + ?p2posY * ?p1dirNY + ?p2posZ * ?p1dirNZ AS
    ↪  ?scalarPartLeftSide) .
    BIND(f:pow(?p1dirNX, 2) + f:pow(?p1dirNY, 2) + f:pow(?p1dirNZ, 2) AS
    ↪  ?lambdaPartLeftSide) .
    BIND((?p1coordFormRightSideValue - ?scalarPartLeftSide) / ?lambdaPartLeftSide AS
    ↪  ?lambda)
    BIND(?p2posX + ?lambda * ?p1dirNX AS ?iX) .
    BIND(?p2posY + ?lambda * ?p1dirNY AS ?iY) .
    BIND(?p2posZ + ?lambda * ?p1dirNZ AS ?iZ) .
    BIND(f:sqrt(f:pow(?p2posX - ?iX, 2) + f:pow(?p2posY - ?iY, 2) + f:pow(?p2posZ - ?iZ,
    ↪  2)) AS ?distanceBtwnPlanes) .
    BIND(IF(?lambda > 0, grasp:isBackToBackParallelTo, grasp:isFrontToFrontParallelTo) AS
    ↪  ?parallelismType).
    BIND(IF(?lambda > 0, grasp:TwoParallelPlanarFacesOutwards,
    ↪  grasp:TwoParallelPlanarFacesInwards) AS ?geoCondClassIri) .
    BIND(IRI(CONCAT(STR(?geoCondClassIri), "_", STRAFTER(STR(?object), "#") , "_",
    ↪  STRAFTER(str(?f1), "#"), "-", STRAFTER(str(?f2), "#"))) AS ?newGeoCondInstanceIri)
    ↪  .
    ?newGeoCondInstanceIri <http://www.fortiss.org/kb/computeOverlappingAreaOfPolygons>
    ↪  (?f1 ?f2) , ?overlappingArea .
}
```

Listing 2: SPARQL query to annotate objects with the geometric condition of TwoParallelPlanarFacesInwards/Outwards

```
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX grasp: <http://kb.local/rest/kb/bd-thesis-dev.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
INSERT {
    ?newGeoCondInstanceIri rdf:type owl:NamedIndividual
      ; rdf:type ?geoCondClassIri
      ; cad:contains ?face
      ; cad:radius ?radius .
}
WHERE {
    <http://kb.local/rest/kb/bd-thesis-dev.owl#CylinderBasic> smerobotics:shape ?compound
    ↪  .
    ?object smerobotics:shape ?compound .
    ?compound cad:contains ?solid .
    ?solid cad:boundedBy ?shell .
    ?shell cad:contains ?face .
    ?face cad:representedBy ?surface . ?surface sesame:directType cad:CylindricalSurface .
    ?surface cad:radius ?radius .
    ?face cad:hasForwardOrientation ?orientation .
    BIND(IF(?orientation = true, grasp:CylindricalFaceOutwards,
    ↪  grasp:CylindricalFaceInwards) AS ?geoCondClassIri).
    BIND(IRI(CONCAT(STR(?geoCondClassIri), "_", STRAFTER(STR(?object), "#") , "_",
    ↪  STRAFTER(str(?face), '#'))) AS ?newGeoCondInstanceIri) .
}
```

Listing 3: SPARQL query to annotate objects with the geometric condition of CylindricalFaceInwards/Outwards

```
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX grasp: <http://kb.local/rest/kb/bd-thesis-dev.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
INSERT {
    ?newGeoCondInstanceIri rdf:type owl:NamedIndividual
      ; rdf:type grasp:PlanarFaceOutwards
      ; cad:contains ?face
}
WHERE {
    <http://kb.local/rest/kb/bd-thesis-dev.owl#CubeBlock> smerobotics:shape ?compound .
    ?object smerobotics:shape ?compound .
    ?compound cad:contains ?solid .
    ?solid cad:boundedBy ?shell .
    ?shell cad:contains ?face .
    ?face cad:representedBy ?plane . ?plane sesame:directType cad:Plane .
    ?face cad:hasForwardOrientation true .
    BIND(IRI(CONCAT(STR(grasp:PlanarFaceOutwards), "_", STRAFTER(STR(?object), "#") , "_",
    ↪  STRAFTER(str(?face), "#"))) AS ?newGeoCondInstanceIri) .
}
```

Listing 4: SPARQL query to annotate objects with the geometric condition of PlanarFaceOutwards

```
PREFIX grasp: <http://kb.local/rest/kb/bd-thesis-dev.owl#>
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
SELECT DISTINCT ?geoCondI ?geoCondClass ?paramName ?paramValue ?noGrasp WHERE {
    ?object smerobotics:shape ?compound .
    FILTER(?object = <http://kb.local/rest/kb/bd-thesis-dev.owl#CubeBlock>)
    ?compound cad:contains ?solid .
    ?solid cad:boundedBy ?shell .
    ?shell cad:contains ?face .
    ?geoCondI rdf:type grasp:GeometricCondition .
    ?geoCondI cad:contains ?face .
    ?geoCondI sesame:directType ?geoCondClass . FILTER (?geoCondClass !=
    ↪  owl:NamedIndividual) .
    ?geoCondI ?paramName ?paramValue .
    FILTER(?paramName != rdf:type) .
    OPTIONAL {
        ?paramValue cad:noGrasp ?noGrasp .
    }
}
```

Listing 5: SPARQL query to fetch geometric conditions of an object together with their parameters

SPARQL queries to match grasping capabilities with geometric conditions

```
PREFIX grasp: <http://kb.local/rest/kb/bd-thesis-dev.owl#>
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX f: <http://www.ontotext.com/sparql/functions/>
SELECT DISTINCT ?graspCapI ?geoCondI ?matchingGeoCond ?geoCondParamName ?geoCondParamValue
↪  ?noGraspFace ?valueCheckingType ?valueToCheckProperty ?valuePassesCheck
↪  ?valueCheckFeasibilityScoreMethod WHERE {
    ?object smerobotics:shape ?compound .
    FILTER(?object = <http://kb.local/rest/kb/bd-thesis-dev.owl#GraspDevObj>)
    ?object smerobotics:shape ?compound .
    ?compound cad:contains ?solid .
    ?solid cad:boundedBy ?shell .
    ?shell cad:contains ?objectFace .
    ?geoCondI rdf:type grasp:GeometricCondition .
    ?geoCondI cad:contains ?objectFace .
    ?geoCondI sesame:directType ?geoCondClass . FILTER (?geoCondClass !=
    ↪  owl:NamedIndividual) .

    <http://kb.local/rest/kb/smerobotics.owl#SchunkWSG50-110Gearbox> sesame:directType
    ↪  ?gripperClass . FILTER(?gripperClass != owl:NamedIndividual)
    ?gripper sesame:directType ?gripperClass .
    ?gripper grasp:hasGraspingCapability ?graspCapI .
    ?graspCapI sesame:directType ?graspCapClass . FILTER(?graspCapClass !=
    ↪  owl:NamedIndividual)
    ?graspCapClass rdfs:subClassOf ?rest1 .
    ?rest1 owl:onProperty grasp:requiresGeometricCondition .
    ?rest1 owl:onClass ?requiredGeoCondClass .
    FILTER(?geoCondClass = ?requiredGeoCondClass) . BIND(?geoCondClass AS
    ↪  ?matchingGeoCond) .

    ?geoCondI ?geoCondParamName ?geoCondParamValue . FILTER(?geoCondParamName != rdf:type)
    OPTIONAL {
        ?geoCondParamValue cad:noGrasp ?noGraspFace .
    }
    OPTIONAL {
        ?graspCapI grasp:needsValueChecking ?valueCheckingType .
        ?valueCheckingType grasp:rangeMin ?rangeMin
          ; grasp:rangeMax ?rangeMax
          ; grasp:valueToCheckProperty ?valueToCheckPropertyStr
          ; grasp:valueCheckFeasibilityScoreMethod ?valueCheckFeasibilityScoreMethod .
        BIND(IRI(?valueToCheckPropertyStr) AS ?valueToCheckProperty) .
        FILTER(?geoCondParamName = ?valueToCheckProperty) .
        BIND(IF(?geoCondParamValue >= ?rangeMin && ?geoCondParamValue <= ?rangeMax, true,
        ↪  false) AS ?valuePassesCheck) .
    }
    BIND(IF(EXISTS{?geoCondParamValue cad:noGrasp ?noGraspFace}, false,
    ↪  IF(BOUND(?valuePassesCheck), ?valuePassesCheck, true)) AS ?valuePassesCheck) .
}
```

Listing 6: SPARQL query to match geometric conditions of an object with the grasping capabilities of gripper and consider value checks

BrepOwlLoader SPARQL queries

```
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
SELECT ?face ?wire ?edge ?vertex ?point WHERE {
  <http://kb.local/rest/kb/bd-thesis-dev.owl#CubeBlock> smerobotics:shape ?compound .
  ?compound cad:contains ?solid .
  ?solid cad:boundedBy ?shell .
  ?shell cad:contains ?face .
  ?face cad:boundedBy ?wire .
  ?wire cad:contains ?edge .
  ?edge cad:boundedBy ?vertex .
  ?vertex cad:representedBy ?point .
}
```

Listing 7: SPARQL query to get all URIs from a face's wire's subelements

```
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
SELECT ?face ?point1 ?point2 ?point3 WHERE {
  <http://kb.local/rest/kb/bd-thesis-dev.owl#CubeBlock> smerobotics:shape ?compound .
  ?compound cad:contains ?solid .
  ?solid cad:boundedBy ?shell .
  ?shell cad:contains ?face .
  ?face cad:triangulatedBy ?triangulation .
  ?triangulation cad:contains ?triangle .
  ?triangle cad:firstNode ?point1 ; cad:secondNode ?point2 ; cad:thirdNode ?point3 .
}
```

Listing 8: SPARQL query to get triangulation points of faces

```
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
SELECT DISTINCT ?point ?coords WHERE {
  <http://kb.local/rest/kb/bd-thesis-dev.owl#CubeBlock> smerobotics:shape ?compound .
  ?compound cad:contains ?solid .
  ?solid cad:boundedBy ?shell .
  ?shell cad:contains ?face .
  ?face cad:triangulatedBy ?triangulation .
  ?triangulation cad:contains ?triangle .
  {
    ?triangle cad:firstNode ?point1 . BIND(?point1 AS ?point)
  }
  UNION {
    ?triangle cad:secondNode ?point2 . BIND(?point2 AS ?point)
  }
  UNION {
    ?triangle cad:thirdNode ?point3 . BIND(?point3 AS ?point)
  }
  ?point cad:x ?x . ?point cad:y ?y . ?point cad:z ?z .
  BIND(concat(str(?x), ",", str(?y), ",", str(?z)) AS ?coords)
}
```

Listing 9: SPARQL query to get points and their coordinates

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
  SELECT ?face ?surface ?surfaceType ?positionCoords ?dirXCoords ?dirYCoords ?dirZCoords
  ↪  ?radius WHERE {
  <http://kb.local/rest/kb/bd-thesis-dev.owl#CubeBlock> smerobotics:shape ?compound .
  ?compound cad:contains ?solid .
  ?solid cad:boundedBy ?shell .
  ?shell cad:contains ?face .
  ?face cad:representedBy ?surface .
  ?surface sesame:directType ?surfaceType .
  FILTER (?surfaceType != cad:Triangulation && ?surfaceType != owl:NamedIndividual) .
  ?surface cad:position ?position .
  ?position cad:x ?Px ; cad:y ?Py ; cad:z ?Pz .
  BIND(concat(str(?Px), ",", str(?Py), ",", str(?Pz)) AS ?positionCoords)
  ?surface cad:directionX ?dirX ; cad:directionY ?dirY ; cad:directionZ ?dirZ .
  ?dirX cad:x ?DXx ; cad:y ?DXy ; cad:z ?DXz .
  BIND(concat(str(?DXx), ",", str(?DXy), ",", str(?DXz)) AS ?dirXCoords)
  ?dirY cad:x ?DYx ; cad:y ?DYy ; cad:z ?DYz .
  BIND(concat(str(?DYx), ",", str(?DYy), ",", str(?DYz)) AS ?dirYCoords)
  ?dirZ cad:x ?DZx ; cad:y ?DZy ; cad:z ?DZz .
  BIND(concat(str(?DZx), ",", str(?DZy), ",", str(?DZz)) AS ?dirZCoords)
  OPTIONAL {
    ?surface cad:radius ?radius .
  }
}
```

Listing 10: SPARQL query to get the surfaces representing faces

Other SPARQL queries

```
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
PREFIX swdl: <http://kb.local/rest/kb/swdl.owl#>
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT * WHERE {
    <http://kb.local/rest/kb/workcell/fortiss_cobot_workcell.owl#FortissCoBotWorkcell>
    ↪  (swdl:succeedingJoint | swdl:succeedingLink)* ?gripper .
    ?gripper rdf:type smerobotics:Gripper .
    ?gripper rdf:type <http://kb.local/rest/kb/smerobotics.owl#SchunkWSG50-110Gearbox> .
    ?gripper (swdl:succeedingJoint | swdl:succeedingLink)* ?node .
    ?node ^(swdl:succeedingJoint | swdl:succeedingLink) ?parent .
    OPTIONAL {
        ?node swdl:transformation ?transMatrix .
        ?transMatrix
            cad:a11 ?a11 ; cad:a12 ?a12 ; cad:a13 ?a13 ; cad:a14 ?a14 ;
            cad:a21 ?a21 ; cad:a22 ?a22 ; cad:a23 ?a23 ; cad:a24 ?a24 ;
            cad:a31 ?a31 ; cad:a32 ?a32 ; cad:a33 ?a33 ; cad:a34 ?a34 ;
            cad:a41 ?a41 ; cad:a42 ?a42 ; cad:a43 ?a43 ; cad:a44 ?a44 .
    }
    OPTIONAL {
        ?node smerobotics:modelUrl ?modelUrl .
    }
}
```

Listing 11: SPARQL query to get the spatial orientation of the grippers parts for displaying it in the grasp planner

```
PREFIX grasp: <http://kb.local/rest/kb/bd-thesis-dev.owl#>
PREFIX cad: <http://kb.local/rest/kb/cad.owl#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX smerobotics: <http://kb.local/rest/kb/smerobotics.owl#>
PREFIX dcterms: <http://purl.org/dc/terms/>
SELECT ?graspCapI ?title ?graspCapClass ?requiredGeoCondClass ?faceList
↪  ?gripperFingerOfFaceList ?face ?cappedByFace ?paramName ?paramValue WHERE {
    ?gripper rdf:type owl:NamedIndividual .
    ?gripper rdf:type <http://kb.local/rest/kb/smerobotics.owl#SchunkWSG50-110Gearbox> .
    ?gripper rdf:type ?gripperClass .
    ?gripperClass sesame:directType smerobotics:Gripper .
    ?gripper rdf:type owl:Class .
    {
        ?gripper grasp:hasGraspingCapability ?graspCapI .
        ?graspCapI dcterms:title ?title .
        ?graspCapI sesame:directType ?graspCapClass . FILTER(?graspCapClass !=
        ↪  owl:NamedIndividual)
        ?graspCapI cad:contains ?faceList .
        ?faceList cad:contains ?face .
        ?faceList cad:containedBy ?gripperFingerOfFaceList .
        ↪  FILTER(?gripperFingerOfFaceList != ?graspCapI) .
        OPTIONAL {
            ?faceList grasp:cappedBy ?cappedByFace .
        }
        ?graspCapClass rdfs:subClassOf ?restriction .
        ?restriction owl:onProperty grasp:requiresGeometricCondition .
        ?restriction owl:onClass ?requiredGeoCondClass .
    } UNION {
        ?gripper ?paramName ?paramValue .
        FILTER(?paramName != rdf:type && ?paramName != rdfs:subClassOf && ?paramName !=
        ↪  grasp:hasCapability && ?paramName != grasp:hasGraspingCapability)
    }
}
```

Listing 12: SPARQL query to get the parameters of the gripper and its grasping capabilities including their required geometric conditions and the involved faces

Bibliography

[1] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration". *Robotics and Autonomous Systems*, 57(5):469 – 483, 2009.

[2] M. Beetz, D. Beler, J. Winkler, J. Worch, F. Blint-Benczdi, G. Bartels, A. Billard, A. K. Bozcuolu, , N. Figueroa, A. Haidu, H. Langer, A. Maldonado, A. L. P. Ureche, M. Tenorth, and T. Wiedemeyer. Open robotics research using web-based knowledge services. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5380–5387, May 2016.

[3] A. K. Bozcuolu and M. Beetz. A cloud service for robotic mental simulations. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2653–2658, May 2017.

[4] A. K. Bozcuolu, G. Kazhoyan, Y. Furuta, S. Stelter, M. Beetz, K. Okada, and M. Inaba. The exchange of knowledge using cloud robotics. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8, May 2018.

[5] Gordon Cheng, Karinne Ramirez-Amaro, Michael Beetz, and Yasuo Kuniyoshi. Purposive learning: Robot reasoning about the meanings of human activities. *Science Robotics*, 4(26), 2019.

[6] J.; Schmiedmayer HB Dollar A.M. Kragic D. Feix, T.; Romero. The grasp taxonomy of human grasp types. In *Human-Machine Systems, IEEE Transactions on*, 2015.

[7] Thomas Feix, Roland Pawlik, and Heinz-Bodo Schmiedmayer. The generation of a comprehensive grasp taxonomy. *In RSS: Robotics, Science and Systems: Workshop on Understanding the Human Hand for Advancing Robotic Manipulation RSS'09, Seattle, USA*, 01 2009.

[8] Sandro Fiorini, Joel Carbonera, Paulo Gonalves, Vitor Jorge, Vitor Rey, Tamas Haidegger, Mara Abel, Signe Redfield, Stephen Balakirsky, and Veera Ragavan Sampath Kumar. Extensions to the core ontology for robotics and automation. *Robotics and Computer-Integrated Manufacturing*, 33, 09 2014.

[9] Ken Goldberg and Ben Kehoe. Cloud robotics and automation: A survey of related work. Technical Report UCB/EECS-2013-5, EECS Department, University of California, Berkeley, Jan 2013.

[10] Martin Kraft and Markus Rickert. How to teach your robot in 5 minutes: Applying ux paradigms to human-robot-interaction. 08 2017.

[11] J. R. Napier. The prehensile movements of the human hand. 38-B:902–13, 12 1956.

[12] J. Persson, A. Gallois, A. Bjoerkelund, L. Hafdell, M. Haage, J. Malec, K. Nilsson, and P. Nugues. A knowledge integration framework for robotics. In *ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*, pages 1–8, June 2010.

[13] A. Perzylo, M. Rickert, B. Kahl, N. Somani, C. Lehmann, A. Kuss, S. Profanter, A. B. Beck, M. Haage, M. R. Hansen, M. Roa-Garzon, O. Sornmo, S. Gestegard Robertz, U. Thomas, G. Veiga, E. A. Topp, I. Kessler, and M. Danzer. Smerobotics: Smart robots for flexible manufacturing. *IEEE Robotics Automation Magazine*, pages 1–1, 2019.

[14] A. Perzylo, N. Somani, S. Profanter, I. Kessler, M. Rickert, and A. Knoll. Intuitive instruction of industrial robots: Semantic process descriptions for small lot production. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2293–2300, Oct 2016.

[15] A. Perzylo, N. Somani, M. Rickert, and A. Knoll. An ontology for cad data and geometric constraints as a link between product models and semantic robot task descriptions. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4197–4203, Sept 2015.

[16] S. Profanter, A. Perzylo, N. Somani, M. Rickert, and A. Knoll. Analysis and semantic modeling of modality preferences in industrial human-robot interaction. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1812–1818, Sept 2015.

[17] Karinne Ramrez-Amaro and Gordon Cheng. Accelerating the teaching of industrial robots by re-using semantic knowledge from various domains. 05 2018.

[18] C. Schlenoff, E. Prestes, R. Madhavan, P. Goncalves, H. Li, S. Balakirsky, T. Kramer, and E. Miguelez. An ieee standard ontology for robotics and automation. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1337–1342, Oct 2012.

[19] Nikhil Somani, Caixia Cai, Alexander Perzylo, Markus Rickert, and Alois Knoll. Object recognition using constraints from primitive shape matching. In *Proceedings of the 10th International Symposium on Visual Computing (ISVC'14)*, pages 783–792, Las Vegas, NV, USA, December 2014. Springer.

[20] D. Song, C. H. Ek, K. Huebner, and D. Kragic. Task-based robot grasp planning using probabilistic inference. *IEEE Transactions on Robotics*, 31(3):546–561, June 2015.

[21] M. Stenmark, M. Haage, E. A. Topp, and J. Malec. Supporting semantic capture during kinesthetic teaching of collaborative industrial robots. In *2017 IEEE 11th International Conference on Semantic Computing (ICSC)*, pages 366–371, Jan 2017.