Towards Autonomous Control of Drilling Rigs

Master thesis
60 credits

Bjørn Tveter
bjorntve@ifi.uio.no

22.05.2009
Abstract

Drilling for petroleum resources in remote and harsh environments requires new technology and operational methods. Recent innovation demonstrates feasibility of having future drilling rigs placed directly on the seabed. In this vision, the drilling rigs are controlled remotely from either an onshore control centre or an offshore supply vessel. Autonomous decision making and advanced control are likely to play a significant role in the realisation of this vision. Powerful methods and constructs brought by the multi-agent paradigm can ease the design and development of such systems. In this thesis we give an introduction to this type of technology, the drilling domain and outline one approach to autonomous control system for drilling rigs. Feasible aspects of this first attempt to address autonomous control of drilling rigs are demonstrated through an experiment conducted in a laboratory setting.
Acknowledgements

This thesis is submitted to the Department of Informatics at the University of Oslo as a part of my Master degree. I would like to thank Stian Aase from Computas AS for his contributions during the planning and realisation of the work conducted in this project. I would also like to thank my supervisor, Roar Fjellheim (Computas and University of Oslo) for proposing the subject of the thesis and his valuable advice and feedback, Jørn Ølmheim from StatoilHydro for contributing with knowledge on multi-agent technology, and the AutoConRig project group for their help and hints.

Oslo, 22.05.09                 Bjørn Tveter (bjorntve@ifi.uio.no)
# Contents

<table>
<thead>
<tr>
<th>1</th>
<th>Introduction ..................................................................................................................</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Motivation ..................................................................................................................</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Project Context ..........................................................................................................</td>
<td>1</td>
</tr>
<tr>
<td>1.2.1</td>
<td>AutoConRig .................................................................................................................</td>
<td>1</td>
</tr>
<tr>
<td>1.3</td>
<td>Research Goals ............................................................................................................</td>
<td>2</td>
</tr>
<tr>
<td>1.4</td>
<td>Research Method .........................................................................................................</td>
<td>2</td>
</tr>
<tr>
<td>1.5</td>
<td>Document Structure ....................................................................................................</td>
<td>3</td>
</tr>
<tr>
<td>1.</td>
<td>Problem Analysis ..........................................................................................................</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Project Description .....................................................................................................</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>Application Area .........................................................................................................</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Scope ............................................................................................................................</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Tripping Sequences ....................................................................................................</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Motivation for an Agent-based Control System ..........................................................</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>State Of The Art ...........................................................................................................</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Background ..................................................................................................................</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Agents Everywhere ......................................................................................................</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>Agent Definitions ........................................................................................................</td>
<td>9</td>
</tr>
<tr>
<td>3.4</td>
<td>Agent Classification ..................................................................................................</td>
<td>11</td>
</tr>
<tr>
<td>3.5</td>
<td>Agent Theories ..........................................................................................................</td>
<td>14</td>
</tr>
<tr>
<td>3.6</td>
<td>Agent Architectures ....................................................................................................</td>
<td>15</td>
</tr>
<tr>
<td>3.6.1</td>
<td>Reactive Agent Architectures ......................................................................................</td>
<td>15</td>
</tr>
<tr>
<td>3.6.2</td>
<td>Deductive Reasoning Architectures .............................................................................</td>
<td>16</td>
</tr>
<tr>
<td>3.6.3</td>
<td>Practical Reasoning Architectures ...............................................................................</td>
<td>17</td>
</tr>
<tr>
<td>3.6.4</td>
<td>Hybrid Agent Architectures .......................................................................................</td>
<td>18</td>
</tr>
<tr>
<td>3.6.5</td>
<td>Layered Architectures ...............................................................................................</td>
<td>18</td>
</tr>
<tr>
<td>3.7</td>
<td>Multi Agent Systems ..................................................................................................</td>
<td>19</td>
</tr>
<tr>
<td>3.7.1</td>
<td>Agent Interactions .....................................................................................................</td>
<td>20</td>
</tr>
<tr>
<td>3.7.2</td>
<td>Agent Organisations .................................................................................................</td>
<td>23</td>
</tr>
<tr>
<td>3.8</td>
<td>Methodologies .............................................................................................................</td>
<td>24</td>
</tr>
<tr>
<td>4</td>
<td>Related work ................................................................................................................</td>
<td>25</td>
</tr>
</tbody>
</table>
13.3 Validity Threats ................................................................. 117
13.4 Experiment Summarised .................................................. 117
14 Conclusion and Future Work .............................................. 119
  14.1 Conclusion ...................................................................... 119
  14.2 Achievements ................................................................ 119
  14.3 Possible Improvements of the Prototype .................................. 119
  14.4 Subjects for Further Research ........................................... 120

Appendices ................................................................................. 123

A. Notation ............................................................................ 125
  A1. PDT Diagram Constructs ................................................ 125
  A2. JACK JDE Graphical Notation ......................................... 125

B. Detailed Interactions ............................................................ 127
  B1. Interactions: Planning .................................................... 127
  B2. Interactions: Hoisting .................................................... 128
  B3. Interactions: Acceleration ............................................. 129
  B4. Interactions: Deceleration ............................................. 130

C. Shared Ontology: State Definitions ....................................... 131
  C1. Bit position .................................................................... 131
  C2. Circulation .................................................................... 131
  C3. Hook position .................................................................. 132
  C4. Park break ..................................................................... 133
  C5. Slips .............................................................................. 133
  C6. Hoisting ....................................................................... 134
  C7. Rotation ....................................................................... 135

D. Implementation Details ......................................................... 137
  D1. Message Descriptors .................................................... 137
  D2. The Business Logic mapped to the Common Ontology ........ 139
  D3. The Planning Algorithm ............................................... 140

Bibliography .............................................................................. 141
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Unmanned Drilling Rigs [4]</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Research Method Used in the Thesis</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Part View of an Agent Typology [8]</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Reactive Agent</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>Layered Architectures A) Horizontal B) Vertical One-Pass C) Vertical Two-Pass</td>
<td>18</td>
</tr>
<tr>
<td>7</td>
<td>Canonical View of MAS [19]</td>
<td>19</td>
</tr>
<tr>
<td>8</td>
<td>Engmo &amp; Hallen Multi-Agent Architecture for Production Optimisation [2]</td>
<td>25</td>
</tr>
<tr>
<td>9</td>
<td>Spillum’s Refined Architecture [24]</td>
<td>26</td>
</tr>
<tr>
<td>10</td>
<td>Prometheus Methodology Overview</td>
<td>30</td>
</tr>
<tr>
<td>11</td>
<td>Screenshot of the JACK Development Environment</td>
<td>32</td>
</tr>
<tr>
<td>12</td>
<td>Screenshot of the Prometheus Design Tool</td>
<td>33</td>
</tr>
<tr>
<td>13</td>
<td>Mud Circulation Explained</td>
<td>35</td>
</tr>
<tr>
<td>14</td>
<td>Simplified Drilling Rig</td>
<td>37</td>
</tr>
<tr>
<td>15</td>
<td>Draw-work From NOV</td>
<td>37</td>
</tr>
<tr>
<td>16</td>
<td>Top-drive Connected to the Hook and Travelling Block</td>
<td>38</td>
</tr>
<tr>
<td>17</td>
<td>Driller’s Cabin with two Cyberbase Chairs</td>
<td>39</td>
</tr>
<tr>
<td>18</td>
<td>Decision Cycles during Drilling</td>
<td>40</td>
</tr>
<tr>
<td>19</td>
<td>System Environment</td>
<td>45</td>
</tr>
<tr>
<td>20</td>
<td>Goal Overview</td>
<td>48</td>
</tr>
<tr>
<td>21</td>
<td>Scenarios</td>
<td>49</td>
</tr>
<tr>
<td>22</td>
<td>Activity Diagram: High-level Business Logic</td>
<td>54</td>
</tr>
<tr>
<td>23</td>
<td>Distribution of Autonomy</td>
<td>55</td>
</tr>
<tr>
<td>24</td>
<td>Information Flow</td>
<td>56</td>
</tr>
<tr>
<td>25</td>
<td>System Roles</td>
<td>57</td>
</tr>
<tr>
<td>26</td>
<td>Mapping the Roles to the Drilling Domain</td>
<td>59</td>
</tr>
<tr>
<td>27</td>
<td>Agent-Role Grouping</td>
<td>60</td>
</tr>
<tr>
<td>28</td>
<td>Agent Acquaintance Diagram</td>
<td>62</td>
</tr>
<tr>
<td>29</td>
<td>System Overview Diagram</td>
<td>63</td>
</tr>
<tr>
<td>30</td>
<td>Combined Lifeline Decomposition and Diagram Referencing</td>
<td>64</td>
</tr>
<tr>
<td>31</td>
<td>Interaction Diagram: Communication Failure Scenario</td>
<td>65</td>
</tr>
<tr>
<td>32</td>
<td>Interaction Diagram: Pre-Communication Failure</td>
<td>66</td>
</tr>
<tr>
<td>33</td>
<td>Interaction Diagram: Above Casing Shoe Scenario</td>
<td>67</td>
</tr>
<tr>
<td>34</td>
<td>Interaction Diagram: Lock Slips</td>
<td>68</td>
</tr>
<tr>
<td>35</td>
<td>Interaction Diagram: Less Than 1 Stand from Casing Shoe Scenario</td>
<td>69</td>
</tr>
<tr>
<td>36</td>
<td>Interaction Diagram: More than 1 Stand in Open Hole Scenario</td>
<td>70</td>
</tr>
<tr>
<td>37</td>
<td>Interaction Diagram: Continually Elevate and Lower the Drillstring</td>
<td>71</td>
</tr>
<tr>
<td>38</td>
<td>Interaction Protocol: PlanningGoalCommand</td>
<td>72</td>
</tr>
<tr>
<td>39</td>
<td>Interaction Protocol: StateSnapshotRetrival</td>
<td>73</td>
</tr>
<tr>
<td>40</td>
<td>Interaction Protocol: OperationCommand</td>
<td>73</td>
</tr>
<tr>
<td>41</td>
<td>Interaction Protocol: MeasurementUpdate</td>
<td>74</td>
</tr>
<tr>
<td>42</td>
<td>Levels of the Common Ontology</td>
<td>75</td>
</tr>
<tr>
<td>43</td>
<td>Identified States for Bit Position</td>
<td>77</td>
</tr>
<tr>
<td>44</td>
<td>High-Level Business Logic Mapped to Definitions from the Common Ontology</td>
<td>79</td>
</tr>
</tbody>
</table>
List of Tables

Table 1: Linda Operations for Tuple Space Control........................................................................ 20
Table 2: Walton and Krabbe Dialogue Types [15]...................................................................... 23
Table 3 JACK Key Programming Constructs [30].......................................................................... 32
Table 4 Process Variables............................................................................................................ 47
Table 5 Specific to General Mapping Scheme............................................................................... 61
Table 6 Example Planning Problem.............................................................................................. 82
Table 7 One Solution to the Example Planning Problem ................................................................ 83
Table 8 Operations/services provided by the ControlInterface................................................ 86
Table 9 Generic Interface to Drilling Machinery .......................................................................... 87
Table 10 Configurations used in the experiment ......................................................................... 101
Table 11 Experiment: Output from Case 1 Compared with Specification. .................................. 103
Table 12 Experiment, Case 2: Actions in Response to Case 2.................................................... 107
Table 13 Experiment, case 3: Sequence of Actions...................................................................... 111
Table 14 Experiment: Auxiliary Test Cases................................................................................ 113
Table 15 Experiment: Case 4.1 – Configuration and Actions....................................................... 114
Table 16 Experiment: Case 4.2 – Configuration and Actions....................................................... 115
Table 17 Experiment: Case 4.3 – Configuration and Actions....................................................... 116
Table 18 Experiment: Case 4.4 – Configuration and Actions....................................................... 116
Table 19 PDT Graphical Notation.................................................................................................... 125
Table 20 JACK JDE Graphical Notation ...................................................................................... 125
Table 21 Message Descriptors....................................................................................................... 137
Table 22 Mapping the Business logic to the Common Ontology.................................................. 139
1 Introduction

Serving as a means to motivate the reader, this chapter describes the project context, problem area, research method and goal. Finally, it presents the structure of this document.

1.1 Motivation

Decreasing reserves of oil in the North Sea makes oil recovery more challenging than ever before. This new setting forces the industry to seek new methods and technology to adjust their operations in order to cope with the decreasing margins. Recent developments, such as wired pipe technology and fibre optics have allowed for better use of software in drilling processes. This has resulted in a growing interest in technology facilitating autonomous decision making and advanced control. However, the interoperability challenges introduced by heterogeneous distributed control systems and the complex and highly dynamic environment, make it difficult to develop systems using traditional software development methods.

Autonomous decision making and advanced control are fields where the benefits of multi-agent technology really excel. Powerful methods and constructs brought by the multi-agent paradigm can help to automate complex processes in distributed highly dynamic environments. This has been demonstrated in a number of projects spanning from various applications in the defence industry [1], to industrial resource scheduling and planning [2].

Despite many potential advantages of multi-agent technology, it remains unknown whether the full potential of this technology can be realised within oil recovery. The technology has to some extent been demonstrated within oil trading and production, but there has generally been little research on multi-agent technology for use within this application area. As far as we know, there is no existing research targeting drilling processes. This is addressed throughout this thesis, as we aim to demonstrate the applicability of multi-agent technology within autonomous control of drilling rigs.

1.2 Project Context

This thesis is a part of my master degree at the University of Oslo in the context of the AutoConRig research project. AutoConRig is part of Integrated Operations in the High North (IOHN), a programme launched by a large industrial consortium including the Norwegian Oil Industry association (OLF). IOHN aims to facilitate collaboration across disciplines to make better use of the Norwegian petroleum resources [3].

My participation in this project is realised through Computas AS (www.computas.com)- a Norwegian software services company with a long tradition of participation with industrial research projects. Computas has experience from a number of relevant projects targeting the oil and gas industry.

1.2.1 AutoConRig

The primary objective of the AutoConRig project as stated in the project proposal is [4]:

“...to analyze, develop and test an autonomous and semi-automated drilling control system for Oil & Gas Drilling in High North areas, where unmanned drilling rigs placed on the sea bottom can be used to eliminate constraints from extreme conditions”.
This outlines the ultimate vision of semi-autonomous, remotely controlled drilling rigs on the seabed. In this vision, the machinery will be safely controlled from an onshore drilling centre or an offshore support vessel (see Figure 1). The AutoConRig project concerns the analysis, development and testing of a control system, capable of autonomous control of the drilling rig during tripping. This system should be realised through the use of multi-agent technology and the final product should comply with the high requirements on safety and environmental impact.

1.3 Research Goals

This thesis addresses the use of multi-agent systems to facilitate autonomous control of drilling rigs. The main goal is to develop a prototype of an autonomous control system using multi-agent technology. Areas of interest are autonomy, robustness and distributed control in a dynamic environment. The work should include a review of state of the art agent technology, detailed analysis of the problem area and outline areas for further research. The scope of the prototype should be limited to a set of scenarios defined in collaboration with the AutoConRig project group where the final product is demonstrated through an experiment.

1.4 Research Method

In this thesis we apply a research method compliant with [5]. This type of technology research is a process consisting of the stages shown in Figure 2.
The figure illustrates the following three stages:

- **Problem analysis** – Interact with possible users and stakeholders to identify a problem which needs to be solved.

- **Innovation** – Develop an artefact that aims to solve the problems identified during the problem analysis phase.

- **Evaluation** – Based on the initial requirements, formulate hypotheses about a prospected solution and use them to evaluate the artefact. If the predictions comply with the artefact, it can be argued that the artefact solves the identified problem.

![Diagram](image_url)

Figure 3 Research Method Used in the Thesis

This is an interactive process where the results are evaluated according to some metric. The cycle may be repeated several times, depending on the result of the evaluation process as this will either strengthen or weaken the hypotheses. Figure 3 describes the phases in the context of this thesis.

### 1.5 Document Structure

The document structure follows the research method described in 1.4 and is therefore split into the following sections.

- **Problem analysis**

  In chapter 2 - *Project Description*, we give a brief description of the project context, scope and motivation for an agent based approach. It is followed by chapter 3 - *State Of The Art*, where an introduction to state of the art agent technology is given. It continues with chapter 4 - *Related work* where we have listed relevant work addressing the oil and gas industry. In chapter 5 - *Tools and Frameworks* we describe the tools and frameworks used within the project, and chapter 6 - *Application Area* describes the drilling domain and the scenarios defining the scope of our prototype.

- **Innovation**

  Here we outline an approach towards autonomous control of drilling rigs using the Prometheus development methodology. This starts with chapter 7 – *System Specification* where the system is specified, and chapter 8 - *Architectural Design* where the architecture is defined. This section continues with descriptions of the common ontology for our system in chapter 9 - *Shared Ontology*, and ends with descriptions of the agent’s internal details in chapter 10 - *Detailed Design and Implementation.*
• Evaluation

This section is initiated by chapter 11 where our approach to autonomous control is discussed. This is followed by a description of the experiment in chapter 12 - Experiment, and the experiment result in chapter 13 - Experiment Results. In chapter 14 - Conclusion and Future Work we conclude our work, list our achievements and describe future work.
I. Problem Analysis
2 Project Description

Introducing the reader to the problem area and narrowing the scope of this thesis are the focus of this chapter. We also describe the motivation for taking the multi-agent approach to the specific application area.

2.1 Application Area

Drilling operations in the High North will be exposed to the same challenges that we have today, in addition they will be exposed to harsh weather conditions and challenges related to remote location [4]. To deal with these challenges, future offshore drilling rigs are likely to be unmanned and located directly on the seabed. The idea is to have these subsea rigs remotely controlled from an offshore supply station or from an onshore control centre.

However, if the required communication links fail during operations, this can have dramatic negative impact on the rig equipment itself and on the well’s future production capability. This thesis addresses how multi-agent technology can facilitate autonomous control and reduce risk in communication-failure scenarios.

2.2 Scope

Much research needs to be undertaken in order to realise the ultimate vision of unmanned drilling rigs. Owing to this it should be clear that neither the work performed for this thesis nor the AutoConRig project as such, is enough to solely realise this vision [6]. More precisely, this research only concerns the development of the software that facilitates autonomous control. Further, it should be understood that the complexity of drilling operations is very high and we should be careful not to underestimate this complexity.

As a realistic scope for this thesis, we have defined a set of tripping scenarios (see 2.2.1) that the control system should be able to handle. These scenarios are developed by the AutoConRig project group as the scope for the first prototype of an autonomous control system. Each scenario describes a situation where the control centre loses control when the rig is in an undesirable state. The control system should then take control over the drilling rig and autonomously perform operations to move the rig into a more desirable state. This way the autonomous control system can ensure the operability of the system and later when the control centre comes back online, the driller can disable the autonomous control system and continue its work. The set of scenarios is described in section 6.5.

2.2.1 Tripping Sequences

Tripping sequences take place during the drilling phase of a well and involve two separate sequences of operations.

- **Trip-in**
- **Trip-out**

Trip-in is concerned with placing the drill-string into the well and trip-out is the process of pulling the drillstring out from the well. Tripping sequences are performed in a number of circumstances, typical scenarios are during well-equipment replacement or preparations to run tests in the wellbore. For instance, if the operators decide to replace the bit during drilling, the whole drillstring needs to be tripped out. Then the bit can be replaced and the drillstring tripped back into the wellbore.
2.3 Motivation for an Agent-based Control System

An agent is a goal oriented autonomous entity which observes, reasons and acts upon the environment it is situated in. When a system consists of multiple interacting agents it can be called a multi-agent system (MAS). Multi-agent systems are particularly relevant for an autonomous control system as the equipment to operate the various drilling machinery is typically delivered by multiple vendors with their own proprietary control interface. An autonomous control system must therefore be able to handle the interoperability challenges introduced by the heterogeneous environment. MAS provide a natural way to integrate heterogeneous systems through resource encapsulation, allowing heterogeneity to be hidden and potential interoperability issues solved. Multi-agent systems are often distributed and are designed to operate in environments spread across both hardware and software. This is relevant as onshore and offshore systems are likely to be integrated.

We can further benefit from the powerful abstraction mechanisms provided by multi agent systems in the development of complex systems. In our case, entities from the domain (machines, systems, roles, techniques etc.) are good alternatives for encapsulation and abstraction. Such abstractions provide the means to decompose the system into a set of components (agents), each representing a functional entity with well understood semantics (roles). The autonomous control system can benefit from this and use abstractions to make the system easier to understand, maintain and control.

The control system will operate in a dynamic environment that can change rapidly over a short time period. Therefore, the autonomous control system should be able to perform its operations while the data from its environment is continually being monitored and processed. While many traditional computer techniques principally perform operations in a single process, operations in multi-agent systems tend to be distributed across both hardware and processes. Thus, tasks are typically executed in parallel, enabling efficient use of the available computational resources. This is feasible for an autonomous control system as process data may be efficiently monitored, enabling fast detection of critical changes in the environment.

Multi-agent systems are often designed with a distributed model of autonomy, enabling decisions to be made on multiple levels. This model facilitates design of robust systems where failure does not need to harm the whole agent-system. This is appealing for an autonomous control system as it may remain operative during software or hardware failure.

In addition the system should be flexible and produce optimal output with respect to the dynamic environment. The behaviour of an agent system is in contrast to conventional computer systems, often not completely wired at design time. Instead, behaviour is determined during runtime, enabling the system to autonomously adapt to its environment. A control system can benefit from this and produce feasible output in situations not foreseen at design time.

Multi-agent systems often combine reactive behaviour with long term proactive behaviour, making them capable of quickly responding to events, while maintaining a long term agenda. These properties are highly relevant in drilling, as critical events, requiring quick response, can occur at any time during long-running operations.

In conclusion, multi-agent technology seems to be a suitable approach towards a robust control system facilitating autonomous control of a drilling rig. The ability to handle situations not foreseen at design time makes agent technology particularly appealing for this specific application area.
3 State Of The Art

This chapter aims to introduce the reader to the state of the art of agent technology.

3.1 Background

The notion of software agents originated in the late 1970s from within the AI community in response to emerging limitations of conventional knowledge based and expert systems with little or no computational distribution and interaction. Yet three decades of scientific research and unprecedented infrastructure and hardware technology advances later, the notion still carries the somewhat disconcerting fact that academic papers discussing it far outnumber real world implementations beyond mere demonstrators or proofs of concept.

It may be argued that the present notion of software agents with its academic lifeline ties well with the pattern of recurring rise and fall of AI fields of focus over the past few decades. Some may argue that a software agent is little more than contemporary wrapping of early visions of machine intelligence as if research is unconsciously leading itself into old traps by its innate desire to replicate human behaviour in silicon and fibre optics.

However, it is widely accepted that software agents, though still adolescent in some respects, are here to stay. Ongoing research attention and growing commercial awareness have boosted confidence in the ideas and delivered an emergent consensus on just exactly what a software agent is. Increased focus on tools and methodologies to support design and implementation of such systems is considered key to success and rollout of this technology.

3.2 Agents Everywhere

Since its inception some forty years ago, the notion of a software agent has enjoyed remarkable generosity from research and industry in the quest for a clear definition. Countless contributions initially leave many still in the dark on what exactly makes an agent an agent rather than just another software component, object, or module. Surely a component or object or module can be designed to represent anything, so why bother messing up the picture?

The term software agent itself seems a good name for what it is (as we will see), but its wide applicability in our daily lives might have added to the confusion more than helped the contrary. Almost every actively participating function in our society (the postman, your GP, the news presenter, your architect or internet service provider, a night shift on an oil rig) can ultimately be termed an agent or agent coalition, so framing a concise, useful, and universally unambiguous definition for software engineering purposes has proved nontrivial.

We will in the following outline some of the more commonly recognised definitions before delving into the essence of agents and show why the properties and capabilities they exhibit become more important than any short-handed definition.

3.3 Agent Definitions

- Russel and Norvig [7]:
  - “An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.”
• Nwana [8]:
  o “We have as much chance of agreeing on a consensus definition for the word ‘agent’ as AI researchers have of arriving at one for ‘artificial intelligence’ itself - nil!”
  o “When we really have to, we define an agent as referring to a component of software and/or hardware which is capable of acting exactly in order to accomplish tasks on behalf of its user. Given a choice, we would rather say it is an umbrella term, meta-term or class, which covers a range of other more specific agent types, and then go on to list and define what these other agent types are.”

• Luck, McBurney and Priest [9]:
  o Agents as a design metaphor -
    “Agents provide designers and developers with a way of structuring an application around autonomous, communicative elements, and lead to the construction of software tools and infrastructure to support the design metaphor. In this sense, they offer a new and often more appropriate route to the development of complex systems, especially in open and dynamic environments.”

  o Agents as design -
    “The use of agents as an abstraction tool, or a metaphor, for the design and construction of systems provided the initial impetus for developments in the field. On the one hand, agents offer an appropriate way to consider complex systems with multiple distinct and independent components. On the other, they also enable the aggregation of different functionalities that have previously been distinct (such as planning, learning, coordination, etc.) in a conceptually embodied and situated whole.”

• Luck, McBurney, Shehory and Willmott [10]:
  o “Put at its simplest, an agent is a computer system that is capable of flexible autonomous action in dynamic, unpredictable, typically multi-agent domains.”

Numerous other definitions have been suggested, some of which are listed by Stan Franklin and Art Graesser in [11].

An alternative approach to defining software agents in terms of natural language descriptions of what they are is to look at the properties they exhibit, i.e. which qualifications they should display in order to qualify as such.

Many sets of properties have been proposed by research, and a particular property may or may not appear in a particular list depending on contextual constraints or the individual author’s interpretation. In other words, some properties may or may not be considered fundamental, i.e. generally applying to all software agents as an unreserved requirement, which inevitably somewhat smudges the line between definition and categorisation of software agents.

Recognising Russel and Norvig’s interpretation that the notion of an agent is more that of an analysis tool than a definition aimed at categorising systems into agents and non-agents, we believe a qualification oriented definition of software agents is more helpful in understanding what lies behind the notion.

Keeping in mind the floating edge between qualifying and classifying capabilities, we include the following characteristics from what seems broadly accepted in literature as key properties which a software component should demonstrate in order to qualify as an agent.
- **Reactive** – Agents are sensitive to changes in their environment and react to these.\(^1\)
- **Proactive/Persistent** – Agents have goals which set their agenda and drive their actions.
- **Autonomous** – Agents exhibit a degree of independence which allows them to make qualified decisions based on their own perception of the environment, optionally in collaboration with other agents.
- **Social** – Agents can collaborate with other agents.
- **Flexible** – Agents can attempt to achieve their goals in several, alternative ways.
- **Robust** – Agents can recover from failure.

In addition, the ability to learn from its environment and thereby accumulate knowledge over time is usually considered a requirement for “intelligence” as a characteristic of agency.

Beside a multitude of definitions, synonyms like knowbots (knowledge based robots), softbots (software robots), taskbots (task-based robots), userbots, personal agents, personal assistants and others [8] have bravely asserted their validity as agents, presumably in attempts to work around the lack of broader consensus by narrowing the scope of individual instantiations.

Though arguably having accomplished some added mystique, the fact that such mutations of the meme have emerged in the first place deserves some justification. Agents inhabit different environments and may serve fundamentally different purposes with different mandates and goals. As observed by Nwana, the various bots and assistants having surfaced in recent years all exhibit properties of agency and have received their names largely from a *role-oriented classification*.

From the root notion of software agents via its tenuous definitions, we now move on to look at the properties and attributes which constitute the basis for a classification, or typology, of agents.

### 3.4 Agent Classification

Agent communities have introduced a host of prefix adjectives describing different types of agents, including *intelligent* agents, *interface* agents, *information* agents, *learning* agents, *collaborative* agents, *presentation* agents, *management* agents, *search* agents, etc. Many researchers introduce their own terminology to explicitly identify, characterise and describe their agent research while typically focussing on a specific area of interest. This often results in competing terms and uncertainty over which terminology to use.

A type should identify the important aspects of an agent whereas a description of an agent’s elements should describe its environment, sensing capabilities, actions, drives, and action selected architecture [11]. It is difficult to establish a common vocabulary for the many variations and combinations of these properties, so an unambiguous, straightforward scheme for categorising agents has yet to break the surface. As asserted by Franklin and Graesser [11]:

*The only concepts that yield sharp edged categories are mathematical concepts, and they succeed only because they are content free. Agents “live” in the real world (or some world), and real world concepts yield fuzzy categories.*

---

\(^1\) This term is sometimes used in separating between *purely reactive* agents with no internal state or temporal knowledge, and *proactive* agents which can take action on their own initiative based on environmental changes and internal state.
The AI community has categorised agents in terms of weak and strong notions of agency [12].

- **Weak notion of agency:** This notion asserts a set of high-level properties on agents which has become widely accepted:
  - *Autonomous:* Agents act autonomously by displaying a degree of self-governing behaviour. They can sense the state of their environment and act upon it without direct intervention from humans (or other agents) in pursuit of their own agenda.
  - *Social:* Agents are aware of other agents and can collaborate with them by means of some agent communication language.
  - *Reactive:* Agents are sensitive to changes in their environment and act upon these in a timely fashion.
  - *Pro-active:* Agents can display goal-directed behaviour by initiating actions upon their environment without being prompted by external events.

- **Strong notion of agency:** The strong notion of agency goes further and requires agents to be designed using concepts that are more commonly associated with humans such as mental and emotional qualities. A popular paradigm under this notion is widely known as the BDI (Beliefs - Desires - Intentions) design scheme or architecture which offers some powerful hooks for defining agent capabilities and behaviour.

The weak and strong notions of agency are useful on a theoretical level, but they fail to address our need for a more fine-grained nomenclature defining the essential properties that constitute an agent.

Nwana has observed that agents exist in a multi-dimensional space and has listed a set of facets that may be used to classify them [8].

- **Mobility:** Agents are defined as either static or mobile.
- **Deliberative vs Reactive** Agents are classified according to whether they exhibit a trigger/response type of behaviour (reactive with no internal state model) or possess state knowledge and reasoning capabilities including deliberation with other agents.
- **Role:** Agents are classified according to the role they play.
- **Qualification:** Agents are classified according to some ideal and primary attributes which they should exhibit. Three such attributes are:
  - *Autonomy:* Agents operate without the direct intervention of humans or others, and have some kind of control over their actions and internal state [12].
  - *Learning:* Agents have the ability to learn from experience and improve their performance over time.
  - *Cooperation:* Agents can collaborate with other agents to perform a task.
- **Hybrid:** Agents are grouped by combining two or more class dimensions.

The three qualifying attributes identified above have been combined by Nwana as underlying characteristics to derive a topology of these attributed agents comprising collaboration agents, collaborative learning agents, interface agents, and smart agents as illustrated in Figure 4.
Acknowledging that the list is somewhat arbitrary, Nwana collapsed the above dimensions and included his interpretation/knowledge of existing types of agents at the time to suggest the following typology of agents:

- **Collaboration Agents:** Collaboration agents are identified by their autonomy and their cooperation with other agents. These identifying aspects are means used to perform tasks on behalf of their owners.
- **Interface Agents:** Interface agents are characterised by their autonomy and their ability to learn in order to assist the user(s).
- **Smart Agents:** This category of agents is autonomous, cooperative and has the ability to learn. Truly smart agents do not yet exist and are as of today more a feasible vision rather than reality.
- **Mobile Agents:** Mobile agents typically move around in a network, traversing and gathering information on behalf of their user, and return home when done. In this sense an agent is either mobile or static.
- **Reactive Agents:** Reactive systems do not maintain an internal representation of the world, instead they act/respond to events from the environment.
- **Information/Internet Agents:** Like mobile agents, these also gather information and are typically classified by their role, i.e. what they do.
- **Hybrid Agents:** Agents which combine two or more agent theories (or philosophies).

King takes a different approach and suggests a role-specific taxonomy of agents where agents are categorised by what they do rather than how they do it [8, 11]. He introduces thirteen different agent types: search agents, report agents, presentation agents, navigation agents, role-playing agents, management agents, search and retrieval agents, domain-specific agents, development agents, analysis and design agents, testing agents, packaging agents, and help agents.

It may be argued that a role-oriented categorisation of agents does not contribute to an unambiguous categorisation scheme, but instead introduces a potential anarchy by inviting each agent to have its own type, thereby blurring the agent terminology further.

A completely different approach was taken by Franklin and Graesser [11]. They suggested a biological classification schema for agents by introducing a starting point for a taxonomy with only a limited set of top classes defined, in anticipation of a gradual expansion by the community. Others have attempted a more complete taxonomy [13].

The above touches only the surface of the work delivered on agent classification. Sustained contribution from research and industry still feeds the debate and will almost certainly continue to do so for years to come. We agree with the view that until a de facto typology is established (if ever), the best we can do is to acknowledge the absence of a universally adopted classification of agents by staying tuned to further progress with an open mind balanced with objective, critical eyes on attempts to oversell the domain or clutter mainstream understanding.
3.5 Agent Theories

Recipes for defining the nuts and bolts of software agents are provided by research concerned with *agent theories*, which offer formalisms which can be used to structure and represent the characteristics deemed compulsory to obtain a set of desired behavioural capabilities.

Agent theory has suggested the *intentional notion of attitudes* as an appropriate abstraction for representing and describing agent behaviour. Two categories have been proposed as the more important [12]:

- **Information Attitudes**: These relate to information the agent has about its environment, such as
  - Belief
  - Knowledge
- **Pro-Attitudes**: These represent the states that in some way may lead to the agent taking action, such as
  - Desire
  - Intention
  - Obligation
  - Commitment
  - Choice

There are multiple theories directed at providing guidance as to which properties to use in different circumstances, the overall goal being to provide software engineers with useful hooks for designing and implementing agents and their behaviour.

A number of different tools, frameworks, and languages are based on these theories [14]. Recent research has increasingly been targeting the construction of new languages to support development of agent oriented software, which has resulted in several new declarative, imperative, and hybrid incarnations.

The paradigm of programming languages using agent-oriented concepts is called *Agent Oriented Programming* (AOP). The most popular agent theory is based on *beliefs, desires, and intentions*. This approach is called the *Belief Desire Intention* (BDI) model, where an agent’s internal representation of the world is represented using these *mental states*:

- **Beliefs**: Beliefs often refer to the perceptions of an agent and represent the information an agent has about the state of its environment. The term *beliefs* is used instead of *information or knowledge* because the elements of information may not necessarily be “true”.
- **Desires**: Desires denote the state of mind the agent (ideally) wants to achieve. An agent may not always be able to realise all its desires due to inconsistency with other desires or because a particular desire is unachievable.
- **Intentions**: Intentions are the subset of desires that the agent is *committed* to achieving. Once an agent has committed to one or more of its desires, those desires become intentions upon which the agent’s focus is directed.
3.6 Agent Architectures

Research on agent architectures has for some time enjoyed a higher degree of consensus than previous topics on agent definitions and classifications, possibly due to its less abstract nature and - in some respects - closer ties with fundamental software engineering principles.

The study of an agent’s architecture focuses on the internal functional constituents defining its overall behavioural capability. This scope extends to communication and collaboration capabilities in multi-agent architectures.

Due to an inevitable correlation between an agent’s public footprint (its affiliation with a certain type or behavioural capability) and its internal architecture, some architectural terminology is reflected in agent types (or vice versa). Hence a reactive agent architecture reflects a reactive agent type (from Nwana’s typology).

There are two main types of agent architectures (leaving the usual slot for a hybrid mutant):

- **Deliberative**: Sometimes referred to as intelligent or cognitive architecture, deliberative agent architectures offer means to represent state knowledge and define reasoning and collaboration mechanisms within the agent. Deliberative agents are commonly divided into deductive reasoning agent architectures and practical reasoning agent architectures.

- **Reactive**: Reactive agent architectures consider agents as entities which merely react to changes in the environment with stimulus/response types of behaviour.

Along with a combination of the two, we arrive at four categories of agent architectures - reactive, deductive reasoning, practical reasoning, and hybrid agents.

3.6.1 Reactive Agent Architectures

A reactive agent is an agent which is designed to react to changes in its environment without reasoning about it [15]. Since purely reactive agents perform no deduction whatsoever, the testing and verification processes are simplified as these agents should always produce the same response to a given sequence of events.

![Environment and Agent Diagram](image)

**Figure 5: Reactive Agent**

There are several aspects pertinent to reactive architectures and their limitations [15], most of which relate to such agents’ limited perception of their environment. We adopt the notion of disabilities to illustrate some key points.

- **Reasoning Disability**: Most reactive agents do not maintain a symbolic representation of the world (as do agents of a deliberative type), i.e. they base their actions on the nature of perceived events only without regard to any state knowledge.
The information available from such event spaces often fails to deliver an accurate print of a particular situation, and may therefore lead to potentially non-optimal courses of action. Furthermore, reactive agents respond poorly to dynamic changes in their environment since they normally demonstrate a short-time view of the world.

- **Learning Disability**: Another problem is how to design a reactive agent that can learn by experience. Having such agents improve their performance over time has been shown to be very difficult.

- **Social Disability**: Reactive agents may perform well in small agent societies with less complex layered architectures. However, when this complexity grows with increased numbers of layers and expanding matrices of collective behaviour combinations, the complexity of inter-layer communication quickly introduces considerable challenges on reactive agent design.

Many approaches can be adopted in design and development of reactive agents, the best known of which is arguably the subsumption architecture [16], which was developed as an alternative to the symbolic approach to agency [12, 17]. Brooks used this architecture to implement several robots to illustrate and support the view that intelligence exists within the system and does not have to be generated.

This architecture is based on a horizontal layer approach (see 3.6.5 Layered Architectures) where the layers are ordered into a subsumption hierarchy. A level in this hierarchy corresponds to a layer, which may also be viewed as a level of competence. A layer of competence can be seen as a set of desired behaviours. Each layer runs unaware of the layers above, but is able to examine and inject data into lower layers through some internal interface. In other words, each layer can viewed as an agent. The lower layers represent primitive behaviour avoiding obstacles and taking precedence over higher layers.

One of the key benefits offered by this architecture is that new layers of competence can be added with no need for alterations to existing layers. Also, the computational simplicity of this approach enables highly efficient architectures.

### 3.6.2 Deductive Reasoning Architectures

The idea of deductive reasoning agents is based on traditional symbolic AI. This suggests that intelligence can be represented in a system by using symbolic notations (logics) to describe the environment and its desired behavioural capabilities. These representations can be modified dynamically by means of symbolic manipulation following some rules of syntax.

As the authors of [12, 15] observe, two essential problems arise when designing deductive reasoning agents:

- **Representation**: The representation problem (also known as the transduction problem) addresses the question of how to represent the environment in an accurate and descriptive way.

- **Reasoning**: The reasoning problem addresses the question of how to make sure that the agent reasons over its available knowledge and takes appropriate action within a reasonable amount of time.

One solution to the first problem is to adopt the traditional AI approach of using declarative languages, e.g. some type of logic. The next issue is traditionally solved using deduction, i.e. theorem proving. Deductive reasoning agents use deduction to make decisions. A purely declarative based approach to building agents have the advantage of having clean and clear semantics, which is also the main reason why this approach is appealing.

On the other hand, pure logics have some drawbacks. The main issue relates to the use of logic based agents in a time-constrained environment with the need for quick response and efficient decision making. Theorem
proving has the disadvantage of potentially being very time consuming (or never reaching a conclusion). In rapidly changing environments, the seemingly appropriate action taken by the agent may be out of date by the time its reasoning completes. If the environment has changed, the outcome of the agent’s actions may be far from optimal or ultimately have severely damaging or fatal consequences in mission critical or high risk environments.

Another problem related to deliberative agents in general is the inherent limitation imposed by the mapping from real world concepts to formal representations. Since we are not able to provide a complete copy of the real world, agents use an abstract and simplified view where details relevant for reasoning may be lost. The reasoning problem faces additional challenges where a concise mapping from a limited real world concept to a symbolic representation is not even readily available. An example of such is the representation of an image by declarative statements [15]. Furthermore, representing temporal information (how a situation evolves over time) in a dynamic environment using logics is a non-trivial exercise. These issues are still under research and remain largely unsolved at this time.

It can be difficult to see the differences between deductive reasoning agents and practical reasoning agents presented in the next section. The key difference is that deductive reasoning is directed toward beliefs while practical reasoning focuses on how we understand human reasoning. Practical reasoning agents reason over which action to perform in order to solve a task. Deductive reasoning agents use deduction to create the appropriate steps to complete a task.

3.6.3 Practical Reasoning Architectures

Practical reasoning agents are agents that reason over which action to perform. The reasoning process is essentially based on how we understand the human reasoning process itself. Human reasoning may be divided into two phases:

- **Deliberation** - what state of affairs we want to achieve
- **Means-end Reasoning** - how we want to achieve the state of affairs identified by deliberation

The following scenario helps establishing a better understanding of the human reasoning process: You are finished at work for the day and sitting in your car. You now have the choice between whether to go home or to the movie theatre. You want to see a movie, but on the other hand you have a wife waiting for you at home. This choice would be an example of the *deliberation phase*. Suppose you choose to go home, the *means-end reasoning* would then find the means that you need in order to drive home.

Means-end reasoning results in a plan or recipe for achieving the desired state of affairs [15]. This plan can then be executed in an *attempt* to achieve that state of affairs. One such attempt does not have to be successful. Since the environment may change, the result may not always be according to plan.

As mentioned earlier, practical reasoning agents try to replicate how humans reason, but they are unfortunately poor in comparison. When we map a specification of a human reasoning process to a computational model, the model will normally encounter several limitations.

One limitation arises from the fact that computers have limited resources at their disposal for executing a reasoning process. An agent will only have a fixed amount of memory and processor power available to carry out its reasoning. This limits the number of computations that can be performed within a timeframe. Since most agents operate in a time constrained environment, they must finish their computations in a timely fashion using the fixed amount of memory and processing resources available. As a result, the scope of deliberation is limited.
Due to these resource bounds, an agent must monitor its deliberation performance. When deliberation fails to complete within a certain time, the agent may have to stop prematurely and commit to the state of affairs. This can lead to poor decisions, which could have been avoided had the deliberation phase been granted more time.

As briefly mentioned earlier, an agent may not be able to achieve all its desires (whether or not a particular desire is also an intention). As we observed with deductive reasoning agents, their reactive reasoning cousins also face challenges related to changes in dynamic environments during reasoning. Yet another problem arises from reasoning processes resulting in competing conclusions. All of these suggest the problem of priority where the agent must be able to make a qualified decision as to which alternative routes to follow in what order. Such decisions must be based on an appropriate policy, e.g. the quickest solution versus the most economically viable. The risk of landing with a non-optimal outcome remains.

3.6.4 Hybrid Agent Architectures

The idea of hybrid agent architectures is to use the better of the two worlds, i.e. pick the properties from both deliberative and reactive architectures deemed optimal for a particular application. This is often accomplished by a layered approach having some reactive and some deliberative layers. It should be obvious that there are multiple ways to form hybrid architectures. Therefore, this category is suitable for composing alternative architectures that fit best with a particular set of requirements.

3.6.5 Layered Architectures

We include a brief outline of so-called layered architectures as they have equal application across both deliberative and reactive streams. Layered architectures are handy from a pragmatic point of view, which is why Walton and Wooldridge [15, 17] advocate the approach. These concepts currently constitute the most popular general approach to agent architectures [15].

Three common ways to organise layers in a layered architecture are shown in Figure 6 [17].

- **Horizontal/Parallel Layering**: The input flows from the environment and into each layer separately. The information is then transformed to actions, which flow back into the environment. We can therefore view each layer as an individual agent, which combines with the other to form a hybrid agent.

- **Vertical/Sequential Layering, One-Pass**: In a vertical architecture the information passes through the layers and out into the environment through actions. One layer is typically responsible for perception (input) and another layer for performing actions (output).

- **Vertical/Sequential Layering, Two-Pass**: The information flows up the layers and back down and into the environment.
The horizontal approach invites the option to develop and deploy layers independently. A new layer can be added to an agent to represent new behaviours. However, the simplicity of this approach comes at the expense of potential conflicts between the layers. Conflicts occur when multiple layers try to take control of the agent at the same time. Handling such conflicts is a non-trivial task and is often delegated to a separate mediator, which forces consistency among the layers. As a consequence, the mediator may introduce a bottleneck inside the agent.

In vertical architectures a layer depends on the presence of other layers. Therefore each layer must be carefully designed to fit with the other layers. In order for a horizontally layered agent to take action, the control must go through all its layers. This traffic could potentially lead to performance issues. Another issue with this approach is the fact that vertical layered agent architectures are not fault tolerant, as a single point of failure in the layer chain could paralyse the entire agent.

3.7 Multi Agent Systems

The preceding discussions on agent architectures have largely focused on *individual* agents. This section looks at *multi-agent systems* (MAS) – systems comprising multiple agents. A multi-agent system can be considered as a naturally distributed set of subsystems, each possessing agent characteristics.

An individual agent is in itself a powerful entity, but the real potential of agents becomes more evident when multiple agents can mingle and interact. A relatively simple set of agents can display comparatively advanced patterns of behaviour.

Take for example the RETSINA calendar [18]. This system enables automatic, intelligent meeting scheduling on behalf of its user. An appointment is initiated by an agent suggesting a time and place for a meeting to a list of recipients. The system then enters a negotiation phase where the agents take their owner’s schedule into account and collectively decide a time and place for the meeting. Most would agree that a RETSINA agent in itself is fairly simple and that the smartness of the system lies in its negotiation capabilities. Some may even argue that the complex interactions generated in such systems are in fact intelligent.

Multi-agent systems tend to be applied in complex, rapidly changing environments. MAS can in many situations be useful as an abstraction mechanism to aid developers in their endeavours to understand and decompose a complex problem area into manageable entities, interactions, and organisational structures.

![Figure 7: Canonical View of MAS](image-url)
As we can see from Figure 7, the technology and techniques used for realising multi-agent systems can be roughly divided into three levels of consideration:

- **Agent Level** - concerned with agents and their internal structures.
- **Interaction Level** - technology and techniques to facilitate agent communication.
- **Organisation Level** - the “top” level; concerned with encapsulating agent interactions into higher organisational structures. Technologies and techniques specify how agents can be grouped together and act in a coherent fashion.

This chapter will discuss the higher levels of interaction and organisation.

### 3.7.1 Agent Interactions

The ability for agents to communicate and understand each other is fundamental in a multi-agent system. This section aims to address technologies and techniques required for successful communication.

Agent systems are distributed by nature and are typically running in different threads or processes distributed across multiple hardware devices. Agents act independently of each other, without central control (they are autonomous). The absence of a governing entity in an environment comprising a number of concurrent processes or threads suggests an asynchronous scheme for inter-process communication. Two common architectural solutions address how this can be accomplished [15]:

- **Blackboard Architecture** - communication realised through a shared state (blackboard) for message exchange. Agents use a shared resource through which they can pass and receive information.

- **Peer-2-peer Architecture** - agents communicate directly with each other without a third-party (point to point communication).

In addition to having a mechanism for message passing, agents must be able to interpret and understand both the context and the content of a message, yielding the need for a means of shared understanding.

### 3.7.1.1 Blackboard Communication Architecture

As the name of this approach suggests, agents communicate via a blackboard to and from which they can add (write) and subtract (clear) information. Adding information to the blackboard is analogous to sending a message whereas subtracting a message can be seen as receiving the message.

The *Linda* architecture is a popular approach to the blackboard model [15]. Agents in this scheme communicate with a central communication component called a **tuple space**. A **tuple** is a collection of fields of any type, which is asserted or retracted from the tuple space depending on the command used.

The set of commands available for communication using the tuple space is listed in Table 1.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>rd</td>
<td>tuple</td>
<td>Attempts to read the given tuple from the tuple space without deleting it.</td>
</tr>
<tr>
<td>in</td>
<td>tuple</td>
<td>Reads the tuple and removes it from the tuple space afterwards.</td>
</tr>
<tr>
<td>out</td>
<td>tuple</td>
<td>Retracts the tuple from the tuple space.</td>
</tr>
<tr>
<td>eval</td>
<td>exp</td>
<td>Writes a tuple to the tuple space if the arguments hold.</td>
</tr>
</tbody>
</table>
The Linda style of communication is very efficient in situations where a message could potentially be relevant for many recipients. Instead of pouring messages across the network to all the agents as in point-to-point communication, the information can be accessed directly from the tuple space by the affected parties.

The implementation of a blackboard architecture is straightforward - its simplicity being a great advantage. However, the central point of communication may act as a bottleneck for agent communication. This architecture would in other words not scale well with a large number of interacting agents.

Another concern with a centralised style communication is the robustness of the application as a whole. If a system breakdown causes the blackboard to vanish, the agents have no way of communicating. Also, since this approach requires messages to be publicly available over time, it quickly poses questions such as how to determine when a piece of information is no longer relevant and who can delete information when from the shared state.

These and other issues have all been addressed by various amendments to the basic scheme. However, as the complexity of the architecture grows, so does the relevance of considering alternative approaches.

3.7.1.2 Peer-2-peer Architecture

The peer-2-peer (P2P) architecture is, in contrast to the blackboard architecture, completely decentralised. The principle of P2P systems is that each node in the network is considered equal to all others as far as communication goes, and messages are exchanged without a central server.

Clients in client-server architectures are restricted to communicate with a server whereas nodes in P2P architectures interact directly with each other. The response time performance delivered by blackboard type architectures is limited by the computational power, memory, and bandwidth of the shared resource and its communication channels. P2P networks do not suffer from the same limitations. If a node needs to get information, it can get it directly from any node in possession of that information. This eliminates the problem overloading a single node in the network, and resolves the critical problem of shared resource failures affecting the entire network.

3.7.1.3 Agent Ontologies

Interaction architectures provide the means for agents to exchange information and otherwise leave it to the agents themselves to figure out a common language to speak. Walton uses human communication as a direct analogy to this situation [15]. If someone communicates to you in a language you don’t understand, you will not be able to interpret the message even if you received it loud and clear. We characterise this problem as an interoperability issue.

To achieve interoperability, agents must be equipped with functionality to interpret and understand messages. This implies that the agents must agree on a form of the message and have shared understanding of how to interpret the contents (common ontology). The techniques available can be broadly separated into two categories:

- **Shared Ontology** - A vocabulary is shared between the agents, typically expressed through ontology languages. An ontology is a formal representation of a set of concepts and the relationships between those concepts, much like a dictionary for a particular domain or subject, or a namespace used in distributed computing. By using concepts from common ontologies, agents have a mechanism to warrant a common interpretation of the same concepts. The relationships between concepts in an ontology provide the basic means for agents to reason about those concepts.

- **Standards** - Interoperability can be achieved using standards. In this approach, involved parties know the structure and underlying semantics of the messages being exchanged.
Much research and development is currently focussing on standards for interoperability of systems. Recent work on shared ontologies includes developments for the oil and gas industry (ISO15926) and within medicine. For shared standards, examples include emerging standards for electronic business information (ebXML) and the Wellsite Information Transfer Standard Markup Language, WITSML, for real-time drilling data in the oil and gas industry.

Another aspect which facilitates agent communication is agent communication languages, ACL’s. These contain a standard vocabulary for describing how a message should be interpreted. ACL’s provide the vocabulary needed to describe all the interaction types which may occur during agent communication. The vocabulary is based on a classification from theory of speech acts called performative verbs. The theory identifies a classification of verbs, performatives, which have characteristics similar to real world actions. Examples include promise, request, and inform. Using this approach, all messages have an associated performative which expresses the intended meaning of the message.

Two popular ACL’s, KQML (Knowledge Query and Manipulation Language) and FIPA ACL, are both high-level communication languages for agents aimed at becoming common language standards for communication between all types of agents. A message in KQML consists of a performative and a number of parameters. These parameters include information about the sender, recipients, context of the message, language, and terminology used to express the content.

Despite KQML’s popularity in the multi agent community, some points have been criticised [15]:

- **Unclear semantics**: One area that has been criticised is the fact that the semantics of the performatives are defined using natural language which is approximate in nature, leaving room for different interpretations of the standard.
- **Lacking commissives**: Another area that was criticised is the absence of commissives in the set of performatives which is required for defining commitments between agents where e.g. an agent wants to coordinate its actions with other agents.
- **Too expressive**: It was also pointed out that KQML contains too many performatives and that the composition looks rather ad-hoc. This last drawback impedes language implementations and makes it difficult to construct meaningful messages.

The critics of KQML encouraged the Foundation for Intelligent Physical Agents (FIPA) to create a new communication language called FIPA ACL. FIPA is an IEEE Computer Society standardisation group for agent technology. The FIPA ACL language has according to Walton successfully overcome the shortcomings of KQML, and has become the current standard for ACL’s. This language is built upon the same principles as the KQML language and has many similarities, including the message structure which is syntactically similar to those used in KQML.

One key difference between KQML and the FIPA ACL is that the semantics of FIPA ACL are described formally using logics as opposed to KQML’s descriptions in natural language. Another distinction is the fact that FIPA ACL consists of 22 performatives compared with 42 in KQML, omitting those that are considered services provided by other agents or viewed as concepts defined within the context of messages or within the messages themselves [20].

Examples include KQML performatives such as register, unregister, recommend, broker, recruit, broadcast, transport-address, forward and advertise. These concepts do not exist in the FIPA ACL specification and it is assumed that if this type of functionality is needed, it should be provided as services from agents.

Further, concepts like ask-one, ask-all, stream-all, eos, standby, ready, next, rest and discard do not exist in FIPA ACL because they are assumed to be embedded into the content of the message. Goal defining
performatives like *achieve* and *unachieve* are in the FIPA standard assumed to be defined in the context of the messages, i.e. in the communication protocols.

Finally, FIPA treats agents as autonomous entities and therefore assumes that it is impossible to directly manipulate beliefs as implicitly assumed by KQML with performatives like *insert, uninsert, delete-one* and *delete-all*.

### 3.7.1.4 Coordination and Negotiation

We have addressed two main architectures for message exchange and how common ontologies can be used to identify the general purpose of a message. We now move on to look at how messages can be associated with a context by describing the structure of the overall purpose of the conversation.

Table 2: Walton and Krabbe Dialogue Types [15]

<table>
<thead>
<tr>
<th>Dialogue Type</th>
<th>Initial Situation</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deliberation</td>
<td>Conflict of interest</td>
<td>Reach a deal or solution to the problem</td>
</tr>
<tr>
<td>Enquiry</td>
<td>General ignorance</td>
<td>Growth of knowledge</td>
</tr>
<tr>
<td>Eristic</td>
<td>Antagonism</td>
<td>Humiliation</td>
</tr>
<tr>
<td>Information seeking</td>
<td>Personal ignorance</td>
<td>Spreading knowledge</td>
</tr>
<tr>
<td>Persuasion</td>
<td>Conflicting point of view</td>
<td>Conflict resolution</td>
</tr>
<tr>
<td>Negotiation</td>
<td>Conflict of interest</td>
<td>Making a deal</td>
</tr>
</tbody>
</table>

A conversation is typically realised with a series of related messages, referred to as a *dialogue*. A summary of different dialogue types is shown in Table 2.

Many problems in this area are studied in other disciplines including *economics, political science, philosophy* and *linguistics*. A host of research and development activities in computer science are therefore concerned with mapping results from prior work into computational theories [10]. This work has resulted in many theories for *negotiation, cooperation and coordination*. We won’t go into the details of these theories, but briefly mention areas central to MAS.

Agents are typically required to cooperate in order to achieve a common goal. One related challenge is to identify a mechanism to enable agents to *coordinate* their actions without direct human intervention. Current research addresses many types of coordination mechanisms, ranging from cooperation without any explicit communication between participants to coordination protocols describing the structure of the conversation, coordination media, and distributed planning [10].

Having all agents in a MAS sharing the same goals is often implausible, and conflicts occur. In some situations it makes sense to have agents which are able to reach agreements. In order to achieve mutual agreements, agents need to be equipped with *negotiation* and *argumentation* capabilities. These scenarios need to be governed by a particular *mechanism* or *protocol* defining the rules of the process [17]. There are a number of protocols designed for negotiation scenarios, ranging from simple auctions to complex argumentation scenarios.

### 3.7.2 Agent Organisations

During agent interaction there is typically some organisational context present which defines the relationship between the agents. Such a context may involve agents working together in a structure of *teams* [19]. By using organisations as abstraction mechanisms, theories from other disciplines including *sociology, anthropology* and *biology* become relevant [10].
Organisational structures are often characterised as either open or closed, where open structures grant agents access to join and leave as they please whereby closed organisations hold a finite number of agents. An example of an environment suited for a freely open society is the Internet. The service potential of agents acting on the Internet is enormous, but freely open societies require infrastructures for supporting security, trust, norms and obligations - all issues receiving ongoing attention from research.

There is an emerging trend to have agent organisations dynamically adapt to emerging goal or environmental changes without explicit external control [10]. The *Autonomous Nano Technology Swarm* project, ANTS, was launched by NASA to investigate the use of swarms of small spaceships to explore an asteroid belt [21]. Autonomy and robustness are imperative success factors in this project, making self organisation essential. When anomalies such as asteroid collisions or hardware malfunctions are detected, roles and functions within the swarm need to be dynamically reconfigured to ensure recovery from failure without the need for external control.

As illustrated by the ANTS project, self organising capabilities increase the autonomy and robustness of the system as a whole, thereby reducing the need for external involvement. Features enabling a MAS to adapt to changes in unpredictable and hostile environments are vital in realising the power of the agent paradigm, particularly in mission critical operations [10].

### 3.8 Methodologies

Software engineering methodologies are sets of guidelines documenting proven methods and techniques for design and development of software systems.

Despite the active research on methodologies targeting agent oriented systems development and the number of proposals made, they are still in their early stages compared with methodologies for object oriented programming [10]. Further research along with broader understanding of agent theories in the industry in general is needed for these methodologies to mature.

Currently available methodologies are weak in a number of areas [10, 22]:

- **Vague Documentation**: One aspect of the maturity issue is to some degree related to poor descriptions in the methodologies of project phases such the transition from design to actual implementation of a system.

- **Testing, Debugging, Maintenance**: Crucial phases in a software life-cycle such as testing, debugging, and maintenance have seen little or no exposure in existing agent systems development methodologies.

- **Estimation and Quality Assurance**: Planning, estimation and quality assurance add to the list of essential activities in a software development project, but are hardly addressed by relevant methodologies at present. These aspects should be incorporated in agent systems development methodologies to make agent technology more attractive to the commercial market.

- **Weak Tool Support**: Proper tool support for a given methodology is ultimately a live-or-die in the competition for community attention and adoption by industry. Many methodologies lack the support from stable modelling tools.

The *Prometheus* methodology is an example of a multi-agent development methodology which is backed by a proper design tool supporting multi-agent systems design, model consistency and checking, and automatic code generation. Prometheus is also closely tied with the *JACK Intelligent Agents* framework, which along with its strong tool support has contributed to the popularity of this methodology.
4 Related work

This chapter gives an overview of agent technology applied in the oil and gas industry. It aims to give the reader an idea of the type of processes that are automated using agent technology within this particular domain.

4.1 Agents in Oil and Gas

During our literature search it appeared that little research has been conducted on agent technology within the petroleum domain. Specifically, all identified material is limited to either oil trading [23] or oil production [24, 25] and can be linked to StatoilHydro (see [26] for an overview). The material outlines potential benefits and application areas for multi-agent technology. However, it is also clear that current research has merely scratched the surface in relation to its full potential within the domain.

We provide descriptions of two multi-agent systems related to oil production, as they seem to be closest to our application domain (drilling). One of those is Hallen and Engmo’s master thesis [25], in which they investigated the applicability of agent technology in oil production. This resulted in a proof of concept system demonstrating how a multi-agent system can be used to optimise production of an oil field.

![Multi-Agent Architecture for Production Optimisation](image)

**Figure 8** Engmo & Hallen Multi-Agent Architecture for Production Optimisation [2]

This optimisation is performed based on parameters from the local wells (sand content, water content and pressure) together with the capacity of the processing plant. An overview of the architecture is shown in Figure 8. To illustrate how their prototype works, a textual description of each agent in this structure is provided below.

- **WellMonitor**: Continuously monitors and records the level of sand in the associated well and alerts *WellController* when sand is detected.
- **WellController**: Is responsible for adjusting the choke of the associated well through calculations based on the wells production goal. The choke is adjusted on reports from *WellMonitor* showing sand in the well, after requests on production rate from *ProductionOptimiser*, and on direct request from the operator. The *WellController* is in addition responsible for the notification of the *FieldCoordinator* on critical situations, like for instance when a dangerously high level of sand is detected in the production facility.

- **FieldCoordinator**: Creates and passes alarms to *OperatorAssistant* in situations where *WellController* or *ProductionOptimiser* reports a critical situation. It also facilitates communication between agents in the control room and the agents in the field.

- **ProductionOptimiser**: Is responsible for initiating the optimisation of the oil production on request from the *PlantMonitor*. The optimisation is performed by collecting water content values from the wells, and based on these the agent determines which wells need adjustments. This also involves the detection of critical situations in the processing plant (e.g. high water levels) which is reported to the *FieldCoordinator*.

- **PlantMonitor**: Continuously monitors and records the level of water in the processing plant. It alerts the *ProductionOptimiser* when the water level passes a given limit.

- **OperatorAssistant**: Interacts with the human operator, receiving instructions and relaying them to the rest of the system.

To test the applicability and performance of their system, a simulator was created. According to the thesis the tests showed improvement of the production, despite this validity threats have been discovered [24].

In Spillum’s project report [24], they argue for an improvement over the architecture above. The prospected improvements include organisational structures using teams, and the concept of autonomy delegation by distributing autonomy among the agents, learning from experience, increased proactive behaviour by forecasting, and the introduction of subsea templates.

![Figure 9 Spillum's Refined Architecture](image-url)
Figure 9 shows the hierarchy of teams introduced in the refined architecture. A short description of the most important features at each level in the authority hierarchy is provided below:

- **Human operators** are on top of the hierarchy and have the highest level of authority. They are capable of manually overriding the system by feeding the operator assistant with instructions. Note: these operators are humans and not software agents.

- **Operator assistant (OA)** is the interface agent for the human operators. It receives and acts according to its input from the human operators. When the operator assistant receives a production target, a negotiation phase with Optimising field Oil Production System is initiated to establish a contract stating an agreed production target. The agent gets notified if the contract is later breached and the production target cannot be met.

- **Optimising field Oil Production System (OFOPS)** signs contracts with the operator assistant stating a reasonable production target, and plans how its obligations should be fulfilled based on plans received from SubSea template Collections. In situations where it gets notified because the Subsea template collection cannot reach its production target, re-planning is performed. If the re-planning does not fulfil the production requirement stated in the contract, the contract needs to be re-negotiated.

- **Subsea Template Collection (STC)** creates alternative plans stating different production rates. The plans are based on forecasts from associated subsea templates and are sent to Optimising field Oil Production System for selection. When the Optimising field Oil Production System has selected a plan, a contract gets established with the subsea templates, stating what the templates should produce according to that plan. If a subsea template reports that it cannot produce according to its contract, the notification must be forwarded to Optimising field Oil Production System. If a sudden change in the reservoir is detected, the other team members should be notified in order to adjust their production rate to be ahead of the upcoming environment change.

- **Subsea Template (ST)** makes multiple plans showing optimal production alternatives for the associated wells. These plans contain combinations of production forecasts provided by the wells, and are sent to the Subsea Template Collection which selects a plan as their contract. After the production-plan is chosen, contracts to ensure the execution of the plan are sent to wells for approval. On notifications from a well stating a contract breach, a compensational action is performed where the other wells belonging to template are asked to increase their production to meet the production loss. If the production loss cannot be compensated for, the associated Subsea Template Collection must be notified. In situations where a sudden change in the reservoir is detected, the other team members should be notified to adjust their production rate to compensate for the prospected situation.

- **Well (W)** creates alternative production plans based on the composition of the production substances. Let the subsea template choose among the production plans and use the selected plan as the contract. If the contract gets breached, the Subsea template should get notified. If a change in the reservoir occurs, the team members should be notified in order to adjust the production rate as early as possible.

Claimed benefits over [25], includes improvements over time by machine learning, proactive behaviour by forecasting, and increased scalability and responsiveness as a result from the introduction of subsea templates and distributing autonomy.

*It should be noted that the systems mentioned here are conceptual systems demonstrating the benefits of agent technology, and they are not developed for or tested against real data or in an actual production environment.*
5 Tools and Frameworks

This section begins by describing how we selected methodologies, tools and frameworks for our prototype. It progresses by providing detailed descriptions of the selected technology stack used in the development process.

5.1 Evaluation of Tools

There are many approaches to multi-agent systems. A commonly used method is to let a software development methodology guide the development process, using proven methods for all the stages of the development lifecycle. However, as addressed in section 3.8, presently available methodologies are generally in premature stages and suffer from significant limitations, such as lack of adequate tool support.

Tools typically boast a large number of both commercial and non-commercial varieties and frameworks supporting design, development, and deployment of autonomous multi-agent systems (See AgentLink [27] for an extensive list of references to educational, open source and commercially available agent oriented tools and frameworks). These systems come in a variety of shapes with varying degrees of support for the different disciplines associated with software engineering life cycles.

A comprehensive evaluation of the available methodologies, tools, and frameworks without any firsthand experience, would be very demanding in time and space. It is not within the scope of this thesis to perform such an analysis. However, the selection of methodology and technology is very important as it is likely to have significant impacts on the development process and the final product. It should also be noted that we attempted without success to locate any relevant, updated papers comparing or reviewing the available technology in the multi-agent field.

To ensure a near optimal selection of technology without conducting a review ourselves, we contacted StatoilHydro, a Norwegian oil company with firsthand experience relating to multi-agent technology through a number of research projects. StatoilHydro shared numerous experiences gained with their current technology stack for developing multi-agent systems. This information gave us an indication of the current state of this technology and a technology stack providing a safe approach towards multi-agent systems. Based on our observations we decided to adopt their technology stack, consisting of the Prometheus development methodology, the Prometheus design tool and the JACK agent platform, all have being described throughout this chapter.

5.2 The Prometheus Development Methodology

Prometheus is a practical methodology aimed at beginners, having successfully entered industrial workshops and university classrooms [28]. The methodology is developed by the University of Melbourne in collaboration with AOS (Agent Oriented Software).

The Prometheus methodology describes three phases (see Figure 10).
**Figure 10: Prometheus Methodology Overview**

- **System Specification** - Aimed at capturing requirements of the system including inputs and outputs.

- **Architectural Design** - Uses results from System Specification to identify agents and the interaction between them.

- **Detailed Design** - Addresses internal details of each agent, such as how they should perform the tasks assigned to them in the Architectural Design phase.

Primarily, the purpose of the System Specification phase is to devise an overall arrangement for the system. The system’s environment is specified by identifying its inputs (percepts), outputs (actions), and available data.

A functional specification identifies goals and functionality/roles needed to achieve them. This specification contains narrow descriptions, specifying exactly what the system as a whole should do through a set of scenarios. These scenarios should contain a name, a description in natural language, related actions, percepts, along with data used and produced.

Identifying the individual agents in the system is the main focus in the architectural design phase. This also involves designing the overall system structure and defining the interactions between the agents. This process is based on the software engineering principles of strong coherence and low coupling, where indications for grouping are related or have similar functionality, i.e. functionality sharing the same data and/or areas with significant interaction. Prometheus uses an agent *Acquaintance Diagram* to visualise and evaluate the agents and their communication paths. This can be used to identify poor designs with problems such as excessively tight coupling between agents.
Once the agents are identified, an *agent descriptor* is used to specify each agent in terms of properties. The *agent descriptors* are consistency checked against the outcome of the *system specification phase* to verify coherence.

The architectural design phase covers details on *percepts* and resulting *events*, as well as specification of which events are handled by which agent. Percepts in this context refer to the raw data that is available from the agents’ environment. Events on the other hand are triggered when something significant to the agent system occurs. Message exchange details are also specified by the architectural design, these include interfaces, message semantics, and communication language. Finally, resources which need to be shared between multiple agents should be identified here.

All the gathered information is visualised graphically by diagrams such as the *System Overview* diagram for static properties and *interaction diagrams* for system dynamics. The System Overview diagram shows the static relationship between agents, events and shared data resources. Interaction diagrams are derived from the use of cases which describe the functionality of the system, and should include an overview of the most important flow of events. The interaction diagrams are supplemented with *interaction protocols* providing exact specifications of the valid interaction sequences.

Finally, the detailed design phase of Prometheus methodology targets the internal structure of each agent and how it achieves its design objectives.

### 5.3 JACK

JACK is a commercial product from AOS providing a complete set of tools to develop, test and run multi-agent systems [29]. It is regarded by many as the market leader in industrial grade agent frameworks and is used in a variety of large-scale commercial and non-commercial research projects.

The key components of JACK are:

- **JACK Agent Language** - JAL is a super-set of the Java programming language providing syntactic and semantic extensions. These add agent oriented reasoning entities through specific classes, interfaces, methods, definitions, and statements.

- **JACK Agent Compiler** - The JACK Agent Compiler pre-processes JAL source files and produces plain Java source code which can run on any Java Virtual Machine (Java VM) compatible target device.

- **JACK Agent Kernel** - Representing the runtime engine for JAL programs, the JACK Agent Kernel consists of a set of classes running behind the scenes to provide the underlying infrastructure which gives these programs their agent oriented functionality. Other classes provided by the kernel are used explicitly in JAL code and are supplemented by callbacks to provide agents with their required agent oriented capabilities.

- **JACK Development Environment** - JDE is a cross platform editor suite for developing JACK based applications. It provides graphical design support for the JACK constructs and powerful debugging tools, all integrated within the JDE.
At the core of the JACK execution model is the BDI (Beliefs, Desires, Intentions) agent architecture (see section 3.5). JAL provides the means to program directly using BDI-constructs, providing an efficient approach towards software capable of autonomous decision making. A summary of JACK’s key programming constructs are outlined in Table 3.

### Table 3 JACK Key Programming Constructs [30]

<table>
<thead>
<tr>
<th>Construct</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Event</strong></td>
<td>Events are the central motivating factor in agents. Without events, the agent would be in a state of torpor, unmotivated to think or act. Events can be generated in response to external stimuli or as a result of internal computation. The internal processing of an agent generates events that trigger further computation.</td>
</tr>
<tr>
<td><strong>Plan</strong></td>
<td>Plans are procedures that define how to respond to events. When an event is generated, JACK computes the set of plans that are applicable to the event and selects the plan that will form its next intention. Plans have a body that defines the steps to be executed in response to the event. The agent can try alternative plans to achieve the same goal.</td>
</tr>
<tr>
<td><strong>Beliefset</strong></td>
<td>Beliefsets are used to represent the agent’s declarative beliefs – what it knows about itself and the world.</td>
</tr>
<tr>
<td><strong>Agent</strong></td>
<td>Agents are autonomous computational entities with their own external identity and private internal state.</td>
</tr>
<tr>
<td><strong>Team</strong></td>
<td>Teams are used to encapsulate the co-ordinated aspects of (multiple) agent behaviour.</td>
</tr>
</tbody>
</table>

In addition to providing its own design tool which is integrated with the JACK Development Environment (see Figure 11), JACK ties well with the Prometheus agent systems design methodology and its supporting Prometheus Design Tool, PDT. PDT provides a graphical environment with support for creating diagrams to define multi-agent system aspects. These include overviews of agent types, inter-agent communication, percepts, actions, data sources, messages, goals, and related functionalities. Many of these constructs map directly to entities in JACK system designs.
5.4 Prometheus Design Tool (PDT)

The Prometheus Design Tool (PDT) [31] provides a structured approach to the design of multi-agent systems. It complements the Prometheus development methodology with tool support. The tool is platform independent and is freely available from the PDT website [31].

![Screenshot of the Prometheus Design Tool](image)

Figure 12 Screenshot of the Prometheus Design Tool

Key features can be summarised through the following [32].

- **Methodology support** – Provides strong tool support for the Prometheus development methodology and is structured around its phases.

- **Graphical interface and structured textual descriptors** – PDT provides a graphical interface to create the diagrams of the Prometheus methodology. In addition to this, it supports textual editing of the various entity descriptors specified in the Prometheus methodology. These are created by a combination of free text and entries based on menus of items.

- **Propagation** – When information is altered in one diagram, it is automatically propagated to all other relevant design views. Similarly, when a new application entity is created, PDT automatically puts this into relevant diagrams.

- **Consistency checking** – Continuously helps the developers to keep the model consistent. This is achieved through constraints based on the meta-model, and logic implemented in the user interface, preventing the user from creating inconsistencies. In addition to these run-time checks, there is also a feature that allows the user to generate a list of errors and warning that can be manually checked by the developer.

- **Report generation** – Creates complete HTML reports from the model with both figures and textual information. It is also possible to create custom reports where the user can specify the elements to include in the report.

- **AUML support** – Supports Agent UML for specifying interaction protocols using a textual notation. Specification in the protocol also populates entities to other relevant design models.

- **Code generation** – The PDT-tool supports code generation targeting the JACK agent language. It supports repeated code generation without altering any user edited code segments.
6 Application Area

The overall complexity of the operations involved in oil and gas recovery forces the scope of this chapter to be limited. It restricts us to having a limited area of focus and prevents us from going into technical details. This section starts off by giving the reader a brief domain introduction with emphasis on the drilling process, and ends with details about the scope of this thesis.

6.1 Oil Recovery

Oil on the Norwegian soil typically occurs offshore in reservoirs located a few hundred meters below the seabed. The geographic location of the reservoirs is one of many complicating factors within oil recovery. To give the reader an overall picture of the processes taking place in modern oil recovery, a description of the complete lifespan of a well is provided. The lifecycle of a well can roughly be divided into five phases [33].

1. **Well planning** - This is the process of creating a well design, describing the optimal path to the oil based on calculation and measurements. This is an interdisciplinary exercise as it requires expertise within geophysics, geology and reservoir engineering [34]. Some fields have been active for a long period of time and consist of a large amount of wells. New wells in these fields have to avoid collisions with existing wells, making the well design extra challenging.

2. **Drilling** - A well is created by drilling a hole into the earth using a drillstring with a bit attached [33]. A controlled portion of the weight of the drillstring pushes the bit forward while it is rotating. After a piece of the hole is drilled, sections of metal casing slightly smaller than the hole are inserted and cemented into the well. Casing acts as isolation from dangers in the formation (e.g. high pressure zones) and strengthens the walls of a newly drilled hole. After the casing is set the driller can continue drilling using a smaller bit until new casing needs to be set.

![Figure 13 Mud Circulation Explained](image-url)
During drilling, drilling fluid aka “mud” is pumped through the wellstring exiting through pores in the bit before returning to surface outside of the drill pipe. The flow of mud is a closed cycle enabling mud to travel up to the surface for analysis, filtering and later pumped back into the wellbore. Mud is a combination of different fluids, carefully mixed to fit the characteristics of the well. The drilling fluid has many functions including cooling of the bit, stabilisation of the pressure in the wellbore as well as removal of rock cuttings as it is swept up by the mud circulation.

During drilling the pipe/drillstring is continuously being extended. This is done by attaching new stands to the pipe. A stand consists of pipes (typically three) mounted together in advance.

3. **Completion** - After a well is drilled it must be prepared for production. This typically involves the making of small holes in the bottom hole casing, called perforations, there are various approaches to achieving this. Perforations enable the flow of fluids from the reservoir to flow into the wellbore. After the path from the reservoir to well is completed, chemicals are injected into the formation to stimulate the reservoir rock to produce hydrocarbons. As a final step a production tubular is lowered into the well and connected to the bottom hole casing.

In many cases the pressure in the reservoir is enough to stimulate the flow of hydrocarbons into the tubular, but this is not always the case. The pressure of the reservoir can by nature be low or it could for instance be lowered by other production wells. These cases require artificial lifting methods like downhole pumps, surface pump jacks or gas lifts.

4. **Production** - It is during a well’s production phase the petroleum from the connected reservoir is being extracted. In this phase the rig used for drilling and completion is replaced with a production facility. After drilling and completion the top of the well is normally equipped with a collection of valves called Christmas tree. The outlet valve of the Christmas tree can then be connected to the production facility and it is ready for production.

5. **Abandonment** – When the production of a well is no longer cost-effective it is shut down and abandoned. In this process, sections of the well are filled with cement, isolating gas from the water and the surface. The casing and tubular is removed and wellhead welded together and buried.

### 6.2 An Introduction to the Drilling Rig

A drilling rig chiefly performs the following three operations;

1. **Hoisting** – One of the basic functions of a drilling rig is its ability to lower and elevate the drillstring into and out of the wellbore.

2. **Rotation** – Another basic function that is required during drilling is the ability to make the drillstring rotate in the wellbore.

3. **Circulation** – During drilling, a drilling rig needs to have functionality to stabilise the pressure in the wellbore, for bit cooling, and to remove cuttings from the wellbore. This functionality is fulfilled by the mud circulation system.
There are many techniques and types of equipment that can be used to perform these operations. We will not go into technical details about how they are performed, but instead provide descriptions of the major functional entities. A sketch of a simplified drilling rig is shown in Figure 14 where the entities of significance for this thesis are outlined. Descriptions of the components based on definitions from [35] are provided below.

The **derrick** is a structure which supports the weight of the **crown block** and the components being hoisted. Actual lifting is performed by a machine called the **draw-work** consisting of a large-diameter steel spool, brakes, a power source and associated auxiliary devices.

The **derrick** is a structure which supports the weight of the **crown block** and the components being hoisted. Actual lifting is performed by a machine called the **draw-work** consisting of a large-diameter steel spool, brakes, a power source and associated auxiliary devices.

The **draw-work** reels the **drilling line** (a large diameter wire rope) in and out in a controlled fashion. The reeling out of the **drilling line** is powered by gravity and reeling in by engines. The end of the **drilling line**, not connected to the **draw-work**, is connected through the **crown block** at the top of the **derrick**, threaded into the **travelling block** and secured to the **drill floor** with the **deadline anchor**. The **travelling block** hangs in the air below the **crown block** and when the **draw-work** reels out or in the **drilling line**, it causes the **travelling block** and whatever may be hanging underneath it, to be lowered or elevated.
At the drill floor, we find a device called **slips**, that can grip the **drillstring** in a relatively non-damaging manner. This device consists of three or more steel wedges that are hinged together, forming a near circle around the **drillpipe**. After the slips is placed around the **drillpipe** the driller slowly lowers the **drillstring**. This downward force pulls the outer wedges down, providing a compressive force inward on the **drillpipe** and effectively locking everything together. Then the upper portion of the **drillstring** can be unscrewed while the lower part is suspended.

Attached to the bottom of the **travelling block**, we find the **hook**. The hook provides a way to pick up heavy loads with the **travelling block**.

![Diagram](image.png)

**Figure 16** Top-drive Connected to the Hook and Travelling Block

The **hook** can be connected to the **top-drive** -the machine that turns the **drillstring** (there are alternative techniques to make the drillstring rotate). The **top-drive** consists of one or more motors (electric or hydraulic) connected with appropriate gearing to a short section of pipe called a **quill**, which may be screwed into the **drillstring** itself.

An alternative to connecting the drillstring to the **quill**, is to connect it to the **elevator**. The elevator is a hinged mechanism that may be closed around the drillstring. This approach is a quick way to connect the **drillstring** to the hoisting components, whilst disabling the rotation functionality of the **top-drive**.

The **mud circulation system** mainly consists of a set of **mud pits**, **mud pumps**, and a cleaning system for mud returns. A drilling rig can have up to 40 **mud pits** and a maximum of 4 **mud pumps**. During circulation a valve is opened, enabling mud to flow into the **mud pump**. The **mud pump** pumps the mud through the **flow line**, up the **standpipe manifold**, into the **top-drive** connecting the mud flow to the **drillstring**. The mud flows down through the drillstring, exiting through the bit into the wellbore. The mud travels together with rock cuttings and other elements up outside of the drillstring and through an important valve at the top of the well called the **blow out preventer (BOP)**. This may be closed if the drilling crew loses control of the formation fluids. By closing the BOP (usually operated remotely via hydraulic actuators), control of the reservoir may be regained and the mud density can be increased until it is possible to open the BOP and retain pressure control of the formation. However, the BOP is connected to a large-diameter pipe, called a **riser**. The **riser** may be considered as a temporary extension of the wellbore to the surface, enabling the mud returns to enter the mud circulation system. The mud returns are cleaned (i.e. rock cuttings removed) and the mud can return to the mud pits.
6.3 Drilling Control Systems

Drilling operations are performed using heavy machinery operated directly from the drill floor. On modern drilling rigs, the drilling crew is protected to some degree from the most dangerous situations involved in the handling of this equipment. This is done by simply enabling the draw-work, the top-drive, mud pumps, and pipe handling equipment to be remotely controlled from a relatively safe location at the rig.

As an extension to this, systems like the Cyberbase workstation from NOV [36], enable the driller to monitor and control the machinery through a single interface (see Figure 17). Using this interface, measurements from sensors are displayed on two embedded screens, and two joysticks together with keypads provide the interface to operate the various machines. Despite such simplified interfaces, most operations are still semi-automatic, leaving it up to the driller to perform the operations. Some of these operations can be fully automated, but are left semi-automated due to the great safety value in having the operations manually performed with a complete overview of the drill floor [37].

The petroleum industry has in comparison to other industries been relatively slow in the exploration of technology enabling tasks to be delegated to computers [26]. This has started to change as recent developments such as wired pipe technology facilitating fast access to downhole measurements and high capacity network through the use of fibre optics have increased the amount of real-time data that are made available for onshore and offshore control centres. Benefits stemming from this new class of tools (like Drilltronics from IRIS and eControl from SINTEF) have simplified drilling by real-time calculation and improved enforcement of safe guards, helping the driller to operate within the well’s safe margins. Other positives include early detection of emerging problems, and optimised execution of operations [34, 38].

6.4 Division of Concerns

The responsibility of operations during drilling is distributed among multiple roles where the exact division of concerns varies between the installations and companies. However, during operation decisions are made on three different levels [34].
6.5 Scenario Descriptions

As outlined earlier, a set of scenarios defines the set of situations which the prototype of the autonomous control system is designed for. These scenarios describe an initial situation and a sequence of actions resulting in desired behaviour with respect to the initial situation. A desirable control system should produce similar output, given the same sequence of events. Scenarios should therefore be used to verify the system’s ability to handle such situations and in addition to this, be used to design the agent architecture. The scenarios introduce the term casing shoe - the lower end of the cased section of the well (see Figure 14). Descriptions of the various scenarios are described below. Note that the system should not perform any form of pipe handling, i.e. not extend or shorten the drillstring.

6.5.1 Scenario 1: Bit above Casing Shoe

The drilling crew run a normal trip-in from a remote control centre when they experience a communication error, leaving the drilling rig disconnected from the control centre. During this scenario an optimal autonomous control system does the following:

1. Detects the communication breach and gains control over the drilling rig.
2. Senses movement on the drillstring (trip-in speed) and reduces it to 0 m/s, following a deceleration curve.
3. Identifies the position of the drilling bit to be above the casing shoe, indicating a relatively small chance of stuck pipe. In practice this means that no vertical movement of the drillstring is needed.
4. Moves over to safe mode by performing the following sequence of actions:
   - Activates the slips: the slips is moved into position
   - Lowers the drillstring to release weight (transfer the weight of the drillstring to the slips).
   - Activates park break
6.5.2 Scenario 2: Bit Less Than 1 Stand in Open Hole Section

The drilling crew run a normal trip-in from a remote control centre when they experience a communication error, leaving the drilling rig disconnected from the control centre. During this scenario an optimal autonomous control system does the following:

1. Detects the communication breach and gains control over the drilling rig.
2. Senses movement on the drillstring (trip-in speed) and reduces it to 0 m/s, following a deceleration curve.
3. Identifies the position of the drilling bit to be in the open hole, but within 1 stand from casing shoe.
4. Hoists the drilling bit up to the well’s cased section because there is a greater chance of stuck pipe below the casing shoe. In this case the casing shoe is within 1 stand from the casing shoe, which means that no pipe-handling is needed to pull the bit up to the cased area.
5. Moves over to safe mode by performing the following sequence of actions:
   - *Activates the slips*: the slips is moved into position
   - *Lowers the drillstring to release weight* (transfer the weight of the drillstring to the slips).
   - *Activates park break*

6.5.3 Scenario 3: Bit More Than 1 Stand in Open hole Section

The drilling crew run a normal trip-in from a remote control centre when they experience a communication error, leaving the drilling rig disconnected from the control centre. During this scenario an optimal autonomous control system does the following:

1. Detects the communication breach and gains control over the drilling rig.
2. Senses movement on the drillstring (trip-in speed) and reduces it to 0 m/s, following a deceleration curve.
3. Identifies the position of the drilling bit to be in the open hole, but more than 1 stand from casing shoe.
4. Since, there is a risk of stuck pipe below the casing shoe an oscillation process is initiated. In this process the drillstring is continually elevated and lowered inside the wellbore, and if possible:
   - *Rotate the drillstring.*
   - *Circulate*

6.5.4 Constraints

Autonomous control of tripping sequences is a topic limited by both time and available resources. Another central constraint is the requirement to work exclusively with equipment that can be used on the planned tests at the Ullrig test rig. These restrictions impose the use of equipment with an available control API. Operations that do not fall into this category are:

- **Manual operations performed on the drill floor** – A typical operation that is not automated on a drilling rig is equipment replacement (e.g. change of drilling bit).
- **Operations that require measurements which are not available** - An example of such an operation is the removing of a stand from the drillstring, as this operation requires the operator to physically see the components on the drill floor. Since, the ICT system has no information about the position of these components, this task cannot be performed autonomously.
II. Innovation
7 System Specification

This chapter describes what we wish to achieve with the autonomous control system. Also discussed is how the business logic from the scenarios can be mapped into the multi-agent system. This corresponds to the system specification phase in the Prometheus methodology. The graphical PDT notation used in this chapter is described in Appendix A1.

7.1 System Description

We would like to develop a system capable of autonomous control of a drilling rig in case of a communication failure. This system should be built upon existing drilling technology and control systems and should be realised through a multi-agent system. The scope of this prototype is limited to the set of scenarios described in section 7.3. Due to the simplicity of these scenarios, it was clear that far more complex scenarios required consideration in the design of the prototype.

Figure 19 System Environment

The system environment is depicted in Figure 19. The red line denotes a significant event flowing into the autonomous control system through the heterogeneous data sources. The blue line shows how this event is handled internally in the multi-agent system, finally resulting in actions flowing back into the environment.

7.2 Assumptions

Due to lack of detailed documentation and limited access to expert resources, we have been forced to make a number of assumptions during the development of the prototype. These are related to technical details of the drilling domain such as information related to the external systems, available sensor data, and auxiliary data sources.
7.3 Interface Descriptions

We assume that the drilling machinery can be orchestrated through a set of functions provided by external control systems. The assumptions made with respect to these interfaces are specified in 7.3.1. We also presume that process-variables are pushed to the multi-agent system during execution. The process variables (or percepts) are described in 7.3.2.

Although these actions and percepts are tightly coupled to the specific interfaces, we aim our design to be generic and not coupled to a specific set of control interfaces. How we achieve this is described in later sections.

7.3.1 Actions

The specific control systems are listed below together with the actions they provide.

Control System for Draw-work:

- *activatePB()*
  It activates the park break on the draw-work, i.e. the draw-work gets secured.

- *deactivatePB()*
  It deactivates the park break on the draw-work.

- *setDWgear( bit direction, int gearmode, int gear)*
  The direction (UP or DOWN), gear mode and gear of the draw-work is set by this action.

  **Parameters:**
  - *direction*
    0 = DOWN,
    1 = UP
  - *gearmode*
    0 = FREE,
    1 = LOW GEAR,
    2 = HIGH GEAR
  - *gear*
    Available gears and their speed depends on the gearmode - parameter:
    
    If Gear mode = 0 then
    0 = 0 m/s
    If Gear mode = 1 then
    0 = 0.1 m/s, 1 = 0.2 m/s, 2 = 0.5 m/s
    If Gear mode = 2 then
    0 = 1 m/s, 1 = 2 m/s, 2 = 5 m/s

Control System for Mud Circulation:

- *setMudCirculation( double setpoint )* 
  It sets the speed of the mud pumps.

  **Parameters:**
  - *setpoint*
    new speed of the mud pumps (legal interval 0 – 100)

Control System for Slips:

- *activateSlips()*
  It places the slips around the drillstring. Note that this does not lock the drillstring as this requires the draw-work to slightly lower the drillstring to transfer the weight of the drillstring from the draw-work to the slips.
• deactivateSlips()

This action removes the slips from the drillstring. This requires that there is no weight on the slips, i.e. the slips is not locked.

Control System for Top-Drive:
• setTDSpeed (int gear, int speed)

It sets both the direction and speed of the top-drive, i.e. controls the rotation of the drillstring.

Parameters:
- gear 0 = FREE (no rotation),
  1 = CW (Clock Wise rotation),
  2 = CCW (Counter Clock Wise rotation)
- speed The rotation speed (0 - 200 RPM)

7.3.2 Percepts

The percepts specified in Table 4 are assumed to be available to the autonomous control system through the systems specified in the Source -column. Note that these systems are the same as the systems specified for the actions.

<table>
<thead>
<tr>
<th>Source</th>
<th>Variable</th>
<th>Data Type</th>
<th>Range</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Auxiliary</td>
<td>BIT_POSITION</td>
<td>double</td>
<td>0-n meters</td>
<td>The position of the bit in the well, measured from the drill floor.</td>
</tr>
<tr>
<td>Auxiliary</td>
<td>CASINGSHOE_DEPTH</td>
<td>double</td>
<td>0-n meters</td>
<td>The depth of the lower casing shoe, measured from the drill floor.</td>
</tr>
<tr>
<td>Auxiliary</td>
<td>TOTAL_DEPTH</td>
<td>double</td>
<td>0-n meters</td>
<td>The total depth of the well, measured from the drill floor.</td>
</tr>
<tr>
<td>Draw-work</td>
<td>DW_GEAR_DIRECTION</td>
<td>bit</td>
<td>0 =DOWN 1 = UP</td>
<td>States whether the draw-work hoists or lowers the drillstring. (see setDWGear in 7.3.1).</td>
</tr>
<tr>
<td>Draw-work</td>
<td>ELEVATOR_STATUS</td>
<td>bit</td>
<td>0 = connected 1 = disconnected</td>
<td>States whether the elevator is connected to the drillstring. If disconnected the drillstring is assumed to be connected to the top-drive.</td>
</tr>
<tr>
<td>Draw-work</td>
<td>PB_STATUS</td>
<td>bit</td>
<td>0 = activated 1 = inactive</td>
<td>States whether the park break is activated on the draw-work.</td>
</tr>
<tr>
<td>Draw-work</td>
<td>DS_TOTAL_WEIGHT</td>
<td>double</td>
<td>0 – n tons</td>
<td>The total weight of the drillstring.</td>
</tr>
<tr>
<td>Draw-work</td>
<td>DW_SPEED</td>
<td>double</td>
<td>0 – 5 m/s</td>
<td>The speed of the draw-work.</td>
</tr>
<tr>
<td>Draw-work</td>
<td>HOOK_LOAD</td>
<td>double</td>
<td>0 – n tons</td>
<td>The weight held by the draw-work (weight of rotation-machine is excluded). E.g. the total weight of the drillstring.</td>
</tr>
<tr>
<td>Draw-work</td>
<td>HOOK_POSITION</td>
<td>double</td>
<td>0 - n meters</td>
<td>The length between the hook and the drill floor.</td>
</tr>
<tr>
<td>Draw-work</td>
<td>MAX_HOOK_POSITION</td>
<td>double</td>
<td>0 - n meters</td>
<td>The highest possible position of the hook. Measured from the drill floor.</td>
</tr>
</tbody>
</table>
### 7.4 System Goals

System goals describe the functionality that the system is going to cover. The main system goals are briefly described below and the complete set of goals and how they relate to each other is depicted in Figure 20.

- **Maintain operativeness** – During communication failure, the main objective for the autonomous control system is to keep the rig operating until the connection to the control centre is re-established. This includes long-term proactive behaviour that prevents the system from getting into dangerous states as well as reactive behaviour where action is taken as a direct response to a significant event.

- **Prevent critical situations** – This goal represents the overall proactive behaviour of the system. If communication failure occurs and the system is in a state likely to affect the future operation of the system, the multi-agent system should proactively perform actions that prevent the system from falling into an undesirable state.

---

<table>
<thead>
<tr>
<th>Draw-work</th>
<th>UPPER_STAND_LENGTH</th>
<th>double</th>
<th>0 - n meters</th>
<th>The length of the upper stand (the stand connected to the hoisting mechanism).</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draw-work</td>
<td>DW_GEAR</td>
<td>int</td>
<td>0 – 2</td>
<td>The current gear of the draw-work. The speed of the gear is depends on the gear mode (see setDWGear in 7.3.1).</td>
</tr>
<tr>
<td>Draw-work</td>
<td>DW_GEAR_MODE</td>
<td>int</td>
<td>0 = FREE, 1 = LOW GEAR, 2 = HIGH GEAR</td>
<td>The gear mode of the draw-work. This indicates the speed range it operates in (see setDWGear in 7.3.1).</td>
</tr>
<tr>
<td>Mud circulation system</td>
<td>MUD_SETPOINT</td>
<td>double</td>
<td>0 – 100</td>
<td>The speed of the mud circulation. The speed is indicated by a set point.</td>
</tr>
<tr>
<td>Slips</td>
<td>SLIPS_STATUS</td>
<td>bit</td>
<td>0 = deactivated, 1 = activated</td>
<td>Stating whether the slips is placed around the drillstring or not. (0 = not, 1 = yes).</td>
</tr>
<tr>
<td>Top-drive</td>
<td>TD_STATUS</td>
<td>bit</td>
<td>0 = disconnected, 1 = connected</td>
<td>Stating whether the top-drive is connected to the drillstring.</td>
</tr>
<tr>
<td>Top-drive</td>
<td>TD_GEAR</td>
<td>int</td>
<td>0 = FREE, 1 = CW rotation, 2 = CCW rotation</td>
<td>The gear used by the top-drive.</td>
</tr>
<tr>
<td>Top-drive</td>
<td>TD_SPEED</td>
<td>int</td>
<td>0 - 200 RPM</td>
<td>States the speed of the top-drive, i.e. how fast the drillstring rotates.</td>
</tr>
</tbody>
</table>
- **React to significant events** – An important goal for the control system is to respond directly to changes in the environment. This is required for the handling of critical situations that can occur at any time.

- **Find optimal strategy** – The purpose of this goal is to find the best long-term strategy to achieve a more desirable state (safe mode).

- **Plan optimal sequence of operations** – During execution, the system should plan the next sequence of operations to apply. This sequence of operations should follow the milestones of the system’s overall strategy towards safe mode.

- **Monitor** – This goal is legitimated by the need to continually keep track of the system state to be able to select optimal strategies and execute appropriate actions. It is also important to detect significant events so they can be quickly handled.

- **Take action** – Reflecting the system’s ability to perform actions, this goal is obviously relevant for long-term agendas (proactive behaviour) as well as in situations where immediate action is required in response to a significant event (reactive behaviour).

### 7.5 Detailed Scenarios

Here we present a detailed description of how the autonomous control system should cope with the scenarios presented in section 6.5. The scenario-descriptions provided here, show traces of optimal sequences of actions. Percepts are input from the external environment and actions are output.

The scenarios are structured differently than the initial scenario descriptions to save space and simply reading. To prevent repeated sequences the scenarios refer to each other and the conditions prior to “communication failure” are separated into an own scenario, *Communication failure*. Also note that “(OR)” denotes situations where there are alternative paths and “..” illustrates evolvement. The relationship between the scenarios is depicted in Figure 21.

![Figure 21 Scenarios](image)

**[S 1] Communication failure**

Communication failure is detected during trip-in and the system takes action to move into a safe state.

**Trigger**: The communication link between the control facility and the drilling rig is breached.

<table>
<thead>
<tr>
<th>Scenario steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Percept SLIPS_STATUS = 0</td>
<td>The percepts here describe the preconditions for the communication failure scenario. These percepts specify the state of the system when the communication failure occurred.</td>
</tr>
<tr>
<td>1.2 Percept MUD_SETPOINT = 0</td>
<td>The most significant information is the percepts indicating movement of the drillstring, such as DW_SPEED, or DW_GEAR and DW_GEAR_MODE.</td>
</tr>
<tr>
<td>1.3 Percept DS_TOTAL_WEIGHT = 50</td>
<td></td>
</tr>
<tr>
<td>1.4 Percept TD_STATUS = 1</td>
<td></td>
</tr>
<tr>
<td>1.5 Percept TD_GEAR = 0</td>
<td></td>
</tr>
<tr>
<td>1.6 Percept TD_SPEED = 0</td>
<td></td>
</tr>
<tr>
<td>1.7</td>
<td>Percept</td>
</tr>
<tr>
<td>1.8</td>
<td>Percept</td>
</tr>
<tr>
<td>1.9</td>
<td>Percept</td>
</tr>
<tr>
<td>1.10</td>
<td>Percept</td>
</tr>
<tr>
<td>1.11</td>
<td>Percept</td>
</tr>
<tr>
<td>1.12</td>
<td>Percept</td>
</tr>
<tr>
<td>1.13</td>
<td>Percept</td>
</tr>
<tr>
<td>1.14</td>
<td>Percept</td>
</tr>
<tr>
<td>1.15</td>
<td>Percept</td>
</tr>
<tr>
<td>1.16</td>
<td>Percept</td>
</tr>
<tr>
<td>1.17</td>
<td>Percept</td>
</tr>
<tr>
<td>1.18</td>
<td>Percept</td>
</tr>
<tr>
<td>1.19</td>
<td>Percept</td>
</tr>
<tr>
<td>1.20</td>
<td>Percept</td>
</tr>
</tbody>
</table>

Communication failure between the remote control centre and the drilling rig is detected.

| 1.21 | Goal | React to significant events |
| 1.22 | Goal | Take action |
| 1.23 | Action | Activate autonomous control |

The communication failure-percept triggers reactive behaviour and action to obtain control over the drilling rig is undertaken as a direct response.

| 1.24 | Goal | Prevent critical situations |

The prevent-goal is triggered to execute operations which prevent the drilling rig from getting into a state that is undesirable for its operation.

| 1.25 | Action | setDwGear(0,2,1) .. setDwGear(0,0,0) |

The trip-in speed is reduced to 0 following an optimal deceleration curve.

| 1.26 | Percept | DW_GEAR = .. |
| 1.27 | Percept | DW_GEAR_MODE = 2 .. 0 |
| 1.28 | Percept | DW_SPEED = 5 .. 0 |

The deceleration process is conducting by gradually lowering the gear of the draw-work until the travelling-block is no longer in motion.

The DW_GEAR and DW_GEAR_MODE percept confirms gear-change, while the DW_SPEED percept indicates the progress of the operation.

| 1.29 | Scenario | Bit above casing shoe |

This scenario covers a further course of action if the bit is located above the casing shoe.

| OR |

| 1.29 | Scenario | Bit less than 1 stand from casing shoe |

This scenario covers a further course of action when the bit is detected to be less than 1 stand from the casing shoe.

| OR |

| 1.29 | Scenario | Bit more than 1 stand in open hole |

This scenario covers a further course of action when the bit is detected to be more than 1 stand in open hole.
Bit above casing shoe

Bit is detected to be above casing shoe and the system takes action to move the system into a more secure state.

Trigger: Communication failure is detected during trip-in, the trip-in process is stopped and the bit position is detected to be above casing shoe.

<table>
<thead>
<tr>
<th>Scenario steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Percept</td>
<td>BIT_POSITION = 1195</td>
</tr>
<tr>
<td>2.2 Percept</td>
<td>HOOK_POSITION = 15</td>
</tr>
<tr>
<td>2.3 Action</td>
<td>activateSlips()</td>
</tr>
<tr>
<td>2.4 Percept</td>
<td>SLIPS_STATUS = 1</td>
</tr>
<tr>
<td>2.5 Action</td>
<td>setDWGear(0, 1, 0)</td>
</tr>
<tr>
<td>2.6 Percept</td>
<td>DW_GEAR = 0</td>
</tr>
<tr>
<td>2.7 Percept</td>
<td>DW_GEARMODE = 1</td>
</tr>
<tr>
<td>2.8 Percept</td>
<td>DW_SPEED = 0 .. 0.01</td>
</tr>
<tr>
<td>2.9 Percept</td>
<td>HOOK_POSITION = 14.9 ..</td>
</tr>
<tr>
<td>2.10 Percept</td>
<td>BIT_POSITION = 1194.9 ..</td>
</tr>
<tr>
<td>2.11 Percept</td>
<td>HOOK_LOAD = 40 ..</td>
</tr>
<tr>
<td>2.12 Action</td>
<td>setDWGear(0, 0, 0);</td>
</tr>
<tr>
<td>2.13 Percept</td>
<td>DW_GEAR = 0</td>
</tr>
<tr>
<td>2.14 Percept</td>
<td>DW_GEARMODE = 0</td>
</tr>
<tr>
<td>2.15 Percept</td>
<td>DW_SPEED = 0.01 .. 0</td>
</tr>
<tr>
<td>2.16 Percept</td>
<td>HOOK_POSITION = .. 14.8</td>
</tr>
<tr>
<td>2.17 Percept</td>
<td>BIT_POSITION = .. 1194.8</td>
</tr>
<tr>
<td>2.18 Percept</td>
<td>HOOK_LOAD = .. 0</td>
</tr>
<tr>
<td>2.19 Action</td>
<td>activatePB()</td>
</tr>
<tr>
<td>2.20 Percept</td>
<td>PB_STATUS = 1</td>
</tr>
</tbody>
</table>
[S 3] Less than 1 stand from casing shoe

Bit is detected to be less than 1 stand from casing shoe and the system takes action to move into a more secure state.

**Trigger:** Communication failure is detected during trip-in, the trip-in process is stopped and the bit position is detected to be less than 1 stand from casing shoe.

<table>
<thead>
<tr>
<th>Scenario steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Percept</td>
<td>BIT_POSITION = 1205</td>
</tr>
<tr>
<td>3.2 Percept</td>
<td>HOOK_POSITION = 5</td>
</tr>
<tr>
<td>3.3 Action</td>
<td>setDWGear(1,1,0), setDWGear(1,2,2)</td>
</tr>
<tr>
<td>3.4 Percept</td>
<td>DW_GEAR_DIRECTION = 1</td>
</tr>
<tr>
<td>3.5 Percept</td>
<td>DW_SPEED = 0 .. 5</td>
</tr>
<tr>
<td>3.6 Percept</td>
<td>BIT_POSITION = 1205 .. 1200</td>
</tr>
<tr>
<td>3.7 Percept</td>
<td>HOOK_POSITION = 5 .. 10</td>
</tr>
<tr>
<td>3.8 Action</td>
<td>setDWGear(1,2,1), setDWGear(0,0,0)</td>
</tr>
<tr>
<td>3.9 Percept</td>
<td>DW_SPEED = 5 .. 0</td>
</tr>
<tr>
<td>3.10 Percept</td>
<td>BIT_POSITION = 1200 .. 1195</td>
</tr>
<tr>
<td>3.11 Percept</td>
<td>HOOK_POSITION = 5 .. 15</td>
</tr>
<tr>
<td>3.12 Scenario</td>
<td>Bit above casing shoe</td>
</tr>
</tbody>
</table>

[S 4] More than 1 stand in open hole

Bit is detected to be more than 1 stand from casing shoe and the system takes action to move into a more secure state.

**Trigger:** Communication failure is detected during trip-in, the trip-in process is stopped and the bit position is detected to be more than 1 stand from casing shoe.

<table>
<thead>
<tr>
<th>Scenario steps</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Percept</td>
<td>BIT_POSITION = 1300</td>
</tr>
<tr>
<td>4.2 Percept</td>
<td>HOOK_POSITION = 5</td>
</tr>
<tr>
<td>4.3</td>
<td>Action</td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>4.4</td>
<td>Percept</td>
</tr>
<tr>
<td>4.5</td>
<td>Action</td>
</tr>
<tr>
<td>4.6</td>
<td>Percept</td>
</tr>
<tr>
<td>4.7</td>
<td>Percept</td>
</tr>
<tr>
<td>4.8</td>
<td>Action</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>4.9</td>
<td>Percept</td>
</tr>
<tr>
<td>4.10</td>
<td>Percept</td>
</tr>
<tr>
<td>4.11</td>
<td>Percept</td>
</tr>
<tr>
<td>4.12</td>
<td>Action</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>4.13</td>
<td>Percept</td>
</tr>
<tr>
<td>4.14</td>
<td>Percept</td>
</tr>
<tr>
<td>4.15</td>
<td>Percept</td>
</tr>
<tr>
<td>4.16</td>
<td>Action</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>4.17</td>
<td>Percept</td>
</tr>
<tr>
<td>4.18</td>
<td>Percept</td>
</tr>
<tr>
<td>4.19</td>
<td>Percept</td>
</tr>
<tr>
<td>4.20</td>
<td>Action</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>4.21</td>
<td>Percept</td>
</tr>
<tr>
<td>4.22</td>
<td>Percept</td>
</tr>
<tr>
<td>4.23</td>
<td>Percept</td>
</tr>
<tr>
<td>4.24</td>
<td>Action</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>4.25</td>
<td>Percept</td>
</tr>
<tr>
<td>4.26</td>
<td>Percept</td>
</tr>
<tr>
<td>4.27</td>
<td>Percept</td>
</tr>
<tr>
<td>4.28</td>
<td>Action</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>4.29</td>
<td>Percept</td>
</tr>
<tr>
<td>4.30</td>
<td>Percept</td>
</tr>
<tr>
<td>4.31</td>
<td>Percept</td>
</tr>
</tbody>
</table>
7.6 High-level Business Logic

The initial scenario descriptions provide the base business logic for our autonomous control system. However, they only capture one level of failure, and do not describe how failures of the failure handling are to handled. Since failure tolerance is important for the robustness of the system, we identified the need to determine how the system should act in case of failure. We have therefore complemented the essential pieces of the business logic with information related to failure. This is described using the UML 2.x activity diagram notation [39] and is shown in Figure 22.

![High-level Business Logic Diagram](image)

**Figure 22 Activity Diagram: High-level Business Logic**

The activities in this diagram denote the goals the system ideally wants to achieve. If a goal is achieved, the normal flow (black lines) is followed, and if an activity fails the error path is followed (red lines).

**Halt operations** - When communication failure occurs the system should try to stop ongoing operations, e.g. stop the drillstring’s vertical movement. If this goal is achieved, the system’s course of action depends on the position of the bit in the well. If the bit position is less than 1 stand from casing shoe then follow the flow-lines to the Move Bit Above casing shoe -activity, else go to the Start Rotation -activity. In the cases where the Halt operations activity fails, the next best action is to try to secure the hoisting machine.

**Move Bit Above Casingshoe** – This activity represents the desirable goal of having the bit above the casing shoe. If the organisation manages to achieve this goal, the next activity is to Lock Drillstring. If it fails to do so, the Start Rotation –activity should be initiated.

**Lock Drillstring** – This is important to secure the drillstring and to release weight on the hoisting entity. This is in practise achieved by locking the slips. Whether this goal is achieved or not the next activity is to Secure the Hoisting Machine.

**Start Rotation** – Since there is a good chance of stuck pipe below the casing shoe, rotation of the drillstring should (if possible) be initiated. Rotation is not absolute requirement which implies that the next activity is to Start Circulation whether the task is achieved or not.

**Start circulation** – If possible, circulate to prevent the drillstring from getting stuck. This activity is not critical for the scenario, so whether the goal is achieved or not the same flow-line is followed.
Secure hoisting machine – The goal of this activity is to lock the hoisting machine e.g. lock the park break on the draw-work. Our current business logic ends here whether this goal is achieved or not.

Move to Upper Hook Position – This activity is relevant if the bit is in the open hole section. The goal is to get the hook in its upper position, i.e. hoist the drillstring as much as possible. This is part of a continuous loop with the Center the hook position – activity. However, if this activity fails, the business logic we have defined indicates that the next activity to start is to Lock Drillstring.

Center the Hook Position – This is related to the Move to Upper Hook Position-activity as they both are part of a process to prevent the drillstring from being stuck in the well. It is achieved by continually lower and elevate the drillstring.

7.7 Organisational Abstractions and Roles

The Prometheus methodology suggests a rather practical approach to multi-agent systems where the functional requirements are directly mapped to a set of roles. These roles are later used to identify agents. The roles of an organisational structure do not necessary map directly to the underlying functional requirements. Therefore we claim that the approach suggested by Prometheus does not consider the advantages of organisational abstractions when identifying agents. However, the Prometheus methodology only acts as development guidelines and allows us to do modifications [40]. We have therefore used an approach inspired by [41] which allows us to incorporate organisational abstractions into our design.

With this approach we designed a vertical layered organisation structure that we believe can cope with the system requirements. The ideas and concepts behind this organisational structure are described in 7.7.1 and the division of concerns with respect to the concepts of roles are described in 7.7.2.

7.7.1 Organisational Structure

We adopted the hierarchical organisation structure that traditionally has dominated large organisations (e.g. most corporations, governments, and organised religions [33]) as it places the system into a well known and understood organisational context for distributed control.

![Figure 23 Distribution of Autonomy](image)

All levels of the organisation distribute their autonomy to lower levels, resulting in local autonomy at all levels. This is illustrated in Figure 23 as we can see that the level of local autonomy follows the layered structure. The upper most level in the hierarchy has the highest level of autonomy and decides upon the overall goals for the system, while the bottom layer has the lowest level of autonomy.

Another important concept related to the dynamics of the organisation is the information flow. Figure 24 shows the information flow. Data from the environment enters at the bottom layer, flows up through the
layers and back down to the environment through actions. This is analogous to the two-pass vertical architecture described in section 3.6.5, but here applied in a multi-agent context.

Figure 24 Information Flow

In this organisation, higher layers operate on a more abstract basis than lower levels. As the information flows up the layers, details are omitted and only the essential pieces of information reach the higher layers. The same principle applies to actions where higher layers have no detailed knowledge on how an action is executed. More specifically, an action begins as a high level goal stated by the top level. This goal is then further transformed into a sequence of operations forming a plan. Each operation in the plan is then refined into actions, executed at the bottom layer.

7.7.2 Roles

The role-concept from Prometheus captures the system’s functionality. Roles in an organisational structure basically cover the same functionality, but some functionality may be implicitly defined within the organisation itself. However, we mapped the levels of our organisational structure (or roles) to the role-concept in Prometheus. The results are described below and depicted in Figure 25.

- **Decision management –role**: Represent the highest authority in the hierarchical organisation. Responsibilities include deciding upon the system’s overall course of action i.e. the high level business logic and to activate the autonomous control system (obtain control) in case of communication failure.

- **Planning –role**: Responsible for carry out the decision made by the decision management-role. This responsibility includes to plan and execute sequences of high-level operations with respect to the current state of the environment. If an operation fails or does not give the expected outcome, re-planning should be initiated with respect to the new updated environment.

- **Operation –role**: This role is concerned with orchestrating the low-level processes made available through the integration –role into meaningful high-level operations for use in planning. These operations should in addition be described in a way that enables the agent (or agents) filling the planning-role to understand when these operations can be applied and how they can be combined into reasonable sequences of operations (plans). With respect to the functional requirements (i.e. scenarios), this role must at least provide operations to perform the following tasks (the final set of operations is listed in 10.3.1).
  - Stop the vertical movement of the drillstring.
  - Hoist the drillstring above the casing shoe.
  - Lower the drillstring until 50% of the upper stand is above the drill floor.
  - Hoist the drillstring until the hook is at its upper most position.
- Facilitate rotation of the drillstring
- Stop rotation of the drillstring
- Lower the drillstring to lock the slips (lock slips).
- Hoist the drillstring to unlock the slips (release slips).
- Wrap the slips around the drillstring (activate slips).
- Move the drillstring away from the slips (deactivate slips).
- Activate the park break on the draw-work.
- Deactivate the park break on the draw-work.
- Start to circulate
- Stop to circulate

- **Integration role**: The main responsibility for this role is to provide a generic interface to the external data sources and control systems. This interface should provide sufficient functionality to ensure that the *Operation* role can achieve its goals (see 10.4.1 for the final set of actions).

Figure 25 System Roles
8 Architectural Design

In this section we define the architecture of the autonomous control system. It roughly corresponds to the architectural design phase in Prometheus. Section 8.1 describes the process of identifying agents and section 8.2 defines the valid interaction sequences. The chapter ends by showing inter-agent communication for the initial scenarios. The graphical PDT notation used in this chapter is described in Appendix A1.

8.1 Agents

During the process of identifying agents, we discovered that elements from drilling operations are good candidates for agent encapsulation and abstraction. In fact, we found an angle of attack that enabled us to map the roles identified in the previous section directly to the drilling domain. We believe real world abstractions are feasible as they could lead towards a set of agents with well understood responsibility and behaviour.

The abstractions adopted from the drilling domain are described in 8.1.1 and the identified agents and how they map to the roles are described in 8.1.2.

8.1.1 Adopted Abstractions

How we mapped the identified roles to the drilling domain is illustrated in Figure 26.

- **Decision management <> Driller supervisor** – The *drilling supervisor* plays an important role in the decision making during drilling operations. More specifically, it contributes in the process of deciding upon the overall course of action. The *decision management*-role has a similar agenda as it concerns the overall decision making in the case of a communication failure. The responsibilities of the *driller supervisor* are therefore quite similar to the *decision management*-role.

- **Planning <> Driller** – While the driller operates the machinery through a control interface e.g. the *cyberbase workbench*, it makes small grained decisions related to the execution of the current operation. The driller uses the information present in the control room (presented through various monitors and alarms), to decide upon the next sequence of operations to perform. These responsibilities are somewhat analogous to the responsibilities associated with the *planning*-role, as
they basically perform the same type of operations i.e. short-time planning of the sequence of operations to perform.

- **Operations <-> Control Interface** – The interface used to orchestrate the drilling machinery serves the means as a good mapping candidate for the operations-role. A typical interface (such as the cyberbase workbench) provides an updated snapshot of the environment to the driller through monitors and alarms. In addition, it holds a complete control interface to the various machines. This is conceptually the same responsibility as we incorporated into the operations-role, as they both hold updated information with respect to the state of the environment and provide a control interface to the drilling machinery.

- **Integration <-> Heterogeneous systems** – The integration-role serves the means to provide an interface to external systems. Here the functional entities of the drilling rig are good candidates for encapsulation and abstraction.

The observant reader would recognise that we will by incorporating these abstractions into our MAS be close to a virtual copy of the drilling rig.

### 8.1.2 Agent Types

As outlined above, the layered organisational structure maps well with the drilling domain. This mapping is pretty straightforward, as each entity from the drilling domain is represented as an agent. The coupling of the organisational roles and the agents is shown in Figure 27.

![Figure 27 Agent-Role Grouping](image)

Note that the *Integration* -role is played by multiple low-level agents. This is to better fit with the distributed and heterogeneous external control systems and data sources. It is feasible as it opens for distributed control with a clear division of concerns, but also because it enables each agent to be closer to its respective control system and provide swifter response in time critical situations.

We have tried to use an abstract naming convention for all the agents as we do not want to have our architecture directly coupled to a specific set of equipment. This is especially feasible as the equipment and techniques used today are likely to evolve in the future. The mapping scheme used for this purpose is described in Table 5.
Table 5 Specific to General Mapping Scheme

<table>
<thead>
<tr>
<th>Specific</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>Draw-work</td>
<td>Hoisting</td>
</tr>
<tr>
<td>Top-drive</td>
<td>Rotation</td>
</tr>
<tr>
<td>Mud circulation</td>
<td>Circulation</td>
</tr>
<tr>
<td>Slips</td>
<td>Slips</td>
</tr>
<tr>
<td>Auxiliary</td>
<td>Well</td>
</tr>
</tbody>
</table>

Descriptions of the agents are provided below, where all are initiated at start-up, and have cardinality 1.

- **Supervisor** – This agent continually monitors the communication links between seabed rig and the operation centre. Communication failure automatically triggers the Supervisor-agent to guide the system into a state, where the system is likely to maintain its operability.

  It is equipped with a complete overview of the high level business logic of the system, and fulfils the decision management-role by using this knowledge to control the system’s long-term agenda. This is achieved by communicating high level goals down to the Driller-agent. These goals may be considered as a path towards a more desirable system state. After a goal is communicated to the Driller, the further course of action is determined based on whether it manages to achieve the state manifested in the goal.

- **Driller** – The Driller receives goal-states from the Supervisor, each describing a state of affairs to work towards. This process starts by the Driller getting an overview of the current situation. It achieves this by interacting with the ControlInterface-agent. This information is then used to determine (plan) the appropriate steps towards the goal-state with respect to the status of the rig and environment. During the planning process, alternative plans consisting of sequences of operations are generated, that will if successfully executed, ultimately put the system in the state specified by the Supervisor. That is, if the planning process resulted in any plans at all. The resulting plans are then evaluated with respect to a cost function, and the most optimal plan selected for execution. The selected plan is then carried out by sending its operations one-by-one to the ControlInterface-agent for execution. If an operation within a plan fails or has unexpected side effects, re-planning is initiated with respect to the new and updated state of the environment. The Driller may either succeed or fail to achieve a goal; the outcome is anyway reported to the Supervisor.

- **ControlInterface** – This agent provides the Driller with a set of high-level operations/services to control the drilling machinery. It also (analogues to monitors and alarms present in a real world control interface) keeps track of the current state of the environment, which at any time is accessible for the Driller. The high-level operations are assembled together by the functionality provided by the low-level agents, i.e. the agents fulfilling the integration-role. Detailed information about the functionality provided by this agent should be sent to the Driller-agent on system start-up.

- **Slips** – The slips-agent integrates the vendor specific control system for the (physical) slips into the multi-agent system. Its responsibility includes to monitor process variables and to provide a generic interface to the slips. Actions associated with this agent are activateSlips and deactivateSlips. It also handles the SLIPS_STATUS-percept.

- **Hoisting** – This agent encapsulates all functionality and percepts related to the hoisting functionality of the rig. These are mapped to the terminology used locally within the agent system. Actions
performed by this agent are deactivatePB, activatePB and setDWgear. The following percepts are handled by this agent: DW_SPEED, DW_GEAR, DW_GEAR_MODE, DW_GEAR_DIRECTION, HOOK_LOAD, HOOK_POSITION, MAX_HOOK_POSITION, DS_TOTAL_WEIGHT, PB_STATUS and ELEVATOR_STATUS.

- **Rotation** – This agent provides an interface to the control systems related to the rotation mechanism on the rig. It also handles percepts related to this functionality. There is one action related to this, setTDSpeed. The percepts handled by this agent are TD_STATUS, TD_GEAR and TD_SPEED.

- **Circulation** – This is also a low-level agent, providing access to the underlying control systems. This particular agent encapsulates the control system for mud circulation. It can perform the SetMudCirculation – action, and handle the MUD_SETPOINT – percept.

- **Well** – The Well continually monitors the well and related equipment. It handles the following percepts, UPPER_STAND_LENGTH, BIT_POSITION, TOTAL_DEPTH and CASINGSHOE_DEPTH.

### 8.2 Agent Interaction

A high degree of the system’s dynamics is within the inter-agent communication. Figure 28 captures how the agents are connected.

![Agent Acquaintance Diagram](image)

**Figure 28 Agent Acquaintance Diagram**

- **Supervisor <> Driller** – There is two-way communication between the Supervisor –agent and the Driller -agent.

- **Driller <> ControlInterface** – The Driller and the ControlInterface -agents both send and receive massages.

- **ControlInterface <> low-level agents** – The ControlInterface -agent communicates using a bi-directional interaction model with the low level agents, i.e. Hoisting, Rotation, Circulation, Slips and Well.

Descriptions of the detailed interactions within these levels are provided in section 8.2.2 and the interaction sequences for the scenarios are described in section 8.2.1.
Figure 29 System Overview Diagram
8.2.1 Interaction Diagrams

We describe the scenarios once again, but this time we indicate how the agents collaborate by showing inter-agent communication. The interactions are visualised using UML 2.x sequence diagrams [39]. The semantics of the messages shown here are described in Appendix D1.

Each lifeline in the upcoming diagrams represents an agent with the exception of Environment and Low-level agents. The Environment-lifeline represents the environment in which the agents are situated in and the Low-level agents-lifeline represents the low-level agents, i.e. Well, Rotation, Circulation, Hoisting and Slips.

Note that the semantics of the interaction diagrams differs from the UML 2.0 specification (see [39]) as we have combined lifeline decomposition and diagram referencing on the “Low-level agents” lifeline. This simplifies the diagrams and enables us to show detailed inter-agent communication between low-level agents, when this is sufficient. An example of combined lifeline decomposition and diagram referencing is shown in Figure 30.

![Figure 30 Combined Lifeline Decomposition and Diagram Referencing](image)
8.2.1.1 Communication failure scenario

The inter-agent communication for the Communication failure scenario, specified in section 7.5 is shown in Figure 31. It captures percepts that enter the system from the external environment and agent interaction.

The OperationSetMsg message is sent from the ControlInterface-agent to the Driller-agent on system startup and contains all the services (operations) which the ControlInterface-agent provides. Each operation has a set of metadata associated with it, describing the conditions that must be fulfilled in order to apply the operation and a description of its effect.

The OperationSetMsg – message is followed by an interaction occurrence referring to the CFS_PreConditions -sequence diagram, shown in Figure 32.
Figure 32 Interaction Diagram: Pre-Communication Failure

This diagram shows low-level details related to the handling of the percepts that occur before communication failure. Percepts (or process) data are pushed to the low-level agents and if a significant change is detected, a MeasurementUpdateMsg-message is sent to the ControlInterface-agent. The ControlInterface-agent captures this information and uses it to maintain an updated snapshot of the environment. The most significant information in this trace is the movement of the drillstring (\texttt{DW\_SPEED} and \texttt{DW\_GEAR\_MODE} are both > 0).

Having explained the interaction occurrence, we continue with the next event in the CommunicationFailureScenario-diagram. The 
\textit{Supervisor} receives a CommunicationFailureMsg from the environment causing it to activate the autonomous control of the drilling rig. It then sends a PlanningGoalMsg to the Driller-agent, requesting all ongoing operations halted. This message or goal triggers a planning process where a sequence of operations leading to the goal is determined (the detailed interactions of the referenced Planning-sequence diagram is shown Appendix B1). The resulting sequence of operations (or plan) consists of one operation: \texttt{haltHoisting}. The Driller sequentially executes a plan by sending its operations one by one to the ControlInterface. This is illustrated by the OperationRequestMsg-message with the attribute pointing to the \texttt{haltHoisting}-operation. The details of the referenced deceleration process are shown in Appendix B2.

After the deceleration process (or \texttt{haltHoisting}-operation) completes, the ControlInterface-agent sends an OperationResultMsg-message to the Driller indicating that the goal of the operation was achieved. Since the Driller’s plan consisted of one operation, this result is forwarded to the Supervisor. The further course of action is determined on the vertical position of the bit in the well. The alt-fragment describes three operands:

1. If above casing shoe, see 8.2.1.2 Above casing shoe scenario.
2. If less than one stand below Casing shoe, see 8.2.1.3 Less than 1 stand from casing shoe.
3. If more than one stand in open hole, see 8.2.1.4 More than one stand in open hole.
8.2.1.2 Above casing shoe scenario

This scenario describes the case when communication failure has occurred, the vertical movement of the bit is stopped and the bit position is detected to be above casing shoe, i.e. in cased section. The cased section is considered a relatively safe section in the well where the chance of stuck pipe is small.

Figure 33 Interaction Diagram: Above Casing Shoe Scenario

The interactions in this scenario are shown in Figure 33. The preconditions for the scenario are illustrated by the first set of messages in this diagram. The BIT_POSITION –percept indicates a bit position at 1195 meters, which implies that the bit is above the casing shoe (Recall the CASINGSHOE_DEPTH –percept from the communication failure scenario, stating the casing shoe depth to be at 1200 meters).

The PlanningGoalMsg – message with the move bit above casing shoe – goal is sent to the Driller-agent. This goal state describes a condition where the bit is above the casing shoe and since this condition is already achieved, it responds with a PlanningGoalResultMsg –message describing that the state was achieved (achieved parameter is set to true).

This message triggers the Supervisor-agent to locate the next goal according to the business logic described in 7.6 and to send it to the Driller-agent. This is the Lock drillstring –goal which is indicated by the
PlanningGoalMsg – message. This message triggers the Driller to start a planning process where the sequence of operations to lock the drillstring is decided upon. This process results in a plan consisting of two operations: ActivateSlips and LockSlips. Each of these operations is then wrapped in an OperationRequestMsg – message and sent to the ControlInterface-agent for execution. The first OperationRequestMsg causes the ControlInterface-agent to send an ActivateSlipsActionMsg to the low-level agents (i.e., the Slips-agent), which invokes the external ActivateSlips() -function. The SLIPS_STATUS –percept confirms that the slips was activated, which is reported back to the ControlInterface in the ActionResultMsg -message. The next OperationRequestMsg – message refers to the locking of the slips-component which Figure 34 describes in detail.

![Figure 34 Interaction Diagram: Lock Slips](image)

As the diagram shows, the locking starts by a LockSlipsActionMsg –message being sent from the ControlInterface-agent to the Hoisting-agent. The Hoisting -agent reacts to this message by setting the draw-work to lower the drillstring in its lowest gear using the setDWGear –function.

The loop-fragment illustrates the waiting period before the HOOK_LOAD (percept) drops to 0. In practise it means that the weight of the drillstring is transferred to the slips-component, i.e. the slips gets locked. When the hook load has dropped to 0, the vertical movement of the drillstring has been forced to a standstill by the grip of the (physical) slips, and the draw-work is stopped. After the hoisting stops the Hoisting-agent sends an ActionResultMsg stating a successfully executed operation.

When the slips is locked the ControlInterface sends an OperationResultMsg back to the Driller, indicating that the locking-operation was successfully executed. The Driller has then successfully achieved the goal-condition (the slips is locked) which is reported back to the Supervisor.

The Supervisor responds by sending another goal, secure the hoisting machine, to the Driller. This is indicated by the PlanningGoalMsg being sent from the Supervisor to the Driller. The Driller -agent plans towards this goal condition and comes up with a plan consisting of a single operation: activatePB. The Driller -agent then triggers the execution of this operation by sending an OperationRequestMsg –message to the ControlInterface with a pointer to it. This triggers the ControlInterface -agent to forward this request to the

68
**Hoisting-agent** (the **Low-level agents** – lifeline in the diagram), which executes the **activatePB**-action. The **PB_STATUS**-percept confirms the success of the operation which is communicated to the **ControlInterface** through the **ActionResultMsg**-message. The **ControlInterface**-agent has also completed its operation which is indicated by the **OperationResultMsg**-message sent to the **Driller**. The **Driller**-agent has then successfully executed all operations in its plan and a **PlanningGoalResultMsg** is sent to the **Supervisor**. The **Supervisor** believes it has achieved a desirable state and takes no further action.

### 8.2.1.3 Less than 1 stand from casing shoe

This scenario depicts how the multi-agent system handles communication failure when the bit position is less than 1 stand in the open hole. As there is a greater chance of stuck pipe in the open hole section, the best strategy is to pull the bit up above the casing shoe (get to the above casing shoe scenario). A system trace for this scenario is described in Figure 35.

![Figure 35 Interaction Diagram: Less Than 1 Stand from Casing Shoe Scenario](image)

The initial messages in this diagram describe the pre-conditions for this scenario. The **BIT_POSITION**-percept indicates that the bit is slightly (**BIT_POSITION** = **CASINGSHOE_DEPTH** = 1205 – 1200 = 5 meters) below the casing shoe and the **HOOK_POSITION** indicates that the drillstring can at maximum, be hoisted (**UPPER_STAND_LENGTH** – **HOOK_POSITION** = 30 – 5 = 25 meters up. This is an ideal opportunity to hoist the bit above the casing shoe with good clearance.

The core-scenario starts by the **Supervisor** sending a **PlanningGoalMsg**-message containing the **Move Bit Above Casingshoe**-goal to the **Driller**. The **Driller**-agent initiates a planning process resulting in a plan with only one operation: **gotoCasedSection** (we refer to Appendix B1 for details of the planning process). It delegates the execution of this task to the **ControlInterface** by sending an **OperationRequestMsg**-message with a reference to the operation. The **ControlInterface** reacts to the request and sends a **HoistActionMsg**-message to the **low-level agents** (Hoisting -agent) with instructions to elevate the drillstring 20 meters. The details of the referenced hoisting –interaction diagram is shown in Appendix B2. After the hoisting operation is successfully executed, the **ControlInterface** gets notified and the message is passed to the **Driller** and from the **Driller** to the **Supervisor**.
The bit is now elevated to 1195 meters and we are now moving into the *Above casing shoe scenario* (see section 8.2.1.2).

**8.2.1.4 More than one stand in open hole**

This scenario illustrates the situation where the bit is located more than one stand below the casing shoe. A system trace describing this scenario is shown in Figure 36.

[Interaction Diagram: More than 1 Stand in Open Hole Scenario]

The conditions characterising this scenario are described by the BIT_POSITION –percept arriving as the first message in this diagram. Given the bit position is lower than the casing shoe depth (CASINGSHOE_DEPTH < BIT_POSITION), and pipe handling is needed in order to travel up to the cased section (max distance to hoist <
distance to casing shoe = (\(\text{UPPER\_STAND\_LENGTH} - \text{HOOK\_POSITION}\)) < (\(\text{BIT\_POSITION} - \text{CASINGSHOE\_DEPTH}\)), the bit cannot be hoisted above the casing shoe.

The \textbf{Supervisor} sends a \texttt{PlanningGoalMsg} message to the \texttt{Driller} describing rotation of the drillstring as a desirable goal. The \texttt{Driller-agent} plans towards this goal and creates a plan consisting of a single operation: \texttt{setRotation}. The execution of this operation is triggered by the \texttt{OperationRequestMsg} message sent to the \texttt{ControlInterface}.

The \texttt{ControlInterface} then forwards the request to the Low-level agents – lifeline. More precisely, the \texttt{Rotation-agent}, which invokes the external \texttt{SetTdGear} command. The \texttt{TD\_SPEED} – percept confirms that the operation was successfully executed which is reported to the \texttt{ControlInterface} through the \texttt{ActionResultMsg} message. This is forwarded further up the hierarchy to the \texttt{Driller} and from the \texttt{Driller} to the \texttt{Supervisor}. Then circulation is facilitated and we see a similar sequence of messages.

Figure 37 Interaction Diagram: Continually Elevate and Lower the Drillstring.

After circulation is been established and reported back up to the \texttt{Supervisor} and a new goal is sent to the \texttt{Driller}. The new goal is to move the hook to its upper position – to prepare for a process where the drillstring is continually lowered and elevated in the wellbore to prevent stuck pipe. The \texttt{Driller-agent} generates a plan to
achieve the goal, consisting of one operation, gotoMaxHookPosition. An OperationRequestMsg with a reference to this operation is sent to the ControlInterface-agent. This causes it to request the drillstring to be elevated 25 meters (see the HoistActionMsg-message). The ActionResultMsg–message sent from the Hoisting-agent to the ControlInterface-agent describes that the operation was successfully executed. Since the Driller-agent’s plan consisted of a single operation; the result is propagated all the way up to the Supervisor.

The autonomous control system now starts to continually lower and elevate the drillstring to prevent stuck pipe. This is shown in Figure 37. Here the continuity of this process is illustrated by the outer most loop-fragment. In this loop the lowering of the drillstring is a result of the CenterHookPosition-goal and the elevation is the effect of the MoveToUpperHookPosition-goal.

8.2.2 Interaction Protocols

The interaction protocols in Figure 29 show an overview of the system with its main entities and how they are connected. The interaction protocols define the valid interaction sequences for our multi-agent system. Detailed descriptions of the levels of interactions are provided below.

- **Supervisor <> Driller**

  The interaction between the Supervisor and the Driller-agent is described using a single protocol: PlanningGoalCommand. This particular protocol captures the Supervisor providing the Driller with a goal to achieve, and how the Supervisor receives feedback on the goal.

  ![Figure 38 Interaction Protocol: PlanningGoalCommand](image)

  Figure 38 shows the actual message exchanges for the protocol. First we see the Driller-agent receiving a PlanningGoalMsg from the Supervisor, containing a description of a goal. Then, after the Driller has tried to achieve the goal, we can see it replying with a PlanningGoalResultMsg-message stating whether the goal was achieved or not.

- **Driller <> ControlInterface**

  Interaction between the Driller-agent and the ControlInterface-agent occurs for three reasons.

  1. *To provide the Driller with instruction of how to use the ControlInterface*: The ControlInterface sends an OperationSetMsg–message, containing a list of its services (operations) together with expressions (pre- and post conditions) stating what they do and when they can be used to the Driller. This message is sent only once at system start-up.
2. To provide the Driller with an updated overview of the state of the environment: The StateSnapshotRetrival protocol describes how the Driller can on request information about the state of the environment.

![StateSnapshotRetrival Diagram]

Figure 39 Interaction Protocol: StateSnapshotRetrival

The ControlInterface-agent receives a SystemStateRequestMsg message - a request for a snapshot of the environment state. The ControlInterface-agent responds with a SystemStateResponseMsg message containing the ControlInterface’s most updated information about the state of the environment.

3. For the Driller to operate the drilling machinery: The OperationCommand protocol describes how the Driller-agent requests operations to be performed by the ControlInterface-agent.

![OperationCommand Diagram]

Figure 40 Interaction Protocol: OperationCommand

The ControlInterface-agent receives an OperationRequestMsg message from the Driller-agent. This message contains a reference to a ControlInterface-operation the Driller wants to have performed. The ControlInterface-agent responds with a message stating whether the operation was successful.

If we compare the autonomous control system with how the drilling rig is operated today, clear comparisons are drawn. The OperationSetMsg can be associated with the driller reading the manual for the control interface to learn about the functionality it provides through the various joysticks and keypads. The StateSnapshotRetrival protocol is analogous to the driller looking at the information being presented through the monitors provided by the control interface. Further, the OperationCommand protocol is equivalent to the driller using the joysticks and keypads provided by the control interface to control the machinery.

- **ControlInterface <> Low-level agents**

Inter-agent communication between the ControlInterface and the low-level agents, i.e. Hoisting-agent, Rotation-agent, Circulation-agent, Slips-agent and Well-agent occur for the following two reasons.
1. **To enable the ControlInterface to carry out actions on behalf of the Driller:** The *CommandMsg*-protocols (see Figure 29) describe how the ControlInterface-agent sends messages that trigger the low-level agents to take action. Common for all these protocols are that the ControlInterface-agent sends a message associated with a specific action to a low-level agent (these actions are listed in 10.4.1). The low-level agent then tries to execute the task and responds with an ActionResultMsg — message, indicating whether the task was successfully executed or not.

2. **For the ControlInterface to maintain an updated view of its environment:** The MeasurementUpdate —protocol defines the valid interactions for new percepts detected by the low-level agents.

![MeasurementUpdate Diagram](image)

*Figure 41 Interaction Protocol: MeasurementUpdate*

Whenever new sensor data is detected by the low-level agents a MeasurementUpdateMsg containing this data is sent to the ControlInterface-agent. This way the ControlInterface-agent can maintain a complete snapshot of the state of the environment. The sensor data may occur simultaneously and be sent in parallel.
9 Shared Ontology

The Prometheus methodology does not address shared ontologies at all, but as described in section 3.7.1.3, they are necessary for the agents to understand each other. This section describes the terminology shared among the agents in the MAS.

9.1 Shared Ontology

We have excluded definitions of the explicit meaning of the individual messages in the common ontology as this is implicitly defined when constructing messages and interaction protocols in Prometheus. However, some of the messages that we have defined carry information that needs to have its meaning explicitly defined. This chapter is dedicated to describe the vocabulary used for this purpose.

![Figure 42 Levels of the Common Ontology](image)

The layers in our organisational structure operate on different levels of abstraction (see information flow in 7.7.1). This invited us to follow the layers of the organisation structure in the making of the shared ontology (see Figure 42). This process resulted in the following vocabularies.

1. **Installation Specific Measurements** – The concepts related to the specific set of installed equipment. This is the process-data/percepts identified in the system specification phase (see section 7.3). This is conceptually not a part of the common ontology as these definitions are local to the agent encapsulating the external resource.

2. **Generic Measurements** – This vocabulary is an abstraction above the installation specific terminology (more general). This is necessary to enable the higher levels of the organisational structure to reason over process data, without concern to the specific terminology used by the low-level control systems.
3. **State Definitions** – We find it necessary for the system to have a shared understanding of the significant states the environment can be in. This simplifies reasoning in the higher levels of the organisational structure as abstract state definitions are easier to reason over than low level process data.

The vocabulary defining the *installation specific measurements* is described in 7.3, and the *state definitions* are described in section 9.2. The terminology for the *generic measurements* was neither specified nor implemented as we found it insignificant for our demonstrator and therefore categorised as future work.

It should be noted that we have not implemented the definitions described here as formal ontologies in e.g. OWL. The installation specific terminology is a simple dictionary and the state definitions are implemented as a hierarchy of standard Java interface-declarations.

### 9.2 State Definitions

As a part of the shared ontology we identified a set of conditions that are especially relevant for the business logic described in the scenarios. These conditions (or environment states) are used to describe the state of the environment to the *Driller-agent*, and to describe the pre- and post conditions of the services (operations) provided by the *ControlInterface-agent*. The *Supervisor-agent* also uses the same taxonomy when dictating goals to the *Driller-agent*. This way, the *Driller-agent* can use the operation-metadata to determine sequences of operations that ultimately lead to a goal stated by the *Supervisor*.

The state definitions describe the state of the external environment in terms of the following.

- *Bit position*
- *Circulation function*
- *Hook position*
- *Hoisting function*
- *Park break*
- *Rotation function*
- *Slips*

We have included an example to show how the concepts above are described. The example is shown in the section below and describes the significant states for the bit position. We refer to Appendix C for the complete set of state definitions.

### 9.2.1 Sample State Definition: Bit Position

The state definitions taxonomy is implemented as a hierarchy of Java interfaces. We have therefore selected to visualise it using UML class diagrams [39]. Note that only leaf nodes in these diagrams can be instantiated, as the other (more abstract) states only exist to simplify logical expressions in planning.

The significant bit positions (states) are depicted in Figure 43. Recall that the bit position is the vertical position of the drilling bit inside the well.
The diagram shows two states at the top level: The bit can either be in the *Cased section* –state or in *Open hole section* –state. Further, if it is in the *Open hole section* –state it is either *Less than 1 stand from cased section* or *More than 1 stand from cased section*.

The states can briefly be described as follows.

- **Cased section** – This indicates that the bit is above the casing shoe, i.e. in cased section. Based on the scenario descriptions, this is the only state that is significant when above the casing shoe, meaning that the state does not need to be decomposed any further.

- **Open hole section** – This is the case when the bit is below the casing shoe. This implies that the bit is either *Less than 1 stand from cased section* or *More than 1 stand from cased section*.

- **Less than 1 stand from cased section** – This is the situation when the bit is identified in a position less than one stand below casing shoe. This means that the drillstring can be hoisted above the casing shoe. Note that this depends on the hook position and not the stand length.

- **More than 1 stand from cased section** – This is the case when the bit position is identified to be more than 1 stand below the cased section (far in the open hole section).

The states shown above represent the bit positions that are important for scope of this thesis. We refer to Appendix C for descriptions of the states for the rest of the concepts.
10 Detailed Design and Implementation

This chapter addresses the details of the entities specified in the previous chapters. This roughly corresponds to the detailed design and implementation phase in the Prometheus methodology. In this part of the development-process we moved from the PDT -modelling environment to the JACK Java Development Environment. The syntax for the JACK diagram is described in Appendix A2.

10.1 Supervisor

The Supervisor should guide the system according to the overall business logic explained in section 7.6. This should be achieved by communicating goals to the Driller. This requires the activities in the high level business logic to be translated to goals that can be interpreted and understood by the involved agents. This is achieved by mapping the overall business logic to goal statements using terminology from the common ontology. Figure 44 shows a UML 2.x activity diagram of a slightly modified version of the high level business logic. The activities in this diagram are annotated with comments showing the corresponding goal expressed using the state definitions from the common ontology (see Appendix D2 for mapping schema).

![Figure 44: High-Level Business Logic Mapped to Definitions from the Common Ontology](image)

The logic for the Supervisor is simply implemented as a finite state machine. The flow can be described through the following steps.

1. The Supervisor starts the process by sending a PlanningGoalMsg –message to the Driller and waits for it to respond. This first message contains the goal the Driller should try to achieve. For example Hoisting = NotHoisting, i.e. stop the ongoing hoisting operations.
2. The Driller then tries to achieve the goal and responds with a PlanningGoalResultMsg –message with attributes describing whether the goal was achieved.
3. If the PlanningGoalResultMsg -message indicates that the goal was achieved the Supervisor follows the normal flow (black flow-line) to the next goal. However, if the message indicates that the goal was for some reason not achieved, the error path is followed (red flow-line) to an alternative goal.
This process is continued until the final state is reached.

Note that the business logic could have been collapsed to a single goal (e.g. above casing shoe, slips locked and park break activated). The Driller-agent would still be able to find the appropriate steps to achieve the very same goal. However, if the system for some reason fails to achieve this goal, it needs to have some mechanism to decide upon a new, alternative goal. The problem is then to decide upon an appropriate goal for the particular state of the environment as well as taking the operation that failed into account. In the stepwise approach we propose, this problem is avoided by the use of milestones (activities in the diagram) and if a particular milestone cannot be achieved the error graph is simply followed without the need for any complex reasoning.

10.2 Driller

The Driller agent decides upon the sequence of operations to achieve the goals stated by the Supervisor. The actual sequence of operations is determined using an algorithm for automated planning. As JACK does not provide this type of functionality, this was created as an external module that can be directly invoked from JACK-plans. A detailed description of the algorithm is provided in section 10.2.1. The other functional entities that make out the agent are described below.

Data

This agent does not use Beliefset-constructs to persist its data. However, it uses the instance of the external planning-algorithm to store temporary information about the state of the environment as well as the set of operations received by the Driller on start-up. This is further elaborated in the descriptions of the capabilities below.

Capabilities

The JACK code for this agent is structured into a single capability, OperationPlanning. This capability is visualised in Figure 45.

![Figure 45 JACK Capability: OperationPlanning](image)

The blue dotted line outlines the functionality that handles the operations received from the ControlInterface on system start-up. A description of how this works is provided below.
1. The **ControlInterface** sends an OperationSetMsg -message on system start-up. This event contains the complete set of operations it can provide to the **Driller**. Each of the operations has its pre- and post condition described using the state definitions from the common ontology (these are described in detailed later).

2. This OperationSetMsg -message is handled by the ReceiveOperationSet-plan, which inserts the operations into the instance of the planning algorithm.

The PlanAndExecute – plan is triggered by the PlanningGoalMsg –message. This message contains the goal condition which the **Driller** should plan towards. The semantics of this plan is described below.

1. Firstly, the goal is extracted from the PlanningGoalMsg –message and added to the external planning algorithm.

2. Next, a snapshot of the state of the environment is collected from the ControlInterface. This is achieved by sending a SystemStateRequestMsg –message to the ControlInterface and wait for it to reply with a SystemStateResponseMsg – message containing a description of the environment.

3. The description of the state of the environment is added to the external planning algorithm.

4. The planning algorithm is then invoked, which creates a sequence of operations (see section 10.2.1).

5. The output from the planning algorithm is then executed in a sequential manner. For each operation in the plan an OperationRequestMsg –message with a pointer to the operation is sent to the ControlInterface for execution. The Driller then waits for an OperationResultMsg –message which states the result of the operation.

6. Based on the content in the OperationResultMsg –message it determines whether re-planning should be performed. If not, the goal is either achieved or it is marked as not achievable which is communicated to the **Supervisor** through the PlanningGoalResultMsg –message.

10.2.1 The Planning Algorithm

The planning algorithm used by the **Driller** -agent is implemented in Java and based on the forward-chaining planning principle. The basic approach to forward-chained planning can be describes as follows [42].

“...forward-chaining planning can be described as search through a landscape where each node is defined by a tuple \(<S,P>\). S is a world state comprised of predicate facts and P is the plan (a series of ordered actions) used to reach S from the initial state. Search begins from the initial problem state, corresponding to a tuple \(<S_0,\emptyset>\). Edges between pairs of nodes in the search landscape correspond to applying actions to lead from one state to another. When an action A is applied to a search space node \(<S,P>\) the node \(<S',P'>\) is reached, where S' is the result of applying the action A in the state S and P' is determined by appending the action A to P. Forward-chaining search through this landscape is restricted to only considering moves in a forwards direction: transitions are only ever made from a node with plan P to nodes with a plan P' where P' can be determined by adding (or ‘chaining’) actions to the end of P...”

Unguided forward-chaining search is obviously a very expensive search strategy, as it investigates every possible combination of actions to reach from the initial state to the goal state. However, by adding some restrictions, the search space can quickly be reduced to an acceptable level (at least for the limited scope of our demonstrator).
The following precautions were made to reduce the search space in the planning algorithm:

- A plan can consist of maximum 30 operations. This prevents the planning algorithm from creating infinitely long plans.
- Each operation is equipped with pre-conditions, specifying the conditions that must be fulfilled in order for the branch to be searched.
- Search branches are only allowed to achieve the same state once. This prevents loops.
- Operations are not allowed to occur twice in a row within the same branch.

The most significant part of the algorithm is shown in Appendix D3.

It should be noted that there are many off the shelf planners that we could have used for this purpose, e.g. STRIPS [43] implementations. However, the benefits of having our own planning algorithm are the ability to tune it towards our needs and use the relations in the state definitions directly in the algorithm.

10.2.2 A Sample Planning Case

This section provides a detailed description of an example solution synthesis on the planning problem described in Table 6. The left table column describes the initial state (the state of the system when planning is initiated) and the column to the right the goal state; both defined using the state definitions from the common ontology (see chapter 9).

<table>
<thead>
<tr>
<th>Initial environment state</th>
<th>Transformation</th>
<th>Goal state</th>
</tr>
</thead>
<tbody>
<tr>
<td>BitPosition = LessThan1StandFromCasedSection</td>
<td>→</td>
<td>BitPosition = CasedSection</td>
</tr>
<tr>
<td>Hoisting = Hoisting</td>
<td>→</td>
<td>Hoisting = NotHoisting</td>
</tr>
<tr>
<td>Slips = InActive</td>
<td>→</td>
<td>Slips = Locked</td>
</tr>
<tr>
<td>ParkBreak = InActive</td>
<td>→</td>
<td>ParkBreak = parked</td>
</tr>
</tbody>
</table>

The planning algorithm starts with the initial state and searches through operations with matching preconditions. Since the states (vocabulary used to define the preconditions) are ordered in a hierarchy, a state will be accepted if the goal is equivalent to it or its parent (super) states. It will for all applicable operations create a branch where the post-condition of the operation is applied to the initial state. The algorithm will do this recursively until all alternative paths from the initial environment state to the goal state are found.

An example of the sequence of operations for an applicable plan is shown in Table 7. It describes an ordered sequence of operations ending in the goal state described in Table 6. The blue text shows the preconditions of the operation being fulfilled while the purple text shows the outcome of the operation. The semantics of the operations used in this plan are described in section 10.3.1.
<table>
<thead>
<tr>
<th>Operations</th>
<th>State</th>
<th>Operation effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 haltHoisting</td>
<td>BitPosition = LessThan1StandFromCasedSection</td>
<td>BitPosition = LessThan1StandFromCasedSection</td>
</tr>
<tr>
<td></td>
<td>Hoisting = Hoisting</td>
<td>Hoisting = ReadyToHoist</td>
</tr>
<tr>
<td></td>
<td>Slips = InActive</td>
<td>Slips = InActive</td>
</tr>
<tr>
<td></td>
<td>ParkBreak = InActive</td>
<td>ParkBreak = InActive</td>
</tr>
<tr>
<td>2 gotoCasedSection</td>
<td>BitPosition = LessThan1StandFromCasedSection</td>
<td>BitPosition = CasedSection</td>
</tr>
<tr>
<td></td>
<td>Hoisting = ReadyToHoist</td>
<td>Hoisting = ReadyToHoist</td>
</tr>
<tr>
<td></td>
<td>Slips = InActive</td>
<td>Slips = InActive</td>
</tr>
<tr>
<td></td>
<td>ParkBreak = InActive</td>
<td>ParkBreak = InActive</td>
</tr>
<tr>
<td>3 activateSlips</td>
<td>BitPosition = CasedSection</td>
<td>BitPosition = CasedSection</td>
</tr>
<tr>
<td></td>
<td>Hoisting = ReadyToHoist</td>
<td>Hoisting = ReadyToHoist</td>
</tr>
<tr>
<td></td>
<td>Slips = InActive</td>
<td>Slips = Active</td>
</tr>
<tr>
<td></td>
<td>ParkBreak = InActive</td>
<td>ParkBreak = InActive</td>
</tr>
<tr>
<td>4 lockSlips</td>
<td>BitPosition = CasedSection</td>
<td>BitPosition = CasedSection</td>
</tr>
<tr>
<td></td>
<td>Hoisting = ReadyToHoist</td>
<td>Hoisting = ReadyToHoist</td>
</tr>
<tr>
<td></td>
<td>Slips = Active</td>
<td>Slips = Locked</td>
</tr>
<tr>
<td></td>
<td>ParkBreak = InActive</td>
<td>ParkBreak = InActive</td>
</tr>
<tr>
<td>5 activatePB</td>
<td>BitPosition = CasedSection</td>
<td>BitPosition = CasedSection</td>
</tr>
<tr>
<td></td>
<td>Hoisting = ReadyToHoist</td>
<td>Hoisting = ReadyToHoist</td>
</tr>
<tr>
<td></td>
<td>Slips = Locked</td>
<td>Slips = Locked</td>
</tr>
<tr>
<td></td>
<td>ParkBreak = InActive</td>
<td>ParkBreak = Locked</td>
</tr>
</tbody>
</table>

If resulting in alternative plans a cost function selects the most optimal plan. In our prototype this is the plan with the fewest operations.

10.3 ControlInterface

The main goal of the ControlInterface is to provide the Driller with necessary instrumentation to control the drilling rig. This includes providing information related to the state of the drilling rig as well sufficient mechanisms to control it.

Data

Information related to the state of the environment is important when deciding upon the specific instructions to send to low level agents in the execution of a particular operation, but also important when determining an operation’s applicability. The ControlInterface-agent uses two beliefsets to store the necessary information to base these decisions upon (see Figure 46).
The first, SystemSnapshot is used to hold the most recent process data, while the second, SystemStates, is used to store abstract states propagated from the data in SystemSnapshot. The structure of these beliefsets is similar to each other as they both are made up of tuples consisting of a simple key-value pair, e.g. ("BIT_POSITION", "2000 meters"). This is feasible as new type of knowledge does not require the structure of these beliefsets to be altered.

Capabilities

The main functionality of this agent is represented by three capabilities described below (see Figure 47).

- **StateReporting** – Encapsulates functionality that handles requests for a snapshot of the current state of the environment (see Figure 48). This functionality is triggered by SystemStateRequestMsg -message and handled by the HandleSendingOfStateResponse -plan. This plan simply extracts high level states from the SystemStates -beliefset, wraps it into a SystemStateResponseMsg -message and uses the @reply construct to send it back to the consumer of the service.

![Figure 46 JACK Beliefs for ControlInterface](image1)

![Figure 47 JACK Capabilities for ControlInterface](image2)

![Figure 48 JACK Capability: StateReporting](image3)
• **Monitoring** – This capability encapsulates functionality related to handling of process data. This includes low-level measurements from the low-level agents as well as propagation of high-level states. The constructs of this capability is shown in Figure 49.

![Figure 49 JACK Capability: Monitoring](image)

The `MeasurementsUpdateMsg` message is sent from a low-level agent, carrying some low-level process data. This message is handled by the `UpdateSystemSnapshot`, which simply adds the process variable to the `SystemSnapshot` beliefset. The `BitPositionStateChange` and `RotationStateChange` are example of events that monitor the beliefset and get posted when certain beliefs arise in the `SystemSnapshot`. This is achieved using the `#posted when` construct that makes events post themselves when a certain condition is fulfilled. If an event is posted, the associated plan extracts the specific state and inserts it into the `SystemStates` beliefset.

• **OperationExecution** – In addition to providing functionality for sending an `OperationSetMsg` message with the complete set of operations to the `Driller` on system start-up, it encapsulates the services or operations which the `ControlInterface`-agent provides to the `Driller`. This is further elaborated in the next section.

### 10.3.1 Operations/Services

We have implemented a set of high-level operations in the `ControlInterface` – agent which the `Driller`-agent can use to plan the actual sequence of operation to apply. The level of granularity of these operations is a trade-off between computation resources in planning and flexibility. A low level of granularity increases the search space, but it also increases the flexibility of planning as the operations may be combined in new ways.

The high-level operations are described below. The name of the operation is its identifier, the precondition is the conditions that need to be in place in order for the operation to be applied, and the postcondition
describes how it affects the environment. Both the preconditions and postconditions are described using the state definitions from the common ontology described in chapter 9.

Table 8 Operations/services provided by the ControlInterface

<table>
<thead>
<tr>
<th>Operation</th>
<th>Precondition</th>
<th>Postcondition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>haltHoisting</td>
<td>H = Hoisting</td>
<td>H = ReadyToHoist</td>
<td>Stops the vertical movement of the drillstring.</td>
</tr>
<tr>
<td>gotoCasedSection</td>
<td>H = ReadyToHoist, B = Lt1Stand, S = Inactive, P = Inactive</td>
<td>H = ReadyToHoist, B = CasedSection</td>
<td>Hoists the drillstring above the casing shoe</td>
</tr>
<tr>
<td>centerHookPosition</td>
<td>H = ReadyToHoist, S = Inactive, P = Inactive</td>
<td>H = ReadyToHoist, HP = Center</td>
<td>Elevates or lowers the drillstring until 50% of the upper stand is above the drill floor. The hook is then roughly halfway to the top of the derrick.</td>
</tr>
<tr>
<td>gotoMaxHookPosition</td>
<td>H = ReadyToHoist, S = Inactive, P = Inactive</td>
<td>HP = Upper, H = ReadyToHoist</td>
<td>Elevate the drillstring as much as possible.</td>
</tr>
<tr>
<td>haltRotation</td>
<td>R = Rotating</td>
<td>R = ReadyToRotate</td>
<td>Stops the rotation of the drillstring</td>
</tr>
<tr>
<td>setRotation</td>
<td>R = ReadyToRotate, S = Inactive</td>
<td>R = Rotating</td>
<td>Starts rotation of the drillstring</td>
</tr>
<tr>
<td>lockSlips</td>
<td>R = NotRotating, S = InPosition, H = ReadyToHoist, P = Inactive</td>
<td>S = Locked, H = ReadyToHoist</td>
<td>Lowers the drillstring to lock the slips</td>
</tr>
<tr>
<td>releaseSlips</td>
<td>S = Locked, H = ReadyToHoist, P = Inactive</td>
<td>S = InPosition, H = ReadyToHoist</td>
<td>Hoists the drillstring to unlock the slips</td>
</tr>
<tr>
<td>activateSlips</td>
<td>S = Inactive, R = NotRotating, H = NotHoisting</td>
<td>S = InPosition</td>
<td>Wraps the slips around the drillstring</td>
</tr>
<tr>
<td>deactivateSlips</td>
<td>S = Locked, H = ReadyToHoist, P = Inactive</td>
<td>S = Inactive</td>
<td>Moves the slips away from the drillstring</td>
</tr>
<tr>
<td>activatePB</td>
<td>P = Inactive</td>
<td>P = Locked</td>
<td>Activates the park break on the hoisting-machinery</td>
</tr>
<tr>
<td>deactivatePB</td>
<td>P = Locked</td>
<td>P = Inactive</td>
<td>Releases the park break on the hoisting-machinery</td>
</tr>
<tr>
<td>setCirculation</td>
<td>C = NotCirculating</td>
<td>C = Circulating</td>
<td>Starts to circulate</td>
</tr>
<tr>
<td>stopCirculation</td>
<td>C = Circulating</td>
<td>C = NotCirculating</td>
<td>Immediately stops the circulation</td>
</tr>
</tbody>
</table>

It may seem that an operation is tightly coupled to a specific JACK plan. This is not the case as no such constraint exists. These operations are not to be confused with method invocations from procedural programming as agents are autonomous entities and their behaviour are not controlled externally. An operation can therefore be handled by multiple plans where the post-condition is the goal of the overall agenda.
Note that the logical expressions in the pre- and post conditions for our prototype are very simple. However, it is possible to define these expressions using far more powerful constructs. The \texttt{haltHoisting} -operation can for example include a halting function that calculates the realistic length to halt with respect to the state of the environment and use this function to determine the \texttt{bit position} as a precondition. This is not implemented as we have aimed to keep the design as general and easy to understand as possible.

10.4 Low-level Agents

The main task of the low-level agents is to provide a simple, generic interface to the control systems. These agents are therefore tightly coupled to the specific underlying systems installed on the drilling rig. In the section below we present the generic interface to the drilling control systems and in section 10.5 we look into the implementation of one of the low-level agents, the Slips-agent.

10.4.1 Generic Interface to Drilling Machinery

A generic interface to the drilling control systems is necessary to hide heterogeneity of the control systems and to integrate their functionality into the multi-agent system. The functionality of the generic interface is shown in Table 9. It describes the functionality, the specific low-level agent that implements it and the messages that act as triggers. See Appendix D1 for description of the triggers.

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Agent</th>
<th>Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sets the direction and the length to hoisting the drillstring. The distance and direction are specified in the trigger.</td>
<td>Hoisting</td>
<td>HoistActionMsg</td>
</tr>
<tr>
<td>Halts any ongoing hoisting-operation.</td>
<td>Hoisting</td>
<td>HaltHoistingActionMsg</td>
</tr>
<tr>
<td>Activates the park break on the draw-work.</td>
<td>Hoisting</td>
<td>ActivatePBActionMsg</td>
</tr>
<tr>
<td>Releases the park break on the draw-work, enabling it to reel in and out the travelling block.</td>
<td>Hoisting</td>
<td>DeactivatePBActionMsg</td>
</tr>
<tr>
<td>Sets the speed of the mud pumps.</td>
<td>Circulation</td>
<td>SetCirculationActionMsg</td>
</tr>
<tr>
<td>Places the slips around the drillstring. Note that the drillstring must be lowered in order for the Slips to be locked.</td>
<td>Slips</td>
<td>ActivateSlipsActionMsg</td>
</tr>
<tr>
<td>Removes the slips from the drillstring. Note that if the slips is locked, the drillstring must be hoisted in order for this operation to be successfully executed.</td>
<td>Slips</td>
<td>ReleaseSlipsActionMsg</td>
</tr>
<tr>
<td>Sets the direction and rotation of the rotation functionality.</td>
<td>Rotation</td>
<td>SetRotationActionMsg</td>
</tr>
</tbody>
</table>
10.5 Slips-agent

We have included detailed descriptions of some aspects of the Slips-agent as an example of how a low-level agent is implemented. This agent is simply a wrapper for the functionality provided by the external control system that provides a general interface to the Slips-component. The data used by this agent and its capabilities are described below.

Data

This agent contains a single ProcessData-beliefset which contains the agent’s current beliefs about the state of the environment. How this beliefset is used is further elaborated in the descriptions of the capabilities.

Capabilities

The functionality of the low-level agent is represented by two capabilities, HandleMeasurements and SlipsActions (See Figure 50).

![Figure 50: JACK Capabilities for Slips](image)

- The **HandleMeasurements** capability provides the means to capture percepts from the environment and forward any updates to the ControlInterface. This is shown in Figure 51.

![Figure 51: JACK Capability: HandleMeasurements](image)

The flow in this diagram can be described through the following.

1. Percepts from the environment are received through MeasurementMsg-messages.
2. This message is handled by the UpdateBeliefsFromMeasurement-plan which adds the process variable to the ProcessData-beliefset.
3. If the added process variable causes the beliefset to be updated, a BeliefUpdate-event containing the changed variable is posted internally.
4. This event is handled by the SendBeliefUpdateToControlInterface-plan, which sends the updated sensor-value to the ControlInterface through the MeasurementUpdateMsg-message.

- The **SlipsActions** encapsulates functionality to map the external control systems to the generic interface provided by this agent. This is shown in Figure 52.
As the diagram indicates, the ActivateSlipsActionMsg and ReleaseSlipsMsg messages act as triggers for the ExecuteActivateSlips plan and the ExecuteReleaseSlips plan. These plans fulfil the requirements to the generic interface to the drilling machinery (for the slips component). The general idea is to select an applicable plan based on the (trigger) message content and the agent’s current beliefs to achieve the implicit goal brought by the message, and after a plan is execution respond with an ActionResultMsg-message stating whether the goal was achieved.
III. Evaluation
11 Discussion

This chapter discusses aspects related to the proposed multi-agent system while outlining some of the achievements accomplished.

11.1 Architecture and Design

We have made an architecture based on the traditional hierarchical organisational structure and identified agents based on abstractions from the drilling domain. Parts of the human organisational (Supervisor, Driller and ControlInterface) are combined with isomorphic design where physical entities on the rig are represented as agents, i.e. Well, Hoisting, Rotation, Circulation and Slips. Our experiences with this approach have been purely positive. Firstly, it has a pragmatic effect on the development process as the approach brings a natural way of decomposing the problem area into entities with well understood semantics. Secondly, the analogy to real world makes the system easy to understand. We have seen this last effect through various presentations of the prototype and its architecture, although we have not explicitly measured the effect.

However, there are several reasons why abstractions that reflects of the real world organisation may not be a good idea [41]. Both the real world organisation and the MAS are likely to evolve over time, which may cause the analogy to the real world to become inappropriate and the abstractions unfeasible. A potential issue is the fact that the identified organisational rules and roles are probably not universally accepted. Real world organisations are typically not well defined and therefore hard to generalise. Another potential issue is the fact that the real world abstractions resulted in a closed agent-organisation (agents cannot be dynamically added and automatically integrated) consisting of a set of tightly coupled agents. This may become unfeasible as it may be good idea to be able to extend the functionality of the system with new agents without changing the the existing agents or configuration. Despite these issues we believe the pragmatic benefits of the abstractions and the analogy to the real world exceed the disadvantages (at least) for the purpose of the first prototype.

The agent-organisation is based on the well known hierarchical organisational structure where the levels of abstraction and autonomy follow the hierarchy. This simple, but powerful approach is considered as a good alternative to advanced distributed control [44]. In our structure agents high up in the hierarchy decide upon the system’s overall course of action while agents in lower levels decide upon the actual sequence of actions. Despite its advantages, disadvantages are apparent as a result of having the business logic centralised in the higher layers. It makes the system vulnerable, as a single failure in an agent high up in the hierarchy may paralyse the whole system. Arguments purport that this is due to bad design, however, we believe it is necessary as the business rules defined through the scenarios describe how the system should act in various situations. This business logic requires the various components to be coordinated, a requirement achieved with ease in our hierarchical structure. It is possible to distribute this logic over multiple agents but we then face the problem of how to best synchronise the various activities. A distributed approach would also make the business logic harder to follow and the system harder to test.

11.2 Common Ontology

We have described a common ontology for the purpose of our conceptual prototype. This vocabulary is not complete and is not defined as a formal ontology in our prototype. However, it outlines some important areas for further research. Initially, a part of this definition is the generic vocabulary used for communication with underlying control systems. This is feasible as the control systems are typically delivered by multiple vendors,
which introduce potential interoperability challenges when integrated. This issue is partly addressed within an AutoConRig subproject, where the aim is to establish a communication standard for drilling control systems.

The state definitions from the common ontology contain concepts to describe the state of the external environment. The state definitions are basically used to describe the current state of the external environment, the goal-state and search space for problems to be solved through automated planning. The state definitions illustrate the point that a computer interpretable description of the problem domain is required to do any sort of reasoning. We believe that logical descriptions of the domain in form of ontologies can provide the means to define such descriptions in a robust and semantically correct way. We see this as future work.

11.3 Decision Making

Decisions are made on all levels in our hierarchical agent-organisation. The top levels make large grained decisions related to the overall course of the system while agents in lower levels of the hierarchy decide upon the low-level details. Decisions are made as a combination of a static representation of the business logic, automatic planning and BDI.

A general issue with decision making in a dynamic environment is the fact that it is difficult to predict the outcome of an action (e.g. the environment may change in an unpredictable way). As a direct consequence it is hard to plan sequences of operations. The business logic is designed as a sequence of steps (or goals) that lead towards desirable system states. These steps are typically not far (many operations) from each other and long sequence of operations can therefore be excluded. This simplifies planning as the algorithm does not have to consider long sequences of operations and both planning and re-configuration (re-planning) can be performed efficiently. The steps also act as checkpoints, so if a step fails, the next best goal from the previous checkpoint is selected. However, this approach may lead to a suboptimal sequence of actions. The stepwise approach can in some situations result in a (less optimal) longer sequence of actions than the shortest possible path to the goal. However, we have for the purpose of our demonstration, prioritised simplicity and the ability to better handle failure to be of higher priority than the need for highly optimised sequences of operations.

11.3.1 Automated Planning

Automated planning is an important ingredient in our autonomous control system. JACK does not provide this functionality, which required us to manually implement it. Planning is necessary because it is undesirable for the control system to proceed with goals which cannot be achieved; as such efforts would lead to unnecessary waste of resources and might leave the system in an undesirable state. With automated planning we can determine whether an organisational goal can be achieved or not, as well as we can find the optimal path to the goal. In our system, the Driller would generate a set of plans, and if no applicable plans are found, the goal is marked as not achievable. However, if this process results in multiple plans, the optimal plan gets selected with respect to a cost function.

The genuine need for planning in our application area is in contrast to systems like [25] where the appropriate action (adjust a choke on a valve) can be coordinated and executed in isolation. Drilling operations on the other hand typically require sequences of actions tailored to the particular state of the environment. Because of the many possible configurations of the external environment it is hard to predict and design these sequences at design time. This fact makes it hard to determine when an action, plan or function should be applied without planning ahead. We have therefore characterised these sequences of actions as subject to planning.

Despite its advantages, automatic planning brings a number of challenges. Planning may be time-consuming and as a result the environment may change in a way that makes the resulting plan irrelevant. We cannot
entirely solve this problem, but we can limit the likelihood of this to happen. The following precautions were made.

- The descriptions of the environment, goal and operations used by the planner are described using high-level descriptions. These abstract definitions do not change as rapidly as low-level process data (percepts), reducing the probability of the environment changing in a manner that makes a plan outdated.
- The business logic is designed as a sequence of steps (or goals) that lead towards desirable system states. These steps are typically not far (many operations) from each other and long sequence of operations will therefore seldom occur.
- A number of low-level restrictions in the implementation of the algorithm (see 10.2.1).

Despite these precautions we do not advocate the specific algorithm used in the prototype if the number of operations (search space) increases. More efficient search/planning/scheduling techniques exist for this purpose. However, a major problem with many of these algorithms is the fact that they are non-deterministic (i.e. it may conclude with different solutions for the same initial conditions). This is a serious issue that should be carefully considered when selecting algorithms for a production environment.

11.4 Robustness

The layered nature of our MAS provides a structured approach to handle changes in the environment. Low-level details are detected and handled by the bottom layers, while higher layers cope with larger (significant) events. To achieve a high level of flexibility in a dynamic environment we have combined JACK’s implementation of the BDI architecture with automated planning.

BDI theory was adopted by the AI community as an approach to reason over appropriate action in resource-bound environments [30]. The light-weight implementation of BDI in JACK provides an efficient and powerful approach to reason over, and find applicable plans among a predefined plan-set. This is feasible for dynamic environments where agents must constantly revise their beliefs and actions to cope with the changing environment. In our system the power of the BDI paradigm really comes to use during execution of low-level operations as swifter response here is a substantial factor for success.

As discussed in section 11.3.1, the applicability of a task or operation on a drilling rig cannot easily be determined in isolation. To decide upon the best sequence of operations to a goal and to determine whether the goal can be achieved it is necessary to do planning. Based on this assumption, we believe that a hybrid approach is good approach to cope with the challenges in drilling operations. Planning is especially good to address situations unforeseen during the design phase. Consider a situation unexpected at design time, e.g. the slips is locked at x meters below the casing shoe and communication failure occurs. In this situation a conventional computer program would typically not be able to give much assistance, but our autonomous control system would through automated planning be capable of finding the appropriate steps (e.g. decide to unlock the slips and ...) to achieve the goals defined through the business logic. This is related to robustness and the system’s ability to adapt to its environment.

Another concept related to the strength of the system is its ability to recover from failure and re-configure its course of action. We have aimed to achieve this using the following four (basic) techniques, although we have not implemented them all.
1. **BDI reconfiguration** - The ControlInterface and the low-level agents are typical BDI-agents built using JACK constructs. JACK is flexible and how it selects plan(s) for execution can be configured in a number of ways. A typical configuration is to enable the JACK runtime environment to generate a set of applicable plans for an event, and if the event fires, the first plan in the plan-set is selected for execution. However, if this plan fails, JACK RE tries to run an alternative plan from the same set of plans. This process continues until a plan has succeeded or all the plans have failed. This is illustrated in Figure 53.

![Figure 53 Example BDI -Reconfiguration](image)

Since this type of re-configuration does not require any time-consuming reasoning it is especially useful to problems that need a quick solution. This type of situation can for example occur if we have a plan that performs a normal (gear wise) deceleration process. If this plan should fail, an alternative plan to prevent a potential crash could be selected for execution. This could for example be to apply the ‘park break’ on the draw-work. This would be a quick, efficient yet still feasible solution to the problem.

2. **Reconfiguration through planning** - This is especially relevant for the following two purposes.

   a. *Recover from failure* – If for example a specific component gets damaged, the Driller-agent could reconfigure its course of action through re-planning. Operations that use the damaged component can be excluded from planning and an alternative path to a goal can be located (see Figure 54).

![Figure 54 Reconfiguration due to Failure](image)

   This functionality is not implemented in our prototype as it requires a lot of work in identifying errors and how they relate to specific actions. However, the planning algorithm can easily be extended to support this type of functionality and provide business value in form of a higher degree of robustness.

   b. *Re-configuration due to unexpected environment change* – This case shows how our design can handle changes in the environment that result in an operation being inapplicable.
Figure 55 shows a general scenario where the Driller handles an unforeseen change in the environment. A brief description of the steps in this scenario is provided below.

1. The Driller -agent receives a goal from the Supervisor -agent.
2. The Driller -agent generates a plan to achieve the goal and starts the execution of the operations in the plan (sending a pointer to the operations to the ControlInterface -agent).
3. An unforeseen event occurs during the execution of the plan, causing an operation to be no longer applicable.
4. The Driller -agent performs re-planning to cope with the changes in the environment, executes the new plan and successfully achieves the goal.

This illustrates a scenario where the system re-configures itself with a new set of operations with respect to the state of the environment.
3. **Goal reconfiguration** – This last case illustrates how the system autonomously recalculates its course of action due to changes in the environment. A general case is illustrated in Figure 56.

![Diagram](image)

**Figure 56 Recalculation of Goal**

This diagram illustrates a situation where a significant change in the environment causes the *Driller* to fail to achieve a desirable state. The *Driller* reports this fact to the *Supervisor*-agent, which generates an alternative goal that the *Driller* manages to achieve.
12 Experiment

As an essential part of the evaluation of our work we designed and conducted an experiment. How this experiment was designed is described throughout this chapter.

12.1 Approach

To ensure the usability of any piece software, it needs to be tested and evaluated. The limited availability of drilling rigs together with the high day rates make it hard to test our prototype in a real setting. It would also require re-implementations of the low-level agents with respect to the API’s used on the specific test rig. These complications made real-world testing impossible for the time and resources available for this thesis. An alternative and simpler approach to testing is through simulation. The environment of the autonomous control system can be replicated and used as testing ground. For the purposes of this thesis, we utilised a simulator tailored toward our needs.

12.2 Requirements for the Simulated Environment

The goal of the simulated environment was to create a test and evaluation environment for the autonomous control system. The simulator was developed after the following requirements.

- It should simulate the physical rig components and its external environment (e.g. well).
- Emulate the interfaces to the external control systems, and simulate the effect its actions have on the rig components and surroundings.
- Simulate sensors and continually update the multi-agent system with information related to the state of the (simulated) environment. The percepts defined in 7.3 should be used for this purpose.
- Produce a log with sufficient information to determine whether the system produces optimal output with respect to the state of the environment and business logic defined in 7.6.
- The system should be able to reproduce the conditions of the initial scenario descriptions (see section 6.5).

12.2.1 The Drilling Rig Simulator

The simulator is implemented as a JACK agent where the state of the environment is stored in a beliefset. If it is updated, the changes are automatically propagated to the autonomous control system using the percepts defined in section 7.3. The simulator implements the actions defined in section 7.3 and can be used to stimulate the virtual environment. Actions can be triggered by the autonomous control system and may be invoked simultaneously. During long running tasks, the simulator keeps the autonomous control system updated by continually posting percepts stating the progress of the operation (about every 200 ms). This is important when for example performing hoisting operations where the position of the bit is significant for the autonomous control system to find the optimal bit position to apply a specific operation.

A configuration file is used to set the initial settings for the simulator’s virtual environment and after configuring the environment, simulations can be performed by starting the simulator. Communication failure will then be simulated after 5 seconds and the autonomous control system should obtain control. Results from a simulation can either be evaluated by reading generated trace logs or through the use of the debugging facilities provided by JACK.
In addition to these debugging facilities, Stian Aase has made an application that performs real time visualisation of the simulated environment together with some details about the system’s internal state. A screenshot of this application visualising some important process variables is shown in Figure 57.

![Figure 57 Stian Aase’s Visualisation of the Simulated Environment](image)

### 12.3 Experiment Success Criteria

The goal of this experiment is to establish whether our prototype acts properly in case of communication failure. The output from the experiment should be compared and evaluated with respect to the scenario specifications in 7.5. A successful experiment should be compliant with the following criteria:

- Test cases that match a scenario from 7.5 should generate the very same output as the output specified in the scenario.
- Course of action should be determined by the state of the environment.
- Actions should fit the particular state of the environment. By this we mean that the percepts together with the system’s beliefs about the environment should actively be used to decide upon the specific actions, their timing, parameter values, and to monitor the progress of operations.
- The experiment should demonstrate autonomous action in situations not explicitly designed for.

Note that our intention is not to conduct a scientific experiment that provides statistically significant answers, but to demonstrate the applicability of our approach to autonomous control of drilling rigs in a laboratory setting.
12.4 Experiment Setup

The scenarios described in section 7.5 were used to create three separate configurations of the virtual environment. The configurations represent the initial setting of the simulated environment before communication failure is simulated. The configurations for the test-cases are described in the table below where the most significant information is highlighted.

<table>
<thead>
<tr>
<th>Case 1 - Bit above casing shoe</th>
<th>Case 2 - Bit less than 1 stand in open hole section</th>
<th>Case 3 - Bit more than 1 stand in open hole section</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLIPS_STATUS 0</td>
<td>SLIPS_STATUS 0</td>
<td>SLIPS_STATUS 0</td>
</tr>
<tr>
<td>MUD_SETPOINT 0</td>
<td>MUD_SETPOINT 0</td>
<td>MUD_SETPOINT 0</td>
</tr>
<tr>
<td>TD_STATUS 1</td>
<td>TD_STATUS 1</td>
<td>TD_STATUS 1</td>
</tr>
<tr>
<td>TD_GEAR 0</td>
<td>TD_GEAR 0</td>
<td>TD_GEAR 0</td>
</tr>
<tr>
<td>TD_SPEED 0</td>
<td>TD_SPEED 0</td>
<td>TD_SPEED 0</td>
</tr>
<tr>
<td>DW_SPEED 5</td>
<td>DW_SPEED 5</td>
<td>DW_SPEED 5</td>
</tr>
<tr>
<td>DW_GEAR 2</td>
<td>DW_GEAR 2</td>
<td>DW_GEAR 2</td>
</tr>
<tr>
<td>DW_GEAR_MODE 2</td>
<td>DW_GEAR_MODE 2</td>
<td>DW_GEAR_MODE 2</td>
</tr>
<tr>
<td>DW_GEAR_DIRECTION 0</td>
<td>HOOK_LOAD 50</td>
<td>HOOK_LOAD 50</td>
</tr>
<tr>
<td>HOOK_LOAD 50</td>
<td>HOOK_LOAD 50</td>
<td>HOOK_LOAD 50</td>
</tr>
<tr>
<td>MAX_HOOK_POSITION 20</td>
<td>MAX_HOOK_POSITION 30</td>
<td>MAX_HOOK_POSITION 30</td>
</tr>
<tr>
<td>DS_TOTAL_WEIGHT 50</td>
<td>DS_TOTAL_WEIGHT 50</td>
<td>DS_TOTAL_WEIGHT 50</td>
</tr>
<tr>
<td>UPPER_STAND_LENGTH 30</td>
<td>UPPER_STAND_LENGTH 30</td>
<td>UPPER_STAND_LENGTH 30</td>
</tr>
<tr>
<td>PB_STATUS 0</td>
<td>PB_STATUS 0</td>
<td>PB_STATUS 0</td>
</tr>
<tr>
<td>BIT_POSITION 1180</td>
<td>BIT_POSITION 1205</td>
<td>BIT_POSITION 1300</td>
</tr>
<tr>
<td>TOTAL_DEPTH 1400</td>
<td>TOTAL_DEPTH 1400</td>
<td>TOTAL_DEPTH 1400</td>
</tr>
<tr>
<td>CASINGSHOE_DEPTH 1200</td>
<td>CASINGSHOE_DEPTH 1200</td>
<td>CASINGSHOE_DEPTH 1200</td>
</tr>
</tbody>
</table>

- **Case 1 - Bit above casing shoe** - The drillstring travels downwards in the well using the highest gear on the draw-work. The bit is above the casing shoe with a clear margin when communication failure is detected. The autonomous control system should then take control over the drilling rig and take action to stop the vertical movement of the drillstring. It should then find the appropriate actions to lock the slips and apply the park break on the draw-work.

- **Case 2 - Bit less than 1 stand in open hole section** - The drillstring is lowered into the wellbore using the draw-work’s highest gear. Communication failure occurs just after passing casing shoe. The autonomous control system should then take control over the drilling rig and take action to stop the vertical movement of the drillstring. When halted, action should be taken to hoist the drillstring to the cased section and then lock the slips and apply the park break.

- **Case 3 - Bit more than 1 stand in open hole section** - Also in this scenario the drillstring is lowered into the wellbore using the draw-work’s highest gear. Communication failure occurs when the bit is far in the open hole portion of the well. The autonomous control system should then take control over the drilling rig and take action to stop the vertical movement of the drillstring. When accomplished, a process where the drillstring is continually oscillated should be started.

A signal stating communication failure is then sent to the autonomous control system to trigger autonomous control of the drilling rig. The simulator generates trace-logs with the input/output from the control system for later analysis.
13 Experiment Results

In this chapter we present the results of the experiment described in Chapter 12. We also describe the result of some auxiliary test-cases that illustrate some feasible aspects of the prototype. After having described the results from the experiment, we comment on the internal and external validity of the experiment and draw our conclusions.

13.1 Results Explained

The trace logs generated by the simulator were used to evaluate the outcome of the experiment. The significant process variables are presented through graphs where the horizontal axis represents time in milliseconds from simulator start-up and the vertical axis the scale used to measure it. The large “dots” reflect actual process data that are sent to the autonomous control system and the line connecting the dots are inserted to simplify reading. (Note that these lines may be misleading in some diagrams as they simply connect the dots). The red dotted lines denote where an action was invoked and the associated label specifies the time and the specific action including parameters.

The results are presented below.

- **Results: Case 1 – Bit above casing shoe**

Table 11 shows the above casing shoe scenario - scenario specification from chapter 7.5 compared with the output from case 1. The column to the right shows the output from the conceptual prototype and the scenario-specification is shown in the left column.

<table>
<thead>
<tr>
<th>Scenario steps</th>
<th>Applied actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication failure scenario*</td>
<td>... ...</td>
</tr>
<tr>
<td>1.21 Action</td>
<td>setDWGear(0,2,1) ... setDWGear(0,0,0) = 6707 setDWGear(0,2,0) 8277 setDWGear(0,1,2) 8882 setDWGear(0,1,1) 9282 setDWGear(0,1,0) 9485 setDWGear(0,0,0) 9691 setDWGear(0,0,0)</td>
</tr>
<tr>
<td>1.22 Percept</td>
<td>DW_GEAR = ..</td>
</tr>
<tr>
<td>1.23 Percept</td>
<td>DW_GEAR_MODE = 2 .. 0</td>
</tr>
<tr>
<td>1.24 Percept</td>
<td>DW_SPEED = 5 .. 0</td>
</tr>
<tr>
<td>...Continuing with “Above casing shoe scenario”</td>
<td>... ...</td>
</tr>
<tr>
<td>2.1 Percept</td>
<td>BIT_POSITION = 1195</td>
</tr>
<tr>
<td>2.2 Percept</td>
<td>HOOK_POSITION = 15</td>
</tr>
<tr>
<td>2.3 Action</td>
<td>activateSlips() = 9903 activateSlips()</td>
</tr>
</tbody>
</table>
As we can see from the table above, the prototype generated a correct sequence of actions with respect to the specification. We can see that the speed of the draw-work is gradually reduced, and actions to activate and lock the slips are initiated. However, it shows little about how they relate to the state of the environment. The important process variables are therefore presented through the graphs shown in Figure 58. The graphs show how the environment changes over time and the effect of the actions.
The timeline for this test run is described below.

- **0 – 5100 ms**: The simulator starts up, receives the initial configuration for the test-case and sleeps 5000 ms while waiting for the system to be initialised before “communication failure” is simulated.

- **5101 – 9901 ms**: The autonomous control system calculates the required length to stop the vertical movement of the drillstring. Due to a safety margin included in this calculation, the first action \((\text{setDWGear}(0,2,1))\) occurs first at 6707 ms from system start-up. This action causes the speed of the draw-work to be reduced to the speed of the gear, and as we can see from
graphs A in Figure 58 - it waits until the speed is reduced until a new gear change is performed. There is a small safety margin for each new gear and some latency involved (message exchange etc.) in a gear switch. This explains the uneven areas in the graph. The process includes 6 gear changes where the initial draw-work speed is gradually reduced from 5 m/s to 0 m/s. This gear down process is optimal with respect to the equipment.

The details of the results of the actions from 9903 – 10000 ms are hidden in the graphs shown in Figure 58. Figure 59 therefore includes some additional graphs that better show this particular portion of the case.

Figure 59 Experiment, Case 1: Parking of the Drillstring

A)

B)

C)
- **9903 –11703 ms:** The `activateSlips` action is invoked at 9903 ms after system start-up and causes the slips to be moved into position (wrapped around the drillstring). In the simulated environment this operation is completed within ca. 1000 ms. The `setDWGear(0,1,0)` is then invoked at 10912 ms, causing the drillstring to be lowered in its lowest gear while the slips increases its grip around the drillstring. From graph A in Figure 59 we can see the speed of the draw-work increases from 0 to 0,03 m/s between 10912 ms and 11480 ms, but as the slips grips the drillstring it affects the speed (the speed decreases). In graph B, we see that the lowering of the drillstring into the slips causes the bit to move slightly downwards, and graph C shows the hookload dropping from 40 to 0 tons. This indicates that the total weight of the drillstring (40 tons) being transferred to the slips component. The `setDWGear(0,0,0)` action invoked at 11684 stops the draw-work, leaving the slips “locked”.

- **11704 – 1200 ms:** The park break is activated on the draw-work component and the hoisting components of the rig are now secured.

**Result: Case 2 - Bit less than 1 stand in open hole section**

Using the pre-conditions from case 2 the following sequences of actions were generated from our prototype.

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>6672</td>
<td><code>setDWGear(0,2,1)</code></td>
</tr>
<tr>
<td>8292</td>
<td><code>setDWGear(0,2,0)</code></td>
</tr>
<tr>
<td>8885</td>
<td><code>setDWGear(0,1,2)</code></td>
</tr>
<tr>
<td>9285</td>
<td><code>setDWGear(0,1,1)</code></td>
</tr>
<tr>
<td>9487</td>
<td><code>setDWGear(0,1,0)</code></td>
</tr>
<tr>
<td>9688</td>
<td><code>setDWGear(0,0,0)</code></td>
</tr>
<tr>
<td>9906</td>
<td><code>setDWGear(1,1,0)</code></td>
</tr>
<tr>
<td>10485</td>
<td><code>setDWGear(1,1,1)</code></td>
</tr>
<tr>
<td>11085</td>
<td><code>setDWGear(1,1,2)</code></td>
</tr>
<tr>
<td>51515</td>
<td><code>setDWGear(1,1,1)</code></td>
</tr>
<tr>
<td>51915</td>
<td><code>setDWGear(1,1,0)</code></td>
</tr>
<tr>
<td>52318</td>
<td><code>setDWGear(1,0,0)</code></td>
</tr>
<tr>
<td>52557</td>
<td><code>activateSlips()</code></td>
</tr>
<tr>
<td>53575</td>
<td><code>setDWGear(0,1,0)</code></td>
</tr>
<tr>
<td>54331</td>
<td><code>setDWGear(0,0,0)</code></td>
</tr>
<tr>
<td>54348</td>
<td><code>activatePB()</code></td>
</tr>
</tbody>
</table>

Table 12 Experiment, Case 2: Actions in Response to Case 2

Also this sequence of actions is equivalent with the scenario specification from 7.5. First actions are applied to halt the downward motion of the drillstring. Then action is taken to hoist the drillstring above the casing shoe. Actions to lock the slips and apply the park break are then taken. How these actions relate to the state of the environment is shown in the graphs in Figure 60.
The timeline is commented on below.

- **0 – 5100 ms**: The simulator starts up, receives the initial configuration for the scenario and sleeps 5000 ms while waiting for the system to be initialised before “communication failure” is simulated.

- **5101 – 9871 ms**: Similarly to case 1, the autonomous control system calculates and performs an optimal deceleration process to stop the vertical movement of the drillstring (see case 1 for a more detailed description of the deceleration process). At 9871 ms from start-up the system has managed to reduce the speed of the draw-work to 0 m/s.

Due to the large time-interval used in the graphs above, the details of some parts of the process are hidden. We therefore provide two additional sets of graphs that show these particular areas in more detail. Figure 61 shows how the system decreases the initial downward motion of the drillstring and the acceleration when hoisting the drillstring above the casing shoe. Figure 62 shows the deceleration of the hoisting operation and the parking of the drillstring.
- **9872 – 51499 ms**: From graph A in Figure 61 we can see the setDWGear(1,1,0) - action occurring at 9906 ms causing the speed of the drawwork to increase from 1071 ms. This is the process of hoisting the drillstring until the bit is above the casing shoe. From graph B in the same figure we can see how this affects the bit position. The gear is sequentially increased until the required length to accelerate to a higher gear plus the length needed to halt (including a safety margin) is greater than the remaining length to hoist. The acceleration process continues until 12684 ms where the gear change that occurred at 11085 ms has taken effect. The very same gear is used until the deceleration is initiated at 51515ms. From Figure 60 – B we can see that the bit passed the casing shoe at 50901 ms and the deceleration process starting.
- **51515 -52511 ms**: The deceleration process actually starts when there is not enough room between the current position and the goal position to accelerate to a higher gear and still be able to stop. It then waits for the optimal position to start gearing down. From Figure 62 B we can see that position seems to be at 1199,698 meters where the first gear change in the deceleration process occurs. This process continues until the draw-work speed is reduced to 0 m/s at ca. 52511 ms.
- **5212 – 54500 ms**: Within this envelope the drillstring is put into parking mode. This is equivalent to the parking of the drillstring in case 1. The process is initiated by the `activateSlips()` command at 52557 ms, causing the slips to be wrapped around the drillstring. The drillstring is then lowered until the hookload drops to 0 tons, indicating that the weight of the drillstring is transferred to the slips.

- **54348 – 54500 ms**: The `activatePB()` action executed at 54348 ms activates the park break on the draw-work and the hoisting mechanism is secured.

- **Result: Case 3 - Bit more than 1 stand in open hole section**

  Using the pre-conditions from case 3 the following sequence of actions were generated from our prototype.

<table>
<thead>
<tr>
<th>Time</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>6929</td>
<td><code>setdwgear(0,2,1)</code></td>
</tr>
<tr>
<td>8522</td>
<td><code>setdwgear(0,2,0)</code></td>
</tr>
<tr>
<td>9127</td>
<td><code>setdwgear(0,1,2)</code></td>
</tr>
<tr>
<td>9536</td>
<td><code>setdwgear(0,1,1)</code></td>
</tr>
<tr>
<td>9741</td>
<td><code>setdwgear(0,1,0)</code></td>
</tr>
<tr>
<td>9935</td>
<td><code>setdwgear(0,0,0)</code></td>
</tr>
<tr>
<td>10895</td>
<td><code>setTDSpeed(1,100.0)</code></td>
</tr>
<tr>
<td>11939</td>
<td><code>setMudCirculation(50.0)</code></td>
</tr>
<tr>
<td>13159</td>
<td><code>setdwgear(1,1,0)</code></td>
</tr>
<tr>
<td>14708</td>
<td><code>setdwgear(1,1,1)</code></td>
</tr>
<tr>
<td>15309</td>
<td><code>setdwgear(1,1,2)</code></td>
</tr>
<tr>
<td>64558</td>
<td><code>setdwgear(1,1,1)</code></td>
</tr>
<tr>
<td>64957</td>
<td><code>setdwgear(1,1,0)</code></td>
</tr>
<tr>
<td>65358</td>
<td><code>setdwgear(1,0,0)</code></td>
</tr>
<tr>
<td>66073</td>
<td><code>setdwgear(0,1,0)</code></td>
</tr>
<tr>
<td>67674</td>
<td><code>setdwgear(0,1,1)</code></td>
</tr>
<tr>
<td>68274</td>
<td><code>setdwgear(0,1,2)</code></td>
</tr>
<tr>
<td>98119</td>
<td><code>setdwgear(0,1,1)</code></td>
</tr>
<tr>
<td>98519</td>
<td><code>setdwgear(0,1,0)</code></td>
</tr>
<tr>
<td>98920</td>
<td><code>setdwgear(0,0,0)</code></td>
</tr>
<tr>
<td>99638</td>
<td><code>setdwgear(1,1,0)</code></td>
</tr>
<tr>
<td>101243</td>
<td><code>setdwgear(1,1,1)</code></td>
</tr>
<tr>
<td>101852</td>
<td><code>setdwgear(1,1,2)</code></td>
</tr>
<tr>
<td>131666</td>
<td><code>setdwgear(1,1,1)</code></td>
</tr>
<tr>
<td>132066</td>
<td><code>setdwgear(1,1,0)</code></td>
</tr>
<tr>
<td>132466</td>
<td><code>setdwgear(1,0,0)</code></td>
</tr>
<tr>
<td>133246</td>
<td><code>setdwgear(0,1,0)</code></td>
</tr>
<tr>
<td>134696</td>
<td><code>setdwgear(0,1,1)</code></td>
</tr>
<tr>
<td>135297</td>
<td><code>setdwgear(0,1,2)</code></td>
</tr>
<tr>
<td>165183</td>
<td><code>setdwgear(0,1,1)</code></td>
</tr>
<tr>
<td>165583</td>
<td><code>setdwgear(0,1,0)</code></td>
</tr>
<tr>
<td>165986</td>
<td><code>setdwgear(0,0,0)</code></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 13 Experiment, case 3: Sequence of Actions

From the actions listed in Table 13, we can see the drillstring is being continually oscillated. The output from this case is also compliant with the specification. Figure 63 shows the first cycles of this experiment and how the actions affect the simulated environment.
0 – 5100 ms: The simulator starts up, receives the initial configuration for the scenario and sleeps 5000 ms while waiting for the system to be initialised before “communication failure” is simulated.

5100 – 10125 ms: The downward movement of the drillstring is reduced using an optimal deceleration process. At 10125 ms the speed is reduced to 0, ending with the bit at 1315,70 meters (see graph B in Figure 63). See case 1 for a more in-depth description of the deceleration process.

10895 ms: The setTDSpeed command causes the drillstring to rotate.

11939 ms: The circulation system is activated.

13159 – 65555 ms: In this time interval the drill bit is hoisted to a position where almost the whole upper stand is above the drill floor. In this process the bit position changes from 1315,70
to 1291.03 (about 1 meter for the uppermost position of the hook). From graph A in Figure 63 we can see that the acceleration and deceleration process is performed with an optimal curve.

- **66073 – 99118**: The drillstring is lowered 15 meters to a position where about 50% of the upper stand of the drillstring is above the drill floor.

- **100838 – N**: The drillstring is continually hoisted 15 meters up to its upper position and then lowered 15 meters. This oscillation process continues until the communication link to the control centre is re-established.

### 13.2 Auxiliary Test-cases

The test-cases above indicate correct behaviour with respect to the predefined scenarios. However, it may be argued that far simpler approaches could have been taken to achieve the same results. We have therefore added some additional test cases that illustrate more autonomy. More precisely, they show situations not explicitly designed for. An overview of the configurations of these tests is shown in Table 14.

<table>
<thead>
<tr>
<th>In open hole</th>
<th>Slips in position</th>
<th>Slips Locked</th>
<th>Park break activated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 4.1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Case 4.2</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Case 4.3</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Case 4.4</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Note that the simulator also here waits 5 seconds before communication failure is simulated. A summary of the configuration of the test cases and the outcome of the tests are provided below.

- **Case 4.1**

The configuration used for this test-case and the actions it resulted in are shown in Table 15. The actions are annotated with a timestamp showing milliseconds from simulator start-up. The configuration of the case describes a particular situation where the bit stands still 5 meters from bottom hole. This together with the hook position limits how much the drillstring can be hoisted or lowered. More precisely, the hook position only allows the drillstring to be hoisted 5 meters (MAX_HOOK_POSITION - HOOK_POSITION), and the bit position indicates that the drillstring can only be lowered 5 meters (TOTAL_DEPTH - BIT_POSITION). In addition to this, the slips is activated (wrapped around the drillstring, but not locked) and the park break is activated.

The case is optimally handled if the system takes action to deactivate the slips and the park break before it starts to oscillate the drillstring. Note that we have not explicitly designed for this case and the control system must therefore autonomously find the appropriate sequence of actions.
Table 15 Experiment: Case 4.1 – Configuration and Actions

<table>
<thead>
<tr>
<th>Initial configuration</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLIPS_STATUS 1</td>
<td>6557  deactivateSlips()</td>
</tr>
<tr>
<td>MUD_SETPOINT 0</td>
<td>7568  setTDSpeed(1,100.0)</td>
</tr>
<tr>
<td>TD_STATUS 1</td>
<td>8666  setMudCirculation(50.0)</td>
</tr>
<tr>
<td>TD_GEAR 0</td>
<td>10254 deactivatePB()</td>
</tr>
<tr>
<td>TD_SPEED 0</td>
<td>11269 setDWGear(1,1,0)</td>
</tr>
<tr>
<td>DW_SPEED 0</td>
<td>12839 setDWGear(1,1,1)</td>
</tr>
<tr>
<td>DW_GEAR 0</td>
<td>13411 setDWGear(1,1,2)</td>
</tr>
<tr>
<td>DW_GEAR_MODE 0</td>
<td>21218 setDWGear(1,1,1)</td>
</tr>
<tr>
<td>DW_GEAR_DIRECTION 0</td>
<td>21629 setDWGear(1,1,0)</td>
</tr>
<tr>
<td>HOOK_LOAD 100</td>
<td>22025 setDWGear(1,0,0)</td>
</tr>
<tr>
<td>HOOK_POSITION 25</td>
<td>22731 setDWGear(0,1,0)</td>
</tr>
<tr>
<td>MAX_HOOK_POSITION 30</td>
<td>24324 setDWGear(0,1,1)</td>
</tr>
<tr>
<td>DS_TOTAL_WEIGHT 100</td>
<td>24942 setDWGear(0,1,2)</td>
</tr>
<tr>
<td>UPPER_STAND_LENGTH 29</td>
<td>41767 setDWGear(0,1,1)</td>
</tr>
<tr>
<td>PB_STATUS 1</td>
<td>41967 setDWGear(0,1,0)</td>
</tr>
<tr>
<td>ELEVATOR_STATUS 0</td>
<td>42376 setDWGear(0,0,0)</td>
</tr>
<tr>
<td>BIT_POSITION 1495</td>
<td>43046 setDWGear(1,1,0)</td>
</tr>
<tr>
<td>TOTAL_DEPTH 1500</td>
<td>44652 setDWGear(1,1,1)</td>
</tr>
<tr>
<td>CASINGSHOE_DEPTH 1000</td>
<td>45248 setDWGear(1,1,2)</td>
</tr>
<tr>
<td></td>
<td>62058 setDWGear(1,1,1)</td>
</tr>
<tr>
<td></td>
<td>62261 setDWGear(1,1,0)</td>
</tr>
<tr>
<td></td>
<td>62660 setDWGear(1,0,0)</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

The actions from Table 15 shows that the appropriate steps were taken by the autonomous control system in the test-run. Action to deactivate slips, facilitate rotation of the drillstring, to start the mud-circulation and to deactivate the park break are taken before the oscillation of the drillstring is started. This is a perfectly valid sequence of actions.

Figure 64 Experiment, Case 4.1: Bit position during oscillation

The bit position for this case are shown in Figure 64 and shows that the autonomous control system adapts to the current situation and does not move the bit below what it believes to be the bottom hole position (including a safety margin).
• **Case 4.2**

The configuration used for this test-case and the resulting sequence of actions are shown in Table 16. The actions are annotated with a timestamp showing milliseconds from simulator start-up. The configuration describes a situation where the slips is locked while the bit is in open hole portion of the wellbore. However, the bit is located in a position that enables it to be hoisted above the casing shoe (the bit is 5 meters below the casing shoe).

This scenario is optimally handled if the control system autonomously unlocks and deactivates the slips before the drillstring is hoisted above the casing shoe and parked the same way as in scenario 2 (see section 6.5.2).

<table>
<thead>
<tr>
<th>Initial configuration</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SLIPS_STATUS 1</strong></td>
<td>6085 setDWGear(1,1,0)</td>
</tr>
<tr>
<td>MUD_SETPOINT 0</td>
<td>7305 setDWGear(1,0,0)</td>
</tr>
<tr>
<td>TD_STATUS 1</td>
<td>7315 deactivatSlips()</td>
</tr>
<tr>
<td>TD_GEAR 0</td>
<td>8325 setDWGear(1,1,0)</td>
</tr>
<tr>
<td>TD_SPEED 0</td>
<td>9890 setDWGear(1,1,1)</td>
</tr>
<tr>
<td>DW_SPEED 0</td>
<td>10486 setDWGear(1,1,2)</td>
</tr>
<tr>
<td>DW_GEAR 0</td>
<td>21311 setDWGear(1,1,1)</td>
</tr>
<tr>
<td>DW_GEAR_MODE 0</td>
<td>21712 setDWGear(1,1,0)</td>
</tr>
<tr>
<td>DW_GEAR_DIRECTION 0</td>
<td>22122 setDWGear(1,0,0)</td>
</tr>
<tr>
<td>HOOK_LOAD 0</td>
<td>23962 lockSlips()</td>
</tr>
<tr>
<td><strong>HOOK_POSITION 20</strong></td>
<td>24968 setDWGear(0,1,0)</td>
</tr>
<tr>
<td>MAX_HOOK_POSITION 30</td>
<td>26967 setDWGear(0,0,0)</td>
</tr>
<tr>
<td>DS_TOTAL_WEIGHT 100</td>
<td>27240 activatePB()</td>
</tr>
<tr>
<td>UPPER_STAND_LENGTH 29</td>
<td></td>
</tr>
<tr>
<td>PB_STATUS 0</td>
<td></td>
</tr>
<tr>
<td>ELEVATOR_STATUS 0</td>
<td></td>
</tr>
<tr>
<td><strong>BIT_POSITION 905</strong></td>
<td></td>
</tr>
<tr>
<td>TOTAL_DEPTH 925</td>
<td></td>
</tr>
<tr>
<td><strong>CASINGSHOE_DEPTH 900</strong></td>
<td></td>
</tr>
</tbody>
</table>

The actions shown in Table 16 indicate correct handling of the case. The actions applied at 6085 ms and 7305 ms show that the slips is unlocked, and the action occurring at 7215 ms completely removes the slips from the drillstring. The drillstring is then free to be hoisted, and from this point the remaining sequence of actions are equivalent to the actions specified in *Less than one stand from the casing shoe* -scenario in 7.5.

• **Case 4.3**

The configuration used for this test-case and the resulting sequence of actions are shown in Table 17. The actions are annotated with a timestamp showing milliseconds from simulator start-up. The configuration of this case describes a situation where the bit is above the casing shoe and the slips is activated (wrapped around the drillstring, but not locked).

Since, in this case we are above the casing shoe and the parking process is already started, optimal course of action would be to lower the drillstring until the slips is locked and when completed, activate the park break.
The sequence of actions applied by the autonomous control system (see Table 17) show that the autonomous control system found and executed the appropriate actions; the drillstring is lowered until the slips is locked and action is then taken to apply the park break.

- **Case 4.4**

  The configuration for this test-case and the resulting sequence of actions are shown in Table 18. The actions are annotated with a timestamp showing milliseconds from simulator start-up. The configuration describes a situation where the drilling rig is already in state that we have characterised as optimal (see the initial scenarios). The optimal solution for this scenario is if no action is taken by the autonomous control system.

Table 18 Experiment: Case 4.4 – Configuration and Actions

<table>
<thead>
<tr>
<th>Initial configuration</th>
<th>Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SLIPS_STATUS 1</td>
<td>6246 setDWGear(0,1,0)</td>
</tr>
<tr>
<td>MUD_SETPOINT 0</td>
<td>8235 setDWGear(0,0,0)</td>
</tr>
<tr>
<td>TD_STATUS 1</td>
<td>8451 activatePB()</td>
</tr>
<tr>
<td>TD_GEAR 0</td>
<td></td>
</tr>
<tr>
<td>TD_SPEED 0</td>
<td></td>
</tr>
<tr>
<td>DW_SPEED 0</td>
<td></td>
</tr>
<tr>
<td>DW_GEAR 0</td>
<td></td>
</tr>
<tr>
<td>DW_GEAR_MODE 0</td>
<td></td>
</tr>
<tr>
<td>DW_GEAR_DIRECTION 0</td>
<td></td>
</tr>
<tr>
<td>HOOK_LOAD 100</td>
<td></td>
</tr>
<tr>
<td>HOOK_POSITION 20</td>
<td></td>
</tr>
<tr>
<td>MAX_HOOK_POSITION 30</td>
<td></td>
</tr>
<tr>
<td>DS_TOTAL_WEIGHT 100</td>
<td></td>
</tr>
<tr>
<td>UPPER_STAND_LENGTH 29</td>
<td></td>
</tr>
<tr>
<td>PB_STATUS 0</td>
<td></td>
</tr>
<tr>
<td>ELEVATOR_STATUS 0</td>
<td></td>
</tr>
<tr>
<td>BIT_POSITION 800</td>
<td></td>
</tr>
<tr>
<td>TOTAL_DEPTH 1500</td>
<td></td>
</tr>
<tr>
<td>CASINGSHOE_DEPTH 1000</td>
<td></td>
</tr>
</tbody>
</table>

As shown in Table 18, the autonomous system performs optimally and does not take any action as the rig is already secured.
13.3 Validity Threats

There are multiple validity threats related to how our experiment was conducted. The most important one may be the fact that we conducted the experiment in a simplified simulated environment. The simulated environment is deterministic as the actions always respond in the correct, predefined way, which may not be the case in a real world setting. Each test was only run once and configured a specific way which may also be a threat to the experiment’s validity.

Also the assumptions we have made with respect to technical details of the drilling domain and information related to the external systems are a threat against the external validity of this experiment. However, the intention of this experiment is not to give theoretic evidence stating that this applies to the real world, but to demonstrate our work in a laboratory setting.

13.4 Experiment Summarised

The results from the experiment may be summarised by the following points:

- The autonomous control system is compliant with the specification (the scenarios in 7.5) in respect to actions selected. In case 1, 2 and 3 the sequences of actions where identical to the scenario specification.

- Actions are synchronised with the state of the environment. Examples of this include;
  - The deceleration process in case 1 (see graph A in Figure 58), 2 (see graph A in Figure 61) and 3 (see graph A in Figure 63), as the gear is continually decreased when the speed of the gear is achieved.
  - The process of locking the slips in case 1 (see Figure 59), 2 (see Figure 62), 4.2, 4.3 and 4.4 where the drillstring being lowered into the slips while the system carefully waits for the hook-load to drop to 0 until the draw-work is stopped.
  - The acceleration process of the hoisting operations in case 2, 3, 4.1 and 4.2 where the gear is gradually increased when the speed catches up with the gears (until it is not feasible to increase the gear with respect to an optimal deceleration process).
  - During the hoisting operations in case 2, 3, 4.1 and 4.2, the deceleration process is carefully timed and coordinated with the state of the environment to ensure accuracy and optimal execution.
  - The oscillation process described in case 4.1 adapts to the environment and only oscillates within the available space.

- A high level of autonomy has been demonstrated as the system plans and correctly executes sequences of actions for situations unforeseen at design time. This is shown in case 4.1, 4.2, 4.3 and 4.4 where the system autonomously finds the appropriate action to achieve its goals.

- Both the course of action and the specific action sequences have been optimal with respect to the specification and the state of the environment. We can see this from all the test cases, but especially case 4.4 where it takes no unnecessary actions as it detects that its business goals are already achieved.

The result from the experiment shows that the autonomous control system handled all of the test-cases successfully and we have fulfilled the experiment’s success criteria.

Although the experiment is conducted in a laboratory setting, it gives an indication on the applicability of multi-agent technology within drilling. We believe that we have with this experiment demonstrated that the proposed architecture and design are an approach that can contribute to the realisation of autonomous control of drilling rigs.
14 Conclusion and Future Work

This chapter describes what we have achieved through our work and areas for further research.

14.1 Conclusion

Throughout this thesis we have presented one approach to autonomous control of drilling rigs using multi-agent technology. We have built a software prototype and through an experiment demonstrated autonomous control of a simulated drilling rig.

From this we can conclude that this first step towards autonomous control of drilling rigs has been successful. However, it is obvious that there is much work to be done if we are to realise the vision of autonomous drilling rigs. Bearing this in mind; we believe that we have gained a great deal of knowledge through our research and achieved some important milestones in the pursuit of the ultimate goal of unmanned subsea drilling rigs.

14.2 Achievements

Our achievements can be summarised through the following.

- Hierarchical decomposition of the domain is identified as a simple and efficient approach to coordinate and distribute activities on a drilling rig.
- Identified powerful abstractions from the real world organisation that make the multi-agent system easy to understand and present.
- Demonstrated autonomous control of the drilling rig in a simulated environment. The results from the experiment show that the autonomous control system successfully handled the scenarios set as the scope of the thesis.
- The results from the experiment show a high level of autonomy as the control system adapted its actions to the state of the environment and perfectly handled situations not overtly designed for.
- Identified the need for automated planning. We claim that it is necessary to look more than one operation ahead to select course of action.
- Outlined and demonstrated a method where a static representation of the predefined business logic is combined with automated planning and JACK’s lightweight BDI implementation. Together providing an efficient approach to cope with situations not foreseen at design time, the highly dynamic environment and failure.
- We have identified the need for ontologies and thorough the prototype provided useful input to the ontology development in AutoConRig.

14.3 Possible Improvements of the Prototype

Throughout this first initiative to address autonomous control of drilling rigs we have demonstrated some important concepts. However, with limited time and available resources we did not manage to finish all parts of the prototype.
Possible improvements of the prototype are summarised below.

- **Robustness**
  - Re-planning should be initiated if the environment changes in a way that makes a scheduled operation non applicable.
  - When an anomaly of some type prevents the system from achieving a goal, re-planning should be initiated to find an alternative sequence of actions to achieve the same goal.

- **Simulator**
  - The simulator should be extended to include unpredictable behaviour in order to better test the system’s ability to re-configure itself (adapt to the environment).

### 14.4 Subjects for Further Research

We have taken a broad approach towards autonomous control of drilling rigs and touched on many subjects without considering them in detail. Some of these areas are described below.

- **Architecture** – The outlined autonomous control system is defined with a finite set of agents, i.e. the agent-organisation is closed and cannot be dynamically expanded. An open agent-organisation would obviously be more flexible as it allows agents to be dynamically integrated into the multi-agent system. The Operations-role could for example dynamically be fulfilled by all discovered agents capable of controlling some function on the rig. Using this approach new functionality to support additional machinery or equipment could be dynamically added and automatically integrated. However, an open MAS also introduce some unsolved issues related to security, trust and infrastructure, such as who can deploy agents, who are responsible, and how can quality of service (QoS) be ensured. All these questions are relevant and should be carefully considered in the future.

- **Planning** - We have shown and demonstrated how automated planning can be combined with the BDI architecture. Planning is used to determine the optimal sequence of operations to apply in order to achieve the system’s goals. BDI is on the other hand used to find a quick solution to more low-level problems where quick response is paramount. However, we have not defined any clear distinction between the problems that should be solved by planning and those to be solved by BDI. Another relevant, but unsolved issue is to identify the operations used in planning and carefully determine the appropriate level of granularity for these. A low level of granularity will enable the planning algorithm to come up with more intelligent and detailed solutions, but this would also require more computation power and likely require more frequent re-planning as opposed to operations defined with a higher level of granularity. However, these problems are all subjects for further research.

The planning algorithm encapsulated by the Driller-agent is fairly simple and does not scale well with a high number of operations (search room). Efficiency and advanced planning techniques have not been a priority in our conceptual prototype, but obviously an important subject for additional research.

Another issue related to planning is how to create and maintain an internal model of the state of the environment, and also how to define pre- and post conditions for the operations used in planning. We have in our prototype taken a simple approach where state definitions that evaluate to either true or false are used for this purpose. We see that this approach comes short when the number of state definitions grows and the complexity of the state increases. However, we have left locating the appropriate techniques needed to address this for a later project.
• **Communication standard** - We have proposed an approach where the ControlInterface -agent communicates with the external control systems using a simple generic interface. This interface is in the prototype implemented by the so-called low-level agents (i.e. Slips, Hoisting, Rotation, Circulation, and Well). The general idea is to create agent wrappers around the external control systems (and other external resources) and solve potential interoperability issues in these wrappers. Remaining work in this area is to identify the appropriate level of granularity, find a suitable format to define them in and identify the specific operations. Solving interoperability issues is a hot topic within the IOHN project is currently being addressed by multiple research projects. Particularly relevant is the drilling communication standard being developed in the AutoConRig project. It remains to be seen how this work can be merged with our work.

• **Ontologies** - We have in the implementation of the prototype taken a simple approach to common ontology. It is essentially a combination of simple dictionary declarations and a taxonomy describing the various configurations the rig and its environment can be in. These definitions are defined on slightly different levels of granularity and have somewhat unclear semantics, leaving room for different interpretations. This will in the future hopefully be defined more formally in for example an ontology language such as OWL.

• **Learning** - A non-existing subject in our prototype implementation is learning. Techniques from machine learning can be applied to continually improve the system’s performance. The system can for example monitor its environment and learn from how actions affect the environment and use this knowledge to make better predictions about the future. The Hoisting agent can improve its performance in hoisting operations by watching how the draw-work decelerates, and this way become more accurate.

• **Parallel agendas** - Another subject that deserves attention is related to how the system can have parallel agendas with individual goals, for example resolve temporary problems while maintaining its agenda. This idea includes goal synchronisation and other contradictions that may arise. This is an interesting and highly relevant topic which hopefully gains attention in the remainder of the AutoConRig project.
This section contains the appendices for the thesis. The appendices are listed below.

- **A. Notation** – Descriptions of the various notations used in this document.
- **B. Detailed Interactions** – Detailed interaction diagrams omitted from the thesis.
- **C. Shared Ontology: State Definitions** – Describes the high-level concepts used to describe the state of the environment.
- **D. Implementation Details** – Implementation details omitted from the report.
A. Notation

This appendix describes the different graphical notations used in this report. (We assume that the reader is familiar with the UML notation [39]).

A1. PDT Diagram Constructs

The graphical symbols used in PDT are shown below in Table 19.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Agent" /></td>
<td>Agent</td>
</tr>
<tr>
<td><img src="image2" alt="Scenario" /></td>
<td>Scenario</td>
</tr>
<tr>
<td><img src="image3" alt="Data" /></td>
<td>Data</td>
</tr>
<tr>
<td><img src="image4" alt="Goal" /></td>
<td>Goal</td>
</tr>
<tr>
<td><img src="image5" alt="Protocol" /></td>
<td>Protocol</td>
</tr>
<tr>
<td><img src="image6" alt="Action" /></td>
<td>Action</td>
</tr>
<tr>
<td><img src="image7" alt="Role" /></td>
<td>Role or functionality</td>
</tr>
<tr>
<td><img src="image8" alt="Percept" /></td>
<td>Percept</td>
</tr>
</tbody>
</table>

A2. JACK JDE Graphical Notation

The graphical constructs provided by the JACK Java Development Environment are summarised in Table 20.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image9" alt="Agent" /></td>
<td>Represents an JACK agent</td>
</tr>
<tr>
<td><img src="image10" alt="Capability" /></td>
<td>Represents an JACK capability</td>
</tr>
<tr>
<td><img src="image11" alt="Event" /></td>
<td>Represents an JACK event of any type</td>
</tr>
<tr>
<td><img src="image12" alt="Plan" /></td>
<td>Represents and JACK plan</td>
</tr>
<tr>
<td><img src="image13" alt="NamedData" /></td>
<td>Represents an NamedData, e.g. Beliefset or external resource.</td>
</tr>
</tbody>
</table>
B. Detailed Interactions

This appendix contains UML sequence diagrams describing details that are removed from the thesis to save space and simplify reading. Note that some of the details specified here are implemented slightly different in the prototype.

B1. Interactions: Planning

The interaction diagram in Figure 65 shows the planning process that is initiated after the Driller-agent receives a goal from the Supervisor.

![Interaction Diagram: Planning](image)

The diagram shows the Driller-agent requesting a snapshot of the current state of the environment. The ControlInterface-agent responds with this information, and the Driller plans sequences of operations. After the planning is complete, it selects the optimal plan for execution.
B2. Interactions: Hoisting

Figure 66 shows a run of a hoisting operation. The interaction diagram takes two parameters, direction and length. The direction-parameter describes the direction of the hoisting operation (0 = DOWN, 1 = UP) and the length-parameter describes the length to hoist.

Figure 66 Interaction Diagram: Hoist

Some calculations are done before the hoisting operation can be performed. This includes the distance to wait until the deceleration process should be initiated. The details of the acceleration/hosting process are shown in the referenced Accelerate-interaction diagram (see B3).

The loop combination fragment which follows the first interaction occurrence, illustrates the waiting period before the deceleration process is started. The guard indicates that the deceleration process waits until the correct position to start reducing the speed. The interactions in the deceleration process are shown in the referenced Decelerate – sequence diagram (see B4).
B3. Interactions: Acceleration

The Accelerate-interaction diagram shown in Figure 67 takes two parameters, \textit{speedtarget}, and \textit{direction}. \textit{Speedtarget} is the speed the acceleration process should achieve, and the direction describes the vertical direction (0 =DOWN, 1 = UP).

![Interaction Diagram: Accelerate](image)

The trace illustrated by the interaction diagram starts with two nested loop-fragments. In the outmost loop the hoisting speed is increased one gear at a time. The innermost level of the loop makes sure that the optimal speed is achieved before the gear is changed. The external function \texttt{setDWGear} sets the gear, and the process variables (percepts) \texttt{DW\_GEAR}, \texttt{DW\_GEAR\_MODE}, \texttt{DW\_SPEED}, \texttt{BIT\_POSITION}, and \texttt{HOOK\_POSITION} indicate the progress.
B4. Interactions: Deceleration

The Decelerate-interaction diagram shown in Figure 68 takes three parameters, *gearmode*, *gear*, and *direction*. The *gearmode* and *gear*—variables represent the current gear of gearbox of the draw-work (equivalent with the *DW_GEAR_MODE* and *DW_GEAR* percepts). The *direction*-variable describes the direction of the operation (0 = DOWN, 1 = UP).

The interaction diagram shows the Hoisting-agent receiving a *HaltHoistingMsg* from the ControlInterface-agent, which triggers the execution of the optimal deceleration process of the vertical movement of the drillstring. This is achieved through a controlled gear-down using the *setDWGear*—function. The gear (*DW_GEAR* and *DW_GEAR_MODE*) is smoothly decreased until the speed (*DW_SPEED*) is 0 m/s. When this is accomplished, an *ActionResultMsg* is sent to the ControlInterface-agent, indicating that the operation was successfully executed.

Figure 68 Interaction Diagram: Decelerate
C. Shared Ontology: State Definitions

In this appendix we describe the state definitions from the shared ontology (see chapter 9). The definitions are implemented as a hierarchy of Java Interface which we have chosen to visualise using UML class diagrams. Note that only leaf nodes in these diagrams can be instantiated, as the other (more abstract) states only exist to simplify logical expressions in planning.

C1. Bit position

Bit position is the vertical position of the drilling bit inside the well. The significant bit positions (conditions) are depicted in Figure 69.

![Figure 69 Identified States for Bit Position](image)

The diagram illustrates that there are two states at the top level related to the position of the bit in the well. The bit can either be in the cased section –state or in open hole section –state. Further, if it is in the open hole section –state it is either less than 1 stand from cased section or more than 1 stand from cased section. The states can briefly be described as follows.

- **Cased section** – This indicates that the bit is above the casing shoe, i.e. in cased section. Based on the scenario descriptions, this is the only state that is significant when above the casing shoe, meaning that we do not need to decompose this state any further.

- **Open hole section** – This is the case when the bit is below the casing shoe. In the scenario descriptions there are two significant states that are important when the bit is in the open hole section. These states are described below.

- **Less than 1 stand from cased section** – This is the situation when the bit is identified in a position less than one stand below casing shoe. This implies that the drillstring can be hoisted until the bit is above casing shoe. Note that this depends on the hook position and not the stand length.

- **More than 1 stand from cased section** – This is the case when the bit position is identified to be more than 1 stand below the cased section (far in the open hole section).

C2. Circulation

The circulation system serves many purposes. However, for the scope of the prototype we are only interested in the states depicted in Figure 70.
The diagram shows that the most significant states of the circulation system are Mudline Connected and Mudline Disconnected. If Mudline Connected its either Ready to Circulate or Circulating. Descriptions of the states are provided below.

**Not circulating** – This state simply indicates that the mud is not circulating in the well.

**Mudline Disconnected** – This state indicates that the mudline is not connected, which means that mud cannot be pumped using the current configuration. This obviously also implies that the not circulating state is active.

**Mudline connected** – This indicates that the circulation system is connected to the drillstring. If this is the case, we must be in one of the states described below.

**Ready to circulate** – This is the case when the mudline is connected, but no mud is being pumped through it, i.e. it is not circulating.

**Circulating** – This state represents a situation when the mudline is connected and mud is being pumped through the system.

### C3. Hook position

The hook position is the vertical position of the hook on the derrick. The various states that are significant for the scenarios are depicted in Figure 71.

At the top level of this diagram we have two conditions, Insignificant – the hook position is not important for the scope of the demonstrator and significant – the hook position is important for the business logic. The states are described below.

**Insignificant** – The hook position is not important for the high level business logic. This is in contrast to the significant-conditions.
Significant – The subclasses of this state describe states that are significant for our business logic, especially scenario 3.

Upper position – This state represents the condition when the drillstring is hoisted as much as possible, i.e. the whole upper stand is above the drill floor. This state is necessary to support scenario 3 as it represents the highest hook position of the oscillated process.

Center position – This state represents the situation when 50 percent of the upper stand is above the drill floor. This condition is also important for scenario 3 as it denotes the lowest hook position when oscillating.

C4. Park break

The park break is a part of the draw-work and is used to lock (secure) the draw-work together with whatever is hanging beneath the hook.

![Diagram of Park Break States]

We have defined two significant states for the park break.

Inactive – The park break is not used to lock the hoisting mechanism (the park break is not activated).

Active – The park break is locked and the hoisting mechanism secured.

C5. Slips

The slips is a component used to grip the drillstring in a non damaging manner. This allows the drillstring to be suspended from the hoisting mechanism. States that are relevant for our prototype is depicted in Figure 73.

![Diagram of Slips States]

The diagram has two significant states at the top level, Active – the slips is wrapped around the drillstring, or Inactive – the slips is not attached to the drillstring. A description of various states is provided below.

Inactive – The slips in not placed around the drillstring. This indicates that the slips is currently suspended from the drillstring and the drillstring can freely travel up and down in the wellbore.
Active – This is the case when the slips is placed around the drillstring. If the slips is wrapped around the drillstring, the slips must be in one of the states below.

In position – This state indicates that the slips is wrapped around the drillstring, but not locked, i.e. the slips-component does not carry the weight of the drillstring. For the slips to be locked it requires the drillstring to be lowered while in position.

Locked – This state is the case when the draw-work has transferred the weight of the drillstring over to the slips, i.e. slips is locked.

C6. Hoisting

The hoisting mechanism for a rig consists of the components that facilitate elevation of the drillstring e.g. draw-work, travelling block, derrick etc. The states that are important for our scope are shown in Figure 74.

![Figure 74](image)

Figure 74 Identified States for the Hoisting functionality

Also this entity has two possible states at the top level, Disconnected – hoisting cannot be performed, and connected to drillstring – the drillstring can be hoisted. More detailed descriptions of the various states are provided below.

Not hoisting – This state simply means that the drillstring is not in motion. From Figure 74, we can see that this can occur if we are in the disconnected – state or in the ready to hoist-state.

Disconnected – This state indicates the drillstring cannot be hoisted e.g. neither the top-drive nor the elevator is connected to the drillstring. This means also that we are in a not-hoisting state.

Connected to drillstring – This is the case when we are in fact connected to the drillstring, i.e. the drillstring is connected to the top-drive or the elevator. This also implies that the hoisting mechanism must be in one of the states described below.

Ready to hoist – This state represents a state where the system is ready to perform a hoisting operation. This is characterised by the fact that we are in the connected to drillstring - state, and that we are in the not hoisting – state, i.e. the drillstring stands still.

Hoisting – This state represents situations where the drillstring is being hoisted. This implies that we are in the connected to drillstring – state, since the drillstring obviously must be connected to the hoisting machine to be hoisted.
C7. Rotation

This represents the functionality that facilitates rotation of the drillstring. The states of importance with respect to the scenarios are shown in Figure 75.

![Diagram of states for rotation functionality]

Figure 75 Identified States for the Rotation function

As with the previous diagrams, this diagram has also two states at the top level. The rotation mechanism can be **Disconnected** or **Connected to drillstring**.

- **Not rotating** – The drillstring is simply not rotating. According to our model, this can only happen if we are in the **Disconnected** - state or in the **Connected to drillstring** –state.

- **Disconnected** – The rotation functionality is disconnected from the drillstring, e.g. the top-drive is not connected to the drillstring. This implies that we are also in the **Not rotating** –state.

- **Connected to drillstring** – This state occurs when the rotation-machinery is in fact connected to the drillstring. This state can only occur if we are in the rotating –state or the ready to rotate-state.

- **Ready to rotate** – This is the case when the rotation mechanism is connected to the drillstring, but drillstring is currently not rotating.

- **Rotating** – This is the case when the drillstring is rotating. This obviously implies that the rotation functionality is connected to the drillstring, i.e. **Connected to drillstring** – state is also active.
D. Implementation Details

This appendix contains information related to the implementation of our prototype.

D1. Message Descriptors

The messages used in our prototype are listed in Table 21. The table contains name and descriptions of the messages, the agents that handles them and their content (parameters).

<table>
<thead>
<tr>
<th>Handled by</th>
<th>Message</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Circulation</td>
<td>SetCirculationActionMsg</td>
<td>double speed</td>
<td>Sent by the ControlInterface to trigger the Circulation-agent to control the circulation with respect to the given speed-parameter.</td>
</tr>
<tr>
<td>ControlInterface</td>
<td>ActionResultMsg</td>
<td>boolean achieved</td>
<td>Sent by the low-level agents in response to a *.ActionMsg – message (a message where action is requested). The parameter indicates whether the task was successfully executed.</td>
</tr>
<tr>
<td>ControlInterface</td>
<td>MeasurementUpdateMsg</td>
<td>String property</td>
<td>Sent by the low-level agents when they discover a new fact about the world. The property-parameter identifies the process-variable updated, and the value-parameter represents its actual value.</td>
</tr>
<tr>
<td>ControlInterface</td>
<td>OperationRequestMsg</td>
<td>String operationid</td>
<td>Sent by the Driller to request a specific operation executed. The operation is identified by the parameter.</td>
</tr>
<tr>
<td>ControlInterface</td>
<td>SystemStateRequestMsg</td>
<td>-</td>
<td>Sent by the Driller to request an abstract snapshot of the current state of the environment.</td>
</tr>
<tr>
<td>Driller</td>
<td>OperationResultMsg</td>
<td>boolean achieved</td>
<td>Sent by the ControlInterface in response to an OperationRequestMsg – message. The parameter indicates the success of the operation.</td>
</tr>
<tr>
<td>Driller</td>
<td>OperationSetMsg</td>
<td>Operation[]</td>
<td>Sent by the ControlInterface on system start-up. Contains descriptions of its operations, i.e. pre- and post conditions and a unique ID.</td>
</tr>
<tr>
<td>Driller</td>
<td>PlanningGoalMsg</td>
<td>Class[] states</td>
<td>Send by the Supervisor and carries an overall organisational goal the Driller should try to achieve. This is</td>
</tr>
<tr>
<td>Driller</td>
<td>SystemStateResponseMsg</td>
<td>Class[] states</td>
<td>also expressed using the state definitions from the common ontology.</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------</td>
<td>----------------</td>
<td>---------------------------------------------------------------------</td>
</tr>
<tr>
<td>Hoisting</td>
<td>HaltHoistingActionMsg</td>
<td>-</td>
<td>Sent by the ControlInterface to trigger the Hoisting-agent to halt the vertical movement of the drillstring.</td>
</tr>
<tr>
<td>Hoisting</td>
<td>HoistActionMsg</td>
<td>int direction double length</td>
<td>Sent by the ControlInterface to trigger the Hoisting-agent to hoist the drillstring the length and direction specified by the parameters.</td>
</tr>
<tr>
<td>Hoisting</td>
<td>LockSlipsActionMsg</td>
<td>-</td>
<td>Sent by the ControlInterface to trigger the Hoisting-agent to lower the drillstring until the slips is locked.</td>
</tr>
<tr>
<td>Hoisting</td>
<td>ActivatePBActionMsg</td>
<td>-</td>
<td>Sent by the ControlInterface to trigger the Hoisting-agent to activate the parkbreak.</td>
</tr>
<tr>
<td>Hoisting</td>
<td>DeactivatePBActionMsg</td>
<td>-</td>
<td>Sent by the ControlInterface to trigger the Hoisting-agent to deactivate the parkbreak.</td>
</tr>
<tr>
<td>Rotation</td>
<td>SetRotationActionMsg</td>
<td>int gear double speed</td>
<td>Sent by the ControlInterface – agent to trigger rotation of the drillstring.</td>
</tr>
<tr>
<td>Simulator</td>
<td>setMudCirculationMsg</td>
<td>double setpoint</td>
<td>Sent by the low-level agents. See 7.3 for description.</td>
</tr>
<tr>
<td>Simulator</td>
<td>ActivatePBMsg</td>
<td>-</td>
<td>Sent by the low-level agents. See 7.3 for description.</td>
</tr>
<tr>
<td>Simulator</td>
<td>DeactivatePBMsg</td>
<td>-</td>
<td>Sent by the low-level agents. See 7.3 for description.</td>
</tr>
<tr>
<td>Simulator</td>
<td>ActivateSlipsMsg</td>
<td>-</td>
<td>Sent by the low-level agents. See 7.3 for description.</td>
</tr>
<tr>
<td>Simulator</td>
<td>DeactivateSlipsMsg</td>
<td>-</td>
<td>Sent by the low-level agents. See 7.3 for description.</td>
</tr>
<tr>
<td>Simulator</td>
<td>SetDWGearMsg</td>
<td>int gear int gearmode bit direction</td>
<td>Sent by the low-level agents. See 7.3 for description.</td>
</tr>
<tr>
<td>Simulator</td>
<td>setTDSpeedMsg</td>
<td>int gear double speed</td>
<td>Sent by the low-level agents. See 7.3 for description.</td>
</tr>
</tbody>
</table>
Slips
ActivateSlipsActionMsg
- Sent by the ControlInterface to trigger the Slips-agent to activate the slips.

Slips
DeactivateSlipsActionMsg
- Sent by the ControlInterface to trigger the Slips-agent to deactivate the slips.

Hoisting
ReleaseSlipsActionMsg
- Sent by the ControlInterface to trigger the Hoisting-agent to hoist the drillstring until the slips is unlocked (released).

Slips, Well, Rotation, Circulation, Hoisting
MeasurementMsg
String property String value Percepts sent by the Simulator to the autonomous control system.

Supervisor
CommunicationFailureMsg
- Sent by the simulator to simulate communication failure.

Supervisor
PlanningGoalResultMsg
boolean achieved Sent by the Driller in response to a PlanningGoalMsg – message. The parameter describes whether the goal was achieved.

### D2. The Business Logic mapped to the Common Ontology

Table 22 shows how we have mapped the high-level business logic to the common ontology. The left column shows the name of the activity/goal and right column how we have described in terms of concepts from the common ontology.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Goal definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halt operations</td>
<td>Hoisting = NotHoisting</td>
</tr>
<tr>
<td>Move Bit Above Casingshoe</td>
<td>BitPosition = CasedSection</td>
</tr>
<tr>
<td>Lock Drillstring</td>
<td>Slips = Locked</td>
</tr>
<tr>
<td>Rotate drillstring</td>
<td>Rotation = Rotating</td>
</tr>
<tr>
<td>Circulate</td>
<td>Circulation = Circulating</td>
</tr>
<tr>
<td>Secure hoisting machine</td>
<td>ParkBreak = Locked</td>
</tr>
<tr>
<td>Move to Upper Hook Position</td>
<td>HookPosition = Upper</td>
</tr>
<tr>
<td>Center the Hook Position</td>
<td>HookPosition = Center</td>
</tr>
</tbody>
</table>
D3. The Planning Algorithm

The body of the planning algorithm is shown below. It is implemented in Java and uses the forward-chaining planning technique.

```java
private static void generatePlans( Plan backlog, State currentstate) {
    boolean isGoalAchieved = State.goalStateAchieved(goalState, currentstate);
    if (isGoalAchieved) {
        return;
    }
    // prevent loops within a plan,
    // by ensuring a state to be visited only once per branch
    boolean stateAlreadyVisited = backlog.getVisitedStates().contains(currentstate);
    if (stateAlreadyVisited) {
        return;
    }
    // set the state as visited
    backlog.addVisitedState(currentstate);
    // prevents long plans
    if (backlog.size() > MAX_PLAN_SIZE) {
        return;
    }
    // iterate through all operations
    for (AbstractOperation op : ops) {
        // prevents an operation to be performed twice in a row
        if (backlog.isLastOperation(op)) {
            continue; // skip
        }
        // investigate branch if the pre-condition of the operation is fulfilled
        if (op.preConditionsFulfilled(currentstate)) {
            // clone path to use in new branch
            State stateClone = currentstate.clone();
            Plan backlogClone = new Plan(backlog);
            // add the operation to the new branch
            backlogClone.add(op);
            // add the post-condition of the operation to the state of the new branch
            op.Execute(stateClone);
            // follow the new branch
            generatePlans( backlogClone, stateClone);
        }
    }
}
```


27. AgentLink. [online accessed; 31 October 2008] www.agentlink.org


   http://www.iris.no/Internet/IntOpera.nsf/wvDocID/4C2AEF53F9828DDDC1257299004613FA


