Semantic Agent Based Process Planning for Distributed Cloud Manufacturing

A dissertation presented to
the faculty of
the Russ College of Engineering and Technology of Ohio University

In partial fulfillment
of the requirements for the degree
Doctor of Philosophy

Arkopaul Sarkar
May 2020
© 2020 Arkopaul Sarkar. All Rights Reserved.
This dissertation titled
Semantic Agent Based Process Planning for Distributed Cloud Manufacturing

by
ARKOPAUL SARKAR

has been approved for
the Department of Industrial and Systems Engineering
and the Russ College of Engineering and Technology by

Dušan Šormaz
Professor of Industrial and Systems Engineering

Mei Wei
Dean, Russ College of Engineering and Technology
ABSTRACT

SARKAR, ARKOPAUL, Ph.D., May 2020, Mechanical and Systems Engineering

Semantic Agent Based Process Planning for Distributed Cloud Manufacturing (241 pp.)

Director of Dissertation: Dušan Šormaz

Cloud Manufacturing (CM) requires a manufacturing process planning systems that can generate a process plan for given product design, based on distributed manufacturing resources, offered as services in the cloud. As opposed to traditional process planning systems, the list of available resources and their current states (capability) cannot be known by the CM planning system beforehand. Distributed cloud-based resources cannot be discovered and integrated into a single application, as they are offered by geographically dispersed manufacturing vendors, using heterogeneous data exchange interfaces. This dissertation has developed a semantic integrated manufacturing planning system (SIMPlanner), suitable for CM, by integrating two methods: multi-agent framework and semantic data integration.

The novel design of the proposed multi-agent framework is based on the belief-desire-intention model. The agents use first-order logic based existential rules to automatically discover and integrate semantic services. The framework is designed to use a functional extension of SPARQL to integrate those rules into the agent-based framework. An ontology for manufacturing product design, resource capability, and planning knowledge is developed to provide a universal semantics. This semantics is used to annotate CM services, encode various components of the agents, as well as represent process planning information. SIMPlanner is built, based on the proposed multi-agent framework and the ontology-based information structure. A number of test part and industrial part designs are used to demonstrate that the prototype is able to generate process plans for their given design specification, respecting different process planning
criteria, such as process capability, alternative processes, required ordering in the application of processes, and precedence among features.
# Table of Contents

Abstract ................................................................. 3

List of Tables ......................................................... 8

List of Figures ......................................................... 9

1 Introduction .......................................................... 12

2 Literature Review .................................................. 20
   2.1 Evolution of Automated Process Planning Systems .......... 20
       2.1.1 Feature-based Process Planning ......................... 21
       2.1.2 Knowledge-based Process Planning ..................... 22
       2.1.3 Agent and Internet-based Process Planning ............ 25
       2.1.4 Service-based Process Planning ....................... 26
   2.2 Data Integration in Manufacturing Process Planning .......... 28
       2.2.1 Manufacturing Standards for Data Integration .......... 29
       2.2.2 Ontology-based Data Integration ....................... 32

3 Research Objectives ............................................... 36
   3.1 A Proposed Architecture of Cloud Manufacturing Planner .... 37
   3.2 Summary of Methods .......................................... 39
   3.3 Document Structure .......................................... 40
   3.4 Domain and Scope of Research ............................... 41

4 Semantic Multi-Agent System for Cloud Service Integration ..... 44
   4.1 Agent Formalism ............................................. 44
   4.2 Anatomy of an Agent ........................................ 51
       4.2.1 Belief of a BDI Agent .................................. 52
           4.2.1.1 Goal of an Agent ................................. 53
       4.2.2 Desire of a BDI Agent ................................ 53
       4.2.3 Intention of a BDI Agent .............................. 56
   4.3 Service Integration .......................................... 58
   4.4 Realization of Agents ....................................... 63

5 SIMPM- a Foundation Ontology for Manufacturing Planning ...... 70
   5.1 Effectiveness of Ontology in Data Integration ................ 71
   5.2 Ontology Design Methodology ................................ 72
   5.3 Product Design Ontology .................................... 73
       5.3.1 Representation and Specifications .................... 74
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.3.2</td>
<td>Part and Assembly</td>
<td>78</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Design Feature</td>
<td>80</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Dimension Tolerance</td>
<td>83</td>
</tr>
<tr>
<td>5.4</td>
<td>Process Capability Ontology</td>
<td>84</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Definition of Capability</td>
<td>86</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Taxonomy of Capability</td>
<td>90</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Capability Information</td>
<td>91</td>
</tr>
<tr>
<td>5.5</td>
<td>Manufacturing Planning Ontology</td>
<td>93</td>
</tr>
<tr>
<td>5.5.1</td>
<td>Manufacturing Process and Resources</td>
<td>95</td>
</tr>
<tr>
<td>5.5.2</td>
<td>Three Notions of Process</td>
<td>97</td>
</tr>
<tr>
<td>5.5.3</td>
<td>Input and Output of a Process</td>
<td>99</td>
</tr>
<tr>
<td>5.5.4</td>
<td>Theory of Manufacturing Planning</td>
<td>100</td>
</tr>
<tr>
<td>5.5.5</td>
<td>Structure of Manufacturing Process Plan</td>
<td>102</td>
</tr>
<tr>
<td>5.5.5.1</td>
<td>Aggregation Dimension</td>
<td>104</td>
</tr>
<tr>
<td>5.5.5.2</td>
<td>Variety Dimension</td>
<td>109</td>
</tr>
<tr>
<td>5.5.5.3</td>
<td>Time Dimension</td>
<td>112</td>
</tr>
<tr>
<td>6</td>
<td>Implementation</td>
<td>119</td>
</tr>
<tr>
<td>6.1</td>
<td>Overall Design</td>
<td>120</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Commands in SIMPlanner</td>
<td>122</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Mapping Prefixes to IRIs</td>
<td>123</td>
</tr>
<tr>
<td>6.2</td>
<td>Knowledge-base Management Agents</td>
<td>123</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Part Design Knowledge</td>
<td>124</td>
</tr>
<tr>
<td>6.2.1.1</td>
<td>Part Design Loader</td>
<td>125</td>
</tr>
<tr>
<td>6.2.1.2</td>
<td>Querying Part Design Graph</td>
<td>134</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Process Capability Knowledge</td>
<td>135</td>
</tr>
<tr>
<td>6.2.2.1</td>
<td>Process Capability Loader</td>
<td>136</td>
</tr>
<tr>
<td>6.2.2.2</td>
<td>Querying Process Capability</td>
<td>140</td>
</tr>
<tr>
<td>6.3</td>
<td>Process Planning Agents</td>
<td>145</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Part Planning Agent</td>
<td>149</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Feature Planning Agent</td>
<td>157</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Process Selector Agent</td>
<td>164</td>
</tr>
<tr>
<td>7</td>
<td>Evaluation</td>
<td>177</td>
</tr>
<tr>
<td>7.1</td>
<td>Setup for Evaluation</td>
<td>177</td>
</tr>
<tr>
<td>7.2</td>
<td>Plan for Evaluation</td>
<td>178</td>
</tr>
<tr>
<td>7.3</td>
<td>Experiment 1a</td>
<td>180</td>
</tr>
<tr>
<td>7.4</td>
<td>Experiment 1b</td>
<td>181</td>
</tr>
<tr>
<td>7.5</td>
<td>Experiment 2a</td>
<td>185</td>
</tr>
<tr>
<td>7.6</td>
<td>Experiment 2b</td>
<td>187</td>
</tr>
<tr>
<td>7.7</td>
<td>Experiment 3a</td>
<td>189</td>
</tr>
<tr>
<td>7.8</td>
<td>Experiment 3b</td>
<td>192</td>
</tr>
<tr>
<td>7.9</td>
<td>Experiment 4a</td>
<td>194</td>
</tr>
</tbody>
</table>
7.10 Experiment 5a ......................................................... 199
7.11 Experiment 6a .......................................................... 203
7.12 Experiment 6b .......................................................... 206
  7.12.1 Test 1 ................................................................. 208
  7.12.2 Test 2 ................................................................. 210
7.13 Experiment 6c .......................................................... 211

8 Research Summary and Future Work ................................. 216

References ................................................................. 224

Appendix A: Semantic Service Schema ............................... 235

Appendix B: Manufacturing Domain Level Taxonomy ............. 236

Appendix C: Example of Design XML file ........................... 240
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Mapping among feature and quality specifications</td>
<td>82</td>
</tr>
<tr>
<td>5.2</td>
<td>Machine and fixture assignments for various Step Activities</td>
<td>108</td>
</tr>
<tr>
<td>6.1</td>
<td>Mapping of prefixes and IRIs</td>
<td>124</td>
</tr>
<tr>
<td>6.2</td>
<td>Methods of the part design loader</td>
<td>127</td>
</tr>
<tr>
<td>6.3</td>
<td>Parameters for applying the capability information for a new capability</td>
<td>142</td>
</tr>
<tr>
<td>6.4</td>
<td>Associations among process and function instances</td>
<td>143</td>
</tr>
<tr>
<td>6.5</td>
<td>Global services offered by planning agents</td>
<td>147</td>
</tr>
<tr>
<td>6.6</td>
<td>The result of query in fig. 6.20 performed on the initial, intermediate, and terminating states of the example, shown in fig. 6.21</td>
<td>175</td>
</tr>
<tr>
<td>7.1</td>
<td>Number of alternatives for each features and number of occurrences at each level</td>
<td>215</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Structural view of proposed Cloud Manufacturing Planning system</td>
<td>38</td>
</tr>
<tr>
<td>4.1 Example of the proposed multi-agent framework</td>
<td>51</td>
</tr>
<tr>
<td>5.1 Relationship among physical product and its design</td>
<td>77</td>
</tr>
<tr>
<td>5.2 Versions of design: change of specifications for number of teeth</td>
<td>78</td>
</tr>
<tr>
<td>5.3 Semantic model of a sample part design with one hole feature</td>
<td>84</td>
</tr>
<tr>
<td>5.4 Process capability as a shared capability among machine and tool</td>
<td>90</td>
</tr>
<tr>
<td>5.5 Surface finish capability may be expressed as a measurement in micron or some standardized code</td>
<td>92</td>
</tr>
<tr>
<td>5.6 Capability describing range (maximum and minimum surface roughness)</td>
<td>94</td>
</tr>
<tr>
<td>5.7 Capability expressed as rule, which expects an argument belonging to the entity the capability</td>
<td>94</td>
</tr>
<tr>
<td>5.8 Common view of manufacturing process</td>
<td>95</td>
</tr>
<tr>
<td>5.9 Triad of manufacturing process, resource, and function</td>
<td>96</td>
</tr>
<tr>
<td>5.10 Manufacturing process planning information represented by three dimensions (variety, time, and aggregation)</td>
<td>102</td>
</tr>
<tr>
<td>5.11 A simple part design with four features and corresponding activity aggregation</td>
<td>108</td>
</tr>
<tr>
<td>5.12 Occurrence tree for the simple part shown in fig. 5.11</td>
<td>113</td>
</tr>
<tr>
<td>5.13 Feature precedence among the features of simple part</td>
<td>114</td>
</tr>
<tr>
<td>5.14 Extended FPN of SimplePart from fig. 5.11</td>
<td>117</td>
</tr>
<tr>
<td>6.1 Overall structure of SIMPlanner</td>
<td>121</td>
</tr>
<tr>
<td>6.2 Rule read-part-name.q</td>
<td>128</td>
</tr>
<tr>
<td>6.3 Rule read-part-features.q</td>
<td>129</td>
</tr>
<tr>
<td>6.4 Rule read-part-features.q</td>
<td>130</td>
</tr>
<tr>
<td>6.5 Rule read-dimension-for-feature.q</td>
<td>131</td>
</tr>
<tr>
<td>6.6 Rule read-dimension-for-feature.q</td>
<td>132</td>
</tr>
<tr>
<td>6.7 Rule read-tolerance-for-feature.q</td>
<td>133</td>
</tr>
<tr>
<td>6.8 Rule read-tolerance-measurement.q</td>
<td>134</td>
</tr>
<tr>
<td>6.9 Query result of sample part specification graph by part command</td>
<td>135</td>
</tr>
<tr>
<td>6.10 create new function</td>
<td>137</td>
</tr>
<tr>
<td>6.11 create new capability</td>
<td>137</td>
</tr>
<tr>
<td>6.12 Every capability demarcating the HoleMaking type of function for process type TwistDrilling</td>
<td>138</td>
</tr>
<tr>
<td>6.13 The capability information structure for type RoundnessCapability and PositiveToleranceCapability demarcating instance holemaking3118</td>
<td>140</td>
</tr>
<tr>
<td>6.14 Read processes and functions by script</td>
<td>141</td>
</tr>
<tr>
<td>6.15 Read capability by script</td>
<td>144</td>
</tr>
</tbody>
</table>
6.16 Control flow among planning agents. ................................................. 148
6.17 Rule to create the root feature specification as output of the root occurrence and associated root feature representation ................................. 150
6.18 Rule to assign root feature representation as the preceding feature representation of the initial feature representations ................................. 151
6.19 The patterns for the two rules given in fig. 6.17 and fig. 6.18. ................. 152
6.20 Rule to select features according to precedence among them .................. 155
6.21 RDF graph of initial, intermediate, and final state of occurrence tree for an example feature precedence. The legend on the left mark the type of the nodes and the relationships among them with different color codes, which is followed in the RDF graphs on the right. ................................................. 158
6.22 Slab making processes and the order of application ............................ 160
6.23 Optional and required precedence among the processes ....................... 160
6.24 Rule to generate the sub-occurrence tree for a given feature specification ... 162
6.25 Graphical representation of the rule fig. 6.24. .................................. 164
6.26 Graphical representation of the rule fig. 6.24. .................................. 165
6.27 transfer-feature-specification.rq ................................................... 167
6.28 Rule ‘specification-capability-matching-limit.rq’ for testing whether the feature has any specification whose value is contained by the range of a matching capability .................................................. 169
6.29 Graphical representation of the rule fig. 6.28. .................................. 170
6.30 Rule ‘create-intermediate-feature.rq’ for creating an intermediate feature. ... 172
6.31 Rule ‘create-final-feature.rq’ for creating a final feature ....................... 173
6.32 Rule ‘create-unsatisfied-feature.rq’ for creating a unsatisfied feature ....... 174

7.1 Part specification loaded into ‘part-1a.rdf’ ........................................ 181
7.2 Sub-occurrence tree of feature ‘Test-Hole’......................................... 182
7.3 The dimensions of output feature created by the planned occurrence for ‘Test-Hole’. ............................................................ 182
7.4 Part specification loaded into ‘part-1b.rdf’ ........................................ 183
7.5 Sub-occurrence tree of feature ‘Test-Pocket’ ..................................... 184
7.6 The dimensions of output feature created by the planned occurrence for ‘Test-Pocket’. ............................................................ 184
7.7 Part specification loaded into ‘part-2a.rdf’ ........................................ 185
7.8 Sub-occurrence tree of feature ‘Test-Slab’......................................... 186
7.9 The dimensions of output features created by the planned occurrences for ‘Test-Slab’. ............................................................ 187
7.10 Part specification loaded into ‘part-2b.rdf’ ........................................ 188
7.11 Sub-occurrence tree of feature ‘Test-Slot’......................................... 189
7.12 The tolerances of output features created by the planned occurrences for ‘Test-Slot’. ............................................................ 190
7.13 Part specification loaded into ‘part-3a.rdf’ ........................................ 191
7.14 Sub-occurrence tree of feature ‘Test-Pocket’ ..................................... 192
7.15 The tolerances of output features created by the planned occurrences for ‘Test-Pocket’. ................................................................. 193
7.16 Part specification loaded into ‘part-3b.rdf’. ................................................................. 193
7.17 Sub-occurrence tree of feature ‘Test-Slot’. ................................................................. 195
7.18 The tolerances of output features created by the planned occurrences for ‘Test-Slot’. ................................................................. 195
7.19 Part specification loaded into ‘part-4a.rdf’. ................................................................. 196
7.20 Sub-occurrence tree of feature ‘Test-Hole’ with specified true position tolerance 0.000144 mm. ................................................................. 197
7.21 Sub-occurrence tree of feature ‘Test-Hole’ with specified true position tolerance 0.000144 mm. ................................................................. 198
7.22 Part specification loaded into ‘part-5a.rdf’. ................................................................. 200
7.23 CAD design and inferred order of machining among features in ‘SimplePart-v2.rdf’. ................................................................. 201
7.24 Sub-occurrence tree of (a) RECTANGULAR_POCKET(3), (b) SLAB_SURFACE(2), and (c) RECTANGULAR_SLOT(7) ................................................................. 202
7.25 Sub-occurrence tree of SIMPLE_HOLE(4) ................................................................. 203
7.26 Complete occurrence tree, planned for part SimplePart-v2. ................................................................. 204
7.27 CAD design and inferred order of machining among features in ‘Netex’ part. ................................................................. 204
7.28 Specifications for Netex part loaded into ‘NetExample-features-prec.rdf’. ................................................................. 205
7.29 Complete occurrence tree, generated for ‘Netex’ part ................................................................. 206
7.30 Part ‘Slider’ with annotation marking its features (a), and precedence among them (b). ................................................................. 207
7.31 Complete order among features for planning conducted in test 1. ................................................................. 208
7.32 Occurrence tree for Slider part, following feature precedence, given in fig. 7.31 ................................................................. 210
7.33 Complete order among features for planning conducted in test 2. ................................................................. 210
7.34 Occurrence tree for Slider part, following feature precedence, given in fig. 7.33 ................................................................. 211
7.35 Part ‘NIST-CTC-01’ with annotation marking its features (a), and precedence among them (b). ................................................................. 213
7.36 Complete order among features of part ‘NIST-CTC-01’. ................................................................. 214

A.1 Schema for publishing semantic service. ................................................................. 235

B.1 Taxonomy of specifications of features and their dimensions, and tolerances ................................................................. 236
B.2 Taxonomy of CNC processes, machines, tools ................................................................. 237
B.3 Taxonomy of functions, borne by CNC machining resources ................................................................. 238
B.4 Taxonomy of capabilities, borne by CNC machining resources ................................................................. 239
1 INTRODUCTION

Manufacturing process planning faces new kind of challenges in the cloud manufacturing (CM) environment over centralized manufacturing. These challenges arise due to two distinctive requirements of CM: the multi-tenancy of distributed manufacturing resources and the automated provisioning of these resources. As described by Xun Xu [130] (definition 1), CM is a new paradigm of manufacturing, which uses shared resources from geographically dispersed locations. Moreover, CM adopts cloud computing and related service-oriented architecture (SOA) in order to achieve a high degree of automation and flexibility in disseminating these shared resources with minimal human intervention. In e-commerce sector, both cloud computing and SOA based platforms have contributed in dramatic transformation (e.g. Amazon EC2, Microsoft’s Azure, Google Cloud, and DigitalOcean). It is easy to imagine similar transformations occurring in the manufacturing world, if manufacturing resources can be offered on cloud platform to enable collaborative design and manufacturing.

Definition 1. CM is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable manufacturing resources (e.g., manufacturing software tools, manufacturing equipment, and manufacturing capabilities) that can be rapidly provisioned and released with minimal management effort or service provider interaction [130]

From the perspectives of manufacturing operation, economy, and business, CM provides some unique benefits over current manufacturing strategies. The following list addresses some of these advantages of CM.

- System integration: Most of the manufacturing technologies, focusing on automating different phases of the product life cycle, generally lack integration among them. The resulting “islands of automation” [4] prevents manufacturing...
industries to integrate their operations across domains and geographical regions. One of the primary aims of CM is to virtualize a wide variety of manufacturing resources by packaging resources - both physical hardware (e.g. machine, equipment, server, tool, material) and software (e.g. CAD, CAM, CAPP), as well as manufacturing capability (e.g. ‘know-hows’, data, standards, employees) - as services in the cloud, such as SaaS (Software as a Service), PaaS (Platform as a Service) and IaaS (Infrastructure as a Service) [130]. By virtue of this ubiquitous ‘servitization’, CM platforms provide an integrated environment for software and hardware applications to be deployed, accessed, and maintained seamlessly with much less undertaking in the maintenance of the compatibility of underlying systems.

- **Data exchange**: A number of industry-wide standards, protocols, and best practices have been adopted by manufacturing industries to facilitate information exchange among internal divisions as well as external partners. However, current manufacturing standards are opinionated, not flexible for an extension, overlap in their target area of application, and suffer for lack of updates from standardization bodies.

On the other hand, being a rendition of Cloud Computing for the manufacturing domain, CM adopts a similar agenda of Service Oriented Architecture (SOA)\(^1\). Following the practice of SOA, information among manufacturing resource services can be exchanged via extremely generic meta-data standards such as WSDL (derived from XML), and JSON-LD, which can represent virtually any type of data structures. Furthermore, manufacturing partners may communicate with the

---

\(^1\) David Sprott and Lawrence Wilkes from Microsoft defines the vision of SOA as: “The policies, practices, frameworks that enable application functionality to be provided and consumed as sets of services published at a granularity relevant to the service consumer. Services can be invoked, published and discovered, and are abstracted away from the implementation using a single, standards-based form of interface” [118].
cloud-hosted resources using web-based communication protocols, such as RPC, SOAP, and HTTP based REST, which are agnostic to the content of the messages. By using these web-based protocols, data communication in CM may be protected by the same level data security, provided by cutting edge encryption methods and cyber-attack prevention measures, applied to most of the web-based transactions.

- **Mass Customization**: As mass-produced goods saturated market demand, manufacturers were gradually compelled to offer product variability, ending up tilting the ‘sellers’ market to ‘buyers’ market [57]. In order to offer product variation and cost-effective at the same time, modern factories employed new manufacturing practices, such as Lean manufacturing, KANBAN, and deferred assembly. Current challenges in mass customization of products are: complexity of control and interactions among sub-systems, difficulty in planning and scheduling of operations, and unpredictability of demand forecasts [14]. Undoubtedly, the multi-tenancy of manufacturing resources lets the manufacturers offer more variability and upgrades in their products with low prior investment. New product lines may be established without a dedicated production system. Furthermore, cloud-based collaboration lets manufacturers and customers share ideas and feedback at almost every step of manufacturing, facilitating customization and tailoring of products.

- **Distributed manufacturing**: At present, manufacturing enterprises face many challenges while expanding production into new locations. First, restrictive and uncertain government policies often demand major adjustments in their business model, forcing manufacturers to produce locally and depend on local resources. Secondly, the labor market has become volatile in recent years, due to the increase in labor cost in the Asian countries \(^2\), migration of low-wage workforce into

\(^2\) https://www.bworldonline.com/rising-wages-said-eroding-aliasas-competitiveness/
high-income countries [23], and demographic transitions, such as aging, in various countries [100]. By integrating manufacturing businesses of every scale in a common ecosystem, CM provides availability and access to quality resources from anywhere let businesses expand and contract in new markets as well as respond to the changing demand in shorter turnaround time. Such a level of flexibility in manufacturing operations has been rightly described by Wolfgang Heinrichs [50] as “Design Anywhere, Manufacture Anywhere” (DAMA). Furthermore, CM is poised to democratize the manufacturing industry [53], by bringing manufacturing resources within the reach of inventors, amateur designers, small businesses, and start-ups.

- **Fourth industrial revolution**: The philosophy of CM also contributes to the vision of smart manufacturing (in the USA), and Industry 4.0 (in Germany), which proposes a network of decentralized, loosely coupled, and autonomous manufacturing modules, in place of rigid and hierarchical control among manufacturing operations [40, 83]. CM provides a virtual manufacturing environment, in which autonomous agents, self-adaptive optimization, and machine learning techniques can be applied in order to achieve an intelligent self-healing, self-adaptable, and self-improving manufacturing execution system [27]. Digital Manufacturing Commons, which is being developed by MXD (formerly DMDII), USA, is an example of an open platform for manufacturing services. Smart Manufacturing Leadership Coalition (SMLC) and National Institute of Science and Technology (NIST) have also undertaken the development of similar open-source community test-bed and virtual manufacturing laboratories.

A number of high-level architectures for CM, proposed by researchers, employ modular and multi-layer approaches to integrate a heterogeneous type of manufacturing
services, where each layer is dedicated to aggregate and control the group of services from the layers below[119, 122, 128]. In spite of these proposals for the architectural and infrastructural requirement for CM, a practical realization of CM still needs to address the following challenges [130].

- Integration of distributed manufacturing resources
- Discovery and selection of manufacturing resources
- Service-oriented manufacturing applications
- Fault tolerance
- Load Balancing
- Security and protection of intellectual property
- Service metering

Among these challenges, the first two are critical for the fruition of CM. Although this dissertation is not about providing a comprehensive solution for CM, a process planning application for CM environment, which this dissertation attempts to build, also needs to address these challenges due to the following reasons.

First, the provisions of the CM services, including discovery and selection, need to be automated. This is because, unlike centralized manufacturing, where the list of available resources is known, CM environment may host any number of resources, offered by different manufacturing partners. Moreover, these resources need to be selected rapidly and utilized on-demand. The resource configuration in CM may even need to be flexibly adjusted in response to any change in their states. Second, both CM services and manufacturing information (design specification, process plan, configuration, controls) need to use a common data model. Without such a unified data model, CM services
cannot be integrated onto a common manufacturing control. This is for the reason that the CM services may use different schema for expressing their input and output structure, as they are offered from heterogeneous sources. More technically, this particular problem is related to the classical data integration problem, i.e. how to universally map information from heterogeneous data sources. These two inherent challenges of CM also influence the process planning task for this type of environment.

Manufacturing process planning lies at the heart of the agenda: “design anywhere, manufacture anywhere” [50], as such a collaborative and virtual production planning system will make cloud manufacturing viable for practical application in distributed production. The task of process planning concerns how a product can be made in the best possible way, that is to generate one or more valid sequences of processes set forth in a set of specification, given in product design [45].

In Computer-Integrated Manufacturing (CIM) environment, Computer-aided Process Planning (CAPP) acts as a bridge between Computer-aided Design (CAD) and Computer-aided Manufacturing (CAM). An efficient manufacturing process planner for CM environment may enable cloud manufacturing customers to compare alternative production plans. In fact, a large number of alternative means of production is highly expected to be generated by CM process planner, as centralized CAPP systems consider only pre-configured knowledge and a much smaller set of available resources. This is due to the availability of diverse kind of manufacturing resources and production methods, offered as services on a cloud platform. Furthermore, the cloud manufacturing planner needs to be sensitive enough to capture the dynamically changing conditions of CM resources, e.g. shop-floor capability, machine-tool health and in-process part and it’s material. In order to satisfy the goal of multi-tenancy (of asset-ful manufacturing resources), the planning system for CM needs to evaluate the varying degree of capability.

\footnote{digitization of manufacturing based on CAx applications, e.g. CAD, CAM, CAPP, CAQ}
for cloud-based distributed manufacturing resources against a certain manufacturing task. Therefore, the planning system needs to consider two different types of information: specifications contained in the product design and manufacturing resources along with their capability. Being sourced from different manufacturing partners, these two types of information may be modeled in various kinds of heterogeneous data models, which are often incompatible in nature.

Considering these requirements of the CM process planning system, this research needs to answer two fundamental questions: (1) How to develop a distributed manufacturing planning system, capable of automatically discovering and orchestrating cloud-based manufacturing services in order to perform process planning for given product design?, (2) How to build an integrated knowledge model, suitable for describing manufacturing resources in the cloud with universally interpretable semantics? The first requirement may be satisfied by a suitable framework, which is able to select manufacturing CM resources by matching their capability with the given set of specifications in the product design. However, the second requirement is crucial for a complete realization of such process planning system for CM, as the integration of heterogeneous systems needs a unified data model. This is not a new problem and various types of common protocols, standards, and languages have been prescribed and are currently in use for the facilitation of system integration in general, including manufacturing systems. These data modeling techniques come with their own set of shortcomings nonetheless, such as conflicting definitions, overlapping coverage, and incompatible meta-model among standards, published by different governing bodies. Moreover, manufacturing vendors often conceive different notions regarding the manufacturing standards and best practices, resulting in semantic inconsistency and thus difficulty in the integration of heterogeneous sources of data. Ultimately, this lack of tools and techniques for integrating CM resources precludes the widespread adoption of CM in
Industry. Therefore, the primary goal of this Dissertation is to build a manufacturing process planning system, which is suitable for addressing some of these concerns. Preliminary results from this dissertation have been published in the following papers.


2 LITERATURE REVIEW

Majority of the automated CAPP systems can be classified based on their adopted methods, such as object-oriented approaches, GA-based approaches, neural-network-based approaches, Petri net-based approaches, fuzzy process planning, feature recognition or feature-based approaches, knowledge-based approaches, agent-based approaches, and internet-based approaches [132][135]. Considering the relevance to cloud-based architecture, section 2.1 mainly focuses on the agent and internet-based methods. Before delving into these two approaches, feature and knowledge-based methods are presented in section 2.1.1 and section 2.1.2 respectively in order to evaluate what kind of information a CM planner needs to consider for planning. As mentioned in the previous chapter, the integration of heterogeneous sources of data is a major concern for a CM planning system. section 2.2.1 includes a brief review of the existing data models, standards, and protocols. We conclude this chapter by an overview of current developments on ontology-based data integration in section 2.2.2.

2.1 Evolution of Automated Process Planning Systems

The importance of the nonlinear process plan concepts, including dynamic and flexible process plans, has been identified and considered as a benchmark for process planning and integration by past researches. Generative CAPP systems, which adopted rule-based generation of process plan based on granular specifications in a product design as opposed to earlier CAPP systems of variant type, came around early 80s. The importance of linking the product designing and manufacturing tasks as well as adoption of the AI techniques in generative process planning research was understood by most of the early researchers [3, 47]. The primary contribution of the concurrent research was the identification of the general workflow of CAPP, which are part interrogation, process
selection, machine selection, cost and time calculation. Most of the later developments consider these developments as the foundation of manufacturing process planning.

### 2.1.1 Feature-based Process Planning

Feature-based design allows a more direct extraction and interpretation of design information directly from CAD systems, enabling tight integration of design and manufacturing. Still, many product designs do not necessarily contain feature data, especially in the older design practice from the 1990s and 2000s, which brought forth interest in automatic feature recognition for product designs in CAD formats. However, feature recognition is a complex problem and in itself an independent research area. In a survey conducted by Babic et al. in 2008 [9], a number of automatic feature recognition methods are surveyed. These methods can be broadly be classified as: graph-based, rule-based, automata-based, syntactic pattern-based, volume decomposition (convex hull, cells), and hint-based. With the advent of reliable feature recognition methods, feature-based process planning attracted many researchers. Han et al. proposed to integrate feature recognition with process planning [48]. Xu and Hinduja recognized the link between feature information and the series of machining operations needed to produce that feature, such as roughing, semi-finishing and finishing operations [131]. In the same year, Sormaz and Khoshnevis (1997) published an integrated process-planning system, using feature precedence reasoning using graph-based methods and volume-based methods [115]. Feature-based CAPP methods were applied to different domains: Sadaiah investigated on prismatic features [95], and Hwang and Miller on planning CNC machining operations for prismatic parts [52]. In recent years, Abu and Md Tap (2010) presented an approach to feature recognition, using rules based on different characteristics specific to each feature [1], and Garcia et al. [39] introduced a method, that utilizes
feature-based modeling for defining both geometric and non-geometric information, which is matched by a set of a high-level capability.

These researches showed that feature-based process planning is an effective way to integrate design information with manufacturing (also in accord with DFM practice). In a CM environment, feature-based planning contributes in two aspects of planning: first, information on the applicability of the processes for manufacturing a type of feature may help the planner to narrow its search to only certain kind of services; second, more granular distribution of tasks may be achieved by breaking the task of manufacturing the complete product into the tasks of making each feature. In this way, only a portion of the complete manufacturing plan needs to be changed if the specification of one feature is changed or originally planned resources become unavailable. Altogether, this strategy may help CM planner to achieve a high degree of flexibility in on-demand re-configuration of the plan. However, a CM planner also needs to address how the geometry, dimension, and tolerance information of the features can be captured, and the mapping from feature to the process can be embedded in the planning system. Some of the proposed information structures from the past studies are mentioned in section 2.2. However, most of these structures are proprietary and each focuses on one particular domain of manufacturing.

2.1.2 Knowledge-based Process Planning

Early knowledge-based CAPP systems remain a popular branch of CAPP research since the 1980s. Several CAPP systems, e.g. GARI by Descotte and Latombe [32], EXCAP by Davis and Darbyshire, SIPP by Nau and Chang, and XPLAN by Lenau and Alting, all of which were developed in the early 90s, can be classified as knowledge-based expert system (KBES). A framework of data integration of CAD, CAM, CAPP, NC and MRP systems was developed by Dhamija et al., based on Unified Data Meta Model (UDMM) [33].
One important characteristic of these systems is the use of knowledge base and rules for inference mechanisms. Due to lack of AI tools and computing power of the hardware, earlier KBES systems employed frame-based, object-oriented and entity-relation data models for building knowledge bases whereas the rules were written in first-generation programming languages, such as FORTRAN, BASIC, PROLOG, and LISP.

Long-standing research in knowledge-based CAPP was initiated by Sormaz and Khosnevis [112, 114], which followed some forbearing works, such as HiMapp [13] and RTCAPP [84]. RTCAPP [58] represents the planning knowledge using frames, which links information on processes, machines, and tools, and a cost analysis system for concurrent part and process design. The 3D planning model, described in section 5.5.4, was developed based on these early investigations.

Along with the progress of technology, a number of knowledge based CAPP systems were developed for different manufacturing domains, leveraging technologies such as rule-based systems, object-oriented languages, and data modeling schema. Some of these CAPP systems are: knowledge-based system EXCATS for selection of cutting tools and configurations of turning operations by Arezoo et al. [6], object-oriented representation of setup information, process application knowledge, and operation control knowledge by Jia et al. [54], another object-oriented representation for planning knowledge by Grabowik and Knosala [44], a holistic process-planning model, including technological and business concerns by Denkena et al. [30], a hybrid approach, consisting knowledge-based rules and geometric reasoning rules to derive sequence from prismatic feature interactions by Liu and Wang [68], a planning approach for 3D rotational parts by Zhang et al. [136], a hybrid procedural and knowledge-based approach to address feature interpretation and recognition by Marchetta and Forradellas [73], a conceptual model, based on methodology of knowledge acquisition (MOKA) to manage different level of planning knowledge by Helgoson and Kalhori [51], and a knowledge-based CAPP system for NC
control tool path for shoe models by Chen et al. [66]. As a continuation of the earlier work of Sormaz and Khoshnevis, IMPlanner system was augmented with CNC machining specific process selection rules [116][110]. During this time, an object-oriented feature and process model was implemented in IMPlanner, in order to capture the hierarchy of domain-specific manufacturing resources and act as an extensible schema, which can be adopted to represent manufacturing processes and design features from diverse domains.

Open formats such as Extensible Markup Language (XML) were successfully utilized to integrate different phases of manufacturing in some of these researches. Sormaz et al. addressed the technologies necessary to achieve distributed, integrative product configuration and mass customization, such as knowledge engineering, alternative process selection, visualization of decision space, product, and process by XML-based data exchange [111]. Many new additions to IMPlanner system, such as integration to Siemens NX CAD system, MasterCAM, and display of design and planning models in a graphical user interface [108] were achieved by an underlying XML schema. XML data structure is also compatible with object-oriented data structure, as the former can be easily generated from a set of classes and their properties, such as java bean objects. XML structures are also suitable for service-based communication, as most of the web-services exchange data in markup languages. However, different manufacturing services may use different XML models to format their communication structures. In order to integrate these services into a single planning system, these heterogeneous data models need to be converted into a unified data model. In order to avoid the manual translation of data models, a common vocabulary needs to be established, which may be used as a canonical model for every manufacturing services in the cloud. Past research in data integration adopted two strategies to develop a unified data model for manufacturing, which we discuss in section 2.2.
2.1.3 Agent and Internet-based Process Planning

Agent-based architecture was recognized as an effective way to realize adaptiveness and dynamism of process planning. Different approaches used in this area mainly focus on negotiations between intelligent agents. Gu et al. [46] implemented an agent society with hierarchical structure and a market-based negotiation strategy, to enable the bidding process between higher-level agents to lower-level agents. McDonnel et al. [76] segregated part management agent and resource handler agents and implemented a recursive auction to facilitate negotiation between these two groups. In a distributed AI-based framework for process planning, proposed by Shih and Srihari [102], the entire production control task is generally broken into several sub-tasks, each of which is implemented by an intelligent agent. Denkena et al. [31] proposed a negotiation based agent society to decentralize various manufacturing activities needed for transforming a centralized rough or template process plan. Both of the above research implement a bidding or negotiation strategy, which is governed by one supervisor agent by applying some rules of engagement on the participating agents. The application of a centralized decision-making process makes a multi-agent system less open to adopt. Moreover, every decision taken by the centralized agent is transitory - as every decision is based on the state of the system at a certain point of time - which makes the system irreversible.

In an agent-based process planning system, called IDCPPS [25], the process planning tasks, such as initial planning, decision-making, and detail planning, is performed collaboratively by distributed agents. Sormaz et al. adopted a service and task-based agent interaction to build mobile agents called task agents, which can search for eligible service agents in a distributed environment to perform the services necessary for different manufacturing activities [42, 98]. In the survey of agent-based manufacturing process planning and scheduling [101], it is shown that currently most of the agent-based planning and scheduling systems use bidding-based or market-like negotiations among agents.
However, the authors of the survey identified the necessity of more robust interaction mechanisms, real-time data exchange, and measurements regarding privacy and security in the agent-based process planning systems.

Recent investigation in agent-internet based process planning mainly focuses on multi-agent systems for achieving decentralization, such as agent-based production environment for mass customization by Blecker and Graf [15], planner with resource independent operation summary to support collaborative manufacturing by Kulvatunyou et al. [64], and web-based collaborative product development for distributed digital manufacturing (DDM), called WebMAS, by Mahesh et al. [72]. Sarkar and Sormaz [109] enhanced the IMPlanner system with Jade-based multi-agent framework, called IMPlanner-MAS, for distributed manufacturing process planning. Soon after, Ghosal and Sormaz developed a multi-agent based simulation platform for flexible manufacturing planning and scheduling, by integrating Anylogic simulation platform to IMPlanner-MAS [42].

In spite of achieving a certain level of decentralization, these multi-agent systems relied on domain-specific data structures for exchanging information among them. In order to avoid using a proprietary data structure, IMPlanner-MAS uses FIPA\(^4\) compliant interaction protocols. However, FIPA interaction protocols are not compatible with REST or SOAP-based web protocols, which prevents the agents in IMPlanner-MAS to interact with cloud services directly.

2.1.4 Service-based Process Planning

CM borrows many components from web and agent-based distributed manufacturing paradigms. Wu et al. [128] mentioned that CM differs from its precursors for two fundamental reasons: multi-tenancy (ubiquitous, on-demand access of a shared pool of

manufacturing resources), and virtualization (encapsulation of wide range of 
manufacturing resources as service). Researchers [119, 121, 129] have designed a number 
of CM architectures to address the multi-tenancy issue. In spite of detailing the hosting, 
provisioning, and management mechanisms, these architectures still do not address how 
the cloud services can be automatically discovered, orchestrated and invoked. Instead, 
these researches leave those decisions to the users, while focusing on creating a 
user-friendly environment for facilitating such manual efforts.

In order to facilitate automatic service discovery, Martin et al. [74] suggested Web 
Ontology Language (OWL) to describe input and output parameters of a service with 
semantic markups, which can be subjected to various reasoning algorithms in order to find 
a suitable service, by evaluating the similarity of the signature of the service to the input 
data. Three ontology-based meta-model, for describing web-services with semantics, are 
Web Service Modeling Ontology (WSMO) [92], Semantic Web Service Language 
(SWSO) [61], and OWL-S [74]. Among them, SWSL and OWL-S are recommended by 
W3C.

The main components of OWL-S are service profile structure, containing service 
name, human-readable description, and contact information, service model structure, 
containing the behavior of the Service in terms of processes, control constructs, and 
groundings to bind input and output variables with the process model as well as WSDL. 5. 
Apart from describing the services structurally, WSMO also provides a meta-language, 
called Web Service Modeling Language (WSML), to let users attach goals and mediators 
to the services. While OWL-S only provides structure to the service provider to advertise 
the service, WSMO provides users to specify their expectations from the service through 
goals. Both OWL-S and WSMO describe the meta-structure using some kind of formal 
languages. However, these languages are not generic as they are built by keeping the

5 Web Service Definition Language - a de facto markup language for describing web service end points
proposed structures in mind. On the other hand, SWSL makes clear a distinction between the ontological meta-structure for describing the services and the language for writing the service profile, composition, and orchestration. Whereas OWL-S strictly follows OWL-DL logical formalism and WSMO follows a layered language, largely based on F-logic and compatible with DL, SWSL tries to capture the full flavor of first-order logic. Moreover, SWSO follows the process specification language (PSL) [99] based axioms to express the composition and orchestration of services. Regarding the composition and orchestration of services, these frameworks slightly differ in their approaches. OWL-S describes the composition of services by classifying tasks of service invocations into simple, atomic, and composite processes, but does not provide details on how services can cater to different process flows. WSMO clearly distinguishes between service composition and choreography, the latter of which describes different behaviors of one service in different situations. Based on the fluent calculus, SWSO defines service composition and orchestration in a real-time scenario.

In spite of the semantic richness provided by these semantic web service frameworks, little is said about the discovery of services. However, the semantic annotations helped researchers in coming up with a number of matchmaking strategies to discover a service, based on its preconditions [86], signatures [62], and capabilities [2], advertised by the service providers. Realizations of ontology-based mapping and automatic matchmaking layer in CM require advancement in semantic service discovery techniques as well as manufacturing ontology [55][43][120] to formalize various manufacturing concepts.

2.2 Data Integration in Manufacturing Process Planning

Researchers in CAPP realized that an efficient process planning system should provide end to end planning for the complete production process, which gave rise to the concept of integrated process planning. One of the major challenges in accommodating
different phases of the product life cycle on a single platform is data integration. Some of the process planning systems, which integrates different phases of product life-cycle, are integrated system of CAPP (ICAPPS) [79] for single-piece, small-lot and make-to-order production, tool management system [78] for selecting alternative tools for scheduling, and integrated process planning and scheduling tool [28] based on IDEF methodology.

However, integration of product design data with the manufacturing process and resource data is extremely important for a planning system, as the planning is conducted based on a given set of design specification. Moreover, modern product development promoted a high level of communication among the team of designers and manufacturers as designers often require designing a product in such a way that it is easy to manufacture and they need frequent feedback on the manufacturability of a design. This practice is often called design for manufacturing (DFM). In the next section, we discuss past studies in feature-based process planning, in which the plan for a complete design is generated by aggregating plans for its features.

2.2.1 Manufacturing Standards for Data Integration

Industries adopted a number of manufacturing standards, usually published by governing bodies and manufacturing consortia, as a mean for data integration in distributed manufacturing. Since 1984, ISO 10303\(^6\) has published a number of application protocols (AP) covering product information in diverse manufacturing domains, such as AP203 (Sheet metal and die), AP210 (electronic assembly), and AP236 (furniture assembly) (STEPTools, n.d.). Two popular AP 203 and 214, which are supported by most of the leading CAD applications, have recently been merged into AP242. In the recent wake of DMfg, AP242 sits at an important position in the product manufacturing landscape and is able to integrate both product design and manufacturing. It provides

\(^6\) an ISO standard for the computer-interpretable representation and exchange of product manufacturing information, colloquially known as STEP (Written in a modeling language called EXPRESS)
extensive ranges of pre-defined 3D machining form features as well as explicit and
parametric surface and 3D shapes along with models for Product and Manufacturing
Information (PMI), which defines functional and structural requirements (GD&T, surface
texture, finish requirements, process notes, material, welding symbols, other annotations).
STEP AP224 data model, which can bridge CAD and CAPP systems, already found its
use in some integrated planning systems [134][88]. In parallel to ISO STEP standards,
American Society of Mechanical Engineers (ASME) published standards for product
definitions and 3D view of surface, volume and contours. ASME Y14.41 defines the core
terminologies of engineering drawing and practices, which has recently been enhanced
with a detailed GD&T information model (Y14.5). NIST conducted a detailed study [67]
on the capability of commercial CAD software to support ASME Y14.5 [67]. STEP-NC
(AP238) is a well-known extension of STEP standard to represent the tool-path and
control instructions, which was largely done by using G-Code. STEP-NC integrates the
CAM systems with CAD and CAPP, which already have well-defined STEP standards
(AP203, AP214, AP242) [133]. AP239 of STEP standard is designed to hold product
life-cycle data along with support, maintenance, repair of complex products and systems
such as ships, aircrafts, engines, and oilrigs. AP239 also extends STEP PDM module for
retrofitting.

It can be argued, however, that a schema built on consensus among members may
also fulfill the purpose of data integration. But such consensus is bound to be local as a
proprietary schema cannot be extended or reused flexibly. These types of schema are also
not durable, as the original intent and rationale behind such schema may evolve over time.
On the other hand, ontology-backed information models are durable, reusable, and most
importantly provide far superior quality. A list of drawbacks, which often plagues the
traditional data structures, is given below.
• Entities in traditional data schema, e.g. UML, often lack proper truthmakers (positive criteria for the existence of an entity), mostly because such data structures are conceived from local view of a domain. For example, in many databases, entities like seller and buyer are associated with an ID, whereas a proper truthmaking analysis of these entities will necessitate the presence of at least one product which the seller wants to sell or the buyer wants to buy.

• Data items have an insufficient ontological commitment in their definition, thus making them difficult to map from one structure to another. While building a data structure for an application, modelers concentrate on fulfilling the requirement of only that application and ignore the relationships needed to define items universally. For example, one system may store only the geometrical information of a 3D object, while another application may need topological information for that object.

• ‘Use-mention distinction’ - the distinction between using a word to refer to something in reality, and mentioning the same word in order to say something about this word itself - is often overlooked in the traditional data models. It is quite common to find the telephone number is being used to refer to the telephone owned by an agent. The processes, for which a machine is suitable, are then stored as codes for process type (polishing, painting, drilling, cutting), and then used to refer to the actual processes themselves. Using proprietary eponyms (brand names used by cultures to suggest the generic artifact itself, e.g. Band-aid, Velcro, Xerox) is a closely related problem, which may often creep into a data structure.

• Traditional data structures do not maintain any thesaurus or glossary for the data item names. Modelers often use terms from their own language as well as social or organizational culture to name data items. Due to the unavailability of proper
translation, thesaurus or glossary of such geographically disparate the data schema need a high level of manual analysis and communication to achieve alignment.

- Terms with polysemy and homonyms make the original intention of the data modeler difficult to understand. This problem is also exacerbated by the lack of glossary in data schema. For example, the word ’man’ may refer to the human species or a particular gender of that species.

- The most problematic data items are those which use an abbreviation, cryptic naming conventions, and jargon. Without any documentation, data structures, containing data items of these sorts, are extremely difficult to interpret. For example, SAP application stores sales order header information in a table called VBAK, with data items named as: VBELN (sales order number), AUDAT (document creation date), and VKORG (sales organization).

### 2.2.2 Ontology-based Data Integration

Utilization of ontology for knowledge-based process planning slowly gained the attention of researchers around mid-2000. Two of the notable early knowledge modeling efforts are: foundation of core ontology in manufacturing by Borgo and Leitão [20] and formalization of product life cycle semantics (PSRL) by Patil et al. [85]. Since then, a number of manufacturing ontology were proposed, for capturing both upper level and domain-specific manufacturing concepts. ADACOR is developed based on DOLCE\(^7\) foundation ontology to facilitate a holonic manufacturing platform [20]. Open assembly model (OAM) proposed by NIST, defines generic concepts to integrate requirement, functional design, kinematic synthesis, and tolerance analysis along with basic part and assembly information [38]. OAM extends the object-oriented Core Product Model (CPM), an earlier model proposed by NIST [35].

\(^7\) Descriptive Ontology for Linguistic and Cognitive Engineering, http://www.loa-cnr.it/Ontologies.html
These efforts were mostly based on UML data models, which even though cannot be qualified as ontology, still provided definitions of some core concepts of the product model. However, the models in these studies focused on a physical product structure, which is not product design in the true ontological sense. In an attempt to provide semantic to STEP standards, ONTOStep provided an ontology to map the STEP functions in OWL model [12]. ONTO-PDM is an ontology developed based on STEP PDM to define fine-grained knowledge structure for product assembly, bill-of-material, and material types [82].

One of the notable ontological studies on product design was conducted by Sanfillipo et al. by expressing a clear distinction between product and its design by something they called ‘semiotic triangle’, which made associations among three different concepts: product type as the intention of the design, specification as the physical description, and product as the extension of such intention [96]. The distinction between the product type and the class of product, presented in their study, allures to the ‘form’ from Platonic idealism – a notion, which contemporary ontological approaches often avoid categorizing in a separate class. Moreover, the instances of the design specification are categorized as ‘Concept’, which makes it difficult to use them for quantitative evaluation of the properties of a physical product, without any suitable translation from the conceptual layer to the physical layer.

Early attempts to build ontologies for manufacturing focused on describing products, manufacturing processes, resources, and controls, that described relations among resources and processes. One of the early works on the core terms of manufacturing was conducted by Borgo and Leitão, who proposed a set of such core concepts related to manufacturing, based on foundation ontology DOLCE [21]. The behavior and function of the manufacturing artifacts were elucidated in their later works [18, 19, 22]. Although these studies established the functionality of products and resources based on behaviorism
and provided some of the conceptual guidelines for understanding capability, they are not exclusively about the manufacturing capability.

MASON ontology proposed by Lemaignan et al. tried to capture the precedence relationship among different manufacturing operations required to produce a part with the desired quality [65]. It also links material, machine, tools, and cost information with every manufacturing operation, thus providing upper-level ontology to represent the complete material requirement and process plan for an assembly. Assembly Relation Model (ARM) proposed by Kim et al. describes a number of widely used assembly processes, such as joining, bonding, riveting, welding, and their sub-processes [59]. Ameri et al. [4] developed an ontological model describing manufacturing processes and resources as services, which included a detailed study on the capability of machine and tool. They also conducted a thesaurus-guided discovery of capability related terms [94] as well as developed a process and resource selection module leveraging the capability knowledge [94]. Lu et al. [69] developed a taxonomic representation of manufacturing processes and resource capabilities, by directly translating concepts from STEP-NC (ISO) standards, using the OntoSTEP methodology, proposed by Barbau and Krima [12]. Luo et al. developed a DL based language called MCDL, which utilized the fuzzy-DL concepts in modeling capability ranges for manufacturing resources [70]. In a recent publication, Järvenpää et al. described a new ontological model for manufacturing capability, named MaRCo [55], which supports capability match-making for resource discovery.

However, most of the existing ontology is limited to a particular industrial domain or some phases of product life-cycle. This is due to the fact that most of the ontology models are built by a bottom-up approach for a particular scenario, and do not comply with upper-level categories. Fragmented ontology models create taxonomy forest (disjoint set of hierarchy trees), overloaded concepts (multiple conflicting definitions for a single term), and closed world models (considering only domain-specific truth). Such
inconsistencies are major impediments in the comprehensive use of ontology in the industry as they often require expensive effort in ontology mapping.

In order to avoid the proliferation of inconsistent ontology models, Industrial Ontology Foundry (IOF)\(^8\) – a joint collaboration among academia, industry, and NIST (National Institute of Science and Technology), started developing a set of reference ontology for manufacturing and supply-chain. The goal of this effort is to come up with standardized definitions of a list of cross-domain terms, which can serve as upper-level reference for domain-level ontology modeling. It is being envisaged that this upper-level ontology will be available to every industry for building taxonomy and semantic models for their scenario. However, the task of developing such cross-domain ontology is extremely challenging due to the diversity of domain-level requirements of multi-disciplinary manufacturing practices [63].

\(^8\) https://www.industrialontologies.org/
3 RESEARCH OBJECTIVES

The importance of manufacturing planning in the realization of a practical cloud manufacturing (CM) system is discussed in chapter 1. Two fundamental questions, posed in section 3.1, are central to the design of a distributed and autonomous CM planning system. The review of past researches on manufacturing process planning, presented in section 2.1, shows that the agent-based methods provide decentralized planning, autonomous execution, and self-governance, all of which are crucial in meeting the requirements of a CM planner. However, current multi-agent systems do not integrate well with the service-oriented architecture, which is a core component of the CM environment. On the other hand, an effective CM planner needs to automatically discover, orchestrate, and invoke manufacturing services. Although such type of automated service integration is proposed in the field of cloud computing, as discussed in section 2.1.4, no CM planner has implemented the same for cloud manufacturing services.

The survey on knowledge-based process planning, presented in section 2.1.2, shows that manufacturing planning is a knowledge-intensive task, as it needs to be aware of the capabilities of available manufacturing resources, as well as the suitability of manufacturing processes for accomplishing a certain design specification. Moreover, a CM planner needs to consider the heterogeneity of the data models, used for encoding this information. As a solution, a common set of semantics for describing the different concepts of manufacturing needs to be established. In section 2.2.1, we discussed many shortcomings of traditional manufacturing standards. In a quest to achieve more flexibility and integrability, this study adopts ontology as a canonical model for integrating heterogeneous data models. Furthermore, such CM planner may become capable of identifying and invoking distributed CM services, if the manufacturing services are decorated with the same set of semantics. Unfortunately, available manufacturing
ontology - so far proposed in the literature, which are reviewed in section 2.2.2 - are mostly built by independent researchers with little to no link with each other.

Due to these reasons, in order to achieve the ultimate goal of this dissertation, that is to build a suitable process planning system for CM, a generic multi-agent based planning architecture and a canonical semantic model for the service-oriented data storage required to be developed first. Specific goals of this research thus can be summarized in the following objectives:

1. Design and development of a multi-agent system, which can autonomously discover, orchestrate, and invoke CM services to communicate with each other and make real-time decision in order to accomplish tasks collaboratively.

2. Development of a manufacturing ontology based on upper-level ontological concepts as consistent descriptions of various terminologies of product design, manufacturing resources, and process planning.

3. Development of a manufacturing planner based on the proposed multi-agent framework for the generation of a set of alternative ways of manufacturing a product. The agents in this planner will use the proposed ontology model to match the process capabilities of a diverse array of CM resources for a known design specification and encode relevant planning knowledge in order to generate the plan.

In order to contextualize the aforementioned goals in terms of the general vision of CM, an architecture of a hypothetical CM planner is presented below.

3.1 A Proposed Architecture of Cloud Manufacturing Planner

Based on the four layers of CM architecture [130], the big picture of a possible architecture of a CM planner is given in fig. 3.1. The bottom-most layer consists of the physical manufacturing resources, commonly known as the machine layer. The semantic
layer on top of the machine layer contains the ‘manufacturing ontology’, which stores the
terminologies (T-Box) for various manufacturing concepts as well as the domain level
concepts in a hierarchical set of semantic models. This layer also stores the manufacturing
information as facts or assertions (A-Box) in the ‘knowledge-bases’ with semantic tagging
derived from the terminologies. The gear wheel, placed in between these two boxes,
denotes various reasoning algorithms, which assures the validity and semantic consistency
of the facts with respect to the terminologies.

Figure 3.1: Structural view of proposed Cloud Manufacturing Planning system
Manufacturing planning is actually controlled by the multi-agent layer, which consists of a society of agents and CM service stack. Every CM service is designated to perform a certain operation, such as computation, query, and machine interaction. These services can behave like a traditional web service or initiate a complex operation by triggering one or more agents. These same services can also operate physical CM resources by virtualizing machine interaction and control. The agents in this layer refer to the semantic layer to detect the changes in knowledge-base due to some action of users or an invocation of some service(s). Every agent is capable of running a set of rules which in turn can invoke one or more CM service, as guided by states of the knowledge-base. In the course of execution, the knowledge-bases are changed by rules, thus triggering more agent and service interaction until the state of the knowledge-base reaches a goal. The end-users of the proposed CM planning system occupy the top layer. These users can be both cloud vendors (e.g. planner, manufacturers, suppliers, transporters), or cloud customers (e.g. designers, innovators, corporate leaders, sales managers). Vendors usually offer their capabilities in the form of launching new agents and services in the manufacturing cloud. The process planning system, developed as part of this research, is based on the multi-agent system and the manufacturing ontology, which are situated in the two middle layers of this architecture.

3.2 Summary of Methods

The proposed design of multi-agent system is unique in two fundamental ways: first, the communication among agents relies on service-oriented data exchange protocols, enabling the agents to freely communicate among each other as well as other native services without changing the communication protocol or message structures; second, the agents allow the use of ‘existential rules’, which can integrate a suitable service by matching the input and output structure of the service with the variables from pre and post
conditions of the rule. In this way, every agent of the proposed multi-agent system is not only capable to discover and invoke available services, but also able to delegate tasks among each other by service-oriented communication. The anatomy of each agent is designed based on ‘belief-desire-intention’ (BDI) model, which proposes an algorithm to direct every agent towards a predefined goal. A practical system is developed based on the proposed design by using RDF to encode the facts and SPARQL CONSTRUCT queries to encode the rules, which is augmented with a formalism for service integration.

In order to develop the manufacturing ontology, named ‘semantically integrated manufacturing planning model’ (SIMPM), most of the upper-level concepts are derived from the foundation ontology, called ‘basic formal ontology’, especially the terminologies in design specification and resource capability. However, the definitions of process and planning related concepts are formulated based on another foundation ontology, called ‘process specification language’. Based on the upper-level concepts, taxonomies of a number of items from Computer Numerical Control (CNC) based processes, resources, and their capabilities are also prepared. These domain level extensions not only act as validations for the proposed upper-level concepts, but also help in building a practical planner as part of this study. In order to evaluate the planner, taxonomies for prismatic features and their specifications are also developed.

### 3.3 Document Structure

Each of the next three chapters of this document describes the methodologies to achieve the three research objectives, given in chapter 3. In section 4.1, the proposed architecture of the BDI-based multi-agent system is presented. Before presenting the rigorous definitions in section 4.2 of different components of a single agent, the overall design is described in section 4.1 with an example. Discussion pays more attention to the ‘plan’ structure, which uses both ‘horn’ and ‘existential’ rules. In section 4.3, the
theoretical grounding for the integration of semantic service with an existential rule is established. Details of a practical implementation of the proposed agent structure are given in section 4.4.

Chapter 5 starts with a list of advantages ontology brings in data integration over other models. Next, the definitions of the upper-level concepts are presented, especially for the concepts used in SIMPM. In section 5.3, product design and its specification related terminologies are defined. Manufacturing process capability is defined in section 5.4. The last section of this chapter is devoted to describe the concepts of manufacturing planning.

A practical implementation of a manufacturing planner, called ‘Semantically Integrated Manufacturing Planner’ (SIMPlanner) is described in chapter 6. This chapter contains three main sections: section 6.2.1 for describing methods to convert design specification into a knowledge-base, section 6.2.2 for describing reading and writing mechanisms of process capability knowledge in knowledge-base, and section 6.3 for describing planning related agents and their structures.

Evaluation of the SIMPlanner is conducted by testing the accuracy of the plans generated for a number of sample part design containing specifications for various types of prismatic features. Different sets of process capabilities of various types of CNC machining are loaded in the planner for this purpose. Setups, configurations, and results of the experiments are discussed in chapter 7.

3.4 Domain and Scope of Research

Considering the fact that present-day manufacturing practices are extensive and diverse in nature, this research has limited coverage of manufacturing domains. Most of the examples, used in the evaluation of the planner, are from common Computer Numerical Control (CNC) machining, mainly drilling, and milling. These methods are
suitable for shaping prismatic part designs. CNC machining is considered as the most popular traditional manufacturing process, which covers a barrage of material removal and shaping methods.

The process planning paradigm adopted in this research also assumes that product designs can be expressed as a collection of machinable features, which can be ultimately broken down into simple 3D design features, such as Hole, Pocket, Slots, Bevel, and Chamfer. Whereas this particular study will only focus on process planning, a number of past studies may be referred to more information on topics, such as feature-based design, machinable feature recognition from CAD-based product designs, and translation to various representational formats, such as STEP (ISO10303). Some of these researches are discussed in chapter 2.

This research also assumes that the machine-tool status, their capabilities, and conditions are updated to the knowledge-bases by machine interface, which sits in between the semantic and machine layer in fig. 3.1. Machine interfacing technologies such as Virtual Machine Monitor (VMM), MTConnect protocol based SCADA, and machine learning may be applied to build this interface. However, such interfacing methods are beyond the topic of this research.

Protection of intellectual property, security from data leakage, and malicious cyber attack are natural concerns for industries regarding CM. A slew of researches is currently engaged in devising cloud security in cyber-physical systems, Internet-of things, and cloud services. Current security measures include building trust models among service providers and consumers [128], assurance of quality of service using service level agreement [130], and various protocols for user authentication, identity check, and information flow tracing [49]. However, these topics are not part of this study as it is

---

9 non-traditional methods include additive manufacturing processes, e.g. 3D printing
assumed that appropriate security measures are adopted for every kind of communications among these cloud based services.

Manufacturing services can also offer virtualized Machine-Tools and other resources for real production. As this research concentrates on manufacturing process planning, no topic related to manufacturing execution and control are covered. Furthermore, only the most cost-effective plan should be selected for manufacturing some products, although many alternative plans may be generated by the process planning system, developed as part of this research. In order to deduce such an optimum process plan out of many alternatives, a suitable optimization algorithm needs to be applied. The optimization of process plan is also out of the scope of this research.
4 SEMANTIC MULTI-AGENT SYSTEM FOR CLOUD SERVICE INTEGRATION

This chapter describes an architecture of the semantic aware multi-agent system, that is designed based on the ‘Belief-Desire-Intention’ (BDI) [91] model. Each agent in this multi-agent architecture uses semantic knowledge-bases as their set of belief, a novel type of logical statements to form their plan structure, and a reasoning algorithm to form the intention structure. Apart from augmenting formalism with belief represented as ontology-driven facts, the majority of the work is invested in defining the plan structure, which is adopted from the existential rule. The most interesting feature of this existential rules is their ability to integrate semantic web services into them. We derive three outstanding benefits from this design: first, the agents can communicate each other based on homogeneous SOA protocols, second, these semantic web-services are invoked automatically by matching the input and output from the pre and post conditions of the rule, and third, a reasoning algorithm can infer the ordering of invocation (automatic orchestration) with the help of a reasoning algorithm. In the next section, the architecture of the proposed multi-agent system is formally presented, before expounding on the individual agent structure.

4.1 Agent Formalism

The principal characteristics of a software agent, given by Wooldridge [126], are: “(1) an autonomous system, making decisions based on its internal state, (2) situated in an environment and being able to perceive it in order to react to the changes, (3) able to take the action and exhibit goal–directed behavior, (4) able to interact with other agents and to co-operate”.
The awareness of an agent of its environment is emphasized by a similar idea, as Norvig [93] states that the agent is “... capable of perceiving the environment through the sensors connected to it”. In a more practical interpretation, we can say that an agent can retrieve various types of information about the environment as well as perceive any change in that information over time. In a CM environment, the environment is mostly comprised of various manufacturing resources. Moreover, the state of these resources, such as the health of a machine, the availability of a tool is constantly changing. The design specifications, which is an input of a manufacturing planner, may go through modifications. In this type of environment, agent-based systems provide more flexibility than traditional computational procedures or shells, as they assume that the input conditions do not change over time. ‘Blackboard’ [29] architecture was one of the early attempts to infuse the awareness of the environment in software modules. In this architecture, multiple ‘knowledge sources’ use a globally shared data structure (blackboard) and reach a decision collaboratively. The control mechanism in blackboard architecture is centralized, often called a ‘control shell’. In contrast, the agents do not use a global repository and their control mechanisms are self-contained. Undoubtedly, decentralization of control brings more flexibility. We have to remember that the manufacturing cloud is an open environment, where any resource can provide some services.

BDI formalism, developed by Rao and Georgeff [91] takes a more realistic viewpoint, concerning “how an agent’s beliefs about the future affect its desires and intentions” [127]. In theory, the BDI formalism adopts the modal logic to express the relationship among these three modalities: belief, desire, and intention. However, the possible worlds of a BDI agent are represented as snapshots of states arranged in a tree structure, in which the agent can transit through by making a decision at branching [91]. For this reason, a BDI agent is required to store the content of its belief and corresponding
action for only the current node in the tree of possible worlds for each iteration. This allows BDI agents to use first order logic to represent the modalities, as formulated in the work of Meneguzzi and De Silva [77].

BDI formalism found widespread application in both academics [17] and industry in the areas of robotics, flight control, space exploration, industrial and military training, and medical diagnostic [87]. Many agent programming languages have also been built based on this formalism, namely Jack [125], jadex [89], and AgentSpeak [90]. BDI based multi-agent system is also used in developing a number of abstract planning methodologies, such as simple task network and hierarchical task network planning. We design the CM planner based on STRIPS (Stanford Research Institute Problem Solver [37]) formalism for its effectiveness in handling first principle planning.

The environment of the proposed multi-agent system is inhabited by three types of entities: a global knowledge-base, consisting of a set of knowledge, multiple agents and a registry, which stores the signatures of a number of web services. The global knowledge-base may either be empty or contain some facts at the initiation of the world. However, this global knowledge-base may change as new facts may be added to it by agents. Formally the global knowledge-base can be described as a series of indexed set of states, i.e. $W = W_0, W_1, W_3, \ldots$. Each agent also possesses a knowledge-base, which is local to that particular agent, e.g. $b_i$ is the local knowledge-base for $i$-th agent, residing in the world. This local knowledge-base makes up the agent’s world view, which at a certain time point contains some facts from the global knowledge-base as well as some facts, generated from the internal activities of the agent. The local knowledge-base of an agent is not persistent, as agents can refresh the local knowledge-base after a transaction is completed. Here, a transaction is defined as the set of operations an agent performs in between the time it receives a service request and the time it generates a response for that request. Before an agent clears the facts from its local knowledge-base, some portion of it
may be transferred to the global knowledge-base for persistence. In that case, the global knowledge-base moves from the current state to the next.

In the proposed model, the services can either execute an external procedure or initiate a transaction for one of the agent. Any of the registered services may be invoked by an external application or another multi-agent system. The same services are also automatically invoked by one agent for communicating with another agent of the same world. The request, denoted by \( I \) and response of the service \( O \) thus equivalent to the incoming and outgoing messages of an agent. Every agent subscribe to the registry by registering the service(s) it offers and in turn can look up other services. It is to be noted that agents are not aware of other agents in the world but can access them only through the services they offer. In this way, the exact location and hosting details of an agent remain anonymous and can only be referred by one of service endpoints. Thus we can say that every agent offers one or more services. In this section, however, we consider that each agent offers only one service for the sake of simplicity. This assumption lets us consider the services as the identifiers for the agents. We can say that the registry contains one or more services, denoted by \( S = s_1, s_2, s_3, \ldots \), acting as surrogates for the agents in the system.

Following the BDI formalism, every agent in this environment is composed of a set of plans \( P = p_1, p_2, p_3, \ldots \), which are, in a simpler term, some kind of recipes to perform some actions. The plans are the central subject of discussion which we will take up in later sections. The plans are only meaningful in a particular context, which is provided by the local knowledge-base of the agent. Therefore, one or more plans become viable when the world in the agent’s view creates the necessary condition for the plans to perform. The abstract description of such a necessary condition is called ‘precondition’ of the plan. Apart from the precondition, the plan also contains instructions for how an action can be performed if the precondition is satisfied.
The model prescribes four types of internal activities that may occur in an agent in the course of a transaction.

- **Belief update on request:** When a new service request is received, the local knowledge-base of the agent is updated by querying the global knowledge-base based on some information contained in the request. This operation is denoted by \(\delta(I) = \{w|w \subseteq W_i\}\), where \(I\) is the request received and \(W_i\) is the state of the global knowledge-base when that request is received. At the completion of this operation, the local knowledge-base is updated with the result of \(\delta\), i.e. \(b_1 = b_0 \cup \delta(I)\), where \(b_0\) is the initial state of local knowledge-base before the request is received and \(b_j\) is the state of the local knowledge-base after \(\delta\) is performed. \(b_0\) may be empty if we assume that the agent refreshes the local knowledge-base after every transaction.

- **Plan selection:** For every change of state of the local knowledge-base occurring during the transaction, the preconditions of every plan of the agent are matched against the local knowledge-base at its current state, and one of the plans is selected, whose precondition returns true. This operation is denoted by \(\rho(b_i, P) = \{p_i|p_i \in P\text{ and } p_i \text{ is ‘applicable’ to the i-th state of the belief } b_i\}\). It may occur that multiple plans are selected as each of their preconditions are matched with the current state of the local knowledge-base. In a naive way, one of the plans can be selected randomly. However, it is prudent to apply some kind of goal-based selection strategy, which we discuss in section 4.2.3.

- **Service integration:** A plan selected by \(\rho\) may integrate a service from the available list of services from the registry. This operation applies suitable methods to discover such service and invokes it. We denote this operation as \(\lambda(p_i, S) = \{s_k|s_k \in S\text{ and } s_k \text{ is ‘integrable’ to plan } p_i\}\). However, this operation is optional as not every plan integrates a service. The structure of a plan itself determines whether such a plan
can be performed without invoking any service. The necessary criteria for a service to be integrable with a plan are discussed in section 4.3. If a plan $p_i$ integrates a service $s_k$, then $\langle p_i, s_k \rangle$ refers to the plan $p_i$, in conjugation with $s_k$.

- **Belief revision:** This operation updates the local knowledge-base after a plan $p_i$ is successfully performed by adding the new facts created by each plan. Precisely, this set of new facts are generated due to the effect of the plan denoted by $\langle p_i \rangle_{effect}$. This operation is called $\Delta(p_i, b_i) = \{b_j | b_j = b_i \cup \langle p_i \rangle_{effect}\}$. If the plan $p_i$ integrates a service $s_k$, then $\Delta(p_i, b_i) = \{b_j | b_j = b_i \cup \langle p_i, s_k \rangle_{effect}\}$

Additionally, the agent may update the global knowledge-base with a subset of the facts in local knowledge-base, as described earlier. This operation is also optional and opted by the agent if it requires to persist some of its evaluated knowledge in local knowledge-base. The agent transfers these facts to global knowledge-base after it has completed a transaction, that the local knowledge-base is in a state which contains information necessary to create the desired response.

The service provided by the agent can respond to the request if and only if the intended output information can be found in one of the resulting states of the local knowledge-base in the course of the internal activities described above. From the descriptions of operations above, it can be noted that every such change in the state of local knowledge-base may occur when a plan is selected by operation $\rho$, and performed successfully by operation $\Delta$. For a plan, which integrates a service, operation $\lambda$ needs to find a matching service before $\Delta$ can be performed. These two situations are mutually exclusive and only one of them can happen for every change of state. We can write these two situations, using algebraic notation as, $\Delta(\rho(b_i, P), b_i) = b_{i+1}$, and $\Delta(\langle \rho(b_i, P), \lambda(\rho(b_i, P), R) \rangle, b_i) = b_{i+1}$. For the agent to respond in a timely manner, it can be assumed that the local knowledge-base will contain the set of facts from which the
response can be derived. In a rigorous term, the response $O$ is a set of information which are a portion of the interpreted model of the local knowledge-base at some state $k$, i.e. $I(b_k)$.

$$O \subset I(b_k) \Rightarrow (\delta(I) = b_1 \subseteq W_I) \land \bigwedge_{i=1}^{k-1} (\Delta(\rho(b_i, P), b_i) = b_{i+1}) \lor (\Delta(\langle \rho(b_i, P), \lambda(\rho(b_i, P), R) \rangle, b_i) = b_{i+1})$$

The above description of the multi-agent system is schematically presented in fig. 4.1 with help of an example. In this example, the participating three agents are shown along with their internal and external communications in the purview of one particular transaction. The global knowledge-base is shown in blue and the local knowledge-base (belief) of each agent is shown in orange. The services offered by the agents are named as $s_1$, $s_2$, and $s_3$. The first agent contains two plans $p_{11}$ and $p_{12}$, the second agent two plans $p_{21}$ and $p_{22}$, and the third agent one plan $p_{31}$. The number in the first place of the two place identifier of $p$ (in the subscript) refers to the index of the agent and the number on the second place refers to the index of the plan. On receiving a request $I_1$, the first agent applies $\delta_1$ over the information embedded in $I_1$ to update its belief with facts from the global knowledge-base. Then it applies $\rho_{11}$ which selects plan $p_{11}$. The plan $p_{11}$ integrates a service, and operation $\lambda_{11}^{11}$ selects the service offered by the second agent. The superscript of $\lambda$ refers to the identifier of the plan and the subscript refers to the agent, which the service triggers. It then applies $\Delta_{11}$ to execute the plan. In order to execute the plan, $\Delta_{11}$ needs to invoke the service, which receives request $I_2$ from plan $p_{11}$. The second agent again applies $\delta_2$ to update its belief, then perform two plans in sequence by applying operations $\rho_{21}$, $\Delta_{21}$, $\rho_{22}$, and $\Delta_{22}$ in succession. The first pair of operations selects and executes plan $p_{21}$ while the second pair selects and executes plan $p_{22}$. None of these plans integrates any service. As a response is returned from the second agent, the first agent updates its belief with the effect of plan $\langle p_{11}, s_1 \rangle$. Then operation $\rho_{12}$ selects plan $p_{12}$, which again integrates a service. This service is provided by the third agent, which
executes its only plan upon receiving request $I_3$. This plan $p_{31}$ also integrates a service, which calls some other agent, which we did not show in this diagram. On receiving response $O_3$, the first agent updates its belief with the effect of plan $\langle p_{12}, s_2 \rangle$. The response $O_1$ then can be created from the facts in this state of the first agent’s belief. The second agent updates the global knowledge-base after it applies both of its plans. The state transition of global knowledge-base is shown by the darkening shade of its color.

Figure 4.1: Example of the proposed multi-agent framework

### 4.2 Anatomy of an Agent

In this section, the abstract definitions of the structure presented above for a single agent, are further represented with the formal system of first-order logic. Such
construction also helps in explaining how an agent performs the internal operations, that is explained in the last section. The formal definition of a BDI agent, given by Meneguzzi [77], takes three modalities, that are belief, desire, and intention of an agent, into account along with a goal and a set of events. As explained in the previous section, every agent is triggered by a service invocation. Therefore, the agent starts its operation on receiving an incoming message, that contains the input values used when the service is invoked. We consider that an event is triggered when the agent receives such a message and is the only type of event the agent acknowledges. This event changes the state of the belief by transferring some facts from the global knowledge-base as well as set the goal of the agent. In the next section, we prescribe the structure of the belief and the goal of an agent.

### 4.2.1 Belief of a BDI Agent

The belief of a BDI agent corresponds to the local knowledge-base, mentioned in section 4.1. The local knowledge-base changes over time during the transaction, which can be represented by a sequence of states $B = b_1, b_2, b_3,...$. At a certain point, an agent may have access to only one of the states, $b_i \in B$. The belief contains a set of facts, each fact being a variable free first-order formula. More rigorously, a set of facts may be expressed as a set of *conjuncts*, each of which is a set of ground atoms (contains only constants and no variable), joined by conjunction [80].

It is to be noted that the global knowledge-base is another repository of facts and after the belief update operation $\delta$, every atom in the belief is also present in the global knowledge-base. This is because the incoming message contains some information (constants in formal terms), which is used to search all matching atoms (containing the constant as one of the arguments). In this way, the agent is able to capture any change in the global knowledge-base, which may occur in between the service is invoked and the
agent receives the service. The operation $\delta$ can be implemented by any type of suitable query and kept open to the implementation detail.

### 4.2.1.1 Goal of an Agent

The goal of an agent is to produce a response upon receiving a service request. As the output structure of the service is pre-determined, the goal of the agent is set for each transaction. The goal is expressed as a formula with some free variables whereas the rest of the variables are mapped by the constants from the incoming service request for a particular transaction. In other words, this type of formula is called *partially grounded*. We can say that the agent reaches its goal (for a transaction), when the variables of the partially grounded formula can be completely mapped from the local knowledge-base. Every time the local knowledge-base is updated by the belief revision operator $\Delta$, it is checked whether such mapping(s) can be derived. We express this partially grounded formula as a *conjunctive query* (CQ) [80].

We can also define the state of belief, when the CQ representing the goal can be answered. A closed-form of CQ is called a boolean CQ (BCQ) [80], which contains no free variable. We can say that the agent has reached its goal, when there is a state $i$ of belief $b_i \models Q$, where $Q$ is the closed-form (BCQ) of the goal.

### 4.2.2 Desire of a BDI Agent

The agent updates its belief by executing one or more plans - sometimes repetitively - which can be viewed as decomposition of the goal into smaller goals. In this sense, the plans can be viewed as tasks to achieve these smaller goals. For example, a coffee brewing agent provides a service which receives coffee beans and returns the brewed coffee. In order to brew the coffee, the agent needs hot water and ground coffee. Although the brewed coffee is the ultimate goal of the agent, it first needs the water to be of a certain temperature and the coffee beans to be crushed to a certain granularity. At the atomic level
of the decomposition, this granular goal can be expressed as a plan, which is defined in
definition 2 (borrowed from the author’s previous publication [97]). A plan recommends
some action, which might be performed only in a certain situation. This preferred
situation for performing an action is expressed as a set of preconditions. The result of the
action is added to the belief if successfully executed.

**Definition 2.** A plan \( p \vdash \langle \text{pre}, a, \beta^+, \beta^- \rangle \) where \( a \) is a pure function with no side effect, \( \text{pre} \)
is the set of preconditions and must be ‘true’ for the action to be invoked, \( \beta^+ \) is the set of
new facts to be added to belief and \( \beta^- \) is the set of existing facts to be deleted from the
belief, if the action is “successful”, i.e. there exists a function \( \Delta \), defined as
\( \Delta : b_i \times p \rightarrow b_{i+1} \); where,

\[
b_{i+1} = \begin{cases} 
(b_i \setminus \beta^-) \cup \beta^+, & \text{when } \Delta \text{ is successful} \\
b_i, & \text{otherwise}
\end{cases}
\] (4.1)

[97]

Here, the function \( \Delta \) is considered pure in a sense that they do not change the belief
of the agent. However, the function may invoke a service, which can trigger another agent
in turn. In that case, the belief of some other agent as well as the global knowledge-base
may be changed by the \( \Delta \).

The questions remain: how to check if the precondition of a plan is true? What is the
content of \( \beta^+ \)? For simplicity, we consider that the belief of an agent is always
monotonically increasing, i.e. no fact is deleted from the belief.

In this model, either of two types of first-order rules is used for expressing a plan.
These are range-restricted positive rule (Definition 5 in [80]) and existential rule
(Definition 6 in [80]). Both of these types have two parts, each being conjunctions of
atoms and called the body and the head of the rule (denoted by \( \text{body}(r) \), and \( \text{head}(r) \),
where \( r \) is an first-order rule). In general, when applied to a fact, rules can infer a new set of atoms (\( \beta^+ \)) based on the existing atoms in the fact.

It can be observed that the existential rule differs from Horn rule by the presence of the existential variable in its head. However, both are closely related, since by the technique called ‘skolemization’ [24], an existential rule can be transformed into a set of Horn clauses with functions. We leverage this technique to integrate a service with an existential rule, which is elaborated in section 4.3. Besides, this transformation yields a reduction, which helps in reducing the entailment problems related to existential rules to entailment problem on a set of Horn clauses. In general, it is easy to observe that the conditional part of a first-order rule acts as the set of preconditions for it.

A rule fires whenever the conditional (body) part of the rule is true in the current state of the belief. We can say that the rule is ‘applicable’ in that condition. The modification of fact as a result of a rule application may be explained by borrowing the concepts of substitution and homomorphism (Definition 1 in [80]). Briefly, a rule is applicable if the homomorphism contains at least one mapping for each variable in the conditional part of the rule. By the application of a FOL rule \( r \), new facts are produced by projecting the derived homomorphism (set of mapping) onto the conclusion part of the rule, i.e. \((head(r))\). This set of new facts forms the content of \( \beta^+ \) from definition 2. Formally, if homomorphism \( h^r_b \) is derived by applying rule \( r \) to a set of belief \( b \) then \( \beta^+ = h^r_b(head(r)) \). This rule application is said to be redundant if \( h^r_b(head(r)) \) is empty.

The application of rule is equivalent to perform the action (\( a \) in definition 2). As we express a plan by a FOL rule, we can say that the action, defined for the plan, is available if the corresponding rule is applicable. A subtle notion of action is that an agent does not necessarily perform an action, even if such action is available. The agent may or may not perform such action. In the next section, we discuss the choice of an agent regarding the actual execution of an action.
It can be observed that the application of an existential rule \( r \) returns a set of partially grounded atoms. This is because there are existential variables in \( \text{head}(r) \) and \( h'_b \) contains the mappings for the frontier variables only. In order to derive the mappings for these existential variables, a suitable service is invoked as part of the action. The integration of such service and mapping of existential variables are discussed in section 4.3.

### 4.2.3 Intention of a BDI Agent

In the classical BDI model, the intention of an agent is described as its preference to execute actions. This preference signifies agent’s deliberative component, as intention structure lets the agent select a particular action from multiple possible alternatives [91]. In the last section, the action of an agent is defined as the application of a rule. At a certain state of belief, multiple such rules may be applicable. In this situation, the agent applies a set of ‘preference criteria’, which induces an priority-based order among the available actions. Generally, different sets of ‘preference criteria’ may be applied to different groups of actions. Following this notion, definition 3 provides a formal definition of the intention of an agent, which is formulated in one of our earlier work [97] closely following the definition, given by Meneguzzi [77].

**Definition 3 (Intention).** An intention structure \( I = \{I_1, I_2, \cdots \} \) is a finite (possibly empty initially) set of intentions, where each intention \( I_i \) is a tuple \( (\rho, \langle A, \prec \rangle) \), where \( \rho \) is a set of ‘preference criteria’, and \( \langle A, \prec \rangle \) is an partially or totally ordered set of actions, such that every action in set \( A \) belongs to complete set of available actions [97].

The simplest possible intention is to have no ‘preference criteria’, which lets an agent to execute every available action in parallel. Another type of intention may apply a random selection of an available action as the ‘preference criteria’. The preference criteria may also prescribe some conditions, which may be tested against the different properties
of action, e.g. the mappings in the homomorphism, type of service to be invoked, and service communication-related parameters.

The priority among the available actions may be automatically deduced by searching the derivation path from the applicable rules to the goal. The derivation path is a sequence of the states, which the belief goes through, when one or more rules are applied to it repetitively. This sequence of derivations is called $R$-derivation (definition 8 from [80]).

The derivation sequences can be created in breath-first forward-chaining search, which is also known as ‘oblivious chase’ [10]. Starting from the initial state of belief $b_0$, at every step, different beliefs can be produced by deriving new homomorphism from the body of different rules to each current beliefs and then applying them. A homomorphism, derived from the body of a rule to a belief at a certain step, is a new derivation if it is not computed in any other derivation from root to the current belief. The set of belief obtained after $k$ such steps is called $k$-saturation of $b_0$. In other words, $k$-saturation of a belief $b_0$ is all beliefs fund at the leaf of the derivation tree of $k−1$ level, of which $b_0$ is the root.

If the goal $Q$ can be answered from one of the belief from the $k$-saturation of the initial belief $b_0$, then the agent has achieved its goal. Strictly speaking, if the conjunctive query $CQ$, representing goal, can be answered by $b^j_k$, which is one of the belief from the $k$-saturation, then there exists a homomorphism from $CQ$ to $b^j_k$ and all answer variable in $CQ$ is contained by the homomorphism. The ‘provenance path’ for $b^j_k$ is the sequence of rules applied to derive $b^j_k$ from initial belief $b_0$; in other words, the path from $b^j_k$ to root in the derivation tree.

The provenance path may be used to decide the priority of the rules. At every step, the next applicable rule is selected on the basis that it produces a new belief, which is on the provenance path. In this way, the belief of the agent gets closer to the goal state, as it moves from one step to another.
However, the construction of an intention structure based on the derivation sequence suffers some serious roadblocks. This is because, a derivation tree can be infinite if existential rules are used as part of the plan, thus not decidable[10]. For forward chaining search, some classes of rules, called ‘finite expansion set’, are known to produce a decidable derivation tree [10, 11]. Investigation on such rules is not part of this study.

4.3 Service Integration

As discussed in the last section, a plan of an agent integrates a service, if written using existential rule. This is because the homomorphism, derived from the body of the rule to the belief, does not contain the mappings for the existential variable. Therefore, the application of rule produces only partially grounded facts. The mappings for these existential variables can be derived by invoking a service. However, a formal notion of these mapping needs to be developed in order to show that such mapping does not violate the entailment of the corresponding semantic model. Furthermore, such mapping illustrates how a service can be discovered and invoked autonomously by a plan represented as an existential rule.

As we described earlier, the existential rules are different from Horn rules because one or more variables may exist in the head which is not introduced in the body. An existential rule with unknown variable may also be transformed into a Horn rule by introducing functions (also called ‘skolem variable’) by applying the method of Skolemization[24]. This method may be used to map the input and outputs of a given function to the existential rule. In order to map the input and output of a function to an existential rule with unknown variables, the input and output of the function need to be semantically tagged. We call such function as semantic service, borrowing the concept from work in semantic specifications of web services, such as Business Process Execution Language for Web Service Modeling Ontology (WSMO) [92], Semantic Web Service
Language (SWSL) [61], and OWL-S [74]. These works define semantic service with different schema, which can be summarized in the following definition.

**Definition 4.** Semantic web services is defined as an executable function or method (usually a web-service) with a profile described by the tuple \( \langle S^I, S^O, S^P, S^R \rangle \), where \( S^I \) is a set of typed input variables, \( S^O \) is a set of typed output variables, \( S^P \) is a set of conjunctive atoms as precondition, \( S^R \) is a set of conjunctive atoms as the result. The service is invoked if the precondition is true at a certain state of the knowledge-base.

The primary purpose of the profile is the specification is the precondition describing the conditions that must be satisfied for the service to be invoked, and the post conditions, describing the effects after the service is successful or returns value. The typed input and output variables can be described by unary predicates, which are atoms with a single variables.

The main idea behind our approach is to derive a mapping of the input, output, precondition, and result of service to the plan. From given the definition 2 of a plan, it can be observed that the precondition of the service \( S^P \) can be represented by the body of the rule associated with the plan \( \text{body}(\text{rule}(p)) \) in toto, whereas the result of the service \( S^R \) can be represented by the \( h(\text{head}(\text{rule}(p))) \), where \( h \) is a homomorphic mapping between \( \text{body}(\text{rule}(p)) \) to the current state of the belief \( b_i \). If such mapping succeeds, the plan may successfully invoke the service (action) [97]. This also opens up the possibility of discovering cloud services by semantically matching the types of input and output concepts with the semantic types of the variables in an existential rule. The following technique of mapping input, output, precondition, and result (IOPR) of the service to variables of an existential rule is paraphrased from the author’s earlier work [97].

In horn rules, frontier variables are treated as universally quantified, with their scope limited to a given rule. As usual, only variables that occur in the antecedent of a rule may occur in the consequent. However, we propose that an existential rule with existentially
qualified variables in consequent, which never occurred in antecedent of the rule, can be converted into horn rule by removing every such existentially qualified with a suitable service invocation, which can map its inputs to the frontier variables of the corresponding rule. Such mappings are derived by applying Skolemization technique [24]. In this model, a function is specifically a cloud service, which is also assumed to be "pure" (cannot change belief by itself) and computable. The following proposition shows how we can transform an existential rule to a range restricted horn rule by mapping the existentially qualified variable with a suitable service. In other words, if such suitable service exists, then the IOPR of the service can be mapped to the variables of the corresponding existential rule.

**Proposition 1.** Let $\mathcal{H}$ be a existential rule of form

$$\bigwedge_{i=1}^{k} a_i \rightarrow \bigwedge_{j=1}^{p-1} \tilde{a}_j \land \tilde{a}_p \land \bigwedge_{j=p+1}^{m} \tilde{a}_j,$$

where $a_i$ is the $i$-th atom of total $k$ atoms in the antecedent, $\tilde{a}_j$ is the $j$-th atom of total $m$ atoms in the consequent of the clause, $\tilde{a}_p$ contains at least one variable, which does not occur in any of the $k + p - 1$ literal before it. Assuming atom is free of negation, $\mathcal{H}$ can be transformed into a DL-safe rule $\mathcal{H}_{safe}$ if there exists a service $S = \langle S_I^T, S_O^T, S_p^T, S_R^T \rangle$, where $S_I^T \subseteq V$, $S_O^T = v_p$, $S_p^T = \bigwedge_{i=1}^{k} a_i$, and $S_R^T = \bigwedge_{j=1}^{m} \tilde{a}_j$, $V$ being the set of every unique variable occurs in $k$ atoms in antecedent [97].

**Proof.** Every variable in the $\mathcal{H}$ is universally quantified at the outer level of rule.

Therefore, existentially quantified variables may occur only in atoms after $a_k$, which is denoted by $v_p$. Let $\tilde{V}_p$ be the set of every unique atom in first $p - 1$ atoms in the consequent and $\tilde{V}_m$ is the set of every unique variable appearing in last $m - p$ atoms. The service, which replaced the existentially qualified variable, needs to complete successfully for the consequent of the rule to fire. If the service does not execute successfully the $\text{naf}$ layer prohibits the rule to fire the consequent. Moreover, the service is considered as "pure", i.e. it cannot change the KB by its internal execution. Therefore, from a point of view of Horn-DL, this is just a syntactic sugar. However, such mapping lets agents map the IOPR of a service to the Horn-DL and thus facilitate automatic service discovery, invocation and orchestration.
given clause can be rewritten in FOL as,

\[ \forall v_i \in V_i \bigwedge_{i=1}^{k} a_i \rightarrow \bigwedge_{j=1}^{p-1} \tilde{a}_j \land \exists v_p \tilde{a}_p \land \bigwedge_{j=p+1}^{m} \tilde{a}_j \]  \hspace{1cm} (4.2)

The above formula in FOL can be converted to prenex normal form by first applying
De-Morgan’s law to remove the implication and then moving all quantifiers to the left
without changing order.

\[ \forall v_i \in V_i \exists v_p \bigwedge_{i=1}^{k} \neg a_i \lor \bigwedge_{j=1}^{p-1} \tilde{a}_j \land \tilde{a}_p \land \bigwedge_{j=p+1}^{m} \tilde{a}_j \]  \hspace{1cm} (4.3)

As the above FOL is in prenex normal form, all existential quantifiers can be eliminated
by using Skolemization. We replace \( v_p \) with a function symbol \( f^{sk} \), which does not occur
anywhere in the formula and corresponds to Skolem function. We will show that such
function \( f^{sk} \) can be replaced by a service \( S = \langle S_I^I, S_I^O, S_I^P, S_I^R \rangle \). \( S_I^I \subseteq V \), as \( v_p \) is in scope
of every universally quantified variable in \( V \). \( S_I^O = v_p \), as \( sk - p \) replaces the existentially
quantified variable \( v_p \). The precondition of the service is the antecedent part of the rule,
i.e. \( S_I^P = \bigwedge_{i=1}^{k} a_i \), because the consequent can only trigger when antecedent is true. This
naturally maps \( R \) to the consequent part of the rule, which is now free of existential
quantifier, thus \( S_I^R = \bigwedge_{j=1}^{m} \tilde{a}_j \). The Skolemized Horn clause has no variable in consequent
which did not occur in the antecedent. Therefore the formula is now DL-safe. \( \square \)

The 4.3 shows that an existential rule can be transformed into a DL-safe Horn rule
when a suitable semantic service is available with a specific semantic property of its input
and output. Although it draws the necessary criteria for such mapping, it is not sufficient
because it does not specify one to one mapping between frontier variables and the input
variables of the semantic service. Such mapping is required for invoking service by
passing data, selected by the body of the rule, to the service as input.

The one to one mappings between the set of input arguments and the set of variables
in the antecedent of the rule is established by matching the types of the variables. As
defined in definition 4, the input arguments of the semantic services are typed and the service providers declare the type of the arguments in the service registry. The types can be either primitive, such as integer, float, or strings, or an ontological class, which can be described by a unary predicate in terms of predicate logic. The implementation of services may also use XSD\textsuperscript{11} data types as primitive types of the arguments. However, we recognize that the ontological types are more appropriate for making the service truly semantic. This is due to the lack of semantic underpinning for the primitive types. XSD data types cannot take part in subsumption relationships and so cannot have a place in any taxonomy and cannot be subject to a binary predicate. Still, the implementation of the proposed framework, described in section 4.4, handles such primitive data types by linking them to an ontological class, using ‘datatype properties’ from Web Ontology Language (OWL) in order to provide a flexible integration to legacy services using only primitive data types.

On the other hand, every variable used in the antecedent of the plan should have an identifiable type. The type may be implicitly declared with a unary predicate, otherwise known as \textit{instance-of} or \textit{member-of} relationship, as an atom in the antecedent clause of the associated rule. For example, a service with input signature $\text{TypeOf}(S_1) = \{C_1, C_2\}$ may be integrated to a rule $r_1$, for which $C_1(v_1) \land C_2(v_2) \subseteq head(r_1)$, where $v_1$ and $v_2$ are some variables, and function $\text{TypeOf}$ returns the collection of types for a set of variables. However, the type may also be interpreted from the existing clauses in conjugation with other rules, in the absence of an explicit declaration of such explicit unary predicate in the head of the rule. For example, the same service may integrate with a rule $r_2$, for which $C_1(v_1) \land C_3(v_3) \land R_1(v_2, v_3) \subseteq head(r_2)$ if there exists another rule $r_3 := C_3(v_3) \land R_1(v_2, v_3) \rightarrow C_2(v_2)$ in the agent’s list of plans. This way, a service may use only the upper-level term for declaring its inputs and still can be integrated to many

\textsuperscript{11} https://www.w3.org/TR/xmlschema-2/
domain specific rules, which may use the sub-types of the upper-level term. For example, a rule \(\text{pyramid}(v_1) \land \text{hasBase}(v_1, v_2) \land \text{square}(v_2) \rightarrow \text{hasArea}(v_2, v_3) \land \text{area}(v_3)\), stating that every pyramid has some base area, may invoke a service \(\text{calculateArea}(\text{quadrilateral}) \mapsto \text{area}\) to calculate the area of its square base. This is only possible in the presence of another rule \(\text{square}(v_4) \rightarrow \text{quadrilateral}(v_4)\), stating that the type ‘quadrilateral’ subsumes (is parent type of) the type ‘square’. The collection of subsumption relationships form a taxonomy, which is typically found in an ontology. Instead of searching these subsumption relationships in the plan structure of the agent, an ontology may be given to the agent to refer from, in order to minimize the number of plans and simplify the service discovery process.

### 4.4 Realization of Agents

So far we described how existential rules, with the existential variable integrated with the output of a matching service, may be used in writing the plans of a single agent. The previous sections described the theory of agents, that uses FO rules to drive the agent towards its goal by changing the state of its belief and existential rules to communicate with other agents by invoking these services. However, a practical solution still needs to be developed based on the theoretical framework, so that these autonomous agents can be deployed for the ultimate realization of cloud-based manufacturing. In this study, we created a java programming library, called ‘FunQL’ (Function Query Language) for writing plans as the FO rules, based on SPARQL query, including an enhancement for the integration of services to existential rules. The library also allows the user to execute these rules on a given knowledge-base in RDF (Resource Definition Framework) format, which basically acts as the belief of the agent. The library also proposes a structure for publishing different types of services (Web, RPC, and java method) with semantic annotations, so that these rules not only can find suitable services from the registry but
also allow the user to write native functions directly in the SPARQL query. However, this implementation does not describe the communication protocols, hosting, maintenance of these services, as many such standards and development tools for cloud-based service implementation already exist.

Although many different query engines exist to run Horn rules and some recent ones even capable of running existential rules, SPARQL is chosen for its popularity in ontology and knowledge-based development community, ability to query graph data, and availability of additional syntax for subquery, filtering, grouping, ordering, and property paths. Instead of the complex reasoning algorithms, such as Tableau along with many heuristics, the basic pattern matching algorithm for SPARQL query engine relies on solving graph Isomorphism problem. Graph isomorphism problem is a well-studied problem in spite of being NP-Complete in complexity. Moreover, the grammar of SPARQL, along with the many useful operators, shares close resemblance to the SQL query, which is almost ubiquitous for querying relational databases. Armed with these features, SPARQL let users write queries with flexibility, unparalleled to other type of query languages.

The genesis of SPARQL\textsuperscript{12} is closely tied to the need of querying the RDF data model, which was released in 1998 as a recommendation of W3C to store information about World Wide Web. RDF is a directed labeled graph data format, which is made of a number of statements. As every statement has three parts: subject, predicate, and object, which is a triple, the RDF data model is often called ‘triple store’. In order to maintain compatibility, the facts in the belief of an agent are stored in RDF format.

The standard ‘SELECT’ clause of SPARQL, returns the mapping for the variables in the ‘basic graph pattern’\textsuperscript{13} of the ‘WHERE’ clause. For standard SPARQL, if var($P$) returns the set of variables in the WHERE clause of the query ($P$) and the $W$ is a set of

\textsuperscript{12} https://www.w3.org/TR/rdf-sparql-query/
\textsuperscript{13} https://www.w3.org/TR/rdf-sparql-query/#BasicGraphPatterns
variables, listed in the ‘SELECT’ clause, then \( W \subseteq var(P) \) [5]. Therefore, the ‘SELECT’ clause cannot list any variable, which is not present in the graph pattern expression in the WHERE clause. In this study, we are interested in ‘CONSTRUCT’ clause, which also contains a basic graph pattern. This type of SPARQL query takes the form ‘CONSTRUCT \( H \) WHERE \( P \)’, which is analogous to the Horn rule. The standard SPARQL CONSTRUCT query is range-restricted as it follows that \( var(H) \subseteq var(P) \), where \( var(H) \) returns the variables in the basic graph pattern of the CONSTRUCT clause (\( H \)) [5]. In this study, we lifted this restriction by allowing the basic graph pattern of the construct query to contain variables, which are not present in the WHERE clause. By allowing these ‘existential’ variables in the CONSTRUCT query, the SPARQL CONSTRUCT query can then be used to express existential rules, although with the limitation on the arity of the predicates, as SPARQL allows only two-place predicates. As the graph pattern in the ‘WHERE’ clause cannot provide the mappings for the existential variables, the mappings for the existential variables are supplied by invoking some service.

In order to integrate a service to the SPARQL CONSTRUCT query with an existential variable, introduce a new clause in the basic grammar of SPARQL is introduced, as follows:

```
ConstructQuery ::= 'CONSTRUCT' (ConstructTemplate WhereClause SolutionModifier [FunctionClause])
FunctionClause ::= 'function' '{' Var '<-' ServiceTemplate '}'
Var ::= Endpoint '.' Method '(' Var* ')'  
Endpoint ::= iri
Method ::= [a-zA-Z0-9_]+
iri ::= IRIREF | PrefixedName
```

The FUNCTION clause mentions one or multiple service end-points. However, it is not mandatory for users to explicitly add the FUNCTION clause. The in-built service discovery algorithm may find a suitable service from the registry and automatically asserted in the query. The service discovery algorithm is implemented in FunQL following section 4.3 to find a suitable semantic service by matching the semantic types of
the input and output. The input and output of every service, along with their semantic
types, need to be published in the registry, encoded in a prescribed structure. This
structure is suggested to avoid coding separate interfaces for different semantic service
structures (mentioned in section 4.3). The proposed meta-structure is given in appendix A.

The core of the extension is done by extracting the input for the service from the
mappings, generated by the WHERE clause and then joining the output of the service to
these mappings to produce the complete mapping for the CONSTRUCT pattern, including
the existential variable. As we relax the constraint $\text{var}(H) \subseteq \text{var}(P)$ of the standard
SPARQL, this operation is performed only when there is an extra variable (existential) in
the CONSTRUCT pattern.

From proposition 1, we know that the input variables of the service can all be found
in the variables from the basic graph pattern in the WHERE clause i.e. $S_f^I \subseteq \text{var}(P)$. The
set of input mapping for the service can be explained by the evaluation of the $p$ over a
RDF graph pattern $G$ (Definition 1.2 from [5]). The input mappings for the service is thus
a subset of the complete mappings, returned by the partial function $\mu$ (from the notation of
SPARQL semantics [5]). If we denote the partial function, containing the mapping for the
input of the service, as $\mu_{S_f^I}$, then the ‘evaluation of WHERE clause $P$ over RDF graph $G$
for the inputs of the service’ ($[[P]]_{G}^{S_f^I}$ [5]) can be defined by extending the definition 1.2
from [5] as follows:

$$[[P]]_{G}^{S_f^I} = \{\mu_{S_f^I} : V \rightarrow T | \text{dom}(\mu_{S_f^I}) = \text{var}(p) \cap S_f^I \ 	ext{and} \ \mu_{S_f^I}(p) \subseteq \hat{G}\}$$

, where $\hat{G}$ is an incomplete graph with some of its nodes being variables, still not
replaced by a mapping. Intuitively, the variables in $\hat{G}$ belong to the set $\{\text{var}(P) \setminus S_f^I\}$.

For a set of mappings $\Omega$, returned by the graph pattern of the WHERE clause, the
service is invoked with each of the restricted mapping ($\mu_{S_f^I} \in \Omega$). The bag semantic
(section 1.4.2 of [5]) can be used to calculate how many times the function is invoked by
counting the number of restricted mappings, which is simply the projection of the
evaluation of $p$ over $G$, restricted by $S_f^I$, but without removing the duplicates. The number of mappings can be written using the cardinality of $\Omega$ [5], as follows:

$$\text{for } \mu_{S_f^I} \in \Omega, \text{card}_\Omega(\mu_{S_f^I}) = \sum_{\mu' \in \mu_{S_f^I}} \text{card}_\Omega(\mu')$$

Upon a successful invocation of the service, a new set of mappings ($\Omega_O$) is returned by the function. From proposition 1, we know that the output of the function is mapped to the existential variable, i.e. $S_f^O = v_p$. Therefore the output mapping of the function can contain only one variable in its domain. Consequently, every input mapping and corresponding output mapping of a function has disjoint domains, as the output mapping maps the existential variable, which does not occur in the $P$, and thus not in the domain of input mapping, as $S_f^I \subseteq \text{var}(p)$. As every disjoint pair of mappings are ‘compatible’ [5], the set of input and output mappings thus can be joined by following the standard AND operator. We can also use the complete set of mappings returned by the graph pattern in the WHERE clause, that is $\Omega_P = \{[P]\}_G$, instead of the input mapping, which is just a restriction over the $\{[P]\}_G$. In this way, we can derive a set of mappings $\Omega_H$ for which $\forall \mu_H \in \Omega_H \text{ var}(H) \subseteq \text{dom}(\mu_H)$, because,

$$\Omega_H = \Omega_P \bowtie \Omega_O = \{\mu_P \cup \mu_o | \mu_P \in \Omega_P, \mu_o \in \Omega_O, \text{dom}(\mu_P) = \text{var}(P), \text{ and } \text{dom}(\mu_o) = v_p, \text{ thus compatible}\}$$

FunQL also allows the more than one existential variable in the CONSTRUCT clause, however, only one of the variables may be mapped by the output of the service in the FUNCTION clause. The rest of the existential variables are mapped by a default function $\theta$, which supplies a new resource to be added to each mapping in the set $\Omega_H$. This is similar to the treatment of blank nodes in SPARQL and makes sure that the final mappings in $\Omega_H$ contain a mapping for every variable in the CONSTRUCT clause, including the existential variables. Finally, a new set of triples can be generated by
replacing the variables in the basic graph pattern \((H)\) in the CONSTRUCT clause by replacing the variable by each mapping in the set \(\Omega_H\).

**Example 1.** Let the following *FunQL* calculate the volume of a cone. The structure of the service [http://geometry.com/solids/calculateVolume](http://geometry.com/solids/calculateVolume) can be found in fig. A.1.

```
PREFIX geom:<http://www.astro.umd.edu/astro-onto/geometry.owl#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ivao:<http://www.astro.umd.edu/astro-onto/IVOAO.owl#>
PREFIX conic:<http://geometry.com/solids>

CONSTRUCT{
  ?v ivao:hasVolume ?vol.
  ?c geom:hasVolume ?v.
  ?v rdf:type geom:volume
}
WHERE{
  ?c astro:hasDiameter ?d.
  ?c geom:hasHeight ?h.
  ?d ivao:hasValue ?dia.
  ?h ivao:hasValue ?ht.
  ?c rdf:type geom:circularCone.
  ?d rdf:type geom:diameter.
  ?h rdf:type geom:height
}
FUNCTION{
  ?vol <- conic:calculateVolume(?dia, ?ht)
}
```

Let the following mapping is returned, when executed against a knowledge-base. XSD type ‘double’ is ommitted for the decimal values.

```
{?d ↦ :dia1, ?c ↦ :cone1, ?h ↦ :h1, ?dia ↦ 1.5, ?ht ↦ 5.0}
```

Clearly, the function `calculateVolume` is invoked with mappings `?dia ↦ 1.5` and `?ht ↦ 5.0` and returns the output mapping `{?vol ↦ 9.817477042468102}`. The other existential variable `?v` is mapped with the default function which maps a random resource.
URI, i.e. {?v \mapsto :volume_ I89yay1}. The combined mapping can be applied to the `CONSTRUCT` pattern, which produces the following graph.

\[
\text{volume}_I89yay1 \implies \text{hasVolume} \implies 9.817477042468102\text{^xsd#double}
\]
\[
\text{cone1} \implies \text{hasVolume} \implies \text{volume}_I89yay1
\]
\[
\text{volume}_I89yay1 \implies \text{type} \implies \text{volume}
\]
5 SIMPM- A FOUNDATION ONTOLOGY FOR MANUFACTURING PLANNING

The goal of this chapter is to derive upper-level semantics for concepts related to manufacturing planning. The purpose of this set of upper-level semantics is to classify various manufacturing domain-level concepts. As we discussed in chapter 3 that the services hosted by various manufacturing partners in a cloud manufacturing environment need a unifying schema in order to achieve seamless integration. The multi-agent architecture, described in the previous chapter, provides a mechanism to express the belief and plan of BDI based agents by using formulas in first-order logic. The plans of the agents are also written in first-order logic-based rules, which automatically integrate semantically tagged services by matching the types of the input and output of the service from the precondition and postcondition of the rule itself. However, an important assumption underpins such a framework - the predicates used in describing the facts and rules of every agent in a single world and the services offered by them are expected to follow a common vocabulary. Without this assumption, the mechanism of service discovery, presented in section 4.3, does not work. For this reason, we adopt ontological analysis to define a set of concepts (unary predicate) for relevant terms in manufacturing planning and relationships (binary predicate) among them. A vocabulary, which is created based on ontological analysis is more robust than traditional data models, including both relational (e.g. RDBMS), and object-oriented (e.g. XSD) schema. Before we delve into the ontological analysis of our domain of discourse, we need to explain what is an ontology and how it is superior to the traditional information modeling system.
5.1 Effectiveness of Ontology in Data Integration

In the traditional understanding, ontology is a study of what is, a philosophical pursuit in understanding the nature of being. For ages, this particular metaphysical study has been undertaken by philosophers to comprehend the entities in the existence, by categorizing and establishing relationships among them. In modern times, information science adopted an ontological study to systematically define concepts and data items for a specific domain of discourse. The primary use of ontology in information science is to enable data integration, as Noy and McGuinness state: “sharing common understanding of the structure of information among people or software agents is one of the more common goals in developing ontologies” [81].

The most general terms in the ontology are expressed in the form of logical statements, which are called ‘axioms’, using primitive concepts (universal), and relationships. A set of axioms, which can satisfy entailment of some members of a domain, may be considered as a theory or model for that domain. The ontological model of a domain is thus separate from the members of the domain and provides a level of abstraction to the instances and the relationship among them. Being separated from the actual information and implementation details, the ontology acts as meta-structure, which capture the knowledge of the domain (semantics), instead of just the physical structure (schema). This primary feature of ontology helps in the interoperability of the systems, which can easily adapt and extend a global ontology to represent information from their respective domains. The axiomatic theory of ontology also places some restrictions on what information can be meaningfully asserted (logically consistent), thus providing an inherent validation of the information stored in the knowledge-base. Furthermore, the logical representation of ontology lets users store less amount of data as a number of reasoning algorithms can extract implicit information from the explicit information, stored in a knowledge-base.
5.2 Ontology Design Methodology

The primary task for the development of an ontology for a particular domain is to formally define a list of general terms, which are necessary to describe the facts for that domain, using some chosen forms of representation.

Once the list of terms to be modeled is identified, formal descriptions of those concepts are derived based on the primitives from a foundation ontology. Ideally, the foundation ontology contains a set of top-level and domain-independent terms, which provides philosophical underpinning for the upper-level terms of a particular domain. This foundation ontology also ensures cross-domain compatibility and interoperability. In this study, Basic Formal Ontology (BFO) [7] is adopted as the foundation ontology. The upper-level terms are formally defined based on axioms, written mostly in the syntax of first-order logic. This set of upper-level terms for manufacturing process planning is called ‘SIMPM-upper’. On the other hand, the taxonomy of various domain-level terms, all of which subsume the upper-level concepts, make up the ontology, called ‘SIMPM-lower’. In this study, SIMPM-lower focuses on the domain of metal fabrication, especially CNC machining. Both of these ontology are stored in web ontology language (OWL) \(^{14}\). OWL is chosen to represent the terminologies and relationships because of three fundamental logistical reasons: 1) As BFO is available in OWL format, it is easier to extend the foundation concept by using the ‘import’ functionality of OWL for the domain-level extensions; 2) OWL can be directly represented in Resource Description Format (RDF), which is used in the realization of the multi-agent system, described in section 4.4; 3) the popular open-source ontology editor ‘protégé’ \(^{15}\) can be used to easily develop, maintain, and also visualize ontology of OWL or RDF format.

---

\(^{14}\) [https://www.w3.org/OWL/](https://www.w3.org/OWL/)
\(^{15}\) [https://protege.stanford.edu/](https://protege.stanford.edu/)
The challenge in constructing an upper-level ontology for planning is that some of the required concepts are abstract (design, plan, specification, capability), while others represent physical objects (material, machine, tool). In order to define these concepts, a set of fundamental terms need to be established first. These concepts go beyond any particular domain and concern elements of reality. In order to maintain semantic conformity, these concepts should be inherited from a foundation ontology. The definitions for product design, manufacturing resources, and capabilities are borrowed from BFO [7, 105, 106, 117]. The upper-level definition of a process, its relationship with manufacturing resources, such as machines and tools, input, and output, as well as planning related concepts are discussed in section 4.2.2, based on theories of process specification language (PSL)[99], however maintaining alignment with processual entities of BFO.

5.3 Product Design Ontology

Colloquially, a product design is the list of specifications for a physical product, conceptualized by the designers. In ontological sense, these specifications are not the actual properties of the product. A physical product has some observable and measurable properties, which can be taken as basis for classifying that product (some sub type of Artifact). However, the specifications, listed in the design are conceptual (imagined) properties and many physical products may be produced following that design.

In close observation, there are two different notions of a ‘design’: the mental model of the product as it is conceptualized, and the quantitative aspect of the model, such as the geometry, dimension, and the tolerance specifications. We tackle this distinction by looking at two aspects of the specification itself: one is which portion of the reality the specification is being conceived for and the other is how such portion of reality is being
conceived. The former is a set of ‘quality references’, which describes the attributes of the product, and the latter is a set of ‘quality specifics’, which specifies those references.

The distinction between quality references and the quality specifics serve two fundamental purposes of the product design: the iterations during designing and the conformance of manufactured product to the design. For any design, the quality specifics may change as the design evolves over time. They may also come from many sources when multiple designers work on a single design using different types of CAD software. Still, different versions of quality specifications still point to the same quality reference. On the other hand, the quality references maintain the semantic linkage between the specifics and the actual products. By maintaining the link, the references help in deciding whether the product conforms to the design, which version of the specification it conforms to. In case the product does not conform then the same linkage helps to identify whether the correct version of the design is used in the manufacturing.

5.3.1 Representation and Specifications

The notion of ‘quality reference’ is denoted by the concept Representation (sub-type of Information Content Entity, which is sub-type of Quality in BFO), which is a quality such that it is about or is intended to be about an entity [106]. Such choice can be justified by the presence of two particular predicates, associated with representation. First, representations can be about (isAbout) some portion of reality which will exist in future or may not exist at all (non-veridical), such as “idea, image, record, or description which refers to or intended to refer to, some entity or entities external to the representation” [106]. Following this, we can define different sub-types of representation, which refers to (isAbout) the intended product or some part of product (e.g.

∀x ArtifactRepresentation(x) → Representation(x) ∧ ∃a Artifact(a) ∧ is_about(x, a)).

Second, by the essence of being a type of quality, a representation (more specifically, a
cognitive representation, which is a mental quality) specifically depends on *(inheres_in)* “an anatomical structure in the cognitive system of organism” [106]. This clearly explains that the specifications do not inhere in the physical product but only in the cognitive realm of the designer.

The ‘quality specifics’ are basically information about the representations. For example, length of a pipe (in design) is a *QualityRepresentation*, but the measurement of the length, e.g. 10 meter is the information about that representation. The information may change without affecting the fact that pipe has some length. Quality specifics are modeled as *Information Content Entity*, which is about *Representation* and such reference is denoted by the property *specifies*, which is sub-property of *is_about*. By the virtue of being a *Generically Dependent Continuant* [7], *Information Content Entity* can support design evolution by migrating from one version of the design to another and support product conformance by transmitting to a physical product after such product is manufactured. As part of this study, various types of specification classes are created for different elements of a product design. In most of the cases, there is a corresponding sub-type of *Representation* class, which the sub-type of *Specification* class *specifies*. The general axiom for *Specification* is given in Axiom 1.

**Axiom 1.**

\[
\forall s \text{ Specification}(s) = \text{InformationContentEntity}(s) \land \exists r \text{ Representation}(r) \land \\
\text{specifies}(s, r) \land \text{DesignDocument}(d) \land \text{genericallyDependsOn}(s, d)
\]

The Representation type denotes only those entities of a product which refer to *(is_about)* physical counterparts in the actual product, such as the assembly, component, features, some geometrical entities along with their dimensions and tolerances. However, typical product design also describes structural information, such as parthood among assemblies and components, the orientation of features, and relations among geometrical
entities and their dimensions. No such relation can be asserted between two instances of Representation(s), as Quality is itself a characterization of some continuant and cannot have another instance of Quality characterizing it further. However, one Information Content Entity may refer to another Information Content Entity, which can be leveraged to describe relationships among specifications. Such mapping is done by the concept Mapping Information Content Entity, which describes the map of two or more Information Content Entity pertaining to Specification. This mapping Information Content Entity is helpful in describing relationship among more than two specifications, akin to the concept of reification. The general axiom of Mapping Information Content Entity, given in Axiom 2, can be specialized into unidirectional mapping, one-to-many, of many-to-many.

**Axiom 2.** \( \forall m \, MappingInformationContentEntity \rightarrow InformationContentEntity(m) \land \exists i_1, i_2 \, describesMapWith(m, i_1) \land describesMapWith(m, i_2) \land (i_1 \neq i_2) \)

The overall relationship among a product component (e.g. ‘gear1’), its Representation and Quality Representation (e.g. number of teeth), and corresponding specifications is presented in fig. 5.1. The figure shows the example of instances of the classes in quotation. The literal value (string, number) can be expressed by some data property attached to the ICE as subject. Apart from string, integer, and decimal data types, which are part of XSD schema and already available in OWL 2.0 (https://www.w3.org/TR/owl2-syntax/#Datatype_Maps), a number of geometrical and dimensional data used in the product design require custom data types, such as vector (for position), matrix (for transformation) and formula (for 3D geometrical entity). These complex data types are not discussed in this document for brevity.

**5.3.1.0.1 Design Document** The Specification entities described so far need to be codified in some kind of storage format. These formats are analogs to the various standards and CAD file types prevalent in design communities, for example, STEP, IGES,
DWF (AutoCAD), prt (NX), and sldprt (Solidworks). These files normally store design specifications in a pre-specified format. The codification format does not change the specifications at all, but just presents it in a way, which helps in storage, exchange or visualization. For example, the same design specification for a part may be stored in a STEP file for data exchange, as well as in a native CAD file, for displaying the geometry and topology on a graphical user interface. In our model, these formats of storages are in category Information Bearing Artifact. Every Specification, generically depends on some DesignDocument (sub-type of Information Bearing Artifact). As they generically depend on the DesignDocument, they can migrate from one document to another as a newer version of design becomes available. In this way, a complete history of the changes in the design can be computed, which is elaborated in fig. 5.2.

5.3.1.0.2 Conformance  A product, when manufactured following a design, needs to be tested to establish that it conforms to the design specification. The Representation entity acts as bridge to perform the conformance test, which is expressed by the rule:

Axiom 3. \( \forall r s e Entity(e) \land Specification(s) \land Representation(r) \land concretize(e, s) \land specified_by(r, s) \rightarrow conforms(r, e) \)
Figure 5.2: Versions of design: change of specifications for number of teeth.

The entity can be an Artifact, a Feature, a Quality, and various other sub-concepts, which are part of a physical product. The \textit{concretizes} predicate needs to invoke a proper function that actually performs a suitable comparison between the actual entity and the specification. The conformance rule described above should start from the measurements of different features, then moving to parts, and finally the entire product, because one of the trivial criteria for conformance of an Independent Continuant (assembly, part, feature) is that the every inhering Dependent Continuants (qualities, such as dimension and tolerance measurements) should conform. However, the strictness of the inspection may vary among manufacturers.

5.3.2 Part and Assembly

Artifact is a “distinct entity in a product, whether that entity is a component, part, subassembly or assembly” [35]. Three principles aspects of an artifact are its function (what the artifact supposed to do), form (structural, geometrical, and material information), and behavior (how the form implements the function). In this study, we will mainly focus on the form, which is also the output of typical CAD software. The artifact can be categorized as an object as both a single component and an assembly, as it respects the criteria of causal unity: the former due to the internal physical force and the later due
to intentional arrangement via some engineering [105]. An Artifact is distinct from an object on the basis that artifacts are intentionally designed by some agents to realize some function. In this study, the teleological aspects of an artifact are not included, which are more appropriate in capturing the relationships an artifact’s structural design shares with the functional requirements for the artifact.

An artifact can be either a part or an assembly. The primary focus of this study is on individual part, but the distinction between part and assembly is kept open for further extension. In a generalized term, ArtifactRepresentation refers to an Artifact (part or an assembly), which is specified by ArtifactSpecification.

**Axiom 4.** $\forall s_a \text{ArtifactSpecification}(s_a) \rightarrow \text{Specification}(s_a) \land \exists r_a \text{ArtifactRepresentation}(r_a) \land \text{specifies}(s_a, r_a) \land \text{Artifact}(a) \land \text{isAbout}(r_a, a)$

Three types of properties can be directly attributed to the ArtifactSpecification, namely structural, geometrical, and material. The structure of a feature-based design is mainly defined by a set of feature specifications, which is discussed in the next section. The material requirement of the part is included in the design, which is important for selecting a suitable tool. The specified material, which is denoted by MaterialSpecification is a type of QualitySpecification) and is associated to ArtifactSpecification by ArtifactQualityMAP [Mapping Information Content Entity]. This geometrical model is composed of geometrical entities, such as solids, surfaces, and curves in 3D space, accompanied by topological information, which elucidates how the geometrical entities are composed to form a valid manifold. These geometrical entities are not elucidated further for the sake of brevity, as they play little role in a feature-based planning.
5.3.3 Design Feature

Design features are some special areas of interest in a part, which are “a set of geometric entities (surfaces, edges, and vertices) together with specifications of the bounding relationship between them and which have engineering function and/or provide assembly aid” (13). The identifiable form features in a physical part can be categorized as fiat object part, which can be distinguished from an Artifact (in the category of Object in BFO) due to the fact that even being a part of the main object (PartFeatureMap associates specifications of features with specification of part). Therefore, design features are represented by FeatureRepresentation (⊆ Representation), which refers to a FormFeature (defined as Fiat Object [103]). The general axiom for FeatureSpecification is given in Axiom 5. Parallel to the taxonomy of FormFeature, separate taxonomy of FeatureSpecification is also developed (Appendix B.1). Some of the sub-types of FeatureSpecification include HoleSpecification, PocketSpecification, and SlotSpecification.

Axiom 5.
\[ \forall s_f \text{ FeatureSpecification}(s_f) \rightarrow \text{Specification}(s_f) \land \exists r_f \text{ FeatureRepresentation}(r_f) \land \text{specifies}(s_f, r_f) \land \text{Artifact}(f) \land \text{isAbout}(r_f, f) \]

In order to establish the links between the specification of part and the specifications of its features, instances of PartFeatureMap (defined in Axiom 6) is used.

Axiom 6. \[ \forall m \text{ PartFeatureMap}(m) \rightarrow \text{MappingInformationContentEntity}(m) \land \exists p_f \text{ PartSpecification}(p) \land \text{FeatureSpecification}(f) \land \text{describesMapWith}(m, p) \land \text{describesMapWith}(m, f) \]

The presence of these fiat boundaries prevents features to be modeled in ‘boundary representation’ (a format for representing topology) effectively as they are non-manifold
object. For these reasons, the design features are normally represented in the parametric form in practice, for example a simple hole with a flat bottom can be specified with a diameter and depth. Following this, we categorize the types of FeatureSpecification based on their distinctive parameters. The set of parameters for each type of feature are separately represented by sub-type FeatureQualitySpecification 
(≡ Specification ∧ ∃ describes(). FeatureQualityRepresentation). The QualitySpecifications are associated with FeatureSpecification with FeatureQualityMap. These quality specifications play a crucial role in the part compliance. When a feature is manufactured, its dimensions and positions need to be compared against these quality specifications to evaluate the result. The mappings from FeatureSpecification to FeatureQualitySpecifications for some of the template features used in CAD drawing is described in table 5.1. For every column, one instance of FeatureQualityMap maps the FeatureSpecification (in the column, e.g. CountersunkHoleSpecification) to every FeatureQualitySpecification (in the row, e.g. TipAngleSpecification), for which it is checked (√). For example, Axiom 7 states that a HoleSpecification is defined by DepthSpecification, DiameterSpecification, and TipAngleSpecification.

**Axiom 7.** ∀h HoleSpecification → FeatureSpecification(h) ∧ ∃a b c m DepthSpecification(a) ∧ DiameterSpecification(b) ∧ TipAngleSpecification(c) ∧ FeatureQualityMap(m) ∧ describesMapWith(m, h) ∧ describesMapWith(m, a) ∧ describesMapWith(m, b) ∧ describesMapWith(m, c)
Table 5.1: Mapping among feature and quality specifications

<table>
<thead>
<tr>
<th>Quality</th>
<th>Hole</th>
<th>Countersunk hole</th>
<th>Counterbore Hole</th>
<th>Cylindrical Pocket</th>
<th>Rectangular Pocket</th>
<th>Slot</th>
<th>Ball-end Slot</th>
<th>U Slot</th>
<th>T Slot</th>
<th>Dovetail Slot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diameter</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tip Angle</td>
<td>√</td>
<td>√</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Countersunk Diameter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Countersunk Angle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counterbore Diameter</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Counterbore depth</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Floor Radius</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taper Angle</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Length</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Corner Radius</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bottom Width</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bottom Depth</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A comprehensive list of the sub-types of FeatureSpecification, representing various types of prismatic features, is given in fig. B.1 (Appendix B).

5.3.4 Dimension Tolerance

Due to the physical randomness, and vibration in machine and tool, it is practically impossible to produce a part that conforms to the dimensions exactly. The tolerance specifications are used for describing the acceptable variation in dimension, which are to be taken into account while evaluating the conformance of a machined part. Normally, two types of tolerance specifications are used against LinearDimensionSpecification, which are PositiveToleranceSpecification (allowed excess from basic dimension) and NegativeToleranceSpecification (allowed shortness from basic dimension). However, these two types of specifications convey the opposite meaning, when applied to RadialDimensionSpecification. These tolerance specifications are mapped to DimensionSpecification by DimensionToleranceMap, which can map either or both types of tolerance specifications.

Apart from the conventional tolerances, a set of geometric tolerances are also used in part design to convey geometric characteristics and of the part, such as straightness, perpendicularity, flatness, angularity, roundness, concentricity, cylindricity, parallelism, and true position. A special type of tolerance called surface roughness is used to specify the quality of the working surface of the part. Most of these tolerances are declared as Specification class with a suitable Mapping Information Content Entity associating them with corresponding SurfaceSpecification. The semantic model of an example of part design with a hole feature specification is shown in fig. 5.3, which is modeled after the proposed design specification ontology.

\footnote{ANSI provides standard for geometric tolerances, e.g. ISO 1101, ISO 2768-1 And ISO 2768-2}
Figure 5.3: Semantic model of a sample part design with one hole feature.

5.4 Process Capability Ontology

In the world of manufacturing, terms such as capability, capacity, and competency are often used with intermingled meaning. Often times, such overlap in the connotations stems from the difference in the context, purpose, and the perspective in which these terms are used. For example, there is always a gap in the understanding of manufacturing systems between the perspective of engineering and management. Luo et al. described these two views as macroscopic, which focuses on the qualitative competency of organizations, and microscopic view, which focuses on the capabilities of the tangible and intangible resources and manufacturing knowledge of an organization [70]. In this study, we will mainly focus on the microscopic view of the capability based on the position that
the capability of resources ultimately defines the competency of the organization from bottom-up. In the ‘Sand Cone’ model of Ferdows and De Meyer, where manufacturing capabilities are stacked in layers – a representation of the cumulative gain of capability from bottom up – the quality of the product forms the ground layer [36]. In turn, the quality of the product is dependent on the conformance of the manufactured product with the design specification, which mirrors customer requirement.

Capability from the standpoint of process engineering is clearly resource-based. As described by Tien-Chieng, Wysk, and Wang in their book ‘Computer-Aided Manufacturing’ [26], the term capability is used in three different contexts which describe their subject of attributions, which are process capability, machine and tool capability, and shop level capability.

In order to clarify the scope of the ontological model presented in this study, a set of the competency questions [81] are provided below:

1. What are the capabilities of an individual resource (machine or tool)?

2. What is the net capability of resources when used together in a process?

3. What are the capabilities, for which some resources (machine or tool) are deemed suitable for a process? e.g. Which machine and tool can be used to twist drill the hole?

4. How the capabilities of one or many resources can be evaluated based on the specification of the task? e.g. Which resources can machine a blind hole of 2” in diameter and 3” in length with a tapered bottom and maximum 8µm of surface roughness?

5. How does one compare the expected result of alternative processes when performed to achieve the same goal? e.g. Could we achieve better/worse surface finish if end
milling is used for making the hole instead of twist drilling in the same factory, using the same machine but with an end-milling cutter?

5.4.1 Definition of Capability

Capability of the manufacturing resource is mostly stated in terms of the function [105] of the resource. In fact, the measurements associated with the capability are basically process level measurement, which assumes that the resource has at least some function to play in that process. It is trivial that the brightness capability (measured in lumen) is associated with a light bulb and not a hammer, because only the former is meant to be used in the process of lighting. It is thus safe to say that the capability of an artifact is an approximation of some function of the artifact. Based on this preliminary analysis, the relationship between capability and function is further detailed based on logical rigor.

The capability of an artifact is defined as a sub-type of disposition. Furthermore, by taking the association of capability with the function of an artifact in accord, the capability is formally defined as follows:

Definition 5. A capability \( c \) is a disposition which inheres in an artifact \( a \), such that

1. demarcates the extent by which some function \( f \), inhering in artifact \( a \), is realized when a participates in some process \( p \), and

2. predicts some change of state for \( a \) or some other object \( a' \), which also participate in process \( p \).

An artifact can bear some functions as well as some capabilities; each of those capabilities puts some limit on one or more functions of the bearing artifact. For example, a lathe has some turning function, which is constrained by the test bed size, spindle speed, position of head and tail stock, which are all expressed as capability. A wrench has both gripping and spanning functions, which are constrained by capabilities such as type of
nuts it can span, and maximum torque it can apply, and angle of the handle. The capability of an artifact is strictly guided by its physical structure and some of the capabilities are close paraphrasing of the artifact’s quality. However, being associated with a function the capability is revealed only when the demarcated function is realized by some process. In other word, the capability of an artifact is always extant but hidden in it.

Every kind of tangible manufacturing resource, including machine and tool, are of type object, whereas the manufacturing processes are of type occurrent. As the functions of the manufacturing resources are only realized when the process actually occurs, the relationship between function, process, and resource, given in Axiom 10, is used for further analysis.

What we broadly mean by the statement ‘capability demarcates function’ is that when such function is realized upon completion of a process, the result of the process can be predicted based on the associated capability. Precisely, on completion of a process, the capability of an artifact predicts some change of state for the participant artifact or some other artifact, which also participates in the same process. For example, the spindle of the milling machine realizes spinning function when it is started. Some associated capability can demarcate such function by asserting that the maximum speed of the rotation is 2400 rpm. Therefore, if the rotational speed of the mill is increased from 0 to $x$ when the mill is started, we can at least predict that $x$ should be less than or equal to 2400 rpm. It is to be noted, that such prediction strongly assumes normative use of the resource – that the resource is not malfunctioning, used in ideal (or close to ideal) way, and designed specifically to be used in the process it participates. There are several ways in which the capability predicts the outcome of the process, which is dependent on the information the capability contains. In section 5.4.3, we describe these information structures in detail. Range is one of the common predictions capabilities provide, which implies that the change of state of the corresponding bearing object in the same process should be within
the range. Assuming the functions (not to be confused with the type ‘function’) \( \text{max}(c) \) and \( \text{min}(c) \) returning the maximum and minimum limit of the range, which some capability \( c \) predicts, the axiom for fluent, given in Axiom 13, can be restated in Axiom 8.

**Axiom 8.** \( \forall p \exists i \ \text{Process}(p) \land \text{Interval}(i) \land \text{occurs}(p, i) \rightarrow \exists o \ t_b \ t_e \ \text{participates}(o, p) \land \text{beginAt}(i, t_b) \land \text{endAt}(i, t_e) \land \neg \text{holds}(o, q, t_b) \land \text{holds}(o, q, t_e) \land (\exists c \ \text{bearerOf}(o, c) \land \text{max}(c) \geq q \land q \leq \text{min}(c)) \)

This axiom fits the description of the intrinsic capabilities of manufacturing resources, when the object \( o \) is specialized as either a machine or tool in the typical manufacturing scenario. However, the capability of an object can also make a prediction for some change of state not only for the object itself but of some other object participating in the same process. Typical process capabilities predict the quality of the output of the process, which is about the material being processed, e.g. the work-piece. For example, different types of milling operations achieve a varying degree of precision, which are normally expressed as range of expected tolerances. These tolerances are qualities of the output feature (e.g. pocket, recess), which belongs to the work-piece. This work-piece is also another participant in the process. The Axiom 8 is restated in Axiom 9 for this type of capability predicting the change of state for some other object than the bearer.

**Axiom 9.** \( \forall p \exists i \ \text{Process}(p) \land \text{Interval}(i) \land \text{occurs}(p, i) \rightarrow \exists o \ t_b \ t_e \ \text{participates}(o, p) \land \text{beginAt}(i, t_b) \land \text{endAt}(i, t_e) \land \neg \text{holds}(o, q, t_b) \land \text{holds}(o, q, t_e) \land (\exists c \ \text{bearerOf}(o, c) \land \text{max}(c) \geq q \land q \leq \text{min}(c)) \land (\exists o' \ \text{participates}(o', p) \land \text{max}(c) \geq q \land q \leq \text{min}(c) \land (o \neq o')) \)

When multiple resources participate in a process, they conjugally achieve some goal. This goal is achieved upon the realization of some function, which inheres in all of these resources as a shared function. The shared function may be different than individual function a resource may have. For example, a screw driver can tighten or loosen a screw when the tip of the screw driver sits on the recessed head of a screw and a user of the
screw driver applies force on it. The process realizes the function of ‘tightening screw’, which is shared by both the user and the screw driver. Some capability may demarcate this particular shared function – for example how fast the screw is driven in – which is enabled by some individual capabilities of the participants. In this case, the capability of the screw driver to match the recess of the screw head (e.g. slotted, cross, square, hexagonal, Phillip, Frearson, Allen head), and capability of the agent to handle a screw driver both enables the capability of the tightening the screw. The interdependency among capabilities mirrors the interdependency of functions as described by Kitamura and Mizoguchi: “A function is achieved by performing(achieving) a series of sub-functions which is called a method of function achievement” [60]. In their model, the justification of the realization an upper-level function in a process based on the structural behavior of its underlying components or co-participants of the process is in fact derived from the FBS ontology of Gero [41]. However, similar justification cannot be strictly derived for the capabilities due to the preconception of the observer as well as the emergent behavior shown by component in the manifestation of a capability. The enabling mechanism of upper-level capability due to participants of components capabilities can be justified only from the teleological sense, instead of causal interaction. Such interdependence can be captured by the property enabledBy (inverse of enables) as shown in fig. 5.4, which may only be asserted artificially from the utilitarian point of view, but never can be reasoned based on the functional of behavioral interplay.

Most of the modern CNC machining functions are shared by some machines and tools. The capabilities, which demarcate them, are realized by various CNC machining processes. The process capabilities mentioned in the section 3.1 are in fact the combined capabilities demarcating the shared function commonly inhering in the machine and tool used in the process. We can distinguish such capability as ManufacturingProcessCapability in order to distinguish from individual capabilities
of machine and tool, which can be categorized as MachineCapability and ToolCapability.

5.4.2 Taxonomy of Capability

So far, the term for capability is defined as an entity, which demarcates the functional potentiality of an artifact. In this section, the modes of such demarcation are explored in detail. The modes of the demarcation form the basis for the classification of capability.

The capabilities can primarily be classified based on type of object, which bears the capability, such as AgentialCapability, which inheres in cognitive agents, and ArtifactCapability, which inheres in artifacts – the artefacts (bfo:Artifact) are a type of bfo:object, which are distinct from the natural objects because they are intentionally created as opposed to natural objects. Naturally, every type of manufacturing resource is a type of artifact and the capabilities borne by them are type of ArtifactCapability. This way further classification can be made for ArtifactCapability based on whether the capability inheres in a machine, tool, equipment, or vehicle.
Furthermore, different types of capability may also be categorized based on the subject of its prediction, which are mainly some quality of the bearer (or another object in case of NonSelfReferringCapability), for example, different types of tolerance capability such as DimensionalToleranceCapability, SurfaceFinishCapability, and ParallelismCapability, SpindlePower, and MaterialCapability. These granular capability classes can build the taxonomy of the MachineCapability and ToolCapability as most of them can be specifically borne by either machine or tool. However, these capabilities can also be either of SelfReferringCapability and NonSelfReferringCapability, which prevents them from building a mono-hierarchical taxonomy.

Several domain-level capability for CNC machines and tools are listed in Appendix B. Combinations of these CNC machines and tools participate in CNC processes, such as twist drilling, boring, end milling, side milling, and face milling. The process level capabilities, borne by some combination of resources, are classified under ManufacturingProcessCapability. The taxonomy of ManufacturingProcessCapability is also shown in Appendix B.

5.4.3 Capability Information

The demarcation imposed by capability on a function is expressed in many different ways in practice. The variation in the methods of recording the information about capability stems from the fact that not every type of capability is measured or apprehended in the same way. While some capabilities are simple tokens such as ‘axis of control’ capability of a CNC machine (‘2’, ‘3’, ‘4-substitution’, ‘4-simultaneous’, and ‘5’), or expressed as grades, some more complex capabilities use mathematical equations as well as conditional rules. For example, the surface finish produced by a cutting tool is dependent on the material of the work-piece, on which the machining is being performed.
These expression structures enable the capability for practical usage by specifying both quantitative and qualitative information for the capability. We build these information structures using Information Content Entity.

By virtue of the characteristics of Information Content Entity [106], multiple formats may be used in tandem to denote the same capability. For example, the surface finish capability may be expressed as a measurement in micron, some standardized code or even a picture of the surface fig. 5.5. Moreover, the data contained by the Information Content Entity may have different provenance, such as tables in machinist’s handbook along with some visual cues and symbols for easy reference in the shop floor, and XML data structure to be referred by computer-sided process planning software [34]. These provenances may be optionally maintained by an Information Bearing Entity, a type of material entity. Thus, the Information Content Entity may inhere in multiple copies of the same information, stored in a different format. Generally, the unit of the measurement value is expressed as Information Content Entity refereeing to some Information Bearing Entity. In this way, same Information Content Entity may have multiple Information Bearing Entity describing the same measurement in different units.

Figure 5.5: Surface finish capability may be expressed as a measurement in micron or some standardized code.
More complex types of information may be expressed by multiple Information Content Entity (s) referring to each other in a pre-ordained fashion. One important way of demarcation is to declare limits and ranges (dual limits) of the capability. The maximum and minimum of limit can be denoted by two different Information Content Entity, each described by its corresponding types of OrdinalICE, either MaximumOrdinalICE, or MinimumOrdinalICE and connected by a NominalMeasurementICE (shown in fig. 5.6). When capabilities are expressed as rule or an equation, they are not determinable in advance. The variables in the rule or equation refer to some elements in which only exist at the time when the bearing artifact takes part in a process. For example, the positive tolerance achievable by applying twist drilling with a suitable vertical drill and drill bit can be expressed as a function of the hole diameter. The equation requires the diameter specification to be available to determine the capability limit precisely. In this case, the positive tolerance capability is a type of ReferentialCapability, because while the positive tolerance capability inheres in the vertical drill and drill bit as shared capability, it predicts the change of state of feature which is being generated by the drilling process. The hole specification acts as the reference entity which the capability refers to and the diameter specification acts as the argument to the equation. In fig. 5.7 the rule is expressed as a JESS math type equation, which is specified by the rule type.

5.5 Manufacturing Planning Ontology

In order to understand the semantics of the process plan, we first need to investigate the concept of the manufacturing process. ‘Process’ is a polysemous term and contains different connotations in different contexts. Traditional data structures, such as ASTM-E3012 defines the manufacturing process with attributes, such as input, output,
Figure 5.6: Capability describing range (maximum and minimum surface roughness)

Figure 5.7: Capability expressed as rule, which expects an argument belonging to the entity the capability

and resources. Three types of resources are used in every kind of manufacturing process, that are machine, tool, and fixture. On the other hand, every process plan is composed of multiple planned operations to be performed by using these manufacturing resources. Different types of processes provide the specification of the behavior of every such operation. Such a typical case for this common understanding of process is depicted in fig. 5.8.

However, the data structure of process contains overlapping meaning and domain-specific assumptions. For example, the term process is used to denote a general type of process as well as an actual occurrence of some type of process. Moreover, the
association of manufacturing resources with the process is not the same as the association of input and output objects to the process. Finally, the data structure does not explain on what basis a machine or a tool is used in a manufacturing process, even though it assumes that such use is warranted.

5.5.1 Manufacturing Process and Resources

Various resources used in manufacturing primarily facilitate a series of well-coordinated processes in order to transform materials into products. In general sense, processes are also central to the dynamic reality we experience around us. Every kind of motion, change, actions, and events are all processes. Fundamentally, every process changes the location and the appearance of some objects, which are involved in that process. We say that objects participateIn in processes. We mostly identify the types of the process by the resultant effects when that process finishes occurring.

In the case of manufacturing, every process is intended human and deliberately arranged. When a person uses a hammer to pound on a stone with enough force, the stone breaks. We can safely characterize such types of hammering is an act of breaking or deforming material. The type of functions the process realizes depends on the objects,
participating in that process. The definition of process in BFO is based on the participating objects and their co-existence in the same temporal region. In manufacturing, most of the resources can be classified as objects, such as Machine and Tool. The particular set of functions, realized during the occurrence of a process, should be borne by the participating objects. From the viewpoint of manufacturing resources, these resources can participate in a process because they bear the necessary functions. It is easy to observe that a chain-lift is primarily designed to achieve the function of lifting heavy material, therefore in a particular act, say ‘lifting’ of a heavy car engine in an auto shop, the ‘function of lifting’ is realized by the ‘process of lifting’. These interrelations among process, object, and function establish a triad (displayed in fig. 5.9), that comes into play in every scene of manufacturing. Whenever a manufacturing process is successfully performed, some resources participate in that process and some functions of those resources are realized (Axiom 10). The manufacturing resources can be of type machine, tool, fixture, or any other artifacts. Here we use the term Resource as a placeholder for these artifacts. Similarly, the term ArtifactFunction acts as a placeholder for upper-level functions associated to resources.

Figure 5.9: Triad of manufacturing process, resource, and function.
Axiom 10. $\forall p \text{ManufacturingProcess}(p) \rightarrow \text{Process}(p) \land \exists f \text{ManufacturingResource}(r) \land \text{participateIn}(r, p) \land \text{ArtifactFunction}(f) \land \text{bearerOf}(r, f) \land \text{realizes}(p, f)$

A general taxonomy of ManufacturingProcess is given in Appendix B. Each of these terms is defined based on the corresponding function type and resource type. The taxonomies of ManufacturingResourceFunction, and manufacturing resources, such as ProductionMachine and ProductionTool are also given in Appendix B.

5.5.2 Three Notions of Process

The three distinct notions of processes are the process universal, process particular, and planned process. Some of the foundation ontology define special classes for universal and particular process. For example, PSL distinguishes between Activity (process universal) and Occurrence (process particular) as the ‘behavior specification’ and ‘a run-time execution of a behavior specification’, respectively [16]. In other words, the process universals describe how a process should occur, such as the participating objects, the configuration of those objects, and necessary circumstances, but do not contain any time related information. On the other hand, a process particular is the actual occurrence, which complies with the behavior, specified by a process universal and must have some beginning and ending time. For example, when we can say that ‘car doors are made by stamping process’, we mean the ‘stamping’ process in general. Here, the term ‘stamping’ is a process universal. But the stamping process, which was performed yesterday for some Toyota car at the Kentucky factory is a process particular. In BFO, the type Occurrent connotes process universal, and every instance of Occurrent is a process particular. In PSL, the participating objects of an occurrence must co-exist with the occurrence, that a resource can participate in a manufacturing process when the former is available (exists) during the same time interval, in which the process is also occurring (occurs). The following axiom is adopted from the axiom 16 of PSL [99] for manufacturing process.
Axiom 11. \( \forall r, p \ Resource(r) \land ManufacturingProcess(p) \land participateIn(r, p) \rightarrow \exists i Interval(i) \land exists(r, i) \land occurs(p, i) \)

, where Interval is defined as a set of two time points marking the beginning and ending of the interval:

Axiom 12.

\( \forall i Interval(i) \rightarrow \exists t_b t_e TimePoint(t_b) \land TimePoint(t_e) \land beginAt(i, t_b) \land endAt(i, t_e). \)

The planned processes are kind of process particular, which still did not occur but planned for some possible future occurrence. A planned process is very similar to a process universal, as multiple occurrences may follow the same instances of planned occurrence. However, the planned occurrences may still have some associated temporal notions, such as a begin and end time points (set in future), or some temporal constraint (temporal precedence). On the other hand, the planned processes are not the same as occurrences. Apart from the fact that the begin and end time for the planned process instances need to be in the future, there is also no constraint of temporal co-existence (ax-process-resource-interval) of the participating resources for a planned process.

Foundation ontology such as BFO, DOLCE, or PSL does not entertain a separate classification for such planned process. For PSL, the beginning and ending time of an Occurrence may determine whether that instance of occurrence is yet to occur or already occurred in the past. One way to express some facts about the planned process is to resort to modal logic. In order to avoid modal operators and stay in the realm of first-order logic, the common-core ontology (a BFO inspired middle level ontology) adopts the theory of ‘modal realism’, proposed by American philosopher David Kellogg Lewis [123], and encourages the use of the same relationships used for actual processes for planned processes, however naming such relationship as Lewsian counterpart of the original relationships. We follow a similar strategy but create a new sub-type of processes, called
PlannedProcess (⊆ Process) to maintain semantic distinctions. We use the concept of PlannedProcess to define the process plan in section 4.2.2.

5.5.3 Input and Output of a Process

The objects participants of a process experience changes in the quality as a result of the process. However, the continuous change in the properties can only be captured in a series of time-indexed snapshots of measurements. The four-dimensionalist view adopted by BFO on the occurrent cannot account for a change in the quality of the participant, as processes are themselves the ‘changes’ [104]. This requires the process instances (particulars) to be instantiated from one or more determinate universals. For example, a twist drilling process occurrence with cutting speed 1200 rpm and feed .0002” is an instance of determinate universal ‘Twist-Drilling with cutting speed x rpm’ and ‘Twist-Drilling with feed .0002”’. This may be helpful in recording the effects of the processes as they are occurring or already occurred but cannot express the expected changes the process brings on its participants.

In PSL, the concept of Fluent is used to capture the change of the state of the world as a result of a process occurring. The prior(f, o) relation specifies that a fluent f is intuitively true before an occurrence o and the holds(f, o) relation specifies that a fluent f is intuitively true after an occurrence o. A fluent is changed by the occurrence of activities, and a fluent can only be changed by the occurrence of the activity. Therefore, holds(f, o) is not true before the process. The fluent can be expressed as the state of an object; for example, some quality is inhering in an object at a certain time point. Objects go through changes when they participate in a process, which means that during the interval the process is occurring, some properties of the object changes during the occurrence of the process. However, these properties, which are quality instances inhering in an object, can only be measured or observed in an object at the beginning and ending time points only.
Axiom 13. \( \forall p \exists i \text{ Process}(p) \land \text{Interval}(i) \land \text{occurs}(p, i) \rightarrow \exists o q t_b t_e \text{ exists}(o, i) \land \text{participates}(o, p) \land \text{beginAt}(i, t_b) \land \text{endAt}(i, t_e) \land \neg \text{holds}(o, q, t_b) \land \text{holds}(o, q, t_e) \)

However useful the concept of fluent is in expressing the changes occurring during a process, complex reification techniques are required to express fluents in OWL [124]. This is because the fluents, which map objects to situations of truth [75], need either a function in a binary relationship or a ternary relationship to express. In order to avoid this complex reification, the input and output of an occurrence are linked with hasInput and hasOutput relationship, both of which are sub-property of participateIn. It is to be noted, that the input and output object of a manufacturing process may remain the same in some cases, even when some of its qualities may change. For example, a face milling process may polish the surface of a work-piece. The surface finish tolerance of the output work-piece improves from the input work-piece by the application of the process but the identity of the work-piece still remains the same. Therefore, the input and output objects of a manufacturing process are distinguished by a change of quality in those objects in Axiom 14.

Axiom 14.

\[ \forall p \exists o_1 o_2 \text{ ManufacturingProcess}(p) \land \text{hasInput}(p, o_1) \land \text{hasOutput}(p, o_2) \rightarrow \exists q_1 q_2 \text{ inheres_in}(q_1, o_1) \land \text{inheres_in}(q_2, o_2) \land (q_1 \neq q_2) \]

5.5.4 Theory of Manufacturing Planning

Every manufacturing process plan (MPP) mainly consists of a series of tasks, performed in a sequence to manufacture a part with the desired quality. However, more often tasks from one step need feedback from previous steps. Production management decision made at every step influences the decision for subsequent steps. This inter-dependency among phases of development is classically tackled by concurrent engineering [71]. The main idea behind concurrent engineering is to integrate various
phases of product life cycle, such as design, manufacturing, quality assurance, and maintenance; so that the designers can design the product keeping the manufacturability of the product in mind (Design for manufacturing/Design for Assembly). In practice, this demands an integrated system able to analyze the design data and compute necessary processes, machines, and tools necessary at the design time. Integrated Manufacturing Planning Model, developed by Sormaz et al. [112], addressed the aforementioned inter-dependency among various planning variables.

Integrated Manufacturing Planning Model uses a three-dimensional object-oriented model to represent the non-linearity in manufacturing process plan information [112]. In this model, various stages of non-linear manufacturing process plan are filed along aggregation dimension. A set of planning variables, such as (features, processes, machines, and tools) are selected at each aggregation level to be part of the overall process plan. Moreover, planning variables at every aggregation level also vary along with the variety and the time dimensions. The variety dimension captures alternative selections of planning variables for a certain manufacturing task, whereas time dimension captures the temporal dependency among the planning variable values at every aggregation level. The aggregation dimension can be expanded to cover planning at other stages of product life cycle, such as material requirement planning (MRP), logistic planning, supplier selection, and distribution. The schematic diagram in fig. 5.10 shows a slice of the manufacturing process plan with the aforementioned three dimensions.

The primary goal of SIMPM foundation ontology is to link planning variables from one aggregation dimension to another by establishing logical links. In particular, every machinable feature of a part design is linked to suitable manufacturing processes, which in turn are linked to suitable machine and tool to use. In this way, SIMPM ontology provides direct connection between design and manufacturing, so that a dynamic process plan can
be generated taking into account any real-time change in either product design or shop floor status.

5.5.5 Structure of Manufacturing Process Plan

A process plan is essentially a collection of planned processes (instance of PlannedProcess). The process plan may be defined by adopting the ‘temporal parthood’ among processes. In simple terms, a process \( p_1 \) can be part of another process \( p_2 \) if the beginning time of \( p_1 \) is after the beginning time of \( p_2 \) and the ending time of \( p_1 \) is before the ending time of \( p_2 \). Following this, we can define a plan as just another instance of PlannedProcess, which contains more than one instances of PlannedProcess as its part. Following this idea, a plan is just a process defined at a certain level of aggregation,
because it contains the processes from lower level of aggregation as its part. For example, a process plan for a part is the process which contains every process, required to generate all of its features, as part. In order to maintain the semantic distinctions among the instances of \texttt{PlannedProcess} at various level of aggregation, we present the techniques to compose a collection of processes, based on the resources they use and type of output they produce in section 5.5.5.1. The model is borrowed from the concept of ‘activity hierarchy’ from PSL [16].

The crux of the process planning is the selection of suitable set of processes for a given specification. In Axiom 19, we describe the criteria for feature-based planning, where the suitability of a given process is decided for a given feature specification. However, there can be multiple processes, which can satisfy such criteria. This builds alternative planning routes for the same feature specification. These alternatives fall in the variety dimension of the plan. The selection criteria of one process or multiple alternative processes for a specification is discussed in section 5.5.5.2.

In a plan, the instances of selected processes need to be applied in a sequence. The determination of such sequence planned under the time dimension. The valid sequence of processes in a plan is done partly based on existing planning knowledge and constrain embedded in the given design specification. In section 5.5.5.3, we describe how these constraints assert precedence criteria among the selected set of processes. As described in section 5.5.2, the selected instances of processes are of type \texttt{PlannedProcess}. However, a number of alternative sequences can be built with these selected processes by respecting the alternatives, precedence, and aggregation. In this study, we adopt the concept of ‘occurrence tree’, proposed by PSL [16], to represent these alternative routes. As the name suggests, the occurrence tree describes the flow of occurrences in a manufacturing system. However, we adopt this concept for representing the process plans, albeit slight modifications, due to its effectiveness in transforming a set of alternative routes from a
selected set of processes (activities), and filtering the valid routes based on some precedence constraint. These topics are also included in section 5.5.5.3.

5.5.5.1 Aggregation Dimension

In PSL, Axiom 16 dictates that an object can participateIn an activity only at those time-points at which both the object existsAt and the process isOccurringAt, whereas an activity-occurrence is the occurrenceOf a single activity [99]18. In manufacturing process plan, the collection and order of the process are not about the actual occurrence of the process but behavior specification based on its allocation of resources and specific configuration, which are better suited to be asserted for activity. In an effort to enable the allocation of resources in the process plan, Solano et al. conjured up an abstract type called Resource, which is used as a reified type to link Object (machine, tool, fixture) to instances of activity [107]. During the actualization of such Activity, the Object instance assumes a role to fulfill the capability and capacity of the Resource attached to the Activity. In reality, every machine and tool is different from each other when their capability is analyzed in detail. Therefore, the essence of the resources is grounded in its capability, which is directly derived from the actual Object, and not from the abstraction of it. The model proposed by Solano et al. still links these actual objects to occurrences [107], whereas the knowledge of the actual objects and their capability is a priori for process planning, thus making the use of abstract type Resource redundant. Rather we propose a modicum extension to PSL, in which the participateIn predicate links object to activity without any predicate requiring the object to existsAt the timepoint. In order to classify processes based on the participants of the process, e.g. the machine, tool, and fixture, three subtypes of activity are defined. The PSL Ontology uses the subactivity relation to capture the basic intuitions for the composition of activities. The subactivity

18 refer to section 5.5.2 for the semantic alignment of the concepts ‘activity’ and ‘activity-Occurrence’ in PSL with the concepts of ‘occurent’ in BFO
relationship is a transitive relationship that holds between two activities [99]. Using this property of the subactivity relationship, the process aggregation can be defined using the sub-types of activity as defined in the following axioms for PartActivity, MachineActivity, and SetupActivity, and ultimately provides a grouping of Step activities. The factor of the aggregation is based on the resources and outputs of the process, which is necessary for creating some desirable order in the process plan. For example, one may want to process every step, which can be processed by the same machine, consecutively in order to minimize the machine transfer cost. Similarly, the steps under the same setup may also be grouped in a similar fashion. Such aggregation will provide the necessary grouping constraints on the ordering of the occurrences of the activity. Here we show that the structure generated by applying the given aggregation is a tree.

5.5.5.1.1 StepActivity Step is an atomic activity, i.e. no other activity can be a sub-activity of StepActivity, in a process plan. An activity is a step activity if the output of a StepActivity is a single feature, i.e., no two features can be the output of the same StepActivity. This definition is based on the definition of working step in ISO 14649, however, while ISO 14649 is focused on CNC machining, this definition is its extension to other manufacturing processes.

Axiom 15. \( \forall a \text{ StepActivity}(a) \rightarrow \text{Activity}(a) \land \exists f \text{ achieves}(a, \text{ partOf}(f, p)) \land (\exists a' \text{ StepActivity}(a') \land (a \neq a') \rightarrow \exists f' \text{ achieves}(a, \text{ partOf}(f', p)) \land (f \neq f')) \)

The predicate achieves is borrowed from the PSL axioms for fluent attached to the occurrence [99]. It can be observed from the definition of StepActivity that every instance of StepActivity is devoted to producing a single feature. We already described the features as fiat objects belonging to some part in section 5.3.3. Here, we introduce a new predicate processingStepOf, which links a StepActivity to that part, containing the
feature produced by StepActivity. This predicate is introduced with little ontological value but to make the subsequent axioms simpler.

### 5.5.5.1.2 SetupActivity

A SetupActivity is composed of one or more StepActivity, each of which is performed using the same fixture, i.e. no two StepActivity, which are subactivity of the same SetupActivity can use different fixture. Also every StepActivity under a SetupActivity works on features, which all belong to the same part.

**Axiom 16.**

\[
\forall a_f \text{ SetupActivity}(a_f) \rightarrow Activity(a_f) \land \exists a \text{ StepActivity}(a) \land subactivity(a, a_f) \\
\rightarrow \\
\exists s \text{ usesFixture}(a, s) \land (\forall s' a' \exists p \text{ StepActivity}(a') \land usesFixture(a', s') \land subactivity(a', a_f) \land processingStepOf(a, p) \land processingStepOf(a', p) \rightarrow (s = s'))
\]

where usesFixture is a sub-property of the inverse of participateIn and Fixture is a sub-type of Object. It should be mentioned here that this definition does not imply that all StepActivity(s) with the same fixture have to belong to the same SetupActivity.

### 5.5.5.1.3 MachineActivity

A MachineActivity is a type of activity such that each of its sub-activity is processes in the same machine, i.e no two activities, which are subactivity of the same MachineActivity, can be processed on two different machines.

**Axiom 17.**

\[
\forall a_m \text{ MachineActivity}(a_m) \rightarrow Activity(a_m) \land \exists a_f \text{ StepActivity}(a) \land subactivity(a, a_m) \rightarrow \\
\exists m \text{ usesMachine}(a, m) \land (\forall m' a_f' \exists p \text{ StepActivity}(a') \land usesMachine(a', m') \land subactivity(a', a_m) \land processingStepOf(a, p) \land processingStepOf(a', p) \rightarrow (m = m'))
\]

where usesMachine is a subproperty of participateIn and Machine is a sub-type of Object.
5.5.5.1.4 PartActivity  A PartActivity is a type of activity such that each of its subactivity makes some change in the workpiece, targeted to make a single part, i.e. no two subactivities of the same PartActivity can have output features belonging to two different parts.

**Axiom 18.**
\[ \forall a \text{PartActivity}(a_p) \rightarrow \text{Activity}(a_p) \land \exists a \text{StepActivity}(a) \land \text{subactivity}(a, a_p) \rightarrow \exists p \text{processingStepOf}(a, p) \land (\forall a' p' \text{StepActivity}(a') \land \text{subactivity}(a', a_p) \land \text{processingPartOf}(a', p) \rightarrow (p = p') \]

Based on the aggregation factor, the activities at every level of the activity hierarchy can be classified as a type of complex activity, except for the StepActivity, which should always be an atomic or primitive activity. The hierarchy of activities may be formed by classifying the activities at each level as one of the activity types, described above.

**Example 2.** The aggregation hierarchy, shown in fig. 5.11, is for a simple part with four features. Table 5.2 shows the machines and tools participating in the instances of StepActivity and corresponding features achieved. The part will be completely done if all four of its features are processed successfully, each of which is the output of one of StepActivity shown as the leaves of the hierarchy. At the root of this hierarchy is PartActivity, which contains every required step as its subactivity. Two instances of MachineActivity are included for two types of CNC machine needed for generating four of its features. Under the MachineActivity, different SetupActivity(s) group step activities base on tool direction. Note that SetupAct1-A and SetupAct2-A have same tool direction, still distinct as they are applied to different machines.
Table 5.2: Machine and fixture assignments for various Step Activities

<table>
<thead>
<tr>
<th></th>
<th>StepAct1</th>
<th>StepAct2</th>
<th>StepAct43</th>
<th>StepAct4</th>
</tr>
</thead>
<tbody>
<tr>
<td>usesMachine</td>
<td>Mill1</td>
<td>Mill1</td>
<td>Drill1</td>
<td>Mill1</td>
</tr>
<tr>
<td>usesFixture</td>
<td>Fixture1</td>
<td>Fixture2</td>
<td>Fixture1</td>
<td>Fixture2</td>
</tr>
<tr>
<td>achieves</td>
<td>Slab1</td>
<td>Slab2</td>
<td>Hole</td>
<td>Slot</td>
</tr>
</tbody>
</table>

Figure 5.11: A simple part design with four features and corresponding activity aggregation.

The primary purpose of the aggregation in a manufacturing plan is to group steps under the same machine, tool, and other resources, so that the resource allocations, which are expressed at the complex activity level, can be deduced from atomic step level. This lets us to collect the required steps to process a part or a single feature. This is possible because PSL declares *subactivity* relationship as transitive. For example, StepAct2 is a subactivity of SetupAct1-B, which is, in turn, a subactivity of MachineAct1. Therefore, StepAct2 uses milling machine Mill1 and fixture Fixture2. In the same way, we can
also retrieve the collection of steps, required to be performed in a particular machine or
under a particular set up. For example, one may query to find which steps can be allocated
to Mill1 and will find three answers: StepAct1, StepAct2, and StepAct4.

However, the collection of steps alone do not represent the manufacturing process
plan entirely. A plan also contains alternatives collection of steps for the same
specification as well as precedence constraints among the collection of steps. These
concepts are discussed in the following sections.

5.5.5.2 Variety Dimension

Presence of alternative choices for a particular manufacturing task stems from the
underlying matching strategy, on which a suitable manufacturing process, and related
resources can be identified for such task. Every manufacturing activity has a list of
specifications to be fulfilled. For example, a hole feature may require specific diameter,
straightness, roundness, perpendicularity and other tolerances [56]. On the other hand,
every manufacturing process needs a suitable machine and tool to perform that process.
Generally speaking, this selection is performed by matching the specifications of lower
level planning variable with the capabilities of upper-level planning variable. Here, the
level of planning is according to the aggregation dimension.

At the bottom-most level, the features of a part are matched against the available
processes. When we say that a process is available, we actually mean that at least one
suitable combination of manufacturing resources (machine, tool, fixture, operator) is
available. As described in section 5.4.1, the combination of manufacturing resources
provides a combined capability of type ManufacturingProcessCapability, which
demarcates some function, realized by the process in turn. During the planning, these
process level capabilities are matched against the dimension and tolerances of the feature.
As most of these capabilities of type ManufacturingProcessCapability are non-self
referential, each with a promise to satisfy a particular type of quality of the feature. A process is considered to be part of the plan for the feature, if at least one of the specification can be satisfied by one of available capability. The exact verification process is dependent on the nature of the specification and the capability information structure. For example, a range type capability can satisfy a quality specification value if the later falls within the range specified by the capability. This particular type of matching is explained in section 5.4.1. However, it is up to the planner to decide whether to select a process if it can satisfy at least one quality or every quality specification, inhering in the feature specification. In reality, however, one process is not enough to generate a feature completely, and often multiple processes are needed to satisfy all specifications of a feature. The task of comparison is similar to simulation, in which it is assumed that the process indeed occurred and its effects are recorded. The effects are captured as some changes in the input object of the process. The changes are calculated based on the capabilities of the process. For example, if a particular occurrence of twist drilling can produce a hole of maximum diameter of 1/4” (because the drill bit used in that instance is of diameter 1/4”), the output feature will have a diameter of 1/4”. This instance of Quality concretizes the corresponding DiameterSpecification, given in the design. It is to be noted, that such ‘concretization’ is not a guarantee that the output feature conforms to the design specification of the hole. In order to generalize the task of comparing specification with capability, we conceive a general function called ‘doesMatch’, which returns a Quality by comparing a pair of ManufacturingProcessualCapability and a FeatureQualitySpecification. The following Axiom 19 (given in a form of a rule) expresses the way a specification for a feature can be satisfied by a capability. The hasSpecification predicate is a composition of the mapping predicates, i.e. describe_map_for ◦ describe_map_for−, which links a
QualitySpecification to a FeatureSpecification through an instance of FeatureQualityMap, as described in section 5.3.4.

**Axiom 19.** \( \forall f, q, s \in \text{hasSpecification}(f, q) \land \text{realizes}(p, f) \land \text{demarcates}(c, f) \land \text{inheresIn}(q, f) \land (\text{doesMatch}(c, q) = q) \rightarrow \text{concretizes}(q, q_s) \)

In rigorous terms, a process can be selected if the input material of the process can be modified by the process in any way to make it more similar to the given specification. The planning rules in this study follows feature-based planning. Therefore, the rules creates a new physical feature as output of the process. This feature resembles the input feature with some new quality added, removed or modified. The output feature is then classified as one of IntermediateFormFeature, UnsatisfiedFormfeature, or SatisfiedFormFeature. The definitions of these types are based on whether the process can produce some ‘improvement’, no ‘improvement’, or the ‘complete’ feature.

Here, we connote the ‘improvement’ as the process yielding a new dimension or tolerance matching (conforms) corresponding dimension or tolerance representation, which are specified by some FeatureQualitySpecification for the FeatureSpecification. A ‘complete’ feature conforms every dimension and tolerance representation, that the feature itself conforms the FeatureRepresentation instance. The following axioms are based on the conformance rule given in Axiom 3. The predicate hasInput and hasOutput are asserted for an occurrence with an assumption that this occurrence still did not occur. However, these occurrences are not PlannedProcess as these rules simulate the effects of the occurrences. The predicate hasQualityRepresntation is an extension of hasSpecification, created by the composition hasSpecification \(\circ\) specified by\(^{-} \) and points to the corresponding QualityRepresentation for the instance of QualitySpecification (e.g. some tolerance).
Axiom 20. $\forall f_o \ IntermediateFormFeature(f_o) \rightarrow FormFeature(f_o) \land \exists p \ PlannedProcess(p) \land hasOutput(p, f_o) \land (\exists f_s d \ FeatureSpecification(f_s) \land hasQualityRepresentation(f_s, d) \land inheresIn(d, f_o) \land concretizes(d, d) \land \neg conforms(d, d))$

Axiom 21. $\forall f_o \ UnsatisfiedFormFeature(f_o) \rightarrow FormFeature(f_o) \land \exists p \ PlannedProcess(p) \land hasOutput(p, f_o) \land (\forall d \exists f_s d \ FeatureSpecification(f_s) \land hasQualityRepresentation(f_s, d) \land inheresIn(d, f_o) \land concretizes(d, d) \land \neg conforms(d, d))$

Axiom 22. $\forall f_o \ SatisfiedFormFeature(f_o) \rightarrow FormFeature(f_o) \land \exists p \ PlannedProcess(p) \land hasOutput(p, f_o) \land (\forall d \exists f_s d \ FeatureSpecification(f_s) \land hasQualityRepresentation(f_s, d) \land inheresIn(d, f_o) \land concretizes(d, d) \land conforms(d, d))$

In general, if a process is selected to be part of the plan, a occurrence of type PlannedProcess is created for the particular type of manufacturing process. According to the classification given above, only the processes producing an IntermediateFormFeature or a SatisfiedFormFeature can be selected to be part of the plan, that is a corresponding occurrence of PlannedProcess is created to be part of the occurrence tree. During the feature based planning, each of such instance of PlannedProcess becomes a StepActivity, that is collected as processingStepOf of the part, the feature belongs to.

5.5.5.3 Time Dimension

The successor relationship between two occurrences of activity occurrences is the basis of the occurrence tree in PSL. “The PSL successor relation associates occurrences with each other to represent all temporal orderings of runtime execution of activities whether they conform to a behavior specification or not, and even including orderings that are physically impossible. The relation forms a tree where every occurrence has exactly one successor for each activity, indicating the possibility of that activity happening next, so the branches represent possible execution traces” [16]. Every node of an occurrence
tree is a primitive or atomic activity occurrence, which can be safely described by the leaf nodes of the activity tree that are the activities which are not decomposed further. Every occurrence in the tree denotes a single step in a possible routing. Therefore every path from the root to leaf occurrence of the occurrence tree provides a possible routing. For example, the activity tree shown in fig. 5.11 has 4 leaves which are instances of \texttt{StepActivity}. The corresponding occurrence tree is shown in fig. 5.12, which contains all possible combinations, generated by the instance of \texttt{StepActivity} from the activity hierarchy in fig. 5.11.

![Figure 5.12: Occurrence tree for the simple part shown in fig. 5.11.](image)

It is apparent that without any constraint on the sequence of the steps, every branch of the occurrence tree is a viable route, or in other words, a possible plan for the part. Two types of precedence constraints govern the validity of a process plan. The first one stems from precedence, imposed by the processing order among features of a part, and the second one from the type of processes to be applied for producing a feature.

\subsection*{5.5.5.3.1 Feature Precedence} Features (e.g. Hole, Slot, Pocket, Chamfers, and Bevel) in product design can only be machined in a specific order due to their overlapping spatial location (position) and orientation (vector) \cite{113}. In general, this type of precedence
constraints is imposed by design specification, such as the dimension datum reference and tolerances.

As described in section 5.5.5.1, features are uniquely tied to the instances of StepActivity. Being primitive activities, the occurrences of the StepActivity instances constitute various branches of the corresponding occurrence tree. PSL prescribes the use of nextSubocc relationship among the subactivities of a complex activity to denote such precedence constraint strictly. During planning, the following axioms impose ordering constraints among the occurrences of the steps (primitive activity) based on the precedence constraints among features. In order to capture the constraints among the features - they are derived from the positioning and orientation of features in the part design - a new predicate hasNextFeature is introduced. This new property can only be held between two features, generating a complete or partial ordering among the features. For example, the partial order among four features of the simple part, shown in fig. 5.11, is given in fig. 5.13. It should be noted here that feature precedence is a minimal set of constraints that the product or part requires for their manufacturing, on the other side process precedence may include more constraints (some will be mention later). Based on this feature precedence, a temporal ordering may be applied among planned activities, using the following minPrecedes axiom.

![Diagram of feature precedence among simple part features](image_url)

Figure 5.13: Feature precedence among the features of simple part.
Axiom 23. $\forall o, o_3, o'_3 \text{minPrecedes}(o_3, o'_3, o) \rightarrow \exists a, a_3, a'_3, f, f' \text{occurrenceOf}(o, a) \land \text{occurrenceOf}(a_3, a) \land \text{occurrenceOf}(a'_3, a') \land \text{subactivity}(a, a) \land \text{subactivity}(a', a) \land \text{achieves}(a, \text{partOf}(p, f)) \land \text{achieves}(a', \text{partOf}(p, f')) \land \text{nextFeature}(f, f')$

For the activity composition shown in fig. 5.11, Slab1 is processed by StepAct1, Slab2 by StepAct2, Slot by StepAct4, and Hole by StepAct3. The constraints among the features can be translated to the order among the corresponding occurrences of these steps by the following facts.

\[
\text{occurrenceOf}(o_4, \text{PartAct}) \land \text{minPrecedes}(o_3, o_1, o_4) \land \text{minPrecedes}(o_4, o_2, o_4) \land \text{minPrecedes}(o_4, o_3, o_4)
\]

PSL suggests the relation legal to specify an atomic occurrence (occurrenceOf a primitive activity) $o$ is an element of the ‘legal’ occurrence tree. A legal occurrence tree is the sub-tree, which captures only the possible branches, of the complete occurrence tree. For example, $\text{occurrenceOf}(o_4, \text{Step4}) \land \text{legal}(o_4) \rightarrow \text{holds}(o_2, \text{partOf}(\text{Slab2}, p)) \land \text{holds}(o_3, \text{partOf}(\text{Hole}, p)) \land \text{prior}(o_4, \text{partOf}(\text{Slot}, p))$.

This set of facts can also be deduced from the following rule by applying the minPrecedes axiom and avoiding the fluent predicate.

\[
\forall o_3 \text{occurrenceOf}(o_4, \text{Step4}) \land \text{legal}(o_4) \rightarrow \exists o_2, o_4 \text{occurrenceOf}(o_2, \text{Step2}) \land \text{occurrenceOf}(o_3, \text{Step3}) \land \text{minPrecedes}(o_4, o_2, o) \land \text{minPrecedes}(o_4, o_3, o).
\]

Similar rules can check the ordering constraint on occurrences of StepAct1, StepAct2, and StepAct3. The branches of the occurrence tree, in which every occurrence is legal, can be considered as a valid process sequence or routing. The valid sequences are marked with green in the occurrence tree shown in fig. 5.12.

5.5.5.3.2 Process Precedence Similar to the precedence among features of the part, a valid process plan also needs to follow a particular ordering for the processes applied in order to make one feature meeting its specification. This sort of situation is normal for CNC machining, in which more than one machining step may be required in order to
make the product feature in order to meet tolerance specification such as dimension, tolerance, and surface finish, however, this may occur in another type of manufacturing depending on the definition of feature adopted. For example, multiple heat treatments may be required to harden a metal bar, a welded joint may require polishing before welding, and paint job may require a coating of primer before the color is applied. The precedence among process stems from the technological limitation of the machine and tool, used in a particular manufacturing method. For example, a pilot hole needs to be drilled before drilling a deep hole with a strict straightness requirement in order to avoid the deflection of the drill bit under mechanical stress. The precedence among processes is best handled if an intermediate object is generated for every step. Following this approach, we may generate intermediate features for the part design whenever more than one process is required to satisfy the specified tolerance requirements. This also corresponds to practice, as such partially completed part may need to be transported from one machine to another (e.g. from a lathe to heat treatment, then to a grinder if grinding operations are required). Those intermediate features become an integral part of the part model for manufacturing planning. Therefore, the extended part model is a union of manufacturing features for the part design and all intermediate features that were identified as necessary during the process selection procedure. In order to complete the extended part manufacturing model, it is necessary to consider the impact of intermediate features on the feature precedence network (FPN). In order to generate the Extended FPN (EFPN), we will assume that the process selection procedure produced the processes for design features as shown in fig. 5.14. From the figure it is visible that to make Slab we need only milling process (M), while for the other three features we need two processes for each: for Hole we need twist drilling (Step3) and boring (Step3i), for Slot we need rough (Step4) and finish (Step4i) milling operations. The activity hierarchy for the corresponding EFPN is shown in Figure 6 by adding the intermediate features.
In order to accommodate the intermediate features, we need to extend our original aggregation to include another level of aggregation, which can combine steps having the feature outputs belonging to one feature specification. Every subactivity of this new type of complex activity \textit{FeatureActivity} is targeted to the same feature specification. The specification of the feature is distinguished from an actual (physical) feature. The intermediate features are a distinct physical feature with different dimensions and properties, whereas the design feature is specified in the product design document. It is expected that only one of the intermediate features will match the specification, and thus may be called the final feature. In Figure 6, The activity hierarchy shown in Figure 5 is redrawn by introducing this new complex activity. The axiom of \textit{FeatureActivity} is given below, in which the property $\text{specificationOf}(f_s, f)$ describes that feature $f$
(possibly intermediate or final) is one of the features created in course of achieving the specification $f_s$.

**Axiom 24.** $\forall a_i \text{FeatureActivity}(a_i) \rightarrow \text{Activity}(a_i) \land \exists a \text{StepActivity}(a) \land \text{subactivity}(a, a_i)$

$\rightarrow \exists p, f, f_s \text{achieves}(a_i, \text{partOf}(f, p)) \land \text{specificationOf}(f_s, f) \land (\forall f', f_s', a' \text{StepActivity}(a') \land \text{subactivity}(a', a_i) \land \text{achieves}(a', \text{partOf}(f', p)) \land \text{specificationOf}(f_s', f'))$

$\rightarrow (f_s' = f_s))$

The aggregation based on the feature can also be imposed on the occurrence tree using the required precedence among processes for subactivities of a FeatureActivity. For example, when applied as subactivity of the same FeatureAct3, twist drilling (Step3i) should precede boring (Step3), which can be stated as

$\forall o_b \text{occurrenceOf}(o_b, \text{Boring}) \land \text{legal}(o_b) \rightarrow$

$\exists o_t, o \text{occurrenceOf}(o_t, \text{TwistDrilling}) \land \text{occurrenceOf}(o, \text{FeatureAct3})$

$\land \text{nextSubocc}(o_b, o_t, o)$. 
6 IMPLEMENTATION

This chapter describes a process planning system, able to generate manufacturing process plans for a given CAD design. The planning is conducted by a society of agents, each of which is modeled after the BDI based agent formalism, presented in chapter 4. However, additional modules are also developed to manage the global knowledge bases, which are used by the planning agent. The plans of the agents and the knowledge bases are written using the terms and relationships following the SIMPM ontology, presented in chapter 5. The common set of semantics enables the data to flows seamlessly in the system. These agents in the system apply rules, which essentially describe the plans of the agents, to transform data from one form to another, communicate with each other, and create new patterns based on existing patterns. Examples include translation of product design specifications from CAD file to RDF graph, planning agents calling different planning services to generate plan for different types of features, and incremental process plan generation by accumulating decisions taken by different planning agents. Based on the fact that every such planning related tasks is guided by semantics, this system is named as ‘Semantically Integrated Manufacturing Planner’ (SIMPlanner).

The functionality and workflow of SIMPlanner system loosely follow the manufacturing process planning system IMPlanner [34], which was developed by Sormaz and other researchers at Russ College of Engineering of Ohio University. IMPlanner system is able to plan many different aspects of manufacturing, such as product design translation, process planning, machine selection, set up planning, process sequencing, and production simulation. In this initial development of SIMPlanner, we only focus on the selection of suitable processes based on available process level capability of distributed manufacturing resources and generating valid routes for manufacturing a given part design. However, SIMPlanner system adopts multi-agent based framework, suitable for a cloud environment, and semantic models for handling heterogeneous sources of data,
instead of the object-oriented programs and data structures, used in IMPlanner. In this way, SIMPlanner is ideal for use as a cloud manufacturing (CM) process planning system. SIMPlanner system is also more flexible than the Jade-based multi-agent manufacturing planner, called IMPlanner-MAS [109], which used proprietary semantic models mimicking the object-oriented model of IMPlanner as well as FIPA protocols for agent communication. SIMPlanner system replaces the propitiatory semantic model with a mono-hierarchical ontology, based on foundation ontology (BFO, PSL), and the FIPA-based communication with standard service-oriented communication.

Before describing the agents in SIMPlanner in detail, we provide a brief overview of the system in section 6.1.

### 6.1 Overall Design

The overall structure of SIMPlanner is given in fig. 6.1, describing different modules, which are mostly implemented by either a single agent or a group of agents, having similar structure. Except the main cluster of agents, engaged in process planning, two additional modules are developed in order to support the process planning with global knowledge management. One of these additional modules is comprised of ‘Part Design Loader’ agent, which populates the design RDF by translating design XML file, and a script to query the design knowledge base. This module is described in section 6.2.1. The other module, described in section 6.2.2, contains two different scripts for populating the process capability knowledge base and querying that knowledge base to find process related information. The process planning module is comprised of three groups of agents: ‘Part Planning’ agent, ‘Feature Planning’ agents, and ‘Process Selector’ agents. Briefly, the ‘Part Planning’ agent generates a set of plans, represented as occurrence tree, for a given part design. Each type of ‘Feature Planning’ agent generates a set of plans, represented as sub-occurrence tree of the complete occurrence tree, for each feature
specification of the part design. The ‘Process Selector’ agents handle the selection of a particular type of process for a feature, according to the capabilities of any available resource, suitable for that type of process. We describe these three groups in section 6.3.

![Figure 6.1: Overall structure of SIMPlanner](image)

The sections in this chapter focus on the structures of individual types of agents. As proposed in section 4.3, the agents offer services, which are then consumed by some plans of other agents to exchange data and collaboratively achieve a goal. These services are also listed and analyzed in the descriptions of the agents. These services also serve a means to invoke the agents from the clients at the users’ end. For this study, these services are implemented in Java and hosted locally. In a practical CM environment, however, these services may be hosted in distributed server with a suitable web-service communication protocol.

The entire development is modularized in a way to resemble a set of agents; however, the communication among these agents are not discussed in detail. This is because the communication protocols and maintenance of agents’ identity in a multi-agent society is
not the focus of this study, which rather focuses on the evaluation of the plans written in FunQL on changing the each agent’s belief and gradually achieve a goal together.

6.1.1 Commands in SIMPlanner

SIMPlanner provides command-based scripts to invoke these services to invoke the agents. These scripts are built purely for evaluation purpose. In a practical CM process planner, a user interface may be built as client for interacting with these agents. The agents may also integrate services, provided by manufacturing resources, such as machines, tools, and sensors, in order to receive data directly from shop floors. The entry point of SIMPlanner is the command-line program, called ‘SIMP’. Under this command, various other sub-commands let user access different modules of SIMPlanner. Each of these commands can be used with a suitable combinations of arguments to invoke various services and queries. These commands and associated arguments are described in the following sections of this chapters. We provide a comprehensive list of every command and argument below.

reads the specifications from a given part design
-d, --dimension=DIMENSION
    dimension type or * for all dimensions
-f, --feature=FEATURE
    feature name or * for all features
-file=FILE
    file path or URL of the specification XML
-g, --graph=PATH
    file path or URL of the RDF graph
-prec=Precedence
    file containing precedence.
-t, --tolerance=TOLERANCE
    tolerance type or * for all tolerances
-u, --unit=Unit
    set the unit, default unit is set to millimeter.
-v, --verbose
    write nodes with complete URI

    [-capa=CapabilityType] [-func=FunctionURI] [-g=PATH]
    [-max=<maxLimit>] [-min=<minLimit>] [-p=ProcessURI]
    [-ref=<reference>] [-u=Unit]
reads the capability of processes
<argms>...
-capa, --capability=CapabilityType
    function type or * for every type of function
-clear
    sure to clear the knowledge base?
-eq=<eqn>
-func, --function=FunctionURI
    function type or * for every type of function
-g, --graph=PATH
    file path or URL of the RDF graph
-max=<maxLimit>
-min=<minLimit> modify capability for a process. the capability is added to new function if '-func' is provided
-mod, --update Add modify capability for a process. the capability is added to new function if '-func' is provided
-new, --create -p, --process=ProcessURI process type or " for every type of process. When used as new pass a process individual URI.
-ref=<reference> -save
-u, --unit=Unit set the unit, default unit is set to millimeter.
-e or --exit to quit

Usage: plan [-feature=Feature] [-g=PATH] [-part=Feature]
[-save=<planPath>] [-u=Unit]
Plan a complete part or an individual feature
-feature=Feature Currently takes a feature name
-g, --graph=PATH file path or URL of the part specification RDF
-part=Feature Currently takes a part name
-save=<planPath> file to save the plan
-u, --unit=Unit set the unit, default unit is set to millimeter.

6.1.2 Mapping Prefixes to IRIs

Every class and property in SIMPM ontology contains an IRI. Separate IRIs are used to distinguish among concepts for design, process, resources, and planning. Moreover, the classes and relationships of these ontology are inherited from basic formal ontology (BFO) and common core ontology (CCO). In this chapter, the classes and relationships bear their respective IRI, which we represent by prefixes. The complete map of prefix and the corresponding IRIs are given in table 6.1. This table also contains prefixes for RDF and OWL schema, which are used for standard predicates from these schema in some rules, given in the following sections.

6.2 Knowledge-base Management Agents

In order to manage the legacy data import to the global knowledge-bases and prepare the test conditions for the planning agents, two types of agents are developed to manage the part design knowledge and process capability knowledge, respectively. In this section, the internal operations and the service endpoints, used for invoking these operations, for these two agents are presented.
Table 6.1: Mapping of prefixes and IRIs

<table>
<thead>
<tr>
<th>Prefix</th>
<th>IRI</th>
</tr>
</thead>
<tbody>
<tr>
<td>design</td>
<td><a href="http://www">http://www</a> ohio.edu/ontologies/design#</td>
</tr>
<tr>
<td>cap</td>
<td><a href="http://www">http://www</a> ohio.edu/ontologies/manufacturing-capability#</td>
</tr>
<tr>
<td>plan</td>
<td><a href="http://www">http://www</a> ohio.edu/ontologies/manufacturing-plan#</td>
</tr>
<tr>
<td>cco</td>
<td><a href="http://www">http://www</a> ontologyrepository com/CommonCoreOntologies/</td>
</tr>
<tr>
<td>rdf</td>
<td><a href="http://www">http://www</a> w3 org/1999/02/22-rdf-syntax-ns#</td>
</tr>
<tr>
<td>owl</td>
<td><a href="http://www">http://www</a> w3 org/2002/07/owl#</td>
</tr>
</tbody>
</table>

6.2.1 Part Design Knowledge

The primary input for a process planning task is a set of part design specifications. The process planning rules, given in section 6.3.3, select suitable processes, which can satisfy the specification for the features in the part, by comparing the specification of the design to the capability of the processes. However, the design specification data needs to be stored in a knowledge-base, before the process selection rules can be executed. The design knowledge-base is represented as graph data, which follows the design specification ontology, described in section 5.3. In the evaluations conducted in this study, only part design specification is considered. Every part design is decomposed into a set of form features. The sample designs, which are considered in the evaluation, are prismatic part design, the features are of type hole, pocket, slot, slab, and chamfer. Before we can transform the design specifications into knowledge-base, two preliminary analysis needs to be conducted on the CAD design file.

First, not all CAD designs are feature-based. In this study, the candidate sample CAD files are generated by Siemens NX CAD application. These designs are intentionally developed using design feature models, given by Siemens NX. However, sketch and
extrusion based CAD designs may also be processed by a feature recognition algorithm. Such transformation of design is not in scope of this study. Second, the precedence among features need to be deduced from the geometry (dimension, orientation, and position), and tolerance information of the feature. In this study, the part design files are first analyzed by a feature precedence deduction algorithm, developed by Arumugam [8] in his master’s thesis and part of the IMPlanner application.

A brief description of integrated process planning application (IMPlanner) is already presented in section 2.1. This application can receive a CAD file (.prt, i.e. a native format produced by NX) and use java application programming interface (API) to invoke various CAD services, provided by NX, to read geometry and dimension information, stored in the file. IMPlanner deduces the precedence among features from the feature interactions, based on geometrical relations. The same application also produces a XML file (in its proprietary schema), which contains the geometry, dimension, and tolerance (GD&T) information for every feature as well as the precedence among them. The XML serves as a combined data model for the design specification and thus it is adopted as the intermediary format in the transformation process from the CAD file to the knowledge-base. A sample XML file containing the the design specification for a simple part is given in appendix C.

6.2.1.1 Part Design Loader

In order to read the XML file, a parser is developed based on a common interface. The common interface provides flexibility as it may be implemented in different types of parser, which may read the design specification from an XML with different schema and also from other sources, if required in future. In this study, the implementation of the interface provides a set of methods, which return the list of features, their types, precedence among them, associated dimensions, and tolerances. These same methods are directly used in the FunQL queries as services. The signatures of the methods are given in
table 6.2. It is to be noted that a typical CAD model contains much more information than what can be extracted by the parser, however, these information are enough for conducting the feature-based process planning. For the same reason, not every axioms from section 5.3 for describing a complete product design, is used in the knowledge graph. The rules are written in FunQL, which are described below. A command-line program in the SIMPlanner can initiate the translation process for a given XML file and save the translated RDF file with the following script.

$ part -g <path to the design RDF to be created> -file <path to the XML file> -u <inch/mm>
<table>
<thead>
<tr>
<th>Method Name</th>
<th>Input</th>
<th>Output</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>readPartName</td>
<td>String</td>
<td></td>
<td>Returns the label of the part design</td>
</tr>
<tr>
<td>readPartProperties</td>
<td>Map of key as String, value as String</td>
<td></td>
<td>Returns the properties of the part e.g. material specification</td>
</tr>
<tr>
<td>readFeatures</td>
<td>List of String</td>
<td></td>
<td>Returns the list of the labels of the features belonging to the part</td>
</tr>
<tr>
<td>readFeatureType</td>
<td>String String</td>
<td></td>
<td>Receives the label of a feature and returns the type of the feature, e.g. hole, pocket, slot, slab</td>
</tr>
<tr>
<td>readFeatureDimensions</td>
<td>String List of key as String, value as String</td>
<td></td>
<td>Receives the label of a feature and returns a list of key-value pair, containing the type of the dimension and corresponding value</td>
</tr>
<tr>
<td>readNextFeature</td>
<td>String String</td>
<td></td>
<td>Receives the label of a feature and returns the label of the next feature as per the precedence among the features</td>
</tr>
<tr>
<td>readPreviousFeature</td>
<td>String String</td>
<td></td>
<td>Receives the label of a feature and returns the label of the previous feature as per the precedence among the features</td>
</tr>
<tr>
<td>readTolerances</td>
<td>String List of key as String, value as String</td>
<td></td>
<td>Receives the label of a feature and returns a list of key-value pair, containing the type of the tolerance and corresponding value</td>
</tr>
</tbody>
</table>
6.2.1.1 Loading part name  The part specification graph is initiated by the rule ‘read-part-name.q’ (fig. 6.2), shown below. The rule first invokes the ‘readPartName’ service to receive the part label and then attaches it to the LabelBearingEntity (subclass of InformationBearingEntity), inhering in the PartSpecification.

```
PREFIX service: <http://www.ohio.edu/services/design/>

CONSTRUCT
{
  ?i1  rdf:type design:LabelBearingEntity .
  ?i1  cco:has_text_value ?pn
}

FUNCTION
{
  ?pn <- service:readPartName()
}
```

Figure 6.2: Rule read-part-name.q

6.2.1.2 Loading features  The features for the part are loaded into knowledge graph in two steps. In the first rule ‘read-part-features.q’(fig. 6.3), the labels for every feature (variable ?f) for the part is read by invoking the service ‘readFeatureName’. The labels are stored as data (variable ?fn) for the instance of LabelBearingEntity, which inheres to each instance of FeatureSpecification. Every instance of FeatureSpecification is associated with the instance of PartSpecification, which is returned by the where clause (variable ?p) by an instance of PartFeatureMap(variable ?pf), following the Axiom 6.
In the second rule ‘read-feature-type.q’ (fig. 6.4), the type of each feature is read by invoking the service ‘readFeatureType’. This service receives the features name (variable \(?f\)\text{n})\), which is returned by the where clause by selecting every instance of FeatureSpecification, already stored in the graph by the rule ‘read-part-features.q’. A mapping is implemented in the service to return a URI from the domain level taxonomy for the feature, given in fig. B.1 (Appendix B), for the feature type (e.g. hole, pocket, slot, slab, bevel, and chamfer). The URI (variable \(?f\)t)\) is then asserted as data for the instance of TypeBearingEntity (variable \(?i2\)\), inhering in the instance of the FeatureSpecification (variable \(?f\)\).
PREFIX service: <http://www.ohio.edu/services/design/>

CONSTRUCT
{}
WHERE
{}

PREFIX cco: <http://www.topquadrant.com//extensions/co/ontologies/

Figure 6.4: Rule read-part-features.q

6.2.1.1.3 Reading feature dimensions  The dimensions of each feature are read and
transformed to knowledge graph in two steps. In the first rule
‘read-dimension-for-feature.q’ (fig. 6.5), the type for every dimension (variable ?dt) for
the feature is read by invoking the service ‘readFeatureDimensionType’. This service
returns all dimensions for the given label of feature (variable ?fn), selected by the where
clause. A mapping is implemented in the service to return a URI from the domain level
taxonomy for the feature dimensions (sub classes of QualitySpecification),
presented in fig. B.1 (Appendix B). Each instance of QualitySpecification (variable
?d) is mapped to the instance of FeatureSpecification by using an instance of
FeatureQualityMap (variable ?fd).
In the second step, the rule ‘read-dimension-measurement.q’(fig. 6.6) reads the measurement for the dimensions by invoking the service ‘readFeatureDimensionMeasurement’, which receives the label of feature (variable ?fn) as well as the URI type of a dimension (variable ?dt), selected by the where clause. The measurement value (variable ?dm), returned by the service, is asserted as data for a new instance of MeasurementBearingEntity, which inheres in the corresponding instance of QualitySpecification (variable ?d), selected by the where clause. The unit of every measurement is hard-coded (taken from the pre-defined instances of units from common core ontology) and asserted as data for the newly created instance of MeasurementBearingEntity.
6.2.1.1.4 Reading feature tolerances The tolerances of each feature are also read and transformed to knowledge graph in two steps. In the first rule ‘read-tolerance-for-feature.q’ (fig. 6.7), the type for every tolerance (variable ?tt) for the feature is read by invoking the service ‘readFeatureToleranceType’, which receives the label of feature (variable ?fn), selected by the where clause. A mapping is implemented in the service to return a URI from the domain level taxonomy for the feature tolerances (sub classes of ToleranceSpecification), presented in fig. B.1 (Appendix B). Each instance of newly created ToleranceSpecification (variable ?t) is mapped to the instance of FeatureSpecification by using an instance of FeatureQualityMap (variable ?ft).
In the second step, the rule ‘read-tolerance-measurement.q’ (fig. 6.8) reads the measurement for the tolerances by invoking the service ‘readFeatureToleranceMeasurement’, which receives the label of feature (variable ?fn) as well as the URI type of a tolerance (variable ?tt), selected by the where clause. The measurement value (variable ?tm), returned by the service, is asserted as data for a new instance of MeasurementBearingEntity, which inheres in the corresponding instance of ToleranceSpecification (variable ?d), selected by the where clause. The unit of every measurement is hard-coded (taken from the pre-defined instances of units from common core ontology) and asserted as data for the newly created instance of MeasurementBearingEntity.
6.2.1.2 Querying Part Design Graph

A RDF graph with complete part design specifications can be queried to retrieve various information, e.g. features, their dimensions, and associated tolerances. A separate command-line program in SIMPlanner allows users to run various SPARQL queries on the specification graph. The program provides the following script to initiate different queries conveniently.

```
part -f <name of feature or '*' for all features> -d <dimension type or '*' for all dimensions> -t <tolerance type or '*' for all tolerances>
```

A sample run of the program is shown in fig. 6.9.
<table>
<thead>
<tr>
<th>FeatureSpecification</th>
<th>FeatureName</th>
<th>FeatureType</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="http://www.ohio.edu/simplanner/design2019/9/6/421158#FeatureSpecification_I7092">http://www.ohio.edu/simplanner/design2019/9/6/421158#FeatureSpecification_I7092</a></td>
<td>&quot;RECTANGULAR_SLOT(7)&quot;</td>
<td>&quot;Slot&quot;</td>
</tr>
<tr>
<td><a href="http://www.ohio.edu/simplanner/design2019/9/6/421158#FeatureSpecification_I7076">http://www.ohio.edu/simplanner/design2019/9/6/421158#FeatureSpecification_I7076</a></td>
<td>&quot;RECTANGULAR_POCKET(3)&quot;</td>
<td>&quot;Pocket&quot;</td>
</tr>
<tr>
<td><a href="http://www.ohio.edu/simplanner/design2019/9/6/421158#FeatureSpecification_I1898">http://www.ohio.edu/simplanner/design2019/9/6/421158#FeatureSpecification_I1898</a></td>
<td>&quot;SIMPLE_HOLE(4)&quot;</td>
<td>&quot;Hole&quot;</td>
</tr>
<tr>
<td><a href="http://www.ohio.edu/simplanner/design2019/9/6/421158#FeatureSpecification_I1740">http://www.ohio.edu/simplanner/design2019/9/6/421158#FeatureSpecification_I1740</a></td>
<td>&quot;RECTANGULAR_POCKET(2)&quot;</td>
<td>&quot;Slab&quot;</td>
</tr>
</tbody>
</table>

Figure 6.9: Query result of sample part specification graph by `part` command.

### 6.2.2 Process Capability Knowledge

The process capability knowledge-base consists of process individuals and associated capabilities. Process individuals are associated function individuals and two types of capability are included in the knowledge-base. The relationships between process, function, and capability individuals follows the capability ontology model, presented in section 5.4.1. Every types of process capability, used in this study, contains upper and
lower limit of the capability as described in fig. 5.6. However, some capability used
equations to express one of its limits instead of crisp limit. The equation information is
asserted following the model described in fig. 5.7.

SIMPlanner provides script to create and read capability knowledge-base. Various
commands for creating and reading capability are given below. The script for loading
capability assumes that process individuals already exist in the knowledge-base (A-Box).
This study uses the domain level ontology for process, function and its capability (T-Box)
stored under IRI http://www.ohio.edu/ontologies/manufacturing-capability#. The
individuals are stored under IRI http://www.ohio.edu/ontologies/capability-implanner#.

6.2.2.1 Process Capability Loader

6.2.2.1.1 Create new function individual  The command process with parameter
-new creates a new function individual for a given function type with argument -p and
assert realizes relationship between a given process individual with argument -func and
the newly created function individual.

$ process -new -p <process individual> -func <function type>

For example, the following command, shown in fig. 6.10, creates a new function
individual for the function type
http://www.ohio.edu/ontologies/manufacturing-capability#PeripheralRoughing and assert
that process individual http://www.ohio.edu/ontologies/capability-implanner#endmillin\nprotect\discretionary{\char\hyphenchar\font}{}}{}}g0101 realizes the new function
individual, which is returned by the command as shown in fig. 6.10.

6.2.2.1.2 Create new capability  The command process with parameter -new creates
new capability individual for a given capability type with argument -capa. The capability
information is applied by additional parameters, which are described in table 6.3.
For example, the command in fig. 6.11 creates a new capability for the given type http://www.ohio.edu/ontologies/manufacturing-capability#Flatness with lower limit 0.0254 mm and upper limit as infinity (we used a large double value i.e. 99999999999.99 mm). The capability demarcates the function individual http://www.ohio.edu/ontologies/capability-implanner#peripheralroughing5703 and predicts limit for the specification http://www.ohio.edu/ontologies/design#FlatnessSpecification. Upon successful creation of the capability instance, the asserted information are returned by the command.

In a practical cloud manufacturing environment, more than one combination of machine and tool can bear same type of function, publishing slightly different set of capabilities. However, we consider only one set of capability for each function type. The capabilities are loaded in both millimeter and inches because sample design specifications used in this evaluation uses either of these two units. For each Information Content Entity, containing the values and equations for the capabilities, two different Information Bearing Entity (s) are used to express the same value or equation in
millimeter and inch. During planning, one of these two Information Bearing Entity(s) is used, as values in a design specification are expressed in only one type of unit. Although the process, function, capability instances may be sourced from different sources, every instance is created under the IRI 
http://www.ohio.edu/ontologies/capability-implanner (prefix default ¹⁹). These instances follow the capability ontology (T-Box) http://www.ohio.edu/ontologies/manufacturing-capability (prefix capa), which is described in section 5.4. The association of process instances to function instances are given in table 6.4. The instance in the column ‘Process Instance’ realizes the instance in column ‘Function Instance’. The types of the process and function instances are given column ‘Process Type’ and ‘Function Type’ respectively.

Figure 6.12: Every capability demarcating the HoleMaking type of function for process type TwistDrilling.

¹⁹ The default prefix is replaced by space in the tables and figures.
Each function instance is demarcated by a set of capabilities. The commands given in section 6.2.2 for reading the process capability can be used to read the capability values for each process and function combination. In order to illustrate how capabilities are stored in the graph, every capability instance demarcating the HoleMaking instance, which is realized by the TwistDrilling instance, are shown in fig. 6.12. The graph is generated by the protégé plugin tool ‘OntoGraf’, plotting directly from A-Box at http://www.ohio.edu/ontologies/capability-implanner. The types of the capabilities are also shown in the same figure.

The capabilities used for this evaluation are NonSelfReferringCapability and contains a range for expressing the possible change of quality in the workpiece. The minimum and maximum values for the ranges are either fixed value or an equation. In order to illustrate these two types of information structure, two capability instances from fig. 6.12 are further expanded in creffig-twist-round-postol. The instance roundness6682 is a RoundnessCapability and can yield as low as 0.004 in or 0.1016 mm roundness in the hole, produced by the process instance twistdrilling0101. As any roundness specification greater than the minimum threshold can be satisfied by this capability instance, the maximum is fixed at a large number 9.999999999999E10. The minimum threshold for the capability instance positivetolerance2649 of type PositiveToleranceCapability is expressed as an equation. The equation is expressed as JESS (MathML) format, e.g. "(+ (* 0.007(sqrt ?arg1)) 0.003)" in millimeter. The argument ('?arg1') for the equation is asserted to the Information Bearing Entity with relationship expects. The maximum threshold for positivetolerance2649 is also set at a large number. For more information on the information structure, section 5.4.3 can be referred. The fig. 6.13 shows only the minimum Information Bearing Entity (s). The data property values and units linked to the Information Bearing Entity (s) are written in text boxes, because of the limitation of OntoGraf tool to display data properties.
Figure 6.13: The capability information structure for type RoundnessCapability and PositiveToleranceCapability demarcating instance holemaking3118

### 6.2.2.2 Querying Process Capability

For a given knowledge-base, a set of commands can parse the processes and their capabilities. In fact, when the parameters \(-p\) and \(-func\) are applied with some process and function type respectively and without the \(-new\) parameter, the program parses the KB to find all individuals for the given process and function type. If the IRIs are not known, every process and function instances are returned by the query if the parameters are applied with wildcard (\(*\)).
A list of drilling processes and functions is extracted from drilling capability KB in fig. 6.14. By applying a particular capability type (wildcard for every existing capability) with parameter `-capa`, the program returns the capability information associated with every function individual, *demarcated* by the given capability type. In fig. 6.15, the capability instances and associated ranges are returned for TwistDrilling process and HoleMaking function. A list of functions types, associated with process types, is given in table 6.4.

```
<table>
<thead>
<tr>
<th>Process</th>
<th>FunctionType</th>
<th>ProcessType</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>:gundrilling0101</td>
<td>j.2:HoleMaking</td>
<td>:gundrilling4318</td>
<td></td>
</tr>
<tr>
<td>:boring0101</td>
<td>j.2:HoleImproving</td>
<td>:boring0101</td>
<td></td>
</tr>
<tr>
<td>:enddrilling0101</td>
<td>j.2:HoleImproving</td>
<td>:enddrilling0101</td>
<td></td>
</tr>
<tr>
<td>:honing0101</td>
<td>j.2:HoleMaking</td>
<td>:honing0101</td>
<td></td>
</tr>
<tr>
<td>:twistdrilling0101</td>
<td>j.2:HoleMaking</td>
<td>:twistdrilling0101</td>
<td></td>
</tr>
<tr>
<td>:centerdrilling0101</td>
<td>j.2:HoleStarting</td>
<td>:centerdrilling0101</td>
<td></td>
</tr>
<tr>
<td>:reaming0101</td>
<td>j.2:HoleImproving</td>
<td>:reaming0101</td>
<td></td>
</tr>
<tr>
<td>:spadeboring0101</td>
<td>j.2:HoleImproving</td>
<td>:spadeboring0101</td>
<td></td>
</tr>
<tr>
<td>:precisionboring0101</td>
<td>j.2:HoleImproving</td>
<td>:precisionboring0101</td>
<td></td>
</tr>
<tr>
<td>:spotdrilling0101</td>
<td>j.2:HoleImproving</td>
<td>:spotdrilling0101</td>
<td></td>
</tr>
<tr>
<td>:holegrinding0101</td>
<td>j.2:HoleImproving</td>
<td>:holegrinding0101</td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 6.14: Read processes and functions by script
Table 6.3: Parameters for applying the capability information for a new capability

<table>
<thead>
<tr>
<th>Argument</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>-max</td>
<td>Maximum measurement limit for the capability</td>
</tr>
<tr>
<td>-min</td>
<td>Minimum measurement limit for the capability</td>
</tr>
<tr>
<td>-eq</td>
<td>Sets equation as maximum or minimum limit along with argument types</td>
</tr>
<tr>
<td>-ref</td>
<td>The quality type for which the capability predicts change of state for</td>
</tr>
<tr>
<td>-unit/-u</td>
<td>Set the unit for the given measurement of the limit</td>
</tr>
</tbody>
</table>
Table 6.4: Associations among process and function instances.

<table>
<thead>
<tr>
<th>Process Type</th>
<th>Process Instance</th>
<th>Function Type</th>
<th>Function Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>capa:CenterDrilling</td>
<td>:centerdrilling0101</td>
<td>capa:HoleStarting</td>
<td>:holestarting8349</td>
</tr>
<tr>
<td>capa:SpotDrilling</td>
<td>:spotdrilling0101</td>
<td>capa:HoleStarting</td>
<td>:holestarting6108</td>
</tr>
<tr>
<td>capa:SpadeDrilling</td>
<td>:spadedrilling0101</td>
<td>capa:HoleMaking</td>
<td>:holemaking8001</td>
</tr>
<tr>
<td>capa:TwistDrilling</td>
<td>:twistdrilling0101</td>
<td>capa:HoleMaking</td>
<td>:holemaking3118</td>
</tr>
<tr>
<td>capa:EndDrilling</td>
<td>:enddrilling0101</td>
<td>capa:HoleMaking</td>
<td>:holemaking3118</td>
</tr>
<tr>
<td>capa:GunDrilling</td>
<td>:gundrilling0101</td>
<td>capa:HoleMaking</td>
<td>:holemaking4318</td>
</tr>
<tr>
<td>capa:Boring</td>
<td>:boring0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving1867</td>
</tr>
<tr>
<td>capa:Honing</td>
<td>:honning0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving6715</td>
</tr>
<tr>
<td>capa:PrecisionBoring</td>
<td>:precboring0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving7947</td>
</tr>
<tr>
<td>capa:Reaming</td>
<td>:reaming0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving9211</td>
</tr>
<tr>
<td>capa:HoleGrinding</td>
<td>:holegrinding0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving1637</td>
</tr>
<tr>
<td>capa:PlungeMilling</td>
<td>:plungemilling0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving1637</td>
</tr>
<tr>
<td>capa:EndMilling</td>
<td>:endmilling0102</td>
<td>capa:HoleImproving</td>
<td>:holeimproving9879</td>
</tr>
<tr>
<td>capa:EndMilling</td>
<td>:endmilling0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving5703</td>
</tr>
<tr>
<td>capa:EndMilling</td>
<td>:endmilling0104</td>
<td>capa:HoleImproving</td>
<td>:holeimproving9134</td>
</tr>
<tr>
<td>capa:SideMilling</td>
<td>:sidemilling0102</td>
<td>capa:HoleImproving</td>
<td>:holeimproving6920</td>
</tr>
<tr>
<td>capa:SideMilling</td>
<td>:sidemilling0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving6586</td>
</tr>
<tr>
<td>capa:SideMilling</td>
<td>:sidemilling0102</td>
<td>capa:HoleImproving</td>
<td>:holeimproving6920</td>
</tr>
<tr>
<td>capa:SideMilling</td>
<td>:sidemilling0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving6586</td>
</tr>
<tr>
<td>capa:SlabMilling</td>
<td>:slabmilling0102</td>
<td>capa:HoleImproving</td>
<td>:holeimproving8250</td>
</tr>
<tr>
<td>capa:SlabMilling</td>
<td>:slabmilling0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving9878</td>
</tr>
<tr>
<td>capa:FaceMilling</td>
<td>:facemilling0102</td>
<td>capa:HoleImproving</td>
<td>:holeimproving1087</td>
</tr>
<tr>
<td>capa:FaceMilling</td>
<td>:facemilling0101</td>
<td>capa:HoleImproving</td>
<td>:holeimproving7067</td>
</tr>
</tbody>
</table>
Figure 6.15: Read capability by script
6.3 Process Planning Agents

The scope of the process planner, designed for this study, is to achieve the following: for a given set design specification and a set of process, a subset of the available processes are selected and necessary orderings among the occurrences of those selected processes are calculated, such that when the selected occurrences are applied maintaining the specified order, a physical part, which meets the given specifications, is produced. The set of suitable processes are selected by matching the capability of processes with the specifications. The ordering among occurrences are calculated based on two ordering constrains: the necessary order of application among the processes and the processing order among the features. The rationality behind these two constraints has already been elucidated in section 5.5.5.3.

As explained in the section 5.4.1, the combinations of different machines and tools reflects in the process level capability. The methods for loading these process level capabilities into the knowledge-base is described in section 6.2.2. In this study no information about the particular shop-floor or manufacturer, who possess these resources, are not considered. It is assumed that some combination of these resources exist for each set of process level capabilities, irrespective of the fact that they can either be situated in geographically dispersed locations or under a single shop-floor. The particular process planning development thus can be used for both centralized and distributed planning. The most important driver for such flexibility is the availability of process and resource related information as linked data, backed by a foundation ontology, which further aid in formulation of planning rules by using only the upper-level terms but still be able to search through the domain level information.

The structure of each process planning agent in a particular group follows generic templates. For example, one agent in the group ‘Feature Planning’ may handle only Hole type feature, and another may handle only Pocket type feature. Similarly, each agent in
‘process selector’ group handles the selection of a particular type of process. However, the planning rules are mostly similar for every agent in a group. Therefore, the discussions of the planning rules, belief structures, and services of the agents are limited to every generic structure, but still mention the parameters, which are variably adopted to create different types of concrete agents.

Apart from the global services, which are mainly responsible for the communication among the three types of planning agents, few other internal services are used by the agents in some cases. These internal services are implemented by the agents themselves and invoked by some of its own plan. These services are discussed in the sections devoted for the respective agent. The list of global services are given in table 6.5.
Table 6.5: Global services offered by planning agents

<table>
<thead>
<tr>
<th>Service and endpoint</th>
<th>Input Type</th>
<th>Output Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PartPlanner. ask_to_plan_part</td>
<td>design:PartSpecification</td>
<td>List</td>
<td>Returns the root occurrences of the planned occurrence tree for the given instance of design:PartSpecification</td>
</tr>
<tr>
<td>FeaturePlanner. ask_to_plan_feature</td>
<td>design:FeatureSpecification</td>
<td>List</td>
<td>Returns the root occurrences of the planned occurrence tree for the given instance of design:FeatureSpecification</td>
</tr>
<tr>
<td>ResourceSelector. ask_to_select_resource</td>
<td>design:FeatureSpecification, List</td>
<td>plan:PlannedProcess</td>
<td>Return an occurrence of the given process type if the given specification can be satisfied by the available resource.</td>
</tr>
</tbody>
</table>
In fig. 6.16, the purported sequence in the invocations of services and underlying executions of plans are shown as a flow chart. In reality, however, some services are invoked in parallel and agents performs these services in independent execution threads.

The global knowledge for the planning agent contains the design specification, which needs to be available for the agents to query from, before the planning starts. Different types of agents are interested in different portions of the specifications, which they extract from the global knowledge-base by executing a suitable query and uploads in their belief, when the corresponding service request is received. For example, the PartPlanner agent query the precedence among the constituent features from the design specification knowledge-base by executing the plan LOAD-FEATURE-PRECEDENCE.RQ and the 'process
selector' agent extracts the dimension and tolerance specification for a given feature by executing the plan `transfer-feature-specification.rq`. The occurrence tree, which depicts the plan with the alternative routing and order of the process occurrences (will be used to mean instance(s) of `plan:PlannedProcess` in further discussion) for each routing, is generated incrementally by different planning agents. Different portions of the occurrence tree thus resides in the beliefs of these agents, mainly the `PartPlanner` and the `FeaturePlanner` agents. As the beliefs of the agents are not persistent, the plan can be stored in a global knowledge-base for long term storage. Each of the occurrences in the tree also has physical features (instance of type `design:FormFeature`) as its input and output, which can be one of the types `design:IntermediateFeature` or `design:UnsatisfiedFormFeature` (section 5.5.5.2). The final output feature of a valid routing concretizes the corresponding instance of `FeatureRepresentation`. For sake of simplicity, no conformance criteria is applied in this development and it is assumed that if an instance of `FormFeature` concretizes an instance of `FeatureSpecification`, then it also conforms it. This is also followed for instances of `InformationQualityEntity`, which inheresIn a `FormFeature`. The instances of `design:FormFeature`, and their relationship with occurrences and specifications are stored in the part RDF.

### 6.3.1 Part Planning Agent

The ‘part planning’ agent offers the service `ask-to-plan-part` which receives the IRI of part specification to be planned. Upon receiving any such request, the ‘part planning’ agent creates the root of the occurrence tree. The root is a dummy occurrence of type `plan:PlannedProcess`, but differentiated by classifying as `plan:RootProcess`, from the other children occurrences. This root occurrence is available in the belief of ‘part planning’ agent in its initial state. An instance of `design:FormFeature` is also created (existential variable ?f0) and asserted as output (`cco:has_output`) of the root occurrence.
The rule, named ‘create-root-feature’ (fig. 6.17), creates a pair of feature specification and representation. These instances do not belong to the given design specification, but only created as placeholders to be used by the rule in fig. 6.20.

```
CONSTRUCT{
  cco:specified_by rdf:type owl:ObjectProperty.
  cco:has_output rdf:type owl:ObjectProperty.
  ?f0 rdf:type design:FormFeature.
  ?p0 cco:has_output ?f0.
  ?r0 rdf:type design:RootFeatureRepresentation;
    cco:specified_by ?f0.
}
WHERE{
  ?p0 rdf:type plan:RootProcess.
  ?p0 rdf:type plan:PlannedProcess.
}
```

Figure 6.17: Rule to create the root feature specification as output of the root occurrence and associated root feature representation

The instance of (RootFeatureRepresentation) is considered to be the root of the ‘feature precedence’ tree, which can be built by the partial ordering among the features of the part to be planned. As explained in ?? 5.5.5.3.1, these orderings are necessary to be maintained in the order of machining of the features, and therefore also guide the ordering of the occurrences in every possible routing planned for the part. Although these ordering relationship (design:has_succeeding_feature) can truly be assigned among machining feature, it is assumed that the design knowledge of the given part already contains these relationships asserted among instances of design:FeatureRepresentation. However, the initial feature representations - features not preceded by any other features - need to be assigned as succeeding feature of the root feature representation. This is necessary for the recursive planning rule to find the initial feature representations, as the planning starts from root feature representation. The rule, named ‘assign-root-representation’ (fig. 6.18) extracts all the instances of FeatureSpecification (?fs) of the given part, after binding
the given URI of the part specification to \( p \). The instances of `design:FeatureSpecification` is mapped to the instance of `design:PartSpecification`, by the instances of `PartFeatureMap`, as asserted by the rule given in fig. 6.3 in times of translating the design specification in RDF. The corresponding instances of `FeatureRepresentation` (\(?r\)) for every instance of `FeatureRepresentation` are queried and among them, only those, which are not succeeded by any other representation (\(?r1\)), are selected by applying the FILTER. In the CONSTRUCT, these selected instances of `FeatureRepresentation` are asserted as succeeding feature representation of the root `FeatureRepresentation` instance (\(?r0\)).

The root feature representation is already created by rule ‘create-root-feature’. The patterns for these two rules are graphically represented in fig. 6.19, which also provides some insight into the resulting RDF graph created after these two rules are executed successfully. The portion of the graph, which is bordered in blue, is asserted in the local knowledge-base (belief) of the ‘part planning’ agent. The green bordered portion shows the pattern, which is queried from the design specification RDF from the global knowledge-base.

```sparql
CONSTRUCT{
  plan:hasSucceedingFeature rdf:type owl:ObjectProperty.
  ?r0 rdf:type design:FeatureRepresentation.
  ?r0 plan:hasSucceedingFeature ?r.
}
WHERE{
  ?pf rdf:type design:PartFeatureMap;
    design:describes_map_with ?p;
    design:describes_map_with ?fs.
  ?fs rdf:type design:FeatureSpecification;
  ?r0 rdf:type design:RootFeatureRepresentation.
  FILTER NOT EXISTS{?r1 plan:hasSucceedingFeature ?r}
}
```

Figure 6.18: Rule to assign root feature representation as the preceding feature representation of the initial feature representations
At this stage, the state of the belief of the ‘part planning’ agent becomes suitable to trigger the core planning rule. Rule ‘plan-features-by-precedence.rq’ (fig. 6.20) is designed in such a way that it may trigger perpetually until there remains no feature to be planned. At each iteration, the rule selects a set of feature specifications, which are still not planned but also not preceded by any other unplanned feature specification. However, these features are not selected just by following the precedence among feature representation. Instead, the WHERE clause of the rule reads every alternative routing so far planned from the current state of the occurrence tree and for each routing, it selects the feature specifications, which are not yet processed by the routing but also not preceded by any other such feature as well. In other words, the SELECT query selects the next available set of feature specifications for each routing so far planned. The selection is highly complex and performed by two nested SELECT queries within the WHERE clause of the rule, which is elaborated in the following paragraphs.
The inner most select query (from line 33 to 46 in fig. 6.20) uses ‘property path’ to find every occurrence, which the root occurrence (variable \( ?p0 \)) \texttt{plan:precedes} and then filters out those occurrences, which do not \texttt{plan:precedes} any other occurrence in order to find the leaf occurrences of the current occurrence tree. This particular pattern, consisting of line 35-37, along with the FILTER clause in line 43, is a ubiquitous pattern, which is used in many rules to find the leaf node of a tree, starting from the root node. In this case, the pattern binds the leaf occurrences to the variable \( ?pCurrent \). Once the leaf occurrences are identified, every possible routing can be found by traversing the path from leaf occurrence to the root occurrence. This is performed by employing the inverse property path \( \texttt{plan:precedes}^* \) (zero or more), which binds all the occurrences in \( ?p3 \) for each routing, the last occurrence for which is denoted by each occurrence in \( ?pCurrent \).

Additionally, the feature representation (\( ?rc \)), connected to the output features (\( ?f \)) for the leaf occurrences as well as for each occurrences in every routing (\( ?r3 \)) are collected. The GROUP BY clause at line 45 groups the feature representations (\( ?r3 \)), covered in each routing by the representation associated with the leaf occurrence (\( ?rc \)). The SELECT clause counts the number of feature representations, covered in each routing and saves the count in variable \( ?frcount \).

The other nested query from line 20 to 31 wraps the innermost select query, and includes the FILTER clause at line 50. The WHERE clause of this select query finds the feature representations (\( ?r4 \)), already covered in each routing, by using the pattern from line 22 to 24, which is similar to the pattern employed by the innermost query from line 40 to 42. Then the query selects the set of available features by binding the set of features in variable \( ?rNext \), which are succeeding feature representations for some covered feature representations but still not planned by the corresponding routing (FILTER clause from line 27 to 31). Additionally, all features which precedes for every feature representations, bound in \( ?rNext \), are selected in variable \( ?rp \), by applying inverse property path
`plan:precedes+` (one or more) on the given precedence relationships among features in design specification. In the SELECT clause, the number (`?fpcount`) of the feature representations, returned in `?rp`, is counted for every feature representation, and bound to `?rNext` (GROUP BY clause in line 48).

Finally, the FILTER clause at line 50 filters only those pairs, bound in `?pCurrent` (last occurrence for every routing) and `?rNext` (next available feature representations to be planned), for which the count in `?frcount` is greater than or equal to the count in `?fpcount`. For every feature representation thus selected, the corresponding feature specification is bound to variable `?fs`. Each of these selected feature specifications is then submitted to a suitable feature planning service.

The service is selected automatically by the query by matching the type of the feature specification instance with the input type of the service. It is assumed that different ‘feature planning’ services, each capable of planning a certain type of feature, are available and known to this planning rule ‘plan-features-by-precedence.rq’. The list of services are given in the FUNCTION block of the rule. Each of these services selects a set of routings, each of which can process the feature completely. These routings can be represented by a occurrence tree. This occurrence tree forms a portion of the complete occurrence tree; therefore, can be called sub-occurrence tree, representing the alternative routes for a feature. Every ‘feature planning’ service returns the set of nodes, which are the direct children of the root occurrence of the sub-occurrence tree. On successful execution of the service, the CONSTRUCT part of the rule ‘plan-features-by-precedence.rq’ appends the returned list of occurrences as children (`plan:precedes`) of the current leaf processes of the current occurrence tree. The leaf occurrences are those, which are bound to `?pCurrent` for the input feature specification in `?fs`. 
CONSTRUCT { ?pCurrent plan:precedes ?pNext. }
WHERE { ?fs cco:represents ?rNext; cco:inheres_in ?i1; cco:inheres_in ?i2. ?i1 rdf:type design:LabelBearingEntity; cco:has_text_value ?fn. ?i2 rdf:type design:TypeBearingEntity; cco:has_URI_value ?ft. }

Figure 6.20: Rule to select features according to precedence among them.
The planning rule ‘plan-features-by-precedence.rq’ fires repeatedly until the occurrence tree is reached in a state, in which all features are covered by each of the possible routing. In order to expound more on the operation of this complex planning rule, three examples are given below to illustrate three states of occurrence tree: the initial, intermediate and final. The examples consider a simple set of feature representations: :r1, :r2, and :r3, for which :r1 and :r2 should be planned before :r3. The result of the query in rule fig. 6.20 for each of these three states of occurrence tree is given in table 6.6. In this table, the bindings of some of the important variables are shown and same bindings for a variable are not repeated for every rows of data. The table also groups the rows together to make them more presentable than what a typical SPARQL query returns.

At the initial state, the occurrence tree contains only the root occurrence :p0, which is linked to the root feature representation :r0. At this stage, there is only one routing, containing the root occurrence as its only member. Therefore, the query binds the root occurrence :p0 to the variable ?pCurrent as the leaf occurrence and :r0 to the variable ?r0. Trivially, the variable ?frcount has value 1, as there is only one member in this route. The next available feature representations of :r0 are :r1 and :r2, which are bound to the variable ?rNext. In the given feature precedence, only the root representation :r0 precedes both :r1 and :r2 (in ?rp). Therefore, the variable ?fpcount has value 1 for both values of ?rp. As the ?frcount is equal to ?fpcount for both values of ?rNext, these two feature representations are selected for planning in this iteration.

At the intermediate state, the occurrence tree already contains the planned occurrences for :r1 and :r2. One occurrence :p11 is planned for representation :r1 and two occurrences :p21 and :p22 are planned for representation :r2. These tree occurrences are added to the root occurrence :p0 as children. At this state, there are three routings, which has :p11, :p21, and :p22 as last occurrences. Therefore, these three occurrences are bound to the variable ?pCurrent, and the corresponding representations :r1 for :p11, and :r2 for
both :p21 and :p22 are bound to ?rc. As there are only two occurrences for every routings, each of which covers two representations, ?frcount has value 2 for every values in ?pCurrent. The list of representations, which is available next for the routing, ending with :p11, are :r1 (after :r0), :r2 (after :r0), and :r3 (after :r1 and :r2). For these values in ?rNext, the feature representations :r0, :r1, and :r2 precedes :r3 and :r0 precedes :r1 or :r2, in the given feature precedence. Trivially, the value :r1 in ?rp, along with the value :r1 in ?rNext, is not considered as it is already covered in the routing, ending with :p11. As there are three values in ?rp for the value :r3 in ?rNext, ?fpcount has value 3. This value of ?rNext is also not considered, because the corresponding count in ?frcount is less than the count of ?frcount. The only value in ?rNext, which can be planned for the routing with last occurrence :p11, is :r2. Similarly, :r1 is the only feature representation, which can be planned for the other two routings, ending with :p21, and :p22, respectively.

The occurrence tree at the final state has four routings with :p31, :p32, :p33, and :p34 as last occurrences. Each of these routing covers three representations; however, in different order. It can be noticed that the orders never violate the given precedence in the feature representations. It is easy to see that ?frcount has count 4 for every routing, that is the tree representations along with the root representation. Moreover, any representation, which can be selected for each of the routing, should be already covered by that routing. Therefore, it can be observed in the table 6.6 that every such row is removed by the filter at line 27.

6.3.2 Feature Planning Agent

The ‘feature planning’ type of agents offer services, which receives an instance of design:FeatureSpecification as input and produce plan for the feature. As different agents handle different types of features, the input type of the services offered by them reflect those feature types. For example, the service offered by HolePlanner agent,
Figure 6.21: RDF graph of initial, intermediate, and final state of occurrence tree for an example feature precedence. The legend on the left mark the type of the nodes and the relationships among them with different color codes, which is followed in the RDF graphs on the right.
which can produce a plan for drilling a hole, should have a type of its input as `design:HoleSpecification`. This way, when this service is invoked with an the instance of `design:FeatureSpecification`, the instance, received by the agent, is guaranteed to be a type of `design:HoleSpecification`. Primarily, the types of processes, which are applicable for generating a feature, differs from one type of feature to another. Therefore, different types of ‘feature planning’ agents consider different sets of processes, while trying to select suitable processes to apply, as well as the order, in which these processes should be applied. These two types of knowledge is inherent in each ‘feature planning’ agent, that they pre-exist in their belief or the local knowledge-base of the agent.

In this study, seven types of standard prismatic features, suitable for CNC machining are considered. Each type of ‘feature planning’ agent captures this knowledge as a rule, which reads all available process individuals for the compatible process types and assert the required order among them. The available process individuals are read from the process capability knowledge-base, which is prepared by using the tools mentioned in section 6.2.2. In reality, this process capability knowledge-base may be distributed among different manufacturers, as they might upload different set of capability, pertaining to the resources they own. In this study, it is assumed that the process capability KB is available to every ‘feature planning’ agent, similar to a global knowledge-base. For sake of brevity, the rule for loading the processes with precedence for only slab type of feature is shown in fig. 6.22). Rules for other types of features are similar but use different process and function types. The precedence among processes of different types are collected from literature and a comprehensive graph is shown in fig. 6.23, which includes both optional and required precedence.
Figure 6.22: Slab making processes and the order of application

Figure 6.23: Optional and required precedence among the processes
The primary rule in a ‘feature planning’ agent is ‘select-process-feature.rq’ (fig. 6.24), which selects suitable set of processes for the given feature specification. The rule recursively selects applicable set of processes from the available processes, respecting precedence and then tests if the capabilities of any process can satisfy the dimension or tolerance specifications of the feature. At every iteration, the processes, satisfying at least some of the specifications of the feature, are added to the occurrence tree by creating a new occurrence for such process type. The rule keeps on triggering repetitively, until no process remains in the precedence graph or the feature specification is already satisfied completely.

However, at every iteration, this rule delegates the task of ‘comparing capability and specification’ to different services, provided by a type of ‘process selector’ agent. In reality, these types of services are provided by the vendors, which owns the resources, because only the vendors should be responsible for confirming whether the given process type can be performed by any resource in their possessions. Furthermore, it is the owner of the resources, who is truly aware of the capabilities of the resources. Therefore, by delegating the task to the process selection services, the ‘feature planning’ agents do not require to be aware of the availability of resources and their process level capability.

If alternative processes are selected for the feature specification, the new occurrences form a tree like structure. Every path from the root of the tree to the leaf occurrence thus forms an alternative route for generating the feature completely. Each such route ends with the leaf occurrence, completely satisfying the feature specification, which means that every dimension and tolerance specifications of the feature is achieved by some occurrence in that routing.

The rule, given in fig. 6.24, attaches only a template service ‘ask_to_select_process’ in the FUNCTION block. This service is just a representative of many such service, which can be offered by different manufacturers. As every agent of type ‘feature planning’
handles a particular type of feature, only the compatible services for that type of the
feature specification can be selectively associated in this rule. However, no such strategy
is considered in this implementation. As a general strategy, these process selection
services receive an instance of design:FormFeature and an instance of the applicable
process type as input. If any resource(s) (from those the service aware of) can be found
suitable to perform the process type and the process level capability of that resource can
satisfy the feature specification at least partially or completely, then an instance of
design:FormFeature is returned as the output. If such output is received, the
CONSTRUCT part of the rule creates a new occurrence for the input process type, and
links the input instance of design:FormFeature as its input and the returned instance of
design:FormFeature as its output.

```prefix resource: <http://www.ohio.edu/services/resources/>
1
2
3 construct{
4 cco:has_output rdf:type owl:ObjectProperty.
5 cco:has_input rdf:type owl:ObjectProperty.
6 plan:precedes rdf:type owl:ObjectProperty.
7 \?f2 rdf:type design:FormFeature.
8 \?pNext1 rdf:type \?pType;
9 cco:has_output \?f2;
10 cco:has_input \?f1.
12 }
13 where{
14 \?p0 rdf:type plan:RootProcess.
15 \?p0 rdf:type plan:PlannedProcess.
16 \?p0 plan:precedes\* \?pCurrent.
17 \?pCurrent rdf:type \?pt.
18 \?pPrevious rdf:type \?pt.
19 \?pPrevious plan:hasSucceedingProcess \?pNext.
20 \?pNext rdf:type \?pType.
21 \?pCurrent cco:has_output \?f1.
22 \?f1 rdf:type design:IntermediateFormFeature.
23 filter not exists {\?pCurrent plan:precedes \?p2}.
24 filter not exists {\?f1 cco:concretizes \?fs}.
25 }
26 function{
27 \?f2 <- resource:ask_to_select_process(\?f1,\?pNext)
28 }
29 ```

Figure 6.24: Rule to generate the sub-occurrence tree for a given feature specification
In order to illustrate the working of the rule in fig. 6.24, a graphical representation of the same is given in fig. 6.25. In this figure, the orange colored nodes and arrows mark the pattern which is in the SELECT clause, and the orange colored nodes and relationships mark the pattern in the CONSTRUCT clause. The area bordered by orange dash-dotted line mark the patterns in the plan RDF and the area bordered by blue dash-dotted line marks the pattern in part RDF. The dashed arrow between two nodes signifies property path.

Similar to the rule in fig. 6.20, the leaf occurrences are queried in ?pCurrent by the pattern from line 19-21 and the FILTER in line 30. The leaf occurrences are the last occurrences of every alternative routing planned so far. However, only those routing are selected, for which the last occurrence still could not satisfy the feature specification completely. The FILTER in line at 31 removes every such routing, for which the output physical feature (instance of type design:FormFeature), bound in variable ?f1, cco:concretizes the given instance of feature specification. The feature specification is bound to the variable ?fs from outside of the rule with the received instance of feature specification, received by the feature planning service. At every iteration, the type of the process in variable ?pt of the occurrence tree is then matched with the types of available process instances, which are bound in variable ?pPrevious. For every value in ?pPrevious, the next available processes are bound in the variable ?pNext by the following the relationship plan:hasSucceedingProcess. The associated service is invoked by every pair of process instance in ?pNext and last intermediate feature in ?f1.

The output feature returned by the service is bound to variable ?f2, which is then linked to a new occurrence ?pNext1 of type in ?pType by cco:has_output. The occurrences in ?pNext1 is added as child of corresponding occurrence in ?pCurrent. The corresponding instance of design:FormFeature in ?f1 is linked to the new occurrence as input feature by cco:has_input.
6.3.3 Process Selector Agent

The agents of type ‘process selector’ offers a service end point, such as ‘ask_to_select_resource’, which receives a request from users (or another agent) for a confirmation on the suitability of a process for a feature specification. In our case, the job is specified by three different inputs: a feature specification - a placeholder of the geometry, dimension, and tolerance information, an input form feature, on which the process should be applied, and a type of process, which the resources need to apply to generate the feature. The generic implementation of this service receives a ‘stock’ feature or an intermediate feature as the input form feature. A stock feature signify that no physical feature has been generated yet and acts a placeholder for any such feature to be generated. The intermediate feature is a physical feature, which partially satisfy the given specifications for that feature in the product design. This particular type of FormFeature
is defined in Axiom 20. Every such instance of physical feature (of type `design:FormFeature`) is linked to the corresponding instance of `design:FeatureSpecification` by the pattern, shown in fig. 6.26 - an instance of `design:FormFeatureIdentifier` (a subtype of `Information Content Entity`) `cco:inheres_in` some instance of `design:LabelBearingEntity`, which bears the label of the given instance of `design:FeatureSpecification`. Therefore, the corresponding `design:FeatureSpecification` can be easily queried from an instance of the `design:FormFeature`. For this reason, the feature specification is not part of the input of this particular implementation of the ‘ask-to-select-resource’ service for this development. This is just a preference and may be different for another implementation without violating the generality.

Figure 6.26: Graphical representation of the rule fig. 6.24.

When the instance of `design:FormFeature` is received by the service, the detailed list of specifications for the feature is retrieved from the global knowledge-base by applying the rule given in fig. 6.27 and loaded in the local knowledge-base of the agent.
The rule copies the entire pattern in WHERE clause into the CONSTRUCT clause, enabling a direct copy of the specifications. However, by applying the FILTER in line 45 and 46, only those dimensions and tolerances of the feature are selected, which is not yet present in the given instance of design/FormFeature. This given instance is bound to the variable ?f1 before the rule is executed. The FILTER clause in line 45 and 46 excludes all the instances of design:QualitySpecification (supertype of both design:DimensionSpecification and design:ToleranceSpecification) in variable ?d, which some of the existing quality instance of the given physical feature eco:concretizes, that is already present in the input FormFeature. A graphical rendition of the rule is given in fig. 6.26. The area bordered by green dash-dotted line mark the patterns in the design specification RDF and the area bordered by blue dash-dotted line marks the pattern in part RDF. The black arrow between two nodes marks the pattern excluded by FILTER clause.
Next, the set of capabilities for the given process type is compared against the set of the capabilities. There can be myriad of ways, in which this comparison may be applied, depending on the type of the capability and how they are measured. In this development, most of the capability information are expressed as numerical range, which is described in section 5.4.3. Every instance of process capability is able to satisfy a specific kind of specification, which is linked to the instance of capability by cco:references. The rule,
given in fig. 6.28, simply tests whether the dimension of the corresponding specification of the feature lies within the maximum and minimum bounds of the capability by applying the FILTER in line 42. The pair of matching capability and specifications are selected by the type of the instance (?dimType) linked to the capability instance (?capa) by cco:references. The pattern to select the maximum and minimum bounds are similar and given in line 27-33 and line 35-41 respectively. The pattern follows the information structure of range capability, in which the maximum and minimum bound bearing entity (?maxIBE and ?minIBE) are contained by the corresponding information content entity variables ?maxICE, and ?minICE, distinguished by the maximum measurement ordinal ?maxOrd and ?minOrd, respectively.
Figure 6.28: Rule ‘specification-capability-matching-limit.rq’ for testing whether the feature has any specification whose value is contained by the range of a matching capability

The pattern of the rule ‘specification-capability-matching-limit’ (fig. 6.28) marks the specifications whose value is contained by the range of a matching capability, as described above. For every satisfied specification, the CONSTRUCT of the rule creates a new instance (?d1) of cco:InformationQualityEntity, which cco:concretizes that specification instance (?d). The instance of cco:InformationQualityEntity is a real quality, which inheres in the physical feature. As every such real quality instance is linked to the corresponding specification instance, every part produced by following the plan can

```sparql
CONSTRUCT{
  ?d1 rdf:type cco:InformationQualityEntity;
  cco:concretizes ?d.
}
WHERE{
  ?f rdf:type design:FeatureSpecification;
  cco:represents ?r.
  ?fq rdf:type design:FeatureQualityMap;
  design:describes_map_with ?f;
  design:describes_map_with ?d.
  ?d rdf:type ?dimType;
  cco:inheres_in ?dm;
  cco:represents ?rd.
  ?dm rdf:type design:MeasurementBearingEntity;
  cco:has_decimal_value ?dim.
  ?capa rdf:type ?capaType;
  cco:references ?ref.
  ?ref rdf:type ?dimType.
  ?capa cco:is_measured_by ?maxICE.
  ?maxICE cco:inheres_in ?maxIBE;
  cco:is_measured_by ?maxOrd;
  ?maxIBE rdf:type cco:InformationBearingEntity;
  cco:has_decimal_value ?max.

  ?capa cco:is_measured_by ?minICE.
  ?minICE cco:inheres_in ?minIBE;
  cco:is_measured_by ?minOrd;
  ?minIBE rdf:type cco:InformationBearingEntity;
  cco:has_decimal_value ?min.

  FILTER (?dim >= ?min && ?dim <= ?max)
}
```
be associated with the corresponding design specification to tiniest detail. Therefore, this relationship is crucial from the perspective of quality assurance. However, these relationships are used by the ‘process selector’ agent to determine whether the newly created physical feature is complete or partial in terms of the given specifications. In order to do so, the specifications, not satisfied by any of the capability for the given combination of resources, are marked by a negative relationship $cco: not\_concretized$. This is done by another rule, which selected all the specifications which is not $cco: concretize(\text{d})$ by the rule ‘specification-capability-matching-limit’. This rule is not displayed as it is trivial in nature.

A graphical rendition of the rule ‘specification-capability-matching-limit’ is given in fig. 6.29. The FILTER clause is given in a green box, which receives the variables selected by the SELECT query. This special rendition of FILTER clause signifies that a service may replace the FILTER clause in order to implement a more complex comparison.

Figure 6.29: Graphical representation of the rule fig. 6.28.
The rules given in fig. 6.30, fig. 6.31, and fig. 6.32 create an instance of design:FormFeature, and also assign subtype design:IntermediateFeature to those instances which do not satisfy the specifications completely, and design:UnsatisfiedFormFeature to those, which was completely rejected by the given combination of resources. The definition of type design:UnsatisfiedFeature is defined in Axiom 21.

These rules primarily rely on the relationships cco:concretize, and not_concretize. The rule ‘create-intermediate-feature.rq’ applies the FILTER in lines 19-27 to test whether there exists any specification (?dm1), which is cco:concretize(d) by some instance of cco:InformationQualityEntity (?d1) as well as any specification (?dm2), which is cco:concretize(d) by some instance of cco:InformationQualityEntity (?d2). If both tests succeeds, then the rule creates a new instance of design:IntermediateFormFeature (?f1), which refers to the same instance of design:LabelBearingEntity (?i1), associated with the feature specification (?f) by having a new instance of cco:FormFeatureIdentifier(?ice1). The new instance of cco:FormFeatureIdentifier cco:designates the new instance of design:IntermediateFormFeature.
CONSTRUCT{
  cco:designated_by rdf:type owl:ObjectProperty .
  ?f1 rdf:type design:FormFeature;
  rdf:type design:IntermediateFormFeature;
  cco:designated_by ?ice1.
  ?ice1 rdf:type cco:FormFeatureIdentifier;
  cco:inheres_in ?i1.
}

WHERE{
  ?f rdf:type design:FeatureSpecification;
  cco:inheres_in ?i1.
  ?i1 rdf:type design:LabelBearingEntity.
  FILTER EXISTS {?fq1 rdf:type design:FeatureQualityMap;
    design:describes_map_with ?f;
    design:describes_map_with ?dm1.
    ?d1 rdf:type cco:InformationQualityEntity;
    ?fq2 rdf:type design:FeatureQualityMap;
    design:describes_map_with ?f;
    design:describes_map_with ?dm2.
    ?d2 cco:not_concretizes ?dm2.}
}

Figure 6.30: Rule ‘create-intermediate-feature.rq’ for creating an intermediate feature.

The rule ‘create-final-feature.rq’ applies the FILTER in lines 19-27 to test if no specification (?dm2) is cco:not_concretize(d) by any instance of cco:InformationQualityEntity. If no such instance is found then the rule creates a new instance of design:FormFeature (?f1), which completely satisfy the the given feature specification. Such instance of design:FormFeature then cco:specified_by the corresponding instance of designFeatureRepresentation (?r). However, a link to the corresponding feature specification, through an instance of cco:FormFeatureIdentifier, is still maintained, which is similar to the intermediate feature.
CONSTRUCT{
cco:designated_by rdfs:domain cco:DesignFeature;
cco:designated_by rdfs:range cco:DesignFeature;
cco:concretizes rdfs:domain cco:DesignFeature;
cco:concretizes rdfs:range cco:DesignFeature;
cco:specified_by rdfs:domain cco:DesignFeature;
cco:specified_by rdfs:range cco:DesignFeature;
?f1 rdfs:domain design:FormFeature;
?f1 cco:designated_by ?ice1;
?f1 cco:concretizes ?f;
?r cco:specified_by ?f1;
?ice1 rdfs:domain cco:FormFeatureIdentifier;
?ice1 cco:inheres_in ?i1;
}

WHERE{
?f rdfs:domain design:FeatureSpecification;
?f cco:inheres_in ?i1;
?f cco:represents ?r;
?i1 rdfs:domain design:LabelBearingEntity;
FILTER NOT EXISTS {?fq2 rdfs:domain design:FeatureQualityMap;
?fq2 design:describes_map_with ?f;
?d2 rdfs:domain cco:InformationQualityEntity;
?d2 cco:not_concretizes ?dm2.}
}

Figure 6.31: Rule ‘create-final-feature.rq’ for creating a final feature.

The rule ‘create-unsatisfied-feature.rq’ applies the FILTER in lines 21-25 to test if no specification (?dm2) is cco:concretize(d) by any instance of cco:InformationQualityEntity. If no such instance is found then the rule creates a new instance of design:UnsatisfiedFormFeature (?f1). A link to the corresponding feature specification, through an instance of cco:FormFeatureIdentifier, is still maintained, which is similar to the intermediate feature.
CONSTRUCT{
  cco:concretizes rdf:type owl:ObjectProperty.
  cco:designated_by rdf:type owl:ObjectProperty.
  ?f1 rdf:type design:FormFeature;
  rdf:type design:UnsatisfiedFormFeature;
  cco:designated_by ?ice1;
  cco:concretizes ?f.
  ?ice1 rdf:type cco:FormFeatureIdentifier;
  cco:inheres_in ?i1.
}

WHERE{
  ?f rdf:type design:FeatureSpecification;
  cco:inheres_in ?i1.
  ?i1 rdf:type design:LabelBearingEntity.
  FILTER NOT EXISTS {?fq1 rdf:type design:FeatureQualityMap;
    design:describes_map_with ?f;
    design:describes_map_with ?dm1.
    ?di rdf:type cco:InformationQualityEntity;
    cco:concretizes ?dm1.}
}

Figure 6.32: Rule ‘create-unsatisfied-feature.rq’ for creating a unsatisfied feature.
Table 6.6: The result of query in fig. 6.20 performed on the initial, intermediate, and terminating states of the example, shown in fig. 6.21

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(ln 27)</td>
<td>(ln 50)</td>
</tr>
</tbody>
</table>

**Initial case**

<table>
<thead>
<tr>
<th>:p0</th>
<th>:r0</th>
<th>:p0</th>
<th>:r0</th>
<th>1</th>
<th>:r0</th>
<th>:r2</th>
<th>:r0</th>
<th>1</th>
<th>:r1</th>
<th>:r0</th>
<th>1</th>
</tr>
</thead>
</table>

**Intermediate case**

<table>
<thead>
<tr>
<th>:p11</th>
<th>:r1</th>
<th>:p0</th>
<th>:r0</th>
<th>2</th>
<th>:r0</th>
<th>:r1</th>
<th>:r0</th>
<th>1</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>:p11</td>
<td>:r1</td>
<td></td>
<td></td>
<td></td>
<td>:r2</td>
<td>:r0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r1</td>
<td>:r3</td>
<td>:r0</td>
<td>3</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>:p21</th>
<th>:r2</th>
<th>:p0</th>
<th>:r0</th>
<th>2</th>
<th>:r0</th>
<th>:r1</th>
<th>:r0</th>
<th>1</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>:p21</td>
<td>:r2</td>
<td></td>
<td></td>
<td></td>
<td>:r2</td>
<td>:r0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r2</td>
<td>:r3</td>
<td>:r0</td>
<td>3</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>:p22</th>
<th>:r2</th>
<th>:p0</th>
<th>:r0</th>
<th>2</th>
<th>:r0</th>
<th>:r1</th>
<th>:r0</th>
<th>1</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>:p22</td>
<td>:r2</td>
<td></td>
<td></td>
<td></td>
<td>:r2</td>
<td>:r0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r2</td>
<td>:r3</td>
<td>:r0</td>
<td>3</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Terminating case**

<table>
<thead>
<tr>
<th>:p31</th>
<th>:r3</th>
<th>:p31</th>
<th>:r3</th>
<th>4</th>
<th>:r2</th>
<th>:r3</th>
<th>:r0</th>
<th>3</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>:p22</td>
<td>:r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:p11</td>
<td>:r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:p0</td>
<td>:r0</td>
<td>:r1</td>
<td>:r3</td>
<td>:r0</td>
<td>3</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:r2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pCurrent</td>
<td>rc</td>
<td>p3</td>
<td>fcount</td>
<td>r4</td>
<td>rNext</td>
<td>rp</td>
<td>fpcount</td>
<td>FILTER</td>
<td>FILTER</td>
</tr>
<tr>
<td>----------</td>
<td>----</td>
<td>----</td>
<td>--------</td>
<td>----</td>
<td>-------</td>
<td>----</td>
<td>---------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Terminating case continued</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:p32</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:p33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>:p34</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
7 EVALUATION

7.1 Setup for Evaluation

In the last chapter, a detailed description of different members of SIMPlanner chapter 6, which are mainly scripts and agents for data transformation and planning, is provided. This chapter evaluates the efficacy and performance of this semantic process planning system, using several different design specifications. The design specifications are based on prismatic part designs, normally created by some CAD applications. These sample part designs are originally designed in Siemens NX CAD application and then analyzed by IMPlanner application to generate the corresponding design XML file, which contains the features, associated dimension and tolerances, and machining precedence among the features. These design XML files are transformed into RDF graph following the methods, described in section 6.2.1. These design XML and part RDF files are provided along with this document.

The goal of the SIMPlanner is to generate process plan for a given design based on available processes and associated capability profiles. As described in section 6.2.2.1, a number of capability profiles, catering to various types of CNC machining processes, are loaded in the system. These capability profiles are assumed to express process level capability, provided by some combination of manufacturing resources, such as CNC mill, drill, and compatible cutting tools. In order to concentrate on the selection of right kind of processes and the generation of valid sequence of them, no particular machine and tools is included in the plan. However, it is trivial to make associations to these actual resources, as the function individual (realized by a particular process) is borne by a specific combination of actual resources. For some of the experiments, the planning agents are restricted to consider only some of the process and associated capability profiles. These restrictions are mentioned in the individual experiment. In this way, these artificial
restrictions simulates the situations, in which different sets of cloud manufacturing resources may be available to the process planner.

As explained in fig. 6.1, the planning agents generate two separate RDF graphs: the plan RDF contains process occurrences and precedence relationships among them, essentially the complete occurrence tree as the plan representation; the part RDF contains the intermediate and final features generated as output of each process occurrence. The results of the experiments are given as an analysis of these two RDF graphs. Four types of outputs are used to convey the results of the planning, conducted under different experiment. First, portion of the plan is shown as raw data by projecting the RDF graph in OWLViz plotter of protége\textsuperscript{20} application, which can serve as editor of both OWL and RDF files. Second, the sub-occurrence tree is displayed with an in-planner plotter, which includes the output feature as colored node (cyan for type SatisfiedFormFeature, red for type UnsatisfiedFormFeature, and blue for IntermediateFormFeature), along with the occurrences. Third, an analyzer program is written to extract the individual routes from the occurrence tree and print them in a human-readable format. Fourth, complete occurrence tree is transformed into a CSV file, containing the precedence relationship, which is then loaded into yEd\textsuperscript{21} graph editor application to display them in a nice layout. We also include the plan and part RDF for each experiment as supplementary medium with this document.

7.2 Plan for Evaluation

In order to verify the accuracy of SIMPlanner, a set of experiment is designed to test various aspects of manufacturing process planning, such as selection of suitable process, selection of alternative processes, planning of multiple processes for satisfying a feature specification completely, and alternative routes for making a feature or a complete part,

\textsuperscript{20} www.protege.stanford.edu/
\textsuperscript{21} www.yworks.com/products/yed
containing multiple features. In the last set of experiments, some real part designs are used to demonstrate the practical capability of SIMPlanner. The origin of these part designs are discussed in the respective experiment. The following list provides a brief overview of the experiments.

1. Testing the suitability of a particular type of process for making a sample feature.

(a) Can twist drilling be applied to drill a hole with radius 1.0 mm and depth 4.0 mm?

(b) Can end milling be applied to make a open pocket with dimension tolerance 0.05 mm and surface finish 55 µm?

2. Testing whether alternative types of processes can be selected for making a sample feature.

(a) Which alternative processes can be selected to make a slab feature, requiring flatness 0.05 mm and surface finish 55 µm?

(b) Which alternative processes can be selected to make a slot feature with dimension tolerance 0.15 mm and surface finish 55 µm?

3. Testing whether a sample feature can be made completely with a set of processes in succession.

(a) How can a closed pocket with dimension tolerance 0.05 mm, flatness 0.05 mm and surface finish 55 µm be made completely?

(b) How can a closed slot with dimension tolerance 0.05 mm, flatness 0.05 mm and surface finish 40 µm be made completely?

4. Testing whether alternative routes can be selected to make a feature completely.
(a) What are the alternative ways to drill a hole of radius 1.0 mm, depth 2.0 mm, requiring roundness tolerance 0.003, and true position .000144 mm completely?

5. Testing whether a part with multiple feature can be produced completely by the available processes.

(a) Can a sample part with four features be produced completely with the available processes?

6. Testing whether the planner can find every alternative routes to produce a part with machining precedence among features.

(a) Planning alternative routes to produce Netex.

(b) Planning alternative routes to produce Slider.

(c) Planning alternative routes to produce NIST-CTC-01 part.

7.3 Experiment 1a

Can twist drilling be applied to drill a hole with radius 1.0 mm and depth 4.0 mm?

The sample part ‘part-1a’ has specifications for one hole feature with radius 1 mm and depth 4 mm. The specification is loaded into specification graph ‘part-1a.rdf’. The specification is read by the script, described in section 6.2.1.2 and the result is shown in fig. 7.1.

The plan is conducted by restricting the planner to consider only the capabilities associated with twist drilling, which realizes hole making function. This setup simulates the situation, in which only one drilling machine with one twist drill bit is available for making the hole ‘Test-Hole’. The plan is saved in the RDF graph ‘part-1a-plan.rdf’. The
Figure 7.1: Part specification loaded into ‘part-1a.rdf’.

sub-occurrence tree created for the feature, created by the in-process plot, is shown in fig. 7.2. It can be observed from the graph that the output feature of the only occurrence is colored in cyan, indicating that it is a final form feature. The result of the analyzer program reads one valid route, with only one occurrence in it. The presence of a valid routing proves that the design specification can be satisfied completely by the available twist drilling process. Upon close inspection of the plan rdf, it is found that the output feature of the occurrence of twist drilling has two qualities, which concretizes the depth and diameter specification instances of the feature. The concretizes relationship among the output feature, the hole specifications and its dimensions are shown in fig. 7.3, which is created by from the ‘part-1a-plan.rdf’ by OWLViz plotter in prot´ég´e.

7.4 Experiment 1b

Can end milling be applied to make a open pocket with dimension tolerance 0.05 mm and surface finish 55 µm?

The sample part ‘part-1b’ has specifications for one open pocket feature with dimension tolerance 0.05 mm and surface finish 55 µm. The specification is loaded into
Figure 7.2: Sub-occurrence tree of feature ‘Test-Hole’.

Figure 7.3: The dimensions of output feature created by the planned occurrence for ‘Test-Hole’.

specification graph ‘part-1b.rdf’. The specification is read by the script, described in section 6.2.1.2 and the result is shown in fig. 7.4.

The plan is conducted by restricting the planner to consider only the capabilities associated with end milling, which realizes the pocket roughing function. This setup
Writing the part graph at ...\part-1b.rdf
Part specification part-with-pocket-1b is loaded from ...\part-1b.xml

\$ part -f * -d * -t *

<table>
<thead>
<tr>
<th>FeatureSpecification</th>
<th>FeatureName</th>
<th>FeatureType</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;:FeatureSpecification_Iz4jgqk&gt;</td>
<td>&quot;Test-Pocket&quot;</td>
<td>&quot;Pocket&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FeatureName</th>
<th>DimensionType</th>
<th>Dimension</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Test-Pocket&quot;</td>
<td>&quot;bottomDistance&quot;</td>
<td>&quot;.1.0&quot;</td>
</tr>
<tr>
<td>&quot;Test-Pocket&quot;</td>
<td>&quot;pocketPoint&quot;</td>
<td>&quot;2 0 2&quot;</td>
</tr>
<tr>
<td>&quot;Test-Pocket&quot;</td>
<td>&quot;normal&quot;</td>
<td>&quot;0 -1 0&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FeatureName</th>
<th>ToleranceType</th>
<th>Tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Test-Pocket&quot;</td>
<td>&quot;dimensionTolerance&quot;</td>
<td>&quot;0.05&quot;<a href="">xsd:double</a></td>
</tr>
<tr>
<td>&quot;Test-Pocket&quot;</td>
<td>&quot;surfaceFinish&quot;</td>
<td>&quot;55.0&quot;<a href="">xsd:double</a></td>
</tr>
</tbody>
</table>

Figure 7.4: Part specification loaded into ‘part-1b.rdf’.

simulates the situation, in which only one milling machine with one end mill cutter is available for making the pocket ‘Test-Pocket’. The plan is saved in the RDF graph ‘part-1b-plan.rdf’. The sub-occurrence tree created for the feature, created by the in-process plot, is shown in fig. 7.5. It can be observed from the graph that the output feature of the only occurrence is colored in blue, indicating that it is an **IntermediateFormFeature**. The result of the analyzer program reads no valid route. The presence of no valid routing shows that the part cannot be completely processed by the available end milling process. Upon close inspection of the plan rdf, it is found that the output feature of the occurrence of end milling has only one quality, which **concretizes** the surface finish tolerance specification instances of the feature. The dimension tolerance specification (0.05 mm) is not satisfied by the end milling process, because the **PositiveToleranceCapability** for the process can ensure 0.0508 mm as minimum. Therefore, the feature is partially satisfied by the end milling process. The **concretizes** relationship among the output feature, the pocket specifications and its dimensions are
shown in fig. 7.6, which is created by from the ‘part-1b-plan.rdf’ by OWLViz plotter in protégé.

Figure 7.5: Sub-occurrence tree of feature ‘Test-Pocket’.

Figure 7.6: The dimensions of output feature created by the planned occurrence for ‘Test-Pocket’.
7.5 Experiment 2a

Which alternative processes can be selected to make a slab feature, requiring flatness 0.05 mm and surface finish 55 \( \mu \text{m} \)?

The sample part ‘part-2a’ has specifications for one slab feature with tolerance requirement of flatness 0.05 mm and surface finish 55 \( \mu \text{m} \). The specification is loaded into specification graph ‘part-2a.rdf’. The specification is read by the script, described in section 6.2.1.2 and the result is shown in fig. 7.7.

\$ part -g "...\part-2a.rdf" -file "...\part-2a.xml" -u mm
Writing the part graph at ...\part-2a.rdf
Part specification part-with-slab-2a is loaded from ...\part-2a.xml
\$ part -f * -d * -t *

\| FeatureSpecification | FeatureName | FeatureType |
\|----------------------|------------|------------|
\| <:FeatureSpecification_Iax9z0d> | "Test-Slab" | "Slab" |
\|----------------------|------------|------------|
\| FeatureName | DimensionType | Dimension |
\|------------|----------------|------------|
\| "Test-Slab" | "pocketPoint" | "2 3 5" |
\| "Test-Slab" | "bottomDistance" | "-1.0" |
\| "Test-Slab" | "normal" | "0 0 1" |
\|------------|----------------|------------|
\| FeatureName | ToleranceType | Tolerance |
\|------------|----------------|------------|
\| "Test-Slab" | "flatness" | "0.05"<xsd:double> |
\| "Test-Slab" | "surfaceFinish" | "55.0"<xsd:double> |

Figure 7.7: Part specification loaded into ‘part-2a.rdf’.

The plan is conducted by allowing the planner to consider alternative processes, that are face milling and slab milling, both of which can realize surface roughing. We want to test whether the planner can select alternative processes for satisfying the tolerance requirement of the feature ‘Test-Slab’. The plan is saved in the RDF graph ‘part-2a-plan.rdf’. The sub-occurrence tree created for the feature, created by the in-process plot, is shown in fig. 7.8. It can be observed from the graph that the output...
features of both occurrences are colored in blue, indicating that the feature can be completely generated by both of these processes. Also, these occurrences are alternative to each other as they are siblings (both are child of root occurrence). The result of the analyzer program reads two valid routes, with one occurrence in each. Upon close inspection of the plan rdf, it is found that the output features of both occurrences have the specified tolerances. This is because both face milling and slab milling process can produce minimum flatness of 0.0254 mm and surface finish of 50 \( \mu \text{m} \), which is less than the specified values. Therefore, the rule in fig. 6.28 assert these two specifications for the output features for both of these processes. The concretizes relationship among the output features, the slab specification and its dimensions are shown in fig. 7.9, which is created by from the ‘part-2a-plan.rdf’ by OWLViz plotter in protégé.

Figure 7.8: Sub-occurrence tree of feature ‘Test-Slab’.
7.6 Experiment 2b

Which alternative processes can be selected to make a slot feature with dimension tolerance 0.15 mm and surface finish 55 $\mu$m?

The sample part ‘part-2b’ has specifications for one slot feature with dimension tolerance 0.15 mm and surface finish 55 $\mu$m. The specification is loaded into specification graph ‘part-2b.rdf’. The specification is read by the script, described in section 6.2.1.2 and the result is shown in fig. 7.10.

The plan is conducted by allowing the planner to consider the alternative processes, that are end milling and side milling, both of which can realize slot roughing function for the associated capabilities. It is assumed that the planner cannot apply processes for slot finishing because necessary tools are not available. We want to test whether the planner can select alternative processes for satisfying the tolerance requirement of the feature ‘Test-Slot’. The plan is saved in the RDF graph ‘part-2b-plan.rdf’. The sub-occurrence
Figure 7.10: Part specification loaded into ‘part-2b.rdf’.

tree for the feature, created by the in-process plot, is shown in fig. 7.11. It can be observed from the graph that the output feature of the occurrence SideMilling_Iq1ngy0 is green, indicating that it is a SatisfiedFormFeature, and the output feature of the occurrence EndMilling_Im0nerp is blue, indicating that it is an IntermediateFormFeature. This tells us that only the occurrence of SideMilling can produce the slab completely, whereas the occurrence of EndMilling only partially satisfies the required tolerances. The analyzer program reads only one valid routes, which means that no alternative processes are available for making the feature ‘Test-Slot’. The feature can be produced in only one way. Upon close inspection of the plan rdf, it is found that the output feature of SideMilling_Iq1ngy0 concretizes both positive tolerance and surface finish, whereas the output feature of EndMilling_Im0nerp concreties only the positive tolerance. This is because end milling and sidelmilling can achieve dimension tolerance of 0.106 mm and 0.0508 mm, both less than the specified tolerance value of 0.15 mm. However, the
specified surface finish of 55 mm can only be satisfied by side milling as it can achieve minimum of 50 µm but not by end milling as it can achieve minimum of 50 µm.

Therefore, the rule in fig. 6.28 assert these two specifications for the output features for both of these processes. The concretizes relationship among the output features, the slot specification and its dimensions are shown in fig. 7.12, which is created by from the ‘part-2b-plan.rdf’ by OWLViz plotter in protégé.

Figure 7.11: Sub-occurrence tree of feature ‘Test-Slot’.

7.7 Experiment 3a

How can a closed pocket with dimension tolerance 0.05 mm, flatness 0.05 mm and surface finish 55 µm be made completely?

The sample part ‘part-3a’ has specifications for one pocket feature with dimension tolerance 0.05 mm, flatness 0.05 mm and surface finish 55 µm. The specification is loaded into specification graph ‘part-3a.rdf’. The specification is read by the script, described in section 6.2.1.2 and the result is shown in fig. 7.13.
The plan is conducted by allowing the planner to consider three process configuration, that are plunge milling realizing plunging by using plunging tool and two end milling process instances, one of which realizes roughing function (type \textbf{PocketRoughing}) by using a rough cutter and another realizing finishing (type \textbf{PocketFinishing}) by a finish cutter for the associated capabilities). The plunging should be applied first for achieving a deep cut in z direction, followed by rough end milling, and then finish end milling. We want to test whether the feature ‘Test-Pocket’ can be completely made by the capabilities of the available resources for the processes, when applied in the aforementioned sequence.

The plan is saved in the RDF graph ‘part-3a-plan.rdf’. The sub-occurrence tree for the feature, created by the in-planner plot, is shown in fig. 7.14. The occurrence tree contains three occurrences, which should occur in sequence, as dictated by \textit{precedes}.
Figure 7.13: Part specification loaded into ‘part-3a.rdf’.

relationship. The output feature of the leaf occurrence EndMilling_Ia369zq is green in

color, which indicates that the feature can be processed completely by applying the

occurrences in succession. The output features of the earlier occurrence are blue in color,

meaning that they are IntermediateFormFeature. Upon close inspection of the plan rdf,

it can be found that the output of the occurrence PlungeMilling_Ijn1jp5 concretizes the

depth specification. The output of the occurrence EndMilling_Ifr6ynh concretizes the

surface finish tolerance in addition to the depth specification. Finally, the output of the

occurrence EndMilling_Ia369zq concretizes the rest of the specifications, which are

flatness and positive tolerance. The concretizes relationship among the output features, the

slot specification and its dimensions are shown in fig. 7.15, which is created by from the

‘part-2b-plan.rdf’ by OWLViz plotter in protégé.
7.8 Experiment 3b

How can a closed slot with dimension tolerance 0.05 mm, flatness 0.05 mm and surface finish 40 \( \mu \)m be made completely?

The sample part ‘part-3b’ has specifications for one pocket feature with dimension tolerance 0.05 mm, flatness 0.05 mm and surface finish 40 \( \mu \)m. The specification is loaded into specification graph ‘part-3b.rdf’. The specification is read by the script, described in section 6.2.1.2 and the result is shown in fig. 7.16.
Figure 7.15: The tolerances of output features created by the planned occurrences for ‘Test-Pocket’.

The plan is conducted by allowing the planner to consider three process configuration, that are plunge milling realizing plunging by using plunging tool and two
side milling process instances, one of which realizes roughing function (type \textbf{SlotRoughing}) by using a rough cutter and another realizing finishing (type \textbf{SlotFinishing}) by a finish cutter for the associated capabilities). The plunging should be applied first for achieving a deep cut in z direction, followed by rough side milling, and then finish side milling. We want to test whether the feature ‘Test-Slot’ can be completely made by the capabilities of the available resources for the given processes, when applied in the aforementioned sequence.

The plan is saved in the RDF graph ‘part-3b-plan.rdf’. The sub-occurrence tree for the feature, created by the in-planner plot, is shown in fig. 7.17. The occurrence tree contains three occurrences, which should occur in sequence, as dictated by \textit{pecedes} relationship. The output feature of the leaf occurrence \texttt{SideMilling.1rgxgzx} is green in color, which indicates that the feature can be processed completely by applying the occurrences in succession. The output features of the earlier the occurrence are blue in color, meaning that they are \textbf{IntermediateFormFeature}. Upon close inspection of the plan rdf, it can be found that the output of the occurrence \texttt{PlungeMilling.13xdkcm} \textit{concretizes} the depth specification. The output of the occurrence \texttt{SideMilling.1m44nc0} \textit{concretizes} the flatness tolerance in addition to the depth specification. Finally, the output of the occurrence \texttt{SideMilling.1qnbqy5} \textit{concretizes} the rest of the specifications, which are surface finish and positive tolerance. The \textit{concretizes} relationship among the output features, the closed slot specification and its dimensions are shown in fig. 7.18, which is created by from the ‘part-3b-plan.rdf’ by OWLViz plotter in protégé.

\section*{7.9 Experiment 4a}

\textbf{What are the alternative ways to drill a hole of radius 1.0 mm, depth 2.0 mm, requiring roundness tolerance 0.003, and true position 0.000144 mm completely?}
The sample part ‘part-4a’ has specifications for one hole feature with radius 1.0 mm, depth 2.0 mm, requiring roundness tolerance 0.003, and true position $1.44 \times 10^{-4}$ mm. The specification is loaded into specification graph ‘part-4a.rdf’. The specification is read by the script, described in section 6.2.1.2 and the result is shown in fig. 7.19.
The planning for the part is conducted by allowing the planner to consider every available processes realizing hole making functions, such as functions of types `HoleStarting`, `HoleMaking`, `HoleImproving`, and `HoleFinishing`. However, the processes maintain a order in their application, following the precedence graph, shown in fig. 6.23. We want to test whether the planner can find every possible ways the feature ‘Test-Hole’ can be drilled, such that the specifications for the feature are satisfied completely.

The sub-occurrence tree for the feature, created by the in-planner plot, is shown in fig. 7.20. It can be observed that output features of every leaf occurrences are colored red (of type `UnsatisfiedFeature`), indicating that no routing is found to satisfy the specifications completely. upon analysis of the resulting plan RDF, which is saved in ‘part-4a-plan1a.rdf’, it can be found that the tolerance specification of true position cannot be satisfied by any available process, as per their capability profile. We can say that the
tolerance specification for this part is too tight. In order to prove this hypothesis, it can be tested whether the part can be completely satisfied by the same set of available processes, if the true position specification is relaxed.

![Sub-occurrence tree of feature 'Test-Hole' with specified true position tolerance 0.000144 mm.](image)

After performing the planning on the same part specifications, except the true position tolerance specification being 0.003 mm, some valid routes are found in the resulting sub-occurrence tree, given in fig. 7.21. This is because output features of some leaf occurrences are colored green, indicating that they are of type **FinalFeature** and they
conform to every specification for the part, including the true position tolerance. The resulting plan RDF for the relaxed specification is saved in ‘part-4a-003-plan1a.rdf’.

Figure 7.21: Sub-occurrence tree of feature ‘Test-Hole’ with specified true position tolerance 000144 mm.

We analyzed the ‘part-4a-003-plan1a.rdf’ with the analyzer program, mentioned in crefsec-eval-setup. The program picks up three valid routes with the following sequence of occurrences, agreeing to the observation from the in-planner graph, shown in fig. 7.21. As this particular specification contains narrow true position tolerance, the planner includes spot drilling as the first process to be applied for every possible route. However it
is required, the true position tolerance does not get satisfied by the application of spot drilling only and needs subsequent processes to be able to achieve such requirement. For this reason, in spite of applying spot drilling, the original requirement of true position 0.000144 mm could not be satisfied by any of the subsequent processes, as it was too narrow for their available range of capability.

1 Number of routings planned : 3
2 Number of occurrences in the routing ending with Reaming_Ir99r1 : 4
3 SpotDrilling_Inbz6yx@Test-Hole -> SpadeDrilling_Iyehhhc@Test-Hole ->
4 PrecisionBoring_Iz3fn6a@Test-Hole -> Reaming_Ihay00r@Test-Hole
5 Number of occurrences in the routing ending with Reaming_I0xgkkm : 4
6 SpotDrilling_Inbz6yx@Test-Hole -> EndDrilling_Ifkexn10@Test-Hole ->
7 PrecisionBoring_I17pyd20@Test-Hole -> Reaming_Iqa15a30@Test-Hole
8 Number of occurrences in the routing ending with PrecisionBoring_Ij5f3e1 : 3
9 SpotDrilling_Inbz6yx@Test-Hole -> TwistDrilling_I5km5jb@Test-Hole ->
10 PrecisionBoring_Ime66c6w@Test-Hole

7.10 Experiment 5a

Can a sample part with four features be produced completely with the available processes?

The sample part ‘part-5a’ has four features of type SlabSpecification, PocketSpecification, HoleSpecification, and SlotSpecification. The specifications are loaded into specification graph ‘part-5a.rdf’. The specification is read by the script, described in section 6.2.1.2 and the result is shown in fig. 7.22. The design of the part induces a precedence among the machining of the features. The CAD design of the part does not contain this particular machining constraint the specification RDF loads this precedence from the inference of IMPlanner application. The side view the part and the corresponding precedence is shown in fig. 7.23.

The planning for the part is conducted by allowing the planner to consider every available processes for every kind of functions they realize, as given in table 6.4. However, the processes maintain a order in their application, following the precedence
Writing the part graph at `...\SimplePart-v2.rdf`
Part `SimplePart-v2` is loaded from `...\SimplePart-v2.xml`

<table>
<thead>
<tr>
<th>FeatureSpecification</th>
<th>FeatureName</th>
<th>FeatureType</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;:FeatureSpecification_Ifzrmxm&gt;</code></td>
<td>&quot;RECTANGULAR_POCKET(3)&quot;</td>
<td>&quot;Pocket&quot;</td>
</tr>
<tr>
<td><code>&lt;:FeatureSpecification_I660m8&gt;</code></td>
<td>&quot;SIMPLE HOLE(4)&quot;</td>
<td>&quot;Hole&quot;</td>
</tr>
<tr>
<td><code>&lt;:FeatureSpecification_I58jbfc&gt;</code></td>
<td>&quot;RECTANGULAR_SLOT(7)&quot;</td>
<td>&quot;Slot&quot;</td>
</tr>
<tr>
<td><code>&lt;:FeatureSpecification_Iak64ej&gt;</code></td>
<td>&quot;SLAB_SURFACE(2)&quot;</td>
<td>&quot;Slab&quot;</td>
</tr>
</tbody>
</table>

On Figure 7.22: Part specification loaded into ‘part-5a.rdf’.
Figure 7.23: CAD design and inferred order of machining among features in ‘SimplePart-v2.rdf’

graph, shown in fig. 6.23. We want to test whether the planner can find every possible way
to produce the part completely.

The plan is saved in the RDF graph ‘part-5a-plan.rdf’. The sub-occurrence trees for
every feature are generated by the in-planner plotter and shown in fig. 7.24 and fig. 7.25.

We analyzed the ‘part-5a-plan.rdf’ with the analyzer program, mentioned in
crefsec-eval-setup. The program found total 36 valid routing, each of which is capable of
making the part completely by fully satisfying the specifications of each features. There is
8 to 9 occurrences for each routing. The complete occurrence tree with only valid routes
are shown in fig. 7.26, which is plotted and radially laid out in yEd graph editor.

In order to verify the accuracy of the routes planned, three routes are sampled from
the set of routes and printed below. In fig. 7.26, the occurrences along these three routes
are highlighted with red, yellow, and blue respectively. The routing It can be observed that
the routes respect the precedence among features, given in fig. 7.23. Furthermore, every
route contains at least one occurrence for each feature of the part. Therefore, the part can
be completely processed by any of these routes planned.

- Red route

Number of occurrences in the routing ending with SideMilling_Irpkrxzyxp42d : 9
SlabMilling_1xc7a4pQSLAB_SURFACE(2) --> EndMilling_Ihrpb8g9yzc95k0RECTANGULAR_POCKET(3) -->
EndMilling_I6gpz7heyzg9RECTANGULAR_POCKET(3) --> SportDrilling_Inbxp84ner990SIMPLE_HOLE(4) -->
Figure 7.24: Sub-occurrence tree of (a) RECTANGULAR_POCKET(3), (b) SLAB_SURFACE(2), and (c) RECTANGULAR SLOT(7)

Yellow route

Number of occurrences in the routing ending with SideMilling_Injpryexzp42d: 8
EndMilling_Lycz05k@RECTANGULAR_POCKET(3) -> EndMilling_Leyzgxd@RECTANGULAR_POCKET(3) ->
FaceMilling_Lzbj8he@SLAB_SURFACE(2) -> SpotDrilling_Inpe3b5ner89@SIMPLE HOLE(4) ->
TwistDrilling_I49ar5hqyq5fp@SIMPLE HOLE(4) -> PrecisionBoring_I42b9jhkpbkh9@SIMPLE HOLE(4) ->
Reaming_Lieejx8de8re9@SIMPLE HOLE(4) -> SideMilling_Lycq2nqfbeb9d90RECTANGULAR SLOT(7) ->
SideMilling_Lrpekrjxzp42d0RECTANGULAR SLOT(7)

Blue route

Number of occurrences in the routing ending with SideMilling_I73rrfmxzp42d: 9
FaceMilling_Lzbj8he@SLAB_SURFACE(2) -> EndMilling_Iggp7pyucz9@RECTANGULAR_POCKET(3) ->
EndMilling_LzmyfbgreysxA@RECTANGULAR_POCKET(3) -> SpotDrilling_Izmnbkmner99@SIMPLE HOLE(4) ->
EndDrilling_I49ar5hqyq5fpq@SIMPLE HOLE(4) -> PrecisionBoring_Lz5h8anb3f@SIMPLE HOLE(4) ->
Reaming_Lz8hzzx2f575nym0@SIMPLE HOLE(4) -> SideMilling_T4kb9qybfeb9d90RECTANGULAR SLOT(7) ->
SideMilling_I73rrfmxzp42d0RECTANGULAR SLOT(7)
7.11  Experiment 6a

Complete process plan generation for ‘Netex’ part

In this experiment, process planning is conducted a sample part design, named ‘NetExample’ (netex). The specifications are loaded into specification graph ‘NetExample-features-prec.rdf’. The design XML of the part induces a precedence among the machining of the features. The side view the part and the precedence among features are shown in fig. 7.27. The complete list of specifications for this part is given in fig. 7.28.
Figure 7.26: Complete occurrence tree, planned for part SimplePart-v2.

Figure 7.27: CAD design and inferred order of machining among features in ‘Netex’ part

The planning for the part is conducted by allowing the planner to consider only one type of process for each type of features. No subsequent processes are considered as this particular part design has no tolerance specification. The plan, generated by SIMPlanner, is saved in the RDF graph ‘netex-plan.rdf’. The analyzer program is used to extract every
Figure 7.28: Specifications for Netex part loaded into ‘NetExample-features-prec.rdf’.

possible routing and a graphical representation of the occurrence tree, which is shown in fig. 7.29. There are a total of 420 unique routes by which Netex part can be processed completely.

In order to verify the accuracy of the routes planned, three routes are sampled from the set of routes and printed below. In fig. 7.29, the occurrences along these three routes are highlighted with red, green and yellow, respectively. The routing It can be observed that the routes respect the precedence among features, given in fig. 7.33. Furthermore, every route contains at least one occurrence for each feature of the part. Therefore, the part can be completely processed by any of these routes planned.

- Red route

Number of occurrences in the routing ending with FaceMilling_Irq72pzbz8r6a : 7

Red route

- Green route

Number of occurrences in the routing ending with SideMilling_Ijh8124a717 : 7

Green route

- Yellow route
7.12 Experiment 6b

Process plan generation for ‘Slider’ part

In this experiment, process planning is conducted a sample part design, named ‘Slider’. The specifications are loaded into specification graph ‘slider-features.rdf’. The design XML of the part induces a precedence among the machining of the features. The side view the part and the precedence among features are shown in fig. 7.30. The part design contains 24 features, including 15 holes, 7 open pockets, and 2 open slots.
The initial attempt of SIMPlanner to generate complete occurrence tree for the part did not finish in reasonable time due huge number of alternative sequences, in which features can be machined. The complete occurrence tree also includes the alternative routes for each feature, further increasing the size of the tree. In order to minimize the size of the problem, planning is conducted for some completely ordered feature precedence,
following the partially ordered feature precedence before the planning. We conducted three tests with three such completely ordered feature precedence. For each test, the planning agents are allowed to select alternative processes. However, the ‘hole planning agent’ is restricted to avail only twist-drilling for hole making function and boring, reaming for hole improving function.

7.12.1 Test 1

In the first test, the planning agents are constrained to follow the precedence (complete order) among features, which is given in fig. 7.31.

SIMPlanner generated total 256 different routes for processing Slider completely. The complete occurrence tree is shown in fig. 7.32. In order to check the accuracy of the plan, three sample routes are printed below. The occurrences along these routes are marked by three different colors, green, red, yellow, signifying three different routes. It can be observed that the routes respect the precedence among features, given in fig. 7.31. Furthermore, every route contains at least one occurrence for each feature of the part. Therefore, the part can be completely processed by any of these routes planned.
• **Red route**

Number of occurrences in the routing ending with Reaming_I4cn48mgkrey: 34

Red route

<table>
<thead>
<tr>
<th>Route Path</th>
<th>Number of Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwistDrilling_I8k5b5g7e0SIMPLE HOLE(19)</td>
<td></td>
</tr>
<tr>
<td>SlabMilling_I4p3m0RECTANGULAR_POCKET(3)</td>
<td></td>
</tr>
<tr>
<td>Boring_InXeS3m5200SIMPLE HOLE(18)</td>
<td></td>
</tr>
<tr>
<td>Reaming_I1W3p0SIMPLE HOLE(31)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I1I0p757kig720SIMPLE HOLE(24)</td>
<td></td>
</tr>
<tr>
<td>EndMilling_I1q4ga20RPC3ANGLE_POCKET(6)</td>
<td></td>
</tr>
<tr>
<td>Boring_I8pab39b0SIMPLE HOLE(11)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_Ipmq83g42830RECTANGULAR_POCKET(4)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I3k25jbk5f670SIMPLE HOLE(18)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I5rkqj920SIMPLE HOLE(17)</td>
<td></td>
</tr>
<tr>
<td>Reaming_I3k25jbk5f670SIMPLE HOLE(17)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I5rpqj9p80SIMPLE HOLE(16)</td>
<td></td>
</tr>
<tr>
<td>SlabMilling_Ipmq83g42830RECTANGULAR_POCKET(4)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I5jhc1b0SIMPLE HOLE(11)</td>
<td></td>
</tr>
<tr>
<td>Boring_I8pab39b0SIMPLE HOLE(11)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_Ipmq83g42830RECTANGULAR_POCKET(4)</td>
<td></td>
</tr>
</tbody>
</table>

• **Yellow route**

Number of occurrences in the routing ending with Reaming_I8x509kgkrey: 34

Yellow route

<table>
<thead>
<tr>
<th>Route Path</th>
<th>Number of Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwistDrilling_I8k5b5g7e0SIMPLE HOLE(19)</td>
<td></td>
</tr>
<tr>
<td>SlabMilling_I4p3m0RECTANGULAR_POCKET(3)</td>
<td></td>
</tr>
<tr>
<td>Boring_InXeS3m5200SIMPLE HOLE(18)</td>
<td></td>
</tr>
<tr>
<td>Reaming_I1W3p0SIMPLE HOLE(31)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I1I0p757kig720SIMPLE HOLE(24)</td>
<td></td>
</tr>
<tr>
<td>EndMilling_I1q4ga20RPC3ANGLE_POCKET(6)</td>
<td></td>
</tr>
<tr>
<td>Boring_I8pab39b0SIMPLE HOLE(11)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_Ipmq83g42830RECTANGULAR_POCKET(4)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I3k25jbk5f670SIMPLE HOLE(18)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I5rkqj920SIMPLE HOLE(17)</td>
<td></td>
</tr>
<tr>
<td>Reaming_I3k25jbk5f670SIMPLE HOLE(17)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I5rpqj9p80SIMPLE HOLE(16)</td>
<td></td>
</tr>
<tr>
<td>SlabMilling_Ipmq83g42830RECTANGULAR_POCKET(4)</td>
<td></td>
</tr>
</tbody>
</table>

• **Green route**

Number of occurrences in the routing ending with Reaming_I8q6ydcgkrey: 34

Green route

<table>
<thead>
<tr>
<th>Route Path</th>
<th>Number of Occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>TwistDrilling_I8k5b5g7e0SIMPLE HOLE(19)</td>
<td></td>
</tr>
<tr>
<td>SlabMilling_I4p3m0RECTANGULAR_POCKET(3)</td>
<td></td>
</tr>
<tr>
<td>Boring_InXeS3m5200SIMPLE HOLE(18)</td>
<td></td>
</tr>
<tr>
<td>Reaming_I1W3p0SIMPLE HOLE(31)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I1I0p757kig720SIMPLE HOLE(24)</td>
<td></td>
</tr>
<tr>
<td>EndMilling_I1q4ga20RPC3ANGLE_POCKET(6)</td>
<td></td>
</tr>
<tr>
<td>Boring_I8pab39b0SIMPLE HOLE(11)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_Ipmq83g42830RECTANGULAR_POCKET(4)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I3k25jbk5f670SIMPLE HOLE(18)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I5rkqj920SIMPLE HOLE(17)</td>
<td></td>
</tr>
<tr>
<td>Reaming_I3k25jbk5f670SIMPLE HOLE(17)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I5rpqj9p80SIMPLE HOLE(16)</td>
<td></td>
</tr>
<tr>
<td>SlabMilling_Ipmq83g42830RECTANGULAR_POCKET(4)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I5jhc1b0SIMPLE HOLE(11)</td>
<td></td>
</tr>
<tr>
<td>Boring_I8pab39b0SIMPLE HOLE(11)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_Ipmq83g42830RECTANGULAR_POCKET(4)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I3k25jbk5f670SIMPLE HOLE(18)</td>
<td></td>
</tr>
<tr>
<td>TwistDrilling_I5rkqj920SIMPLE HOLE(17)</td>
<td></td>
</tr>
<tr>
<td>Reaming_I3k25jbk5f670SIMPLE HOLE(17)</td>
<td></td>
</tr>
</tbody>
</table>
7.12.2 Test 2

In the first test, the planning agents are constrained to follow the precedence (complete order) among features, which is given in fig. 7.33.

SIMPlanner generated total 256 different routes for processing Slider completely. The complete occurrence tree is shown in fig. 7.34. In order to check the accuracy of the plan, two sample routes are printed below. The occurrences along these routes are marked by two different colors, red and green, signifying three different routes. It can be observed that the routes respect the precedence among features, given in fig. 7.33. Furthermore, every route contains at least one occurrence for each feature of the part. Therefore, the part can be completely processed by any of these routes planned.

- Red route
Figure 7.34: Occurrence tree for Slider part, following feature precedence, given in fig. 7.33
In this experiment, process planning is conducted a sample part design, named ‘NIST-CTC-01’, which is shown in fig. 7.35a. The specifications are loaded into the specification graph ‘nist-ctc-01.rdf’. The design XML of the part induces a precedence among the machining of the features, which is shown in fig. 7.35b. The side view the part and the precedence among features are shown in fig. 7.30. The part design contains 26 features, including 10 holes, 15 open pockets, and 1 open slots.
Figure 7.35: Part ‘NIST-CTC-01’ with annotation marking its features (a), and precedence among them (b).
Due to the large number of features, planning for this part is conducted for some totally ordered feature precedence, following the partially ordered feature precedence before the planning. The total order FPN is shown in fig. 7.36. In this experiment, the planner is allowed to choose alternative processes for pockets but not for other types of features. Also, it is assumed that only one process can completely satisfy the specifications for each feature. Our goal is to show that the number of alternative routes can be calculated from the number of alternatives for each feature, when each such route processes the features in the same order.

![Complete order among features of part ‘NIST-CTC-01’](image)

Figure 7.36: Complete order among features of part ‘NIST-CTC-01’.

Total number of alternative routes generated by the planner for ‘NIST-CTC-01’ is 32768. As every feature can be made by only one process and every route follows the same order (total) among features, it can be trivially observed that every planned occurrence at each level of the tree has same feature as its output. Therefore, the number
of alternatives for a feature denotes the branching factor for that level of the tree. The number of leaves of the occurrence tree can then be calculated by multiplying the branching factors for each features. In table 7.1, the features are listed in the same sequence as in the precedence in fig. 7.36. The corresponding number of alternative processes are then multiplied, denoting number of occurrences at each level of the tree. The last level contains 32768 leaves, which is same as the number of alternative routes.

Table 7.1: Number of alternatives for each features and number of occurrences at each level.

<table>
<thead>
<tr>
<th>Level</th>
<th>Feature</th>
<th># of alternatives</th>
<th># of occurrences</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RECTANGULAR SLOT(15)</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>RECTANGULAR POCKET(7)@</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>RECTANGULAR POCKET(8)</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>RECTANGULAR POCKET(5)</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>RECTANGULAR POCKET(6)</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>RECTANGULAR POCKET(21)</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>7</td>
<td>RECTANGULAR POCKET(4)</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>8</td>
<td>RECTANGULAR POCKET(3)</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>9</td>
<td>RECTANGULAR POCKET(22)</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>10</td>
<td>RECTANGULAR POCKET(34)</td>
<td>1</td>
<td>32</td>
</tr>
<tr>
<td>11</td>
<td>SIMPLE HOLE(25)</td>
<td>2</td>
<td>64</td>
</tr>
<tr>
<td>12</td>
<td>RECTANGULAR POCKET(35)</td>
<td>1</td>
<td>64</td>
</tr>
<tr>
<td>13</td>
<td>SIMPLE HOLE(26)</td>
<td>2</td>
<td>128</td>
</tr>
<tr>
<td>14</td>
<td>RECTANGULAR POCKET(10)</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>15</td>
<td>RECTANGULAR POCKET(9)</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>16</td>
<td>RECTANGULAR POCKET(23)</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>17</td>
<td>RECTANGULAR POCKET(24)</td>
<td>1</td>
<td>128</td>
</tr>
<tr>
<td>18</td>
<td>SIMPLE HOLE(12)</td>
<td>2</td>
<td>256</td>
</tr>
<tr>
<td>19</td>
<td>SIMPLE HOLE(14)</td>
<td>2</td>
<td>512</td>
</tr>
<tr>
<td>20</td>
<td>SIMPLE HOLE(11)</td>
<td>2</td>
<td>1024</td>
</tr>
<tr>
<td>21</td>
<td>SIMPLE HOLE(13)</td>
<td>2</td>
<td>2048</td>
</tr>
<tr>
<td>22</td>
<td>RECTANGULAR POCKET(16)</td>
<td>1</td>
<td>2048</td>
</tr>
<tr>
<td>23</td>
<td>SIMPLE HOLE(20)</td>
<td>2</td>
<td>4096</td>
</tr>
<tr>
<td>24</td>
<td>SIMPLE HOLE(17)</td>
<td>2</td>
<td>8192</td>
</tr>
<tr>
<td>25</td>
<td>SIMPLE HOLE(18)</td>
<td>2</td>
<td>16384</td>
</tr>
<tr>
<td>26</td>
<td>SIMPLE HOLE(19)</td>
<td>2</td>
<td>32768</td>
</tr>
</tbody>
</table>
8 Research Summary and Future Work

The cloud manufacturing (CM) planning system, proposed in this document, pivots on two main topics: design of a multi-agent society based on semantic aware BDI agents, capable of integrating CM services autonomously, and development of a unified semantic data model for representing distributed manufacturing resources and planning knowledge. The context of such development is elaborated in chapter 1, where we discussed the benefits of comprehensive servitization of manufacturing resources in a cloud environment to the distributed and collaborative manufacturing of the future. A brief overview of the recently proposed architectures for facilitating collaboration among manufacturing service providers and consumers is included in section 3.1. We identified some of the open issues, yet to be addressed for a practical realization of CM. These are how organizationally disconnected manufacturing partners can establish seamless data exchange among CM services, which often use heterogeneous data models and how these services can be integrated in an automated fashion. The overarching goal of this study is to address these issues in the light of manufacturing process planning, as process planning in a CM environment needs to consider diverse kind of manufacturing resources as well as production methods, therefore needs to be flexible enough to handle certain level of uncertainty during planning regarding the availability of these resources. The framework for the multi-agent society, presented in chapter 4, is built as an apparatus to facilitate autonomous service discovery, orchestration, and invocation, making further development of a CM process planner possible. In order to ensure that these agents communicate in a uniform language, a canonical information model is proposed in chapter 5, based on ontological analysis of various aspects of manufacturing process planning, e.g. product design, resource capability, and planning knowledge.

The bedrock of the proposed architecture of a multi-agent society is based on belief-desire-intention formalism, which is elaborated in section 4.1. In section 4.2, we
described the internal components of a generic agent. Broadly, a set of conjuncts makes up the belief; a set of plans, written in FOL rules (definition 2), makes up the desire of the agent. It is shown in section 4.2.3 that the belief of the agent moves from one state to another, as new facts are added by the repetitive application of matching FOL rules, eventually having one of those states to possibly satisfy the goal of the agent. We also discuss the possibility of developing some search strategies based on forward chase algorithm to optimize the selection of matching rules at every iteration. The most original contribution of the proposed design is the use of existential rules for writing the plans of an agent. Two novel purposes are achieved by the employment of existential rules as plan, first, semantic services are integrated to the rule based on formal theories of logic, given in section 4.3, and second, semantic services are automatically discovered by matching the input and output types of the service with the types of variables in the rule, as explained in section 4.3. From the utilitarian point of view, the proposed framework adopted service-oriented communication instead of the traditional message-based communication among agents. As services, provided by some agents, are automatically discovered, orchestrated, and invoked by other agents, fully automated communication among agents is achieved. In this way, agents of the proposed multi-agent society are able to solve a problem autonomously and collaboratively. Furthermore, a practical solution for realizing the theoretical framework of agents is given in section 4.4, in which resource definition framework (RDF) is used to represent both global knowledge base and local belief of the agent and SPARQL CONSTRUCT query language is used as the vocabulary to write the plans of the agent. In order to accommodate the logic of existential rules as well as automatic service integration, an extension to the standard SPARQL query is proposed and implemented. This extended version of SPARQL, as described in section 4.4, along with development related to service integration, is named as ‘Function Query Language’ (FunQL). This particular development may be prospective in many other applications.
beyond the proposed multi-agent system as it provides the ability to use any type of function and web service to map ‘unknown variables’ in CONSTRUCT clause of a SPARQL query.

Chapter 5 presents an upper-level ontology for describing concepts related to manufacturing process planning with strong philosophical and logical underpinning. As its primary purpose, the proposed ontology serves as a common vocabulary for CM agents to describe their plans and services they offer; for example, CM resources can advertise their capability with common information structure, product design specifications can be expressed in a unified data model, and process planning knowledge can be captured in a uniform manner. Additionally, this ontology helps in storing manufacturing design, process, resource, and planning related information in semantically linked graph databases by providing the suite of formal definitions and taxonomy of manufacturing terminologies. As opposed to many other manufacturing ontology, previously published, every concept in the proposed ontology is defined based on domain-independent foundation ontology. Concepts related to the physical entities and their characteristics are derived from ‘basic formal ontology’ (BFO). On the other hand, manufacturing process related concepts are derived by augmenting the definitions of BFO with theories from ‘process specification language’ (PSL). The upper level ontology is divided into three parts: section 5.3 describes product design specification, section 5.4 describes manufacturing resource capability, and section 4.2.2 describes manufacturing process planning. The unique attribute of these models is that the terminologies are defined based on ontological analysis with a realist viewpoint instead of following existing object-oriented data models, such as UML. Every definition is formally expressed as statement of first order logic to maintain rigor.

Furthermore, several taxonomies are built with domain level terms from CNC machining area. Taxonomies of prismatic part feature, their dimensions, and tolerance,
manufacturing process, function, resource and process capability form a lower-level ontology, which can be built based on the upper level concepts. This domain level ontology caters to a small portion of metal fabrication manufacturing but helps in the evaluation of the prototype planning system, which tests scenarios from CNC machining area. However, the domain level ontology also serves as a proof of concept, which can be replicated for other domains of manufacturing.

The key feature of product design ontology is the tripartite differentiation among the representations of various elements of product design, their specifications, and the information attributed to those specifications. Another distinct idea is modeling the design features as type of fiat objects. In the area of resources, a significant amount of effort is spent in section 5.4.1 for deriving a formal definition of manufacturing resource capability and establishing its association with manufacturing process level capability. Similar to the design ontology, different information models are designed in section 5.4.3 to capture various types of manufacturing capability.

The most critical part of the ontology development is planning ontology, which aims to capture the three dimensional information structure of manufacturing process plan, comprised of aggregation, variety, and time dimension. This particular structure has long been proposed by some earlier studies, but represented formally by a set of axioms by this study, in section 5.5.5. The axioms are heavily motivated by the concepts of process compositions (section 5.5.5.1) and temporal relationship among occurrences (section 5.5.5.3), which are borrowed from PSL with some modifications to adapt to the situation of planning. The output features of the planned processes are also classified based on their conformance to the given specification. Finally, manufacturing process plan is represented as an occurrence tree, which collects every possible route for manufacturing a part completely is represented in a tree structure.
The prototype process planning system SIMPlanner, described in chapter 6, is built on the proposed multi-agent framework and uses types and predicates from the proposed ontology to write the rules of the agents. The prototype is equipped with scripts that can be invoked by commands to load part design specifications and process capabilities in RDF graphs. Three types of agents are discussed in section 6.3, which collaboratively generates plan for a given design specification. The plans of these agents are written in FunQL, which lets ‘part planning agent’ to delegate the sub-task of planning processes a feature to a suitable ‘feature planning agent’, based on the feature type, and the ‘feature planning agent’ to delegate the sub-task of selecting a particular type of process for the feature to a suitable ‘process selector agent’, based on the process type. The rule, written in ‘part planning agent’ (fig. 6.20), to plan completely ordered routes from the partially ordered features is an interesting one, as it uses a complex pattern based on SPARQL property path to do so. However, this plan is also expensive, as transpired, that it takes exponentially increasing amount to time to query the occurrence tree, growing in size during planning.

The prototype is evaluated by a number of sample parts and by constraining it to select from a specific set of alternative processes. In chapter 7 the results of these experiments are presented by various types of plots and visuals, in order to show the quality of the generated plan, the alternative routes found, as well as both intermediate and final features, produced by each occurrence. The accuracy of these plans is mostly guaranteed by evaluating the criteria, such as whether the plan contains some occurrence for every feature, whether the alternative routes obey the constrains of feature precedence and process precedence, and whether every feature is completely processed or if not, for what reason. Some experiments are conducted with industrial parts, usually containing numerous features, e.g. Netex, Slider, and NIST-01, in order to show the applicability of SIMPlanner in practical cases. However, a complete occurrence tree could not be generated for Slider and NIST-01 in a single run, due to the time-consuming rule in ‘part
planning agent’. Instead, the experiments are conducted with completely ordered feature precedence, derived from the given partial order among the features, in order to minimize the load on the rule in fig. 6.20.

A number of assumptions are made in the development and evaluation of the prototype process planning system. First of all, the prototype is not truly cloud-based, as the agents are hosted locally and the services act like common methods, although the input and output mapped to a semantic types. In order to host the agents in cloud with fully enabled service-oriented communication, the agents need to be implemented in a reactive, dynamic, fault-tolerant platform. This platform is required to address several topological issues, such as encapsulation, registration, hosting, and clustering, which are related to the practical implementation of a multi-agent society. The structural framework, proposed in this thesis, does not go into such implementation related topics. Among many commercial alternatives, including container services, e.g. Docker™, cloud computing services, e.g. AWS lambda, Microsoft Azure, and open sourced Node.js javascript environment, the most promising platform is Akka framework, which lets users to build highly concurrent, distributed, lightweight applications with strong isolationist viewpoint. Additionally, Akka-compatible Lagom framework may be used to build microservice oriented communication environment. Furthermore, an user-friendly toolkit is required to add semantic annotations to the services. Although having some proposals in the literature, there is no reliable toolkit available for annotating services with semantics. Such a toolkit should be developed with the capability of identifying and suggesting semantics from a set of given ontology.

Secondly, the service discovery strategy, proposed in section 4.3, needs further improvements. The current implementation naively searches through every available service for a matching input and output semantics. Moreover, only the exact matches is considered, without considering super types and inferred types. In case of a large number
of available services, the search time needs to be minimized by optimization and memorization.

Third, every applicable plan is performed at every state of the belief in the current version of implementation, which may result in a large derivation tree. Essentially, the current implementation of the intention structure of the agents applies no priority to the plans. As explained in section 4.2.3, a search strategy needs to be implemented in order to guide the belief to the shortest possible derivation path. Forward-chaining chase algorithm is considered as a good candidate for this purpose.

The current version of SIMPlanner also needs a number of improvements to be applicable in industry-scale process and resource planning. As observed in the experiments with a large number of features, the planner consumes an extraordinary amount of time. In a CM environment, this problem is exacerbated by the sheer volume of alternative resources. A number of space-search algorithms, such as A*, may be applied with a suitable heuristic to avoid expanding some of the alternative routes further. These type of bounding algorithms need to be embedded in the rules of part planning and feature planning agents.

The current development considers only one combination of machine and tool for every function individual, given in table 6.4. Therefore, actual machines and tools are not included in the plans, given in chapter 7. For the same reason, only the process level capability is used in the selection of a suitable process for a given feature specification. In a CM environment, multiple combinations of machines and tools may incur different sets of capability profile for a single type of process. Moreover, selection of process is also contingent on the resource level capability; for example, surface finish tolerance of a slab type feature may satisfy the corresponding capability of face milling, but the process still may not be selected if the workpiece cannot fit the available milling machine, which will be used for the process.
In order to enable SIMPlanner for handling process planning for other domains of manufacturing, such as assembly, metal stamping, forging, injection molding, and 3D printing, the domain level ontology needs to be furnished with taxonomies of concepts from these domains. For this purpose, a detailed investigation needs to be conducted for each of these domains to gather the list of processes, functions, and capabilities of machines and tools, used in these domains. Furthermore, various domains level planning knowledge, precedence among processes, material conditions, and general best practices need to be encoded in the rules of the agents. Some of the planning related concepts in other domain may even require certain adjustments in the definitions of upper level concepts. Such an endeavor definitely coincides with the current work of Industrial Ontology Foundry in developing a global ontology for manufacturing. In the future, the proposed model of upper level manufacturing ontology is expected to converge.
REFERENCES


Appendix A: Semantic Service Schema

```
{
  "serviceProfile": {
    "serviceName": "Name of the Service",
    "textDescription": "Long description of the service",
    "actor": {
      "input": [
        {
          "parameter": 1,
          "type": "classURI"
        },
        {
          "parameter": 2,
          "type": "classURI"
        }
      ],
      "output": {
        "type": "classURI"
      },
      "result": [
        "propertyURI",
        "propertyURI"
      ]
    }
  }  

  "serviceGrounding": {
    "inputGrounding": {
      "parameter": 1,
      "grounding": [
        {
          "arg": 1,
          "dataProperty": "propertyURI",
          "dataType": "xsd:string"
        },
        {
          "arg": 2,
          "dataProperty": "propertyURI",
          "dataType": "xsd:string"
        }
      ],
      "parameter": 2,
      "grounding": [
        {
          "arg": 3,
          "dataProperty": "propertyURI",
          "dataType": "xsd:string"
        },
        {
          "arg": 4,
          "dataProperty": "propertyURI",
          "dataType": "xsd:string"
        }
      ]
    },
    "outputGrounding": {
      "grounding": [
        {
          "arg": 1,
          "dataProperty": "anURI",
          "dataType": "xsd:string"
        },
        {
          "arg": 2,
          "dataProperty": "anURI",
          "dataType": "xsd:string"
        }
      ]
    }
  }
}
```

Figure A.1: Schema for publishing semantic service.
APPENDIX B: MANUFACTURING DOMAIN LEVEL TAXONOMY

Figure B.1: Taxonomy of specifications of features and their dimensions, and tolerances
Figure B.2: Taxonomy of CNC processes, machines, tools
Figure B.3: Taxonomy of functions, borne by CNC machining resources
Figure B.4: Taxonomy of capabilities, borne by CNC machining resources
APPENDIX C: EXAMPLE OF DESIGN XML FILE

The following XML file is generated by IMPlanner application.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<MfgPartModel partName="SimplePart-v2" partMaterial="CarbonSteel" batchSize="50">
  <Stock>edu.ohiou.mfgresearch.implanner.geometry.Stock needs to implement writeXML() method</Stock>
  <featureList>
    <edu.ohiou.mfgresearch.implanner.features.Slot featureName="RECTANGULAR_SLOT(7)">
      <Dimensions slotPoint = '2 1 2'
                   normal = '0 -1 0'
                   sweep = '1 0 0' width = '1.0'
                   bottomDistance = '-0.0'
                   positiveSweepLength = '3.0'
                   negativeSweepLength = '3.0'/>
      <previousFeature name = "SIMPLE HOLE(4)" />
      <previousFeature name = "RECTANGULAR_POCKET(3)" />
      <Tolerances>
        <Parameter dimensionTolerance="0.002"/>
        <Parameter surfaceFinish="55"/>
      </Tolerances>
      <processList />
    </edu.ohiou.mfgresearch.implanner.features.Slot>
    <edu.ohiou.mfgresearch.implanner.features.Hole featureName="SIMPLE HOLE(4)">
      <Dimensions radius = '1.0'
                   depth = '2.0'
                   holeaxis = '0 0 1'
                   axisPoint = '1.91351 2.92519 4'
                   bottomDistance = '-2.0'/>
      <nextFeature name = "RECTANGULAR_SLOT(7)" />
      <previousFeature name = "RECTANGULAR_POCKET(2)" />
      <Tolerances>
        <Parameter roundness="0.001"/>
      </Tolerances>
      <processList />
    </edu.ohiou.mfgresearch.implanner.features.Hole>
    <edu.ohiou.mfgresearch.implanner.features.Pocket featureName="RECTANGULAR_POCKET(3)">
      <Dimensions pocketPoint = '2 0 2'
                   bottomDistance = '-1.0'
                   normal = '0 -1 0'/>
      <nextFeature name = "RECTANGULAR_SLOT(7)" />
      <Tolerances>
        <Parameter dimensionTolerance="0.002"/>
        <Parameter surfaceFinish="55"/>
      </Tolerances>
      <processList />
    </edu.ohiou.mfgresearch.implanner.features.Pocket>
    <edu.ohiou.mfgresearch.implanner.features.Slab featureName="RECTANGULAR_POCKET(2)">
      <Dimensions pocketPoint = '2 3 5'
                   bottomDistance = '-1.0'
                   normal = '0 0 1'/>
      <nextFeature name = "SIMPLE HOLE(4)" />
      <Tolerances>
```
</Tolerances>

<processList />

</edu.ohiou.mfgresearch.implanner.features.Slab>

</featurelist>

</MfgPartModel>