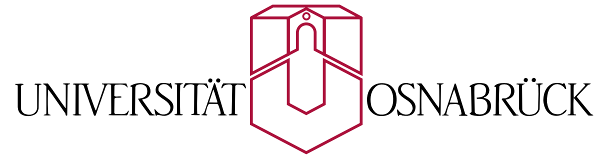


INSTITUTE OF COMPUTER SCIENCE
KNOWLEDGE-BASED SYSTEMS RESEARCH GROUP

Doctoral Dissertation

to obtain the doctoral degree

Doktor der Naturwissenschaften (Dr. rer. nat.)
(Doctor of Natural Sciences)

Fusing DL Reasoning with HTN Planning as a Deliberative Layer in Mobile Robotics

Ronny Hartanto

12.08.2009

First Supervisor: Prof. Dr. Joachim Hertzberg
Second Supervisor: Prof. Michael Beetz, Ph.D.

Abstract

Action planning has been used in the field of robotics for solving long-running tasks. In the robot architectures field, it is also known as the deliberative layer. However, there is still a gap between the symbolic representation on the one hand and the low-level control and sensor representation on the other. In addition, the definition of a planning problem for a complex, real-world robot is not trivial. The planning process could become intractable as its search spaces become large. As the defined planning problem determines the complexity and the computationability for solving the problem, it should contain only relevant states. In this work, a novel approach which amalgamates *Description Logic* (DL) reasoning with *Hierarchical Task Network* (HTN) planning is introduced.

The planning domain description as well as fundamental HTN planning concepts are represented in DL and can therefore be subject to DL reasoning; from these representations, concise planning problems are generated for HTN planning. The method is presented through an example in the robot navigation domain. In addition, a case study of the *RoboCup@Home* domain is given. As proof of concept, a well-known planning problem that often serves as a benchmark, namely that of the blocks-world, is modeled and solved using this approach.

An analysis of the performance of the approach has been conducted and the results show that this approach yields significantly smaller planning problem descriptions than those generated by current representations in HTN planning.

Acknowledgements

This dissertation is not the result of a few weeks or months of work, but rather of years of work. It would not have been possible without the help and support of many people. Therefore, I thank all those who have directly or indirectly supported me in this endevour. Of these generous people, I take this opportunity to thank a few by name.

First, I thank my first supervisor ("*Doktorvater*"), Prof. Joachim Hertzberg, for his tremendous support and guidance over these past years. In particular, for this opportunity to work on such an engaging research topic. My sincere thanks to my second supervisor, Prof. Michael Beetz, for his valuable suggestions and guidance. I express my gratitude to Prof. Gerhard Kraetzschmar, Iman Awaad, and Christoph Mies for their time and effort in reading this work and for their valuable feedback.

Working in robotics research would not be possible without robots and research labs. Therefore, I thank Prof. Kurt Ulrich-Witt, Prof. Erwin Praßler, Prof. Paul G. Plöger, and Prof. Gerhard Kraetzschmar for providing me with the opportunity to work in the b-it funded research group and supporting me in my work on this dissertation. I thank my colleagues, past and present: Walter Nowak, Iman Awaad, Azamat Shakhimardanov, Jan Paulus, and Anastassia Küstenmacher. During the time spent on this work, I have had the pleasure of being a part of, not only the b-it research group, but also the EU-funded project, XPERO. Therefore, I extend my thanks to my XPERO-colleagues: Timo Henne, Björn Kahl, and Katharina Hoffmann.

If this is the second page you have read so far, you will not yet have noticed one more important aspect of this work, namely the *b-it-bots* RoboCup team. I gratefully acknowledge my team members: Dirk Holz, Jan Paulus, Thomas Breuer, Geovanny Giorganna, Frederik Hegger, Christian Müller, and Jin Zha. My gratitude also goes to one very special member, namely Johnny Jackanapes "the robot", without whom the team would not be able to participate in any RoboCup events.

I owe my deepest gratitude to my dear Meidjie Ang for her love and continuous encouragement and to my family in Indonesia for their endless support.

Contents

Abstract iii

Acknowledgements v

List of Figures xi

List of Tables xv

1. Introduction 1
- 1.1. Motivation 2
- 1.2. Problem Statement 3
- 1.3. Challenges 4
- 1.4. Scope 5
- 1.5. Previous Work 6
- 1.6. Contributions 9
- 1.7. Outline 11

2. The Hybrid Deliberative Layer 13
- 2.1. System Architecture 13
  - 2.1.1. A Brief Overview of Robot Control Architectures 13
  - 2.1.2. The Hybrid Deliberative Layer Architecture 15
- 2.2. HDL Components 17
  - 2.2.1. Knowledge Bases 17
    - 2.2.1.1. A Brief Overview of Knowledge Representation and Description Logic 17
    - 2.2.1.2. KB in HDL Implementation 23
  - 2.2.2. Description Logic Reasoner 23
  - 2.2.3. Planner 24
  - 2.2.4. Planning Problem Extractor 25
- 2.3. Concept 26
- 2.4. Ontologies 26
  - 2.4.1. HTN Planning Definition 27
  - 2.4.2. HTN Planning TBox 29
- 2.5. Reasoning 34
  - 2.5.1. Algorithm for HTN Planning Domain/Problem Generator 35
  - 2.5.2. Algorithm for Generating SHOP2 Code 37

3. HDL Systems in the Robotics Domain 41
- 3.1. Modelling the HTN Planning Problem in the HDL System 41
- 3.2. Navigation Domain 42
  - 3.2.1. Step 1: Define the Actions and Objectives 43

3.2.2. Step 2: Define the Task Networks 44
3.2.3. Step 3: Program the Planning Domain 46
3.2.4. Step 4: Test the Planning Domain 47
3.2.5. Step 5: Define the HTN ABox in the HDL System 49
3.2.6. Step 6: Modelling and Instantiating the States in the HDL System . . . 50
3.2.7. Step 7: Testing the HDL System 51
3.3. Exploiting the HDL system 51
3.3.1. Results 54
3.4. Pick-and-Place Domain 55
3.4.1. Problem Specification 56
3.4.2. Modelling Actors and Objects 56
3.4.3. Defining the HTN Planning Domain for Pick-and-Place Tasks 57
3.4.3.1. Partial Pick-and-Place Domain 58
3.4.3.2. Complete Pick-and-Place Domain 64
3.4.4. Results 65

4. Case Study: “Johnny Jackanapes” 73
4.1. RoboCup@Home 73
4.1.1. Test Scenarios 74
4.1.2. Challenges 75
4.2. Johnny Jackanapes, The Robot 76
4.2.1. Hardware Components 76
4.2.2. Software Components 77
4.2.3. Applications 80
4.3. The HDL system in Johnny Jackanapes 83
4.3.1. Using the HDL System as Knowledge Base Component 85
4.3.1.1. Navigation and Localisation 85
4.3.1.2. Object Detection and Manipulation 87
4.3.1.3. Human Robot Interaction 89
4.3.2. Solving RoboCup@Home Tasks with the HDL System 90
4.3.2.1. Modelling the ABox of RoboCup@Home Environment . . . 90
4.3.2.2. Solving the Scenario with Pick-and-Place Domain 92
4.3.2.3. Adding New Methods in the ABox 93

5. HDL Systems in the AI Domain 97
5.1. Blocks-World 97
5.1.1. Problem Statement 97
5.1.2. The HTN Planning Domain 98
5.2. HDL Implementation of Blocks World 103
5.2.1. HDL’s Blocks World Domain 103
5.2.2. Modelling Blocks World in HDL 104
5.2.3. Enhanced Model of Blocks World Domain in HDL 107
5.3. Experimental Results 110
5.3.1. Simple Blocks World 110
5.3.2. Two Simple Blocks World Problems 111
5.3.3. Complex Blocks World Problems 113

6. Results and Evaluation **117**
6.1. Complexity of the HDL System 117
6.2. Experiment Design 121
6.3. Experiments 124
6.3.1. Navigation Domain 124
6.3.1.1. Analysing the Result 125
6.3.2. The Blocks World Domain 129
6.3.2.1. Analysing the Results 130
6.4. Concluding Remarks 134

7. Discussion **135**
7.1. HTN Blocks World Anomaly 135
7.2. Inconsistencies in the Model 136
7.3. Defining Usable Objects 137
7.4. HDL and Plan-Based-Control Approaches for Robotics 138
7.5. Plan Recovery from Failures 139
7.6. Operator Cost and Plan Optimisation 139
7.7. HDL Versus Other DL-Planning Approaches 140

8. Conclusions **143**
8.1. Summary 143
8.2. Strengths and Limitations 144
8.3. Future Work 144
8.3.1. Affordance-Based Planning 144
8.3.2. Using Other HTN Planning Implementations 145
8.3.3. Collocate the Planner Using Web-Service 145
8.3.4. Application in Plan-Based Robot Control 145
8.3.5. Using DL Inference Engine for Plan Repair 146

Bibliography **147**

Appendices **159**

A. Generated Planning Domain **161**
A.1. Navigation Domain 161
A.1.1. An Example from the Planning-Domain Instance 161
A.1.2. An Example from the Method Instance 164
A.2. Pick-and-Place Domain 166
A.2.1. Partial Pick-and-Place Domain 166
A.2.2. Complete Pick-and-Place Domain 170
A.3. Johnny Jackanapes Domain 175
A.3.1. Solution Using Pick-and-Place Domain 175
A.3.2. Bring an Object Planning Domain 176
A.4. Blocks World Domain 181
A.4.1. Simple Blocks World Planning Example 186
A.4.2. Two Blocks World Planning Example 188
A.4.3. Complex Blocks World Example 193

B. HDL ABox Assertion **199**
B.1. Navigation Domain 199
B.2. Pick-and-Place Domain 200
B.2.1. Partial Pick-and-Place Domain 200
B.2.2. Complete Pick-and-Place Domain 202
B.3. Blocks World Domain 203

C. Software Engineering **207**
C.1. HDL Plan Suite 207
C.2. UML Diagrams 211
C.2.1. HDL Plan Option Module 212
C.2.2. OWL Actor Editor Module 213
C.2.3. OWL Module 213
C.2.4. OWLPlanner GUI Module 214
C.2.5. Planner Module 215
C.2.6. SHOP Support Module 216

List of Figures

1.1. Hybrid Deliberative Layer (HDL) in a robot control architecture. 5

2.1. Robot control system spectrum 15
2.2. HDL in a layered robot control architecture. 16
2.3. A figure with some different representations. 17
2.4. Architecture of a knowledge representation system based on Description Logics. 20
2.5. Layers of OIL. 21
2.6. A simple conceptual model for planning 25
2.7. A schematic diagram on how the planning problem is passed from the DL representation to the HTN planner. 26
2.8. Planning knowledge representation in DL TBoxes and ABoxes. 27
2.9. Illustration on how HTN expands the planning problem into subtasks. 28
2.10. HTN planning ontology. 29
2.11. An example of HDL planning domain instance for domain1. 30
2.12. Task network of method m1. 32
2.13. Graph representing the task network of method m1 32
2.14. An example of the HDL method instance for m1. 33
2.15. An example of the HDL operator instance for o2. 34
2.16. A reasoning process over DL representation to extract a concrete HTN planning problem. 35
2.17. Simple navigation domain example. 37

3.1. HTN planning ontology. 43
3.2. Navigation task network. 45
3.3. Navigation domain's assertions in the HDL. 49
3.4. Navigation domain's states concepts. 50
3.5. Simple navigation domain with door(s) connecting between rooms, which can be either open or closed. 52
3.6. Extended navigation domain's states concepts. 53
3.7. Pick-and-place domain with two robots and six manipulable objects. 56
3.8. Extended pick-and-place domain's states concepts. 57
3.9. Partial pick-and-place domain task network. 58
3.10. Method "moveobject_p" and "moveobject2_p" assertions in the HDL. 59
3.11. Method "getobject_p" and "getobject2_p" assertions in the HDL. 61
3.12. Method "putobject_p" assertion in the HDL. 62
3.13. Operators assertion in the HDL for pick-and-place domain. 63
3.14. Planning domain assertion in the HDL for pick-and-place domain. 63
3.15. Complete pick-and-place domain task network. 64
3.16. Method assertions in the HDL for complete pick-and-place domain. 66
3.17. Planning domain assertions in the HDL for complete pick-and-place domain. . 67

3.18. The complete pick-and-place plan and its relation to the partial pick-and-place plan and the navigation domain. 69
3.19. Illustration the solution plan for robot2. 70

4.1. (a) Introduce (RoboCup 2008), (b) Fetch & Carry (RoboCup 2009), (c) Fast Follow (German Open 2009), (d) Supermarket (German Open 2009), (e) Party Bot (RoboCup 2009), (f) Final (RoboCup 2008). 74
4.2. Hardware specification of Johnny Jackanapes. 76
4.3. Components and operating systems platforms. 79
4.4. State machine diagram of the open challenge scenario. 81
4.5. RoboCup@Home arena points of cloud from mapping module with Johnny trajectories during open challenge scenario. 82
4.6. (a) Johnny receiving a command, (b) grasping the coke, (c) bringing the coke to the guest, (d) handing the coke to the guest. 83
4.7. HDL system architecture in Johnny Jackanapes. 84
4.8. Ontology for a RoboCup@Home environment that is relevant for the navigation and localisation module. 86
4.9. The RoboCup@Home environment with the dining table rotated 90°. 88
4.10. The concept Person and its subclasses. 89
4.11. RoboCup@Home ontology. 91

5.1. Blocks world domain with four blocks. 98
5.2. Blocks world task network decomposition tree. 103
5.3. An instance of Planning-Domain represents the blocks world domain. 103
5.4. Blocks world domain ontology. 105
5.5. Two blocks world domains with four blocks each. 107
5.6. Enhanced blocks world domain ontology. 108
5.7. Complex blocks world domains. 109
5.8. Solution plans for the simple blocks world problem (without dummy operators). 111
5.9. Solution plans for the two blocks world problem (without dummy operators). . 113
5.10. Solution plans for the complex blocks world problem (without dummy operators). 115

6.1. The HDL system's internal component structure. 117
6.2. Main components of the Pellet reasoner. 118
6.3. HTN planning generator components. 119
6.4. JSHOP2 compilation process. 121
6.5. Stored states in DL and possible generated HTN planning domain. 122
6.6. Experiment scenario in navigation domain. 123
6.7. Test scenario in navigation domain. 124
6.8. Number of facts in the planning problem for navigation domain. 126
6.9. Number of generated plans (logarithmic scale) in the navigation domain. 127
6.10. Computation time (logarithmic scale) in the navigation domain. 128
6.11. Computation time (logarithmic scale) in the navigation domain for planning one solution. 129
6.12. Experiment scenario for the blocks world domain. 130
6.13. Number of facts in the planning problem for the blocks world domain. 131
6.14. The number of generated plans (logarithmic scale) in the blocks world domain. 132
6.15. Computation time (logarithmic scale) in blocks world domain. 133

6.16. Computation time (logarithmic scale) in the blocks world domain for planning one solution. . . . 134

C.1. Snapshot of the OWLPlannerGUI. . . . 207
C.2. Snapshot of the JSHOP2Editor for editing the domain. . . . 208
C.3. Snapshot of the JSHOP2Editor for editing the domain. . . . 209
C.4. Snapshot of the JSHOP2Editor for displaying the planning results. . . . 210
C.5. Snapshot of the ActorNavigator window. . . . 211
C.6. Snapshot of the OWL options window. . . . 212
C.7. Snapshot of the JSHOP2 options window. . . . 213
C.8. HDLPlan-Option UML. . . . 214
C.9. OWL-Actor-Editor UML. . . . 215
C.10. OWL module's UML diagram. . . . 216
C.11. OWL Planner GUI UML diagram. . . . 217
C.12. Planner module UML diagram. . . . 217

List of Tables

2.1. Common DL constructors and their correspondence with language name. . . . 21
2.2. OWL constructors. 22

3.1. Generated states for different specialisations of the Fixed-Object concept. . . . 55
3.2. Generated planning domain description from different instances of Method or Planning-Domain. 68

4.1. RoboCup@Home concepts and their instances. 93

5.2. Results generated by the DL reasoner for the simple blocks world domain. . . . 110
5.3. Result of DL reasoner over two blocks world domains. 112
5.4. Results from the DL reasoner for the two blocks world domains for first problem. 112
5.5. Result of DL reasoner over two blocks world domains for problem number 2. . 113
5.6. Result of the DL reasoner for the complex blocks world domain. 114

6.1. Complexity and computability of HTN planning. 120

Glossary

ADL	Action Description Language
CORBA	Common Object Resource Broker Architecture
CWA	Closed World Assumption
DAML	DArpa Markup Language
DCOM	Distributed Component Object Model
DL	Description Logics
FOPL	First-Order Predicate Logic
HDL	Hybrid Deliberative Layer
HDLP	Hierarchical Description Logic Planner
HRI	Human Robot Interaction
HSSH	Hybrid Spatial Semantic Hierarchy
HTN	Hierarchical Task Network
ICE	Internet Communication Engine
IDL	Interface Definition Language
KL-ONE	Knowledge Language One
LPS	Local Perceptual Space
NHC	Nested Hierarchical Controller
OIL	Ontology Inference Layer
OOP	Object-Oriented Programming
OWA	Open World Assumption
OWL	Ontology Web Language
PDDL	Planning Domain Description Language
PLEXIL	Plan Execution Interchange Language
RAP	Reactive Action Package
RCS	Realtime Control System
RDF	Resource Description Framework
RDFS	RDF Schema
RPC	Remote Procedure Call
RPL	Reactive Plan Language
SA	Subsumption Architecture
SHOP	Simple Hierarchical Ordered Planner
SHPWA	Semantic Hierarchical Planning through World Abstraction
SIFT	Scale-Invariant Feature Transform
SLAM	Simultaneous Localisation And Mapping
SPA	Sense-Plan-Act
SSH	Spatial Semantic Hierarchy
STRIPS	STanford Research Institute Problem Solver
TCA	Task Control Architecture
XML	eXtensible Markup Language

Typographic Conventions

In this document, the following typographical conventions are used:

- `serif` for representing instances of DL in the *ABox*.
- `typeset` for representing source code, including that in the SHOP syntax.
- $mathnormal$ for representing concepts of DL in the *TBox*.
- $mathitalic$ for representing properties of a concept.

1. Introduction

In recent years, the robotics field has seen rapid advancements in both software and hardware areas. Good algorithms for perception, learning, planning, navigation and inferencing have been researched and used in autonomous robots. They enable the robot to be more useful in daily life, for example as a courier, or a museum guide, such as RHINO [BCF+99, BAB+01], MINERVA [TBB+00], and ROBOX [ATS03, SAB+03]. Several new robot platforms have been developed which support major research in robotics, from advanced mobile robots, such as legged robots, to lightweight manipulators.

A mobile robot with a manipulator is called a mobile manipulator. It has the capabilities of a mobile robot in addition to the functionality associated with its manipulator. Tasks which could not be accomplished with a mobile robot can be achieved with such a mobile manipulator. Examples of such tasks are fetching an object.

A mobile manipulator is not simply the combination of a mobile robot and a manipulator, rather, additional sensors and actuators, such as cameras, laser scanners, and pan-tilt units, are also added to these platforms. Each component has a different interface and different properties. Thus, system integration is another important aspect of mobile manipulator development. This integration is very closely related to the architecture of the robot. Defining interfaces for each component such that they can communicate with each other is an example of what such an integration task includes. A hybrid control architecture is one of the widely used architecture types for controlling a robot. In this architecture type, the components are grouped into several different layers based on the time scale or representation hierarchy. The low level controller, that controls the robot locomotion, runs in a higher frequency than the deliberative layer, that plans the robot actions.

A robot would normally receive commands from a human user either through a terminal through which the commands are entered or through speech. The latter is preferable since it is more natural for the user. Regardless of the approach being used, the robot will receive the command, for example “bring me the green tea from the side board to the dinner table”. Such a command consists of several different sub-tasks, some of which may be relevant to the mobile component while others may be relevant to the manipulator. A symbolic planner is required to execute such a command. It would generate sequences of sub-tasks consisting of several actions needed to successfully achieve the given goal. In terms of hybrid control architecture, the symbolic planner is implemented as the topmost component which is called the *deliberative layer*.

In this work, the deliberative layer of a hybrid control architecture is enhanced and named

"*Hybrid Deliberative Layer*" (HDL). This dissertation focuses on the planning task for a mobile manipulator. The planner presented here is enhanced with a knowledge-based system. It minimises the gap between the deliberative layer, the perception system and the control layer. It also enables more natural *human robot interaction* (HRI). The HDL system is used for solving the planning problem in the RoboCup domain for our robot, Johnny Jackanapes.

1.1. Motivation

Mobile robotics has been among the motivations for developing planning in AI. Often the main objective in mobile robotics is to move from location A to location B. The planner is used to plan a sequence of way points between the original location and the destination. In the mobile manipulator case, the robot will also have manipulation capability, thus, additional objectives are usually specified. For solving different objectives, the planner needs different models. Therefore, a robot will have several models to enable it to work in its environment.

Most planning applications are intended to be used by a specific robot. In the worst case, the model of the robot, which is used for planning purposes is difficult to use on other robot platforms even though the objective may be the same. This is due to the different commands used by each robot. Having a planning system which is portable enough and can work within several different environments is a major goal of the field.

One can model the environment in such detail that it contains almost every aspect of the different objects. Although this might sound trivial, modelling of the environment, or the domain as it is referred to, is far from a trivial task. The more complex the domain is, the more difficult it is to model. Additionally, it is also difficult to test whether the model is correct or not. Besides that, providing a complex domain to the planner might cause the planner to break. The planner extracts the actions from the domain, the larger the domain, the more time is needed to search within it. A planner that takes hours to plan is not acceptable, especially if the robot is to interact with humans.

Recall the task, "bring me the green tea from the side board to the dinner table". This task is stated very specifically. If this task were asked of another human being, it would most likely not include the location of the green tea. Given the large memory that a robot has, it should suffice to state the task as "bring me the green tea to the dinner table". For human beings such incomplete knowledge is no problem at all. However for the robot, it means that only partial information is available. The question is then how the robot can bridge this gap in its information in order to successfully generate a viable plan to achieve its goals.

The commands given by the user are presented in terms of symbolic objects, for example, green tea, side board, or dinner table. This symbolic information can be processed by the symbolic planner. As a consequence, the output of the planner is also given as a sequence of symbolic actions. However, such a symbolic action might not be useful for the control system. The control system needs a numerical value, in this case the poses of the objects. Additionally,

the perception system needs a reference, for example, a *Scale-Invariant Feature Transform* (SIFT) feature [Low04] of the object's texture. It is also possible to implement a local database which stores additional information about these objects. However, this solution might not be elegant and may prove difficult to maintain in a complex system like a mobile manipulator.

Adding a knowledge base (KB) to the system could minimise the gap between each component and provide a more elegant solution. The KB contains the domain model of the planning problem in addition to the information used to map symbolic objects and their numerical counterparts. The KB provides the other components with information, thus facilitating the updating of the information and ensuring its consistency for the whole system.

1.2. Problem Statement

Computation time and memory remain the main technical problems that a planning system deals with. The size of the planning problem is related to the required time to extract the solution plan and the required space (memory) for the planner. Although the development of microprocessor speed and memory in the computer has been staggering, these problems have remained. The HTN planning system is semi-intractable depending on the problem's description [NSE98]. The size of the problem is one of the reasons for this. The first problem that is tackled in this work is that of keeping the planning problem as small as possible while preserving its validity.

A KB system and a planning system can be categorised as a knowledge representation system and an inference system. The difference between these systems is found at several levels. Firstly, the planning system is used for searching the domain for a plan to achieve the goal. The KB system is a more general system which is intended to represent any knowledge and to reason about its stored knowledge. The reasoning process in the KB is not bound solely to any specific application. The planning system, is thus more specific than the KB system. Secondly, they use different languages or syntaxes to represent their knowledge. As such the second problem which is addressed in this work is how to efficiently combine the planning system and the KB into a coherent system.

The third problem addressed here is how to model the planning problem in the DL syntax. DL can be used to represent the knowledge of an application domain in a formal well-understood and structured way. Thus, it is possible to model the planning domain in DL. However, there is no fixed guideline on how to model an ontology or even on modelling a planning domain in the DL KB. The DL ontology should also serve as the main representation used by the robot.

Navigation is one of the capabilities of a mobile robot. Hence, the navigation domain is one of its planning domains. Adding a manipulator to the robot provides the robot with additional capabilities, such as pick-and-place task. Thus, the robot also needs the pick-and-place domain representation. A complex goal to manipulate an object may also require the robot to navigate. Therefore, the fourth problem addressed here is how to model the planning domains, such that they can be stored in the same DL model and also increase their re-usability.

1.3. Challenges

Gil [Gil05] surveyed work on combining expressive knowledge representation and planning techniques. She stated that real-world planning applications are naturally rich in knowledge. She summarises the challenges for a planning algorithm:

> "Typical planning systems use very small amounts of knowledge, and the representations of the domain or the planning tasks are typically not very expressive. A challenging area of future research is the integration of existing planning algorithms with rich representations of domain-specific knowledge about planning tasks and objectives, actions and events as part of a plan, and the complex relations among them." Gil [Gil05]

Although the surveyed works have used DL, they have not been incorporated within state-of-the-art planning algorithms.

Combining DL reasoning with HTN planning to form a deliberative layer in mobile robotics poses several challenges and difficulties such as:

- Defining the planning domain in the DL system.
- Reasoning about the objects described in DL in order to generate an HTN planning problem.
- Capturing real world information in the DL model and using it as input to the planning problem.
- Limiting the number of states in the planning problem while preserving its soundness.

Defining the HTN planning domain in the DL system is a crucial part of the solution presented here. However, planning domains consist of several terminologies and structures. In particular, the HTN planning domain consists of heuristics in the form of task networks that provide the planner with a solution to the problem. A process is needed that transfers this information to the DL representation. It is important that the relationships between tasks in the DL model are maintained such that the task networks can be reproduced.

Once the planning domains are stored in the DL model, the next challenge is how to retrieve the information from the model. Retrieving information as it is stored is one problem but a more interesting one is how this information can be used to gather additional information or even new knowledge from the model. Hence, the modelling of the planning domain has a direct effect on how the other information would be stored in the DL model.

The stored information must also be used to generate the planning problem. Hence, capturing the world model, where the robot is operating, is another challenge. Which states and how they are stored for enabling the reasoning system to infer the states for the planning problem is yet another challenge.

Last but not least, it is challenging to determine which information is relevant for a planning problem. For the planning system alone, it is not possible to determine which information can be omitted without affecting its results. A naive solution is to omit the states that do not appear in the goal states. This might work in some specific domains such as the simple blocks world problem. However, in the robotics domain, specifically in the navigation domain, the planner would not be able to determine which path is relevant for moving from one place to another. This information is available to the system after the planning process is completed. The challenge is how to model the DL terminologies in a way such that the irrelevant states can be filtered out from the generated planning problem.

1.4. Scope

This work focuses on the deliberative layer of a robot architecture. It deals with a novel approach by amalgamating a DL reasoner and an HTN planner. However, this work is neither developing nor improving any DL reasoner or HTN planner. It uses an existing DL reasoning system, namely *Pellet*, and the HTN planner JSHOP2. The work focuses on developing algorithms and concepts in DL for automatically generating the planning problem.

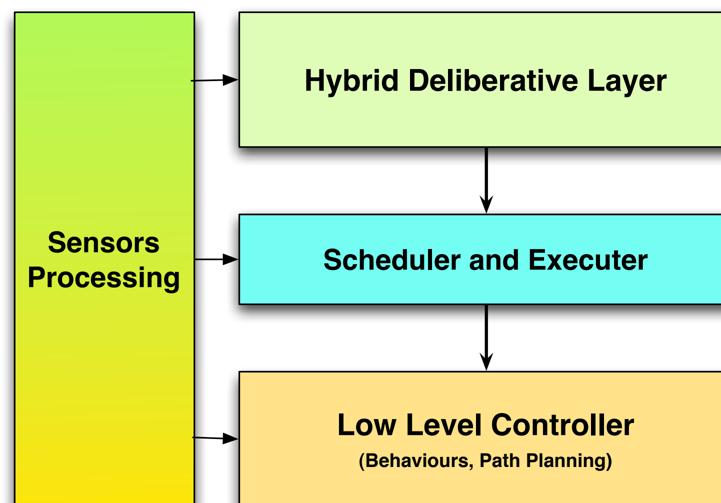


Figure 1.1.: Hybrid Deliberative Layer (HDL) in a robot control architecture.

Figure 1.1 depicts the HDL in a robot control architecture. This work covers neither the executive layers ("scheduler and executer" and "low level controller") nor the perception layer ("sensor processing") of the robot architecture. However, a case study is shown in Chapter 4 on how the HDL system could be integrated into existing systems, namely in our RoboCup robot. In other words, this work does not deal with robot control but rather with planning, representation and their extensions.

1.5. Previous Work

The HDL system is a novel approach that combines knowledge representation techniques and planning systems for application in the robotics field. A number of researchers have worked with planning for mobile robotics or representation in robotics. In this section, some of these approaches are briefly surveyed.

Beetz (2002) developed a plan representation for robotic agents [Bee02b]. It was triggered by the development of robotic agents, such as XAVIER [SGH+97], MINERVA [TBB+00], and RHINO [BCF+99, BAB+01]. These robotic agents use plans to improve their competence. His work emphasises a plan representation that not only provides guidelines for the robot but also handles the control structure of the robot. Thus, his proposed plan representation has four characteristics. Firstly, *Representational Adequacy*, the plans must have the expressiveness of *Reactive Plan Language* (RPL) [McD91, Fir87] and they should mirror the control pattern of the robot's behaviour. Secondly, *Inferential Adequacy* where the plan management mechanism must have inference techniques for performing various kinds of operations such as plan generation, plan elaboration, environment monitoring, and so on. Thirdly, *Inferential Efficiency* where plans must maintain economic inference and plan management. In case of failure, the planner should not re-plan the entire course of action. Fourthly, *Acquisitional Efficiency* where representations should support learning of sub-symbolic control processes and routine activities. This plan representation was implemented on the RHINO robot whose main role is that of the courier robot. Details of plan-based control of robotic agents are presented in [Bee02a].

Fox & Long (1998) developed a GraphPlan-based planner, named STAN [LF99]. Its name is due to the uses of the STate ANalysis technique to enhance its performance. The state analysis is accomplished using the Type Inference Module (TIM) [FL98]. TIM analyses a planning domain in order to reveal a domain's implicit type structure and various kinds of state invariants. This information is then used by the STAN planner to filter inconsistent states from the domain, thus speeding up the planning algorithms. Besides that, domain designers can use this information to check the consistency of the domain. TIM can be used by any planner regardless of its underlying architecture. However, originally TIM accepted only planning domains expressed in the basic language of STRIPS. An extension of TIM to support a subset of the ADL language is presented in [CFL02].

Asada (1989) developed dynamic semantic constraints for building a mobile robot's world model [AS89]. The system models the environment in a hierarchical form from sensor-based maps to global maps with both numerical and symbolic descriptions. Domain specific knowledge is used to organise semantic-constraints of objects and the relations between them as production rules. The domain-specific knowledge is described within the frame structures of object models.

Beeson et. al. (2007) developed the *Hybrid Spatial Semantic Hierarchy* (HSSH) for mapping, navigation and communication in robotics [BMM+07]. Kuipers introduced *Spatial Semantic Hierarchy* (SSH) in 2000 [KBG+00], but it provides only the abstractions for reasoning

about large-scale spaces. Hence, SSH is too large for the robot sensory horizon. The HSSH is optimised for a small-scale space by factorising the spatial reasoning about the environment. This representation is used at four different levels: *local metrical*, *local symbolic*, *global symbolic*, and *global metrical*. The *local metrical* level is used to reason about the geometry of the robot's immediate perceptual surroundings. The *local symbolic* models the local surround symbolically by giving navigational affordances of a place, i.e. the entrances and exits. The *global symbolic* level stores global environment symbolically, such as topological maps of a building. The *global metrical* adds metric information to the topological map such that the robot can estimate the travel distance from one place to another.

In 2007, Galindo et al. developed *Semantic Hierarchical Planning through World Abstraction* (SHPWA) [GFMGS07]. SHPWA is based on their previous work on hierarchical planning called HPWA. It runs with an embedded planner, such as Metric-FF. SHPWA uses a multi-hierarchical symbolic representation of its workspace. It entails two hierarchies that represent the environment from two different perspectives, namely *spatial perspective* and *semantic perspective* [GSC+05]. The *spatial hierarchy* symbolises particular elements of the environment. This process is accomplished by anchoring physical elements in the real world model which are recognised by the perception unit. The *semantic hierarchy* maps information from the *spatial hierarchy* into semantic concepts. The semantic reasoner is based on description logic DL. The multi-hierarchical structure is constructed using a mathematical model based on graphs called Multi-AH graph (MAH-graph) [GGFM04]. It has been proved that MAH-graph could reduce computational effort of robot operations, for example in the path-search task.

The *CoSy* project (2007), an EU-funded project, has looked into the integration of robotic systems for spatial understanding and situated interaction in indoor environments [ZJOMM+07]. The motivation behind this project is service robots for home and offices where the robot should understand and work together with humans. Thus, multidisciplinary approaches are necessary to fulfil the tasks. One of the aspects that are similar to this work is how they model the robot's environment in the conceptual spatial representations [ZOMMJ+08, OMMJZ+07]. This information is stored in four layers. The first layer is the *Metric Map* which contains a metric representation of the environment from the *Simultaneous Localisation And Mapping* (SLAM) component. The second layer is the *Navigation Map* which contains navigation graphs that model the free space and its connectivity. The third layer is the *Topological Map*, which groups some navigation graphs into areas. The areas are interconnected with others, i.e. through a doorway. The fourth layer is the *Conceptual Map*, which is the highest level of abstraction that provides the semantic meaning of the topological map (areas). This layer is implemented in OWL-DL.

The surveyed work, presented in [Gil05] is the most closely related to our approach. It discusses four main uses of DL for representing planning knowledge, namely: *object taxonomies* to reason about the planning state, *action taxonomies* to reason about action types at different levels of abstraction, *plan taxonomies* to reason about plan subsumption of partially ordered

plans, and *goal taxonomies* to reason with expressive representation of goals and their parameters [Gil05]. CLASP [DL96] is a system developed to reason about action taxonomies and action networks in a telephony domain. It was integrated into a software information system called LaSSIE [DBSB91]. CLASP represents the objects in their respective domains in DL, thus it was able to represent actions, plans, and goals that can be supported by DL reasoners. To reason about its action taxonomies, CLASP used CLASSIC's [BMPS+91] classifier. The plan taxonomies are used to organise, validate, and retrieve plans in a library. These plans are defined in CLASP's language. EXPECT is an interactive knowledge acquisition system for problem solving and reasoning [Gil94, SG95, GM96, BKRG01]. It uses LOOM [MB87, Mac91] as its reasoning system to exploit structured representation of goals and capabilities to support sophisticated matching during problem solving. Thus, it facilitates the generation of natural language paraphrases of problem-solving knowledge. EXPECT represents the objects in its respective domain in DL. It uses goal taxonomies for matching, which is represented as verb clauses using a case-grammar formalism. SUDO-PLANNER is a system developed for medical applications, which was designed to reason trade-offs in decision making under uncertainty. It uses plan subsumption to control the search during plan generation. The reasoning system was implemented using NIKL [KBR86]. Like CLASP and EXPECT, it represents the objects in DL for reasoning about its object taxonomies. The actions were represented as concepts, with action parameters as concept roles, and action constraints as role restrictions. Therefore, it can exploit the actions taxonomies by checking the inheritance and classification of their action types. It uses plan taxonomies to guide its plan generation using plan space search [Wel99]. PHOSPHORUS is a multiagent system for an office environment that is used as agent matchmaker and was implemented within the Electric Elves architecture [GR01, CGK+02]. It has the same goal and capabilities representations as EXPECT system. An application on a mobile robot, TINO, is shown in [DGINR96]. The robot uses DL to generate high-level plans, in which the representation domain (includes static axioms) is used to represent background knowledge that does not change as actions are executed. The dynamic axioms represent the changes caused by the actions. It uses CLASSIC as its knowledge representations system.

In order to improve the current approaches, some researchers incorporated semantic information in their work. [NWL+05] uses the semantic knowledge to improve the 3D-scan matching algorithms. Each 3D point is labelled with semantic information, such as floor, ceiling or object. This method has improved the scan algorithms and added valuable information to their results. [PHK04] uses a semantic symbol language to enhance human-robot interaction (HRI). The robot's environment and actions are expressed semantically as objects, places, and actions. Thus, the user's commands can be interpreted more accurately by the robot.

1.6. Contributions

The integration of existing planning algorithms with rich representations of domain-specific knowledge is a challenging future research area in AI planning [Gil05]. In her survey [Gil05], she presented previous work on combining planning techniques with Description Logics (DL) to reason about tasks, plans, and goals. Although the surveyed approaches and systems have used DL, it has not been incorporated in state-of-the-art planning algorithms or systems. This work presents a novel approach, that combines a DL reasoner and an HTN planner into a coherent system, to be used in the robotics domain. The system is called the *Hybrid Deliberative Layer* (HDL), as it extends the deliberative layer in a robot control architecture (see Figure 1.1). The reasoning capability of DL reasoners provides valuable advantages to the HTN planner. The user now has additional leverage to customise the planning problem such that an intractable planning problem is avoided. The HDL system increases the re-usability of any existing planning domain in its model and facilitates the use of the planning system for non-planning expert users.

The major contributions of this work are:

1. Modelling HTN planning domains in DL terminology. HTN planning uses a different kind of representation than the DL reasoner. Hence, modelling the HTN planning domain as DL concepts is essential for the HDL system. In this work, the planning domain is defined in DL such that the DL reasoner can use it for generating the planning problems.

This modelling gives us some advantages, three of them are explained in the following sentences; Firstly, the planning domains are now represented in a structured and well-understood manner in DL instead of the usual planning domain descriptions, which are in the form of a planning system's dependent formatting (e.g. PDDL, ADL, or LISP). Thus, non expert users can benefit from using the HDL system without having advanced knowledge in planning. Secondly, planning domains can be stored in the same DL model in the HDL system. Thus, the system can be used as the planning domains' repository, where the user can browse and choose which domain he or she wishes to work with. Thirdly, The HTN planning domains are represented as instances of the respective DL planning concepts, which means that each method and operator in a planning domain is also represented as an instance in DL planning concepts. Thus, domain designers can benefit from re-using available planning domains, or even some of their methods or operators only. They can compose new planning domains using available instances. This is shown as an example in Section 3.4.

2. Designing algorithms for generating HTN planning problems. The HDL system may have several planning domains in its model. The system must be able to generate HTN planning problems from its model based on the user's selection. Therefore, two algorithms are developed for this purpose. These algorithms are required to reason about the planning concepts in the HDL system. In the usual scenario, the user chooses any available planning domain from the set of instances of domain concepts in the DL model. The user can also choose instances of

planning objectives instead and the planning domain will be generated based on this selection. The first algorithm is used for generating the planning domain. Once the planning domain is ready, the second algorithm is then applied to generate the states to complete the planning problem. It analyses the selected planning domain and filters the relevant states from the DL model.

The idea of pre-processing the planning domain before the planning extraction has been previously applied in the STAN system [LF99]. It uses the Type Inference Module (TIM) to analyse the planning domain to reveal the domain's type structure and state invariants [FL98]. It is then able to filter inconsistent states from the domain using this information. However, TIM requires the planning domain to be expressed in STRIPS or ADL [CFL02]. Hence, this approach cannot be applied directly in the HDL system, because the planning domains are represented in DL concepts instead.

Some benefits can be gained from having these algorithms in the HDL system. First, users can select the planning problem based on the goals or tasks. Thus, the users do not have to know which domain needs to be selected in order to perform those tasks or goals. Second, valid planning problems are generated based on the user's selection, in spite of the number of stored planning domain instances in the system. Third, users can limit the size of the planning problem by selecting which DL concepts are involved in the given tasks. Thus, intractability can be avoided. For example, users can choose to use an instance of $Driveable\text{-}Room$ concept instead of $Room$ concept. Or even, only $Driveable\text{-}Room$ in building A instead, because the robot is located in that building. The performance of the HDL system and HTN planning with respect to the planning problem size is evaluated in Chapter 6.

3. Designing and implementing the HDL system. In this work, the HDL system is implemented using an off-the-shelf HTN planner, namely JSHOP2, as its planning system and *Pellet* as its DL reasoner. It shows that such readily available planning systems and DL reasoners can be fused into a coherent system where benefits from both systems are gained. Any other HTN planning system or DL reasoning system can also be used with our proposed approach. The presented methods might also be used for integrating any other planning system besides HTN with similar approaches and some effort.

In this work, the HDL system's implementation is presented through a number of use cases. Two robotics domains, namely the navigation domain and the pick-and-place domain are presented and explained in detail. In addition, the well known AI planning domain, blocks world, is also solved using the HDL system. One might also implement other planning domains in the HDL system by using the same approach as shown in these use cases.

A real robot implementation is shown as a use case in the RoboCup@Home domain. The HDL system is applied to our robot, Johnny Jackanapes, and the RoboCup@Home domain is modelled in the system as DL concepts. It is demonstrated how the HDL system is applied to an existing robot system, namely Johnny, to solve its planning problems. Beside that, the benefits

of using the HDL system as a common KB system in Johnny are described. Therefore, the HDL system can also be implemented in other robots, which need a deliberative layer to plan their actions.

1.7. Outline

Chapter 2 presents the concept and basic principle of the *Hybrid Deliberative Layer* HDL. The chapter also defines the ontology of the *Hierarchical Task Network* HTN planner in the HDL system. The algorithms' use for generating the HTN planning problem are detailed.

Chapter 3 explains the manner in which the HDL system is used through a number of examples. Two planning problems in robotics, one from the navigation domain and the other from the pick-and-place domain, are used. For this purpose, the modelling of the actors and objects concepts in the HDL system are also presented. These are necessary for capturing the environment and generating the planning problem automatically. The examples show the advantages of the HDL system, in particular the re-usability of the existing domains and hierarchical representation of them in the model.

Chapter 4 shows an integration example of the HDL system and an existing robotic system. This robotic system is a mobile manipulator that is used for the RoboCup@Home competition. The robot is called "Johnny Jackanapes". In this chapter, two integration benefits are presented. Firstly, HDL as a common knowledge representation system for all components in Johnny is shown. Secondly, HDL as a deliberative layer that plans Johnny's actions.

Chapter 5 gives an example of the well known AI planning problem, blocks world, which is implemented using the HDL system. The domain is an existing one, which is included as one of the examples in the JSHOP2 source. Therefore, the HDL system is not restricted to be used in the robotics domains but also in available planning domains.

Chapter 6 presents the complexity analysis of the HDL system. It is supported by empirical results obtained by using two domains, which are presented as examples. The results compare the HDL system and the HTN planning system. Hence, the benefit of using the HDL system is shown quantitatively.

Chapter 7 discusses related issues of the HDL system. These issues are benefits of the HDL system, anomalies in the HTN blocks world, dealing with inconsistency, and qualitative comparisons with related systems.

Chapter 8 concludes this dissertation. It gives an overview of possible future work aimed at expanding the HDL system.

2. The Hybrid Deliberative Layer

In this chapter, a brief introduction to robot control architectures is presented. It highlights the need for a deliberative layer in robotics and its role as a planning system of sorts. In large domains, computation time is in danger of exploding, as the size of the domain grows. A novel approach that amalgamates *Hierarchical Task Network* (HTN) planning with *Description Logic* (DL) reasoning is presented to keep the planning domain size within limits. With this approach the planning domain is modelled using the DL representation instead of being modelled directly in the planning representation. Finally, algorithms to automatically generate planning problems from the DL system are shown.

2.1. System Architecture

An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators [RN03, Chapter 2]. Thus, a robot is also an agent. It has some sensors, e.g. camera, laser scanner, and odometry. It also has actuators, usually motors which are used to move the wheels or the manipulator. These components need to interact with each other. This makes robot systems quite complex. Therefore, the architecture needs to structure the system in a way that helps managing this complexity. It can be considered as a product of software engineering. Currently, there exists no single architecture that fulfils the requirements of all applications [KS08].

2.1.1. A Brief Overview of Robot Control Architectures

The oldest approach which has been used to compose robot systems is the *Hierarchical Paradigm*, often referred to as the *SENSE, PLAN, ACT* (SPA) approach. The *Hierarchical Paradigm* is sequential and orderly. The sensors first sense the environment then this data is processed and used to plan the next action. Finally, the action is performed using the actuators [Mur00, Chapter 2].

Two architectures were based on the *Hierarchical Paradigm*: the *Nested Hierarchical Controller* (NHC) and the *NIST Realtime Control System* (RCS) [Mur00, Chapter 2]. The need for a world model or a knowledge base is already present in the *Hierarchical Paradigm* approach. In the NHC system, the knowledge base is located in the *SENSE* block. The information in the knowledge base is shared with the planning component. It is clear that the knowledge base contains the state of the robot, objects, and environment, which are needed by the planner to

generate the plan. The major disadvantage with this paradigm is a result of the sequence and ordering of the signal flow. The plan usually needs more time to compute the actions. As a consequence, the output signal to the actuator is delayed, resulting in a 'shakey' or 'jittery' behaviour. This is not acceptable in dynamical environments such as that of a kitchen or office, or one which includes other agents.

The problems with the SPA paradigm led scientists to search for other approaches. One of the resulting approaches was the Reactive Paradigm. One notable work that was based on this paradigm was developed by Brooks [Bro86]. The main idea behind his *Subsumption Architecture* (SA) was to decompose tasks vertically instead of horizontally as in the *Hierarchical Paradigm*. This vertical decomposition is built on layers of interacting finite-state machines. Each layer connected sensors to actuators directly with some data processing. Thus, the robots can react to a dynamic environment quickly. Some scientists had been looking to *ethology* (the study of animal behaviour) and relating their observations to robotics. To them, the layer which connects sensors and actuators, is known as the behaviour. Hence, the architecture is called *Behaviour-based Architecture* or *Behavioural Robotics* [Ark98]. Another reactive approach is based on the *potential fields* methodology [Mur00, Chapter 4]. This methodology was introduced by Khatib and was intended to control a *PUMA 560* manipulator [Kha86]. Other reference architectures based on the *Reactive Paradigm* are *Motor Schemas* [Ark87] and *Dual-Dynamics* [JC97].

In *Behaviour-based Architecture*, each behaviour can be active at certain time points. Especially, several behaviours, can be active in parallel. However, some behaviours might give complementary commands to the actuators, which could cause *local minima* problems. To overcome these problems, additional behaviours are programmed to control the behaviours. In *Subsumption Architecture*, these behaviours can inhibit or suppress other behaviours. In *Dual-Dynamics*, these behaviours are implemented as high-level behaviours, which have a similar function to those in subsumption architecture. The effect of having several behaviours which are active at the same time is known as *emergent* behaviour. Nevertheless, *Behaviour-based Architecture* remains reactive and the lack of a planning system makes it difficult to achieve complex tasks. In order to enable complex tasks, one *Motor Schema* architecture, namely the *Autonomous Robot Architecture* (AURA), added a navigation planner and a plan sequencer [KS08]. [HSMH04] shows an example of a behaviour-based system with a target-oriented approach for performing office tasks, thus overcoming this limitation. In order to enable complex tasks, scientists were driven to work on new paradigms that incorporate a planning system but remain reactive. Such a paradigm is referred to as the *Hybrid Deliberative/Reactive Paradigm* or *Hybrid Control Architecture*.

The *Hybrid Control Architecture* can also be described as *PLAN*, then *SENSE-ACT* [Mur00, Chapter 7]. It adds a deliberative layer (*PLAN*) on top of the reactive layer (*SENSE-ACT*). Thus, it is able to perform complex tasks with the help of the planner while still preserving the reactive property from the behaviour-based layer. Figure 2.1 compares the reactive and deliberative

approaches. Due to the layered implementation, this paradigm is also known as the layered robot control architecture [KS08]. The first AI robot, *Shakey*, at Stanford University used this paradigm [Nil84]. Some recently implemented robot architectures have also been based on this architecture. These include *Autonomous Robot Architecture* (AURA) [Ark87], *Reactive Action Package* (RAP) [Fir94], *Task Control Architecture* (TCA) [Sim94], *Saphira* [KMRS97], and *DD&P* [SH02]. Details of the *3T* architectures can be found in [KS08].

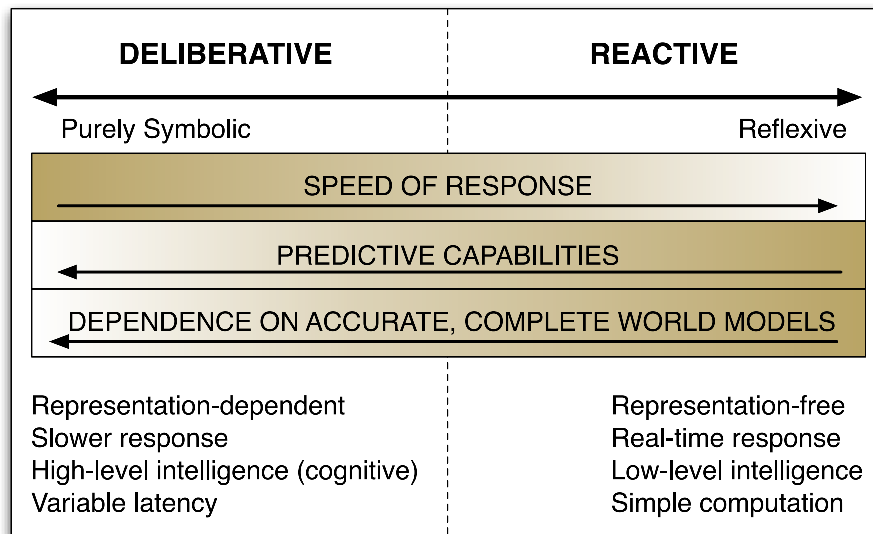


Figure 2.1.: Robot control system spectrum [Ark87, Chapter 1].

As shown in Figure 2.1, the deliberative layer uses a purely symbolic representation and the reactive layer is free to choose its representation model. The reactive layer is usually represented in a way that facilitates the translation into actuators' commands. Thus, there is a need for a common world model or knowledge system which shares information between these layers. In the *Saphira* architecture, a *Local Perceptual Space* (LPS) is used as a cartographer and to improve the quality of the robot's overall behaviours [Mur00, Chapter 7]. TCA uses *Global World Models* to share common knowledge between several components; such as path planning, navigation, and obstacle avoidance [Sim94]. Nevertheless, the world model is mainly intended to share information between different layers for consistency.

2.1.2. The Hybrid Deliberative Layer Architecture

The term "*hybrid*" has been used often in robotics to represent the combination of two approaches to improve the current one. In this work, a novel approach is introduced which amalgamates the knowledge base system and the planning system in the deliberative layer. Hence, it is called the *Hybrid Deliberative Layer* (HDL).

Figure 2.2 depicts the HDL within a layered robot control architecture. Let us first view the HDL block as a normal deliberative layer so as to analyse how the HDL interacts with other

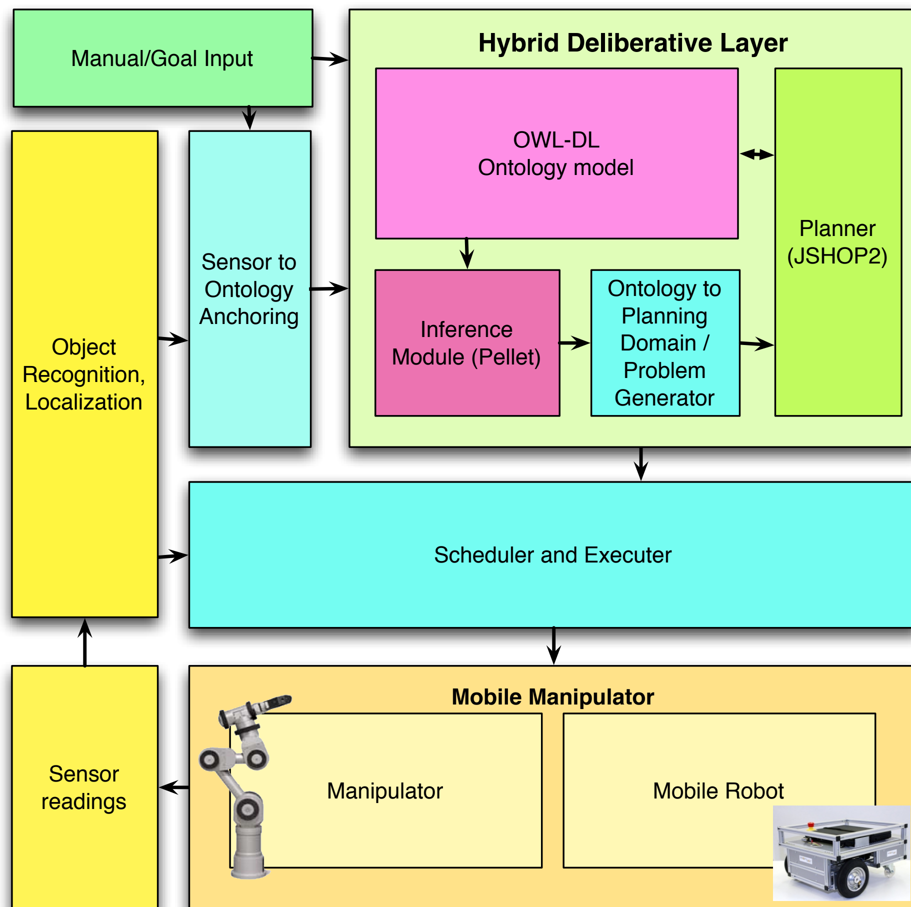


Figure 2.2.: HDL in a layered robot control architecture.

components. The HDL generates sequences of actions to achieve the objectives. These actions are represented in symbolic form, hence, an intermediate block which includes the *scheduler and executor* is needed to translate the actions into commands that the reactive layer can use. Additionally, it should allocate the necessary resources for command execution and then monitor this execution. The bottom layer, *mobile manipulator*, is the reactive part of the overall system. This bottom layer could be a manipulator, a mobile robot or even a mobile manipulator as shown in the figure. For the sake of clarity, the *sensor reading* is drawn separately from the *mobile robot*. The *object recognition and localisation* module processes numerical information from sensors into quasi-symbolic information, e.g. robot pose, object pose, and object type. The output is fed back to the *scheduler and executor* in order to maintain the reactivity of the overall system and to the *sensor to ontology anchoring* to be processed further. The *sensor to ontology anchoring* block translates its input information into a symbolic one, which is understandable to the HDL. The user can interact with the system by adding or manipulating information directly or by giving a goal to the system.

The architectural approach, as shown in Figure 2.2, is an example of how HDL could be

used in the robotic scheme. In general, any robotic system can use HDL for the deliberative layer in their system. This work focuses on the HDL. Two robotics domains, namely navigation and pick-and-place, are implemented using the HDL and discussed in Chapter 3. A case study showing a possible implementation of the HDL for a real robot is presented in Chapter 4.

2.2. HDL Components

As shown in Figure 2.2, there are four subcomponents in the HDL block. These are *OWL-DL ontology model*, *Inference Engine*, *Ontology to Planning Domain/Problem Generator*, and *Planner*. Each of these subcomponents is detailed in the following section. The main concept of the HDL is explained in more detail in Section 2.3.

2.2.1. Knowledge Bases

What is a knowledge base and why should it be implemented in the deliberative layer? According to the *Knowledge Representation Hypothesis* of philosopher Brian Smith, [BL04, Chapter 1] systems for which the *intentional stance* is grounded by design in symbolic representations are called *knowledge-based systems* and the symbolic representations involves their *knowledge bases* (KBs). Thus, the deliberative layer is the right place to implement KBs. This is also seen in Figure 2.1. An *intentional stance* is the most abstract layer of three abstraction layers which are proposed by Daniel C. Dennett [Den87]. This layer is normally found in the software or minds level that concern things like belief, thinking, goals, hopes and intent. Therefore, having knowledge-based systems in our agent is necessary in order to provide it with these features. It enables the agent to have an intention, for example "move to charging station" because it believes that its battery is almost empty.

2.2.1.1. A Brief Overview of Knowledge Representation and Description Logic

The knowledge that the agent has is abstract. How do we represent this knowledge in a machine-understandable manner? Figure 2.3 shows an example of how a figure can be represented in several different forms. The picture represents a cartoon character, namely "Bart Simpson". In the LaTeX form, it is represented as "\Bart". The "0x5C42617274" is a hexadecimal representation and "010111000100...0" is the machine representation on the disk or in memory. Using this example, it can be seen that knowledge can be represented at different levels. These levels are categorised by Ron Brachman (1979) as follows [Sow00, Chapter 3]:

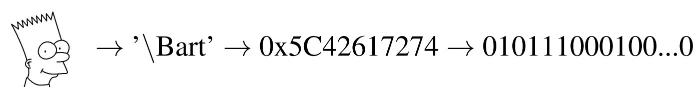


Figure 2.3.: A figure with some different representations.

1. *Implementational:* The level of data structures such as atoms, pointers, lists and other programming notions.
2. *Logical:* Symbolic logic with its propositions, predicates, variables, quantifiers and Boolean operators.
3. *Epistemological:* A level for defining concept types with subtypes, inheritance, and structuring relations.
4. *Conceptual:* The level of semantic relations, linguistic roles, objects, and actions.
5. *Linguistic:* The level of arbitrary concepts, words, and expressions of natural languages.

A robot has an explicit representation of parts or aspects of its environment [HC08]. How should this knowledge be represented? [HC08] explains the answer by looking into two aspects, *epistemological adequacy* and *computational adequacy*. The first aspect decides whether the formalism can express the knowledge precisely and compactly. The second aspect decides whether it can be inferred effectively and efficiently. These aspects are like two sides of one coin, having a formalism which is very rich and expressive typically leads to an intractable or undecidable inference problem, and vice versa [HC08].

Logical representation is the level above the implementational, hence, AI logicists believe that knowledge is best represented using formal logic. They postulate that *First-Order Predicate Logic* (FOPL), along with its modifications, is a language that is particularly well-suited to capturing reasoning, due to its expressivity, its model-theoretic semantic and its inferential power [LMP08]. A more detailed analysis of FOPL's adequacies in the robotics domain is found in [HC08]. Other limitations of FOPL, such as its inability to represent *transitive closure* or *numerical existentialism*, are explained in detail in [LMP08].

Though logical representation is able to represent knowledge that can be effectively inferred, it is still not expressive enough to represent knowledge from the linguistic level, such as in words or natural language. Another drawback of FOPL representation is that its representation is *flat*. It can not represent structured knowledge (*epistemological level*) or semantic relations (*semantical level*). Some scientists searched for other kinds of representations to overcome these problems. The *Frame formalism* is one approach that is used to represent knowledge with their structure in frames. The knowledge is captured by grouping objects into more generic ones (*generic frames*) and the object itself as an *individual frame* [BL04, Chapter 8]. The *frame formalism* is capable of representing knowledge in a hierarchy and has some inheritance which is slightly similar to that of *Object-Oriented Programming* (OOP). The scientists worked on both the formalism and the reasoning issues. They mapped these representations into logic, such that they could benefit from the inferential power of logic. Originally, Marvin Minsky (1975) described a frame as "a network of nodes and relations". *Knowledge Language One* (KL-ONE), which describes a network notation in its representation, was designed by Ronald Brachman (1979). KL-ONE uses *definitional logic* or *terminological logics* as its inference engine [Sow00]. Although there are still more knowledge representation formalisms that have been introduced, their description is beyond the topic of this work. Further reading on these

topics can be found in [Sow00, AGPC04, BL04].

Description Logics (DL) is currently state-of-the art in knowledge representation form. It is based on *semantic networks* and *frame systems*. [BHS08] explains DL as follows:

> The name *description logics* is motivated by the fact that, on the one hand, the important notions of the domain are described by concept *descriptions*, i.e., expressions that are built from atomic concepts (unary predicates) and atomic roles (binary predicates) using the concept and role constructors provided by the particular DL; on the other hand, DLs differ from their predecessors, such as semantic networks and frames, in that they are equipped with a formal, *logic*-based semantics.

DL can be categorised a mature knowledge representation. [BHS08] describes the development of DL through four distinct phases:

- *Phase 0* (1965-1980) is the pre-DL phase. In this phase *semantic networks* and *frames* were introduced for representing knowledge. Ronald Brachman's KL-ONE, which structures the knowledge in inheritance networks, is the first DL system.
- *Phase 1* (1980-1990) was concerned with the implementation of DL systems, such as KL-ONE [BS85], CLASSIC [BMPS+91], K-REP [MDW91], KRYPTON [BFL83], BACK [PSKQ89], NIKL [KBR86], and LOOM [MB87, Mac91]. A *structural subsumption algorithms* for inferring the knowledge was used by these systems.
- *Phase 2* (1990-1995), during which a new algorithmic paradigm was introduced into DLs, namely *tableau-based algorithms*. The first systems with these algorithms were KRIS [BH91] and CRACK [BFT95]. They showed a performance gain resulting from the use of these new algorithms.
- *Phase 3* (1995-2000) saw the development of inference procedures for very expressive DLs. Some DL reasoners were introduced such as FACT [Hor98], RACE [HM99] and DLP [VHH+05]. Section 2.2.2 describes DL reasoners in more detail.
- *Phase 4* (2000-present) sees industrial-strength DL systems with very expressive DLs are under research. DLs are used in applications such as Semantic Web, Medical or Bio-Informatics. This work is one such application that uses DL for robotics.

A DLs' knowledge base consists of two main components, namely the *terminological knowledge* (*TBox*) and the *assertional knowledge* (*ABox*) as shown in Figure 2.4. The *TBox* contains the general *concepts* of the domain and their relationships, which together are also known as the *upper ontology*. In the *frame formalism*, the *TBox* can be seen as a container for *general frames*. The *concepts* denote sets of individuals, such as $Robot$, $Door$, $Location$, $Container$ and their unary predicate. In the *frame formalism*, *concepts* represent classes of objects. They can build a hierarchy of sub-concepts, for example $Room \sqsubseteq Location$ which means $Room$ is a sub-concept of $Location$. *Roles* correspond to a binary relationship, such as $adjacentTo$ one $Location$ to another one. By using *concepts* and *roles*, one captures the knowledge and

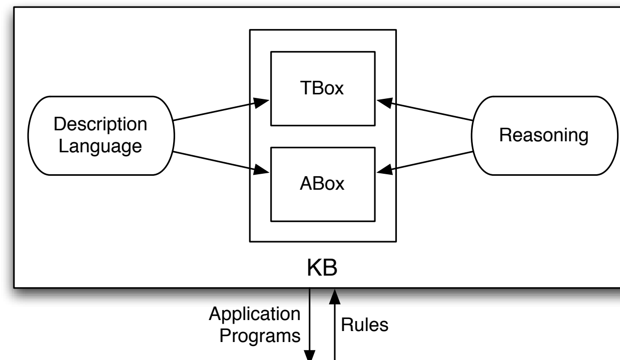


Figure 2.4.: Architecture of a knowledge representation system based on Description Logics [BN03].

represents it in the DL formalism. The other box, *ABox*, contains the individuals or instances of the defined *concepts* and *roles*. In a *frame formalism*, the *ABox* can be seen as a container for *individual frames*. Some examples of *ABox* are $\texttt{Room(C102)}$, $\texttt{adjacentTo(C102, C101)}$ or $\texttt{isOpen(DoorC102)}$. Basically, DL systems represent the ontology with three components: *concepts*, *roles* and *individuals* [AGPC04].

Concepts and *roles* are described with terminological descriptions that are built from a set of constructors, such as conjunctions, disjunctions, value restriction and negation. The combination of these constructors defines different DL languages, as shown in Table 2.1. A basic DL is defined as DL $\mathcal{ALC}$ and it is the most widely used DL reasoning service [BHS08]. The $\mathcal{U}$nion and $\mathcal{E}$xistential restrictions can be represented with $\mathcal{C}$omplement (negation). Hence, DL $\mathcal{ALC}$ is basically equivalent to DL $\mathcal{ALCUE}$. DL $\mathcal{ALC}$ extended with transitive roles or DL $\mathcal{ALC}_{R+}$ is often represented as DL $\mathcal{S}$. The DL can be very expressive and are closely related to *Modal Logic*. [BHS08] describes how the $\mathcal{ALC}$-concepts are described in modal $\mathbf{K}$ formula.

Semantic Web is one of the application domains that try to represent knowledge in such a way that enables it to be shared among users over the Internet. It began with the introduction of the *eXtensible Markup Language* (XML) which enabled more flexible and platform independent data structures. One of the first frameworks to model a semantic web is the *Resource Description Framework* (RDF) (Lassila and Swick, 1999). It captures the semantics by using the relation of subject, predicate and object, known as the RDF triple [Pow03]. Although it seems very simple, it is a very powerful one. Several triples, described in the RDF/XML format, can describe a complex property of an object. The RDF data model consists of three components: *resources*, *properties*, and *statements*. However, this data model can not describe the relation between properties and resources. Therefore, the RDF Vocabulary Description Language (Brickley and Guha, 2003) was introduced. It is also known as the *RDF Schema* (RDFS). The *Ontology Inference Layer* (OIL) is an extension language based on RDF(S) (Horrocks et. al. 2000). It is derived from frame-based knowledge representation techniques and uses DLs to gain clear semantics over RDF(S). Figure 2.5 depicts the levels of OIL. OIL can also express

Table 2.1.: Common DL constructors and their correspondence with language name [AGPC04].

<table>
<tr><th>Construct</th><th>Syntax[1]</th><th colspan="4">Language[2]</th></tr>
<tr><td>Concept</td><td>A</td><td rowspan="4">$\mathcal{FL}_0$</td><td rowspan="7">$\mathcal{FL}^-$</td><td rowspan="8">$\mathcal{AL}$</td><td rowspan="11">$\mathcal{S}$</td></tr>
<tr><td>Role name</td><td>R</td></tr>
<tr><td>Intersection</td><td>$C \sqcap D$</td></tr>
<tr><td>Value restriction</td><td>$\forall\, R.C$</td></tr>
<tr><td>Limited existential quantification</td><td>$\exists R$</td><td rowspan="3"></td></tr>
<tr><td>Top or Universal</td><td>$\top$</td></tr>
<tr><td>Bottom</td><td>$\bot$</td></tr>
<tr><td>Atomic Negation</td><td>$\neg A$</td><td colspan="2"></td></tr>
<tr><td>Negation[3]</td><td>$\neg C$</td><td colspan="3">$\mathcal{C}$</td></tr>
<tr><td>Union</td><td>$C \sqcup D$</td><td colspan="3">$\mathcal{U}$</td></tr>
<tr><td>Existential restriction</td><td>$\exists\, R.C$</td><td colspan="3">$\mathcal{E}$</td></tr>
<tr><td>Number restrictions</td><td>$(\leq n\ R)\ (\geq n\ R)$</td><td colspan="3">$\mathcal{N}$</td><td></td></tr>
<tr><td>Nominals</td><td>$\{a_1 \ldots a_n\}$</td><td colspan="3">$\mathcal{O}$</td><td></td></tr>
<tr><td>Role hierarchy</td><td>$R \sqsubseteq S$</td><td colspan="3">$\mathcal{H}$</td><td></td></tr>
<tr><td>Inverse Role</td><td>R^-</td><td colspan="3">$\mathcal{I}$</td><td></td></tr>
<tr><td>Qualified number restriction</td><td>$(\leq n\ R.C)\ (\geq n\ R.C)$</td><td colspan="3">$\mathcal{Q}$</td><td></td></tr>
</table>

[1] A refers to atomic concepts, C and D refers to any concept definition, R refers to atomic roles and S refers to role definitions

[2] $\mathcal{FL}$ is used for structural DL languages and $\mathcal{AL}$ for attributive languages [BCM+03]. $\mathcal{S}$ is the name used for the language $\mathcal{ALC}_{R+}$, which is composed of $\mathcal{ALC}$ plus transitive roles.

[3] $\mathcal{ALC}$ and $\mathcal{ALCUE}$ are equivalent languages, since union ($\mathcal{U}$) and existential restriction ($\mathcal{E}$) can be represented using negation ($\mathcal{C}$).

enumeration, hence it can be categorised as DL $\mathcal{SHIQ}$ language. A combination of *DArpa Markup Language* (DAML) and OIL, which is known as DAML+OIL, was developed as an extension of RDF(S). This language extends RDF(S) directly instead of building a layer over it. DAML+OIL is an extended DL $\mathcal{SHIQ}$. It can also represent datatypes and nominals. The most recent semantic web language is the *Ontology Web Language* (OWL), which has been developed by W3C Web-Ontology Working Group. OWL is a derivative of DAML+OIL which is built upon RDF(S) [AGPC04].

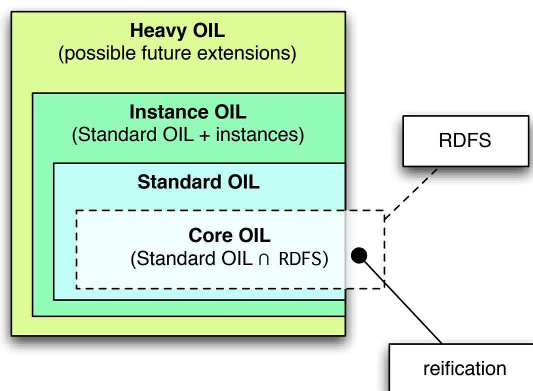


Figure 2.5.: Layers of OIL [AGPC04].

Table 2.2.: OWL constructors [BHS08].

Constructor	DL syntax	Example
intersectionOf	$C_1 \sqcap ... \sqcap C_n$	$Human \sqcap Male$
unionOf	$C_1 \sqcup ... \sqcup C_n$	$Doctor \sqcup Lawyer$
complementOf	$\neg C$	$\neg\ Male$
oneOf	$\{x_1 ... x_n\}$	{john,mary}
allValuesFrom	$\forall\ P.C$	$\forall\ hasChild.Doctor$
someValuesFrom	$\exists\ R.C$	$\exists\ hasChild.Lawyer$
hasValue	$\exists\ R.\{x\}$	$\exists\ citizenOf.${USA}
minCardinality	$(\geq n\ R)$	$(\geq 2\ hasChild\ )$
maxCardinality	$(\leq n\ R)$	$(\leq 1\ hasChild\ )$
inverseOf	R^-	$hasChild^-$

OWL has three expressive sublanguages, namely OWL-Lite, OWL-DL, and OWL-Full. OWL-Lite is the least expressive of these. It supports primarily classification hierarchies and simple constraints. It supports only cardinality constraints with a value of 0 or 1. Hence it is categorised as DL $\mathcal{SHIN}$. OWL-DL has the maximum expressiveness and still retains computational completeness. Thus, it is named after the *Description Logics* DL. It is equivalent to DL $\mathcal{SHOIN}$. OWL-Full is not really a sublanguage. It permits users to use maximum expressiveness and syntactic freedom of RDF, however it cannot guarantee the tractability of the reasoner. Every valid OWL-Lite is also a valid OWL-DL and every valid OWL-DL is a valid OWL-Full [MvH04, DSB+04, BHS08].

Just like RDF, OWL describes the concepts and roles from DL syntax in XML format. Table 2.2 summarises the OWL constructors with their relations to DL syntax. A snapshot of XML serialisation for expressing $Human \sqcap Male$ would be written as follows [BHS08]:

```
<owl:Class>
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Human">
    <owl:Class rdf:about="#Male">
  </owl:intersectionOf>
</owl:Class>
```

In the same way $(\geq 2\ hasChild\ )$ would be written as:

```
<owl:Restriction>
  <owl:onProperty rdf:resource="#hasChild">
    <owl:minCardinality
      rdf:datatype="&xsd;NonNegativeInteger">2
    </owl:minCardinality>
  </owl:onProperty>
```

```
</owl:Restriction>
```

The two examples above have shown how the OWL constructor represents DL syntaxes in the XML format. They have also shown how some parts are represented using the RDF schema.

2.2.1.2. KB in HDL Implementation

In this work, OWL-DL is used for representing knowledge that the robot has. For editing and viewing the ontology which is written in OWL, PROTEGE is used [HKR+04]. PROTEGE is a free, open source ontology editor that is based on Java. However, PROTEGE is only used for defining the ontology. The system runs independent of it. The DL reasoners are detailed in the following section.

2.2.2. Description Logic Reasoner

One of the differences between the approach presented here and others is the reasoner. HDL is equipped with a DL-reasoner, such that it can retrieve the stored information from the DL-model and also reason about the knowledge therein. Including a reasoner in the robot enables it to perform advanced deliberative actions rather than simply reactive actions that are based on sensor readings.

Most of the DL-reasoners are actually *decidable* fragments of FOPL. However, some of them provide operators such as transitive closure of roles or fixpoints that require second-order logic. DL-reasoners are closely related to Modal Logics, but they have been developed independently [BHS08]. The DL reasoner works on both conceptual knowledge (*TBox*) and assertional knowledge (*ABox*). The basic reasoning on concept expressions is subsumption, which is written as $C \sqsubseteq D$. In the previous subsumption example, concept D (the subsumer) is considered more general than concept C (the subsumee). In OOP programming, C can be seen as being a subclass of D. Besides subsumption, there are still some reasoning services that can be applied to a *TBox*, such as “consistency”, “satisfiability”, “equivalence”, and “disjointness”. In *ABox* reasoning, the main tasks are to determine whether an individual is an instance of a given concept or whether the role assertion is valid. The reasoner can also identify inconsistency within the *ABox*. For example, the *TBox* defines $Male \sqcap Female \sqsubseteq \bot$ but the *ABox* contains the assertions: `Male(Johnny)` and `Female(Johnny)` which are inconsistent.

One major challenge for the reasoner is tractability. There are several reasoning techniques which have been researched to reason about more expressive DL representations. A few other reasoning techniques are tableau-based reasoning, automata-based reasoning, and structural reasoning [BHS08]. In this work, a new technique is not introduced, neither is an existing one improved. Instead, an available DL-reasoner is used. Examples of such reasoners are *Pellet* (an open source OWL-DL reasoner written in java) [SPG+07], FACT++ (a new generation of FACT - *Fast Classification of Terminologies*) [TH06], and *RacerPro* (RACER - *Renamed* ABOX *and Concept Expression Reasoner*) [HM01, HM03]. In this work, *Pellet* is used.

2.2.3. Planner

Planning is an ambiguous word in robot control architecture. On the low level controller, one might use the term planning to mean path planning. However, in this work, planning is used for the deliberative layer, which works on the symbolic level. Hence, the planning or planner terms used in this work refer to the symbolic planner unless otherwise noted.

The planner is also a reasoning system. It has a specific purpose, namely to extract sequences of actions required to achieve the given objectives. The first robot planning system that was used for "*Shakey*" the robot was called STRIPS. It worked under specific constraints: only one action can occur at a time, actions are instantaneous, and nothing changes except as the result of planned actions [BL04]. In STRIPS, actions are represented as operators. The operators are specified by pre- and post-conditions. The environment is defined in the world model. These operators act upon the world model and trigger events which change the state from one into another. The planning process is completed when a state which is implies the goal conditions is found.

Figure 2.6 depicts how the planner can influence the system Σ by executing the plans on the controller block. In general, the planner needs three inputs, the description of Σ, the initial state, and the objectives. Σ is also known as a state transition system. Formally, it is a 4-tuple: $\Sigma = (S, A, E, \gamma)$ [GNT04, chapter 1] where:

- $S = \{s_1, s_2, ...\}$ is a finite or recursively enumerable set of states;
- $A = \{a_1, a_2, ...\}$ is a finite or recursively enumerable set of actions;
- $E = \{e_1, e_2, ...\}$ is a finite or recursively enumerable set of events;
- $\gamma = S \times A \times E \rightarrow 2^s$ is a state transition function.

Modelling an exact or dynamic environment in Σ is difficult to achieve in the current planning system. Hence, some restrictions or assumptions are needed for the planning model, such as [HC08]:

- *Finiteness* (the domain has only finitely many objects, actions, events and states)
- *Information completeness* (the planner has all relevant information at planning time)
- *Determinism* (all actions have deterministic effects)
- *Instantaneousness* (actions take effect immediately and have no relevant duration)
- *Idleness* (the environment does not change during planning)

These assumptions help the planning algorithms to work more efficiently. However, a restrictive model would not be able to capture a real world dynamic environment. The generated plan might be invalid at execution time. For example: state of a door, s_{door_1} might be detected

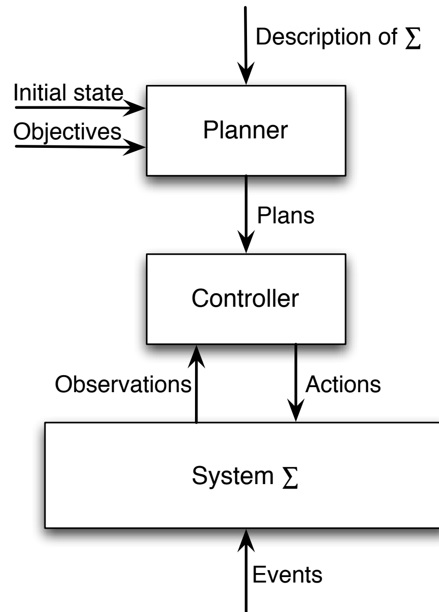


Figure 2.6.: A simple conceptual model for planning [GNT04, chapter 1].

and modelled as an open door. However, during the planning time it might change such that during execution time the door is not open and the plan might fail. Most robot actions are not instantaneous, but rather need some time to execute. This might be in seconds or minutes.

Modelling the real world to create the planning domain is not a trivial problem. The first step is to take a snapshot of the current state or freeze it and write as the planning domain. Then the planner extracts possible actions to achieve the objectives. This process might be time consuming depending on the size of the domain. Real world problems might have lots of states in the domain. Thus, the sequence of actions could become inconsistent with the environment. One possible way out of this problem is to model the domain with a focus on keeping it small. However, this might lead to another problem, namely the "*qualification problem*". This is the problem of not knowing which states are relevant for the particular problem. Hence, it might not properly model the real world. One approach to cope with this problem is to divide the goal into subgoals. The overall goal might be seen as a hierarchy. The extension approach to the hierarchical approach was implemented with "ABSTRIPS". This approach minimises the size of the problem.

Another approach, used to make the search faster, is the use of heuristics. A *Hierarchical Task Network* (HTN) is a planning scheme which uses heuristics and a hierarchical approach to solve planning problems. The heuristic in this case is defined by the human who designs the planning problem. In this work, the planning system is implemented using an HTN planner.

2.2.4. Planning Problem Extractor

DL and HTN use different representation formalisms. Thus, it is not possible to have a unified

representation for both. In addition, DL might have additional knowledge which is not relevant for the planning system. For example, it might have the robot specific commands, descriptions of some objects, and so on. The planning problem extractor's task is to translate the DL representation into planner-specific syntax. Hence, it filters and processes the information which is relevant for the HTN planner. The rest of this chapter presents the algorithms that are used for extracting planning problems from the DL representation.

2.3. Concept

The basic idea of this approach is to split tasks between DL reasoning and HTN planning. DL representation is more expressive than HTN planning representation. The planning domain as well as basic HTN planning concepts are modelled in DL. In fact, so are the sensor readings, detected objects, actuators, and the robots. The DL reasoner is then used for automatically extracting a tailored representation for each and every concrete planning problem, and handing this problem to the planning system. Note that the DL model can be much larger than the planning domain. It may include any information that the system should have; in particular, it may span information that otherwise would have been modelled in separate planning domain descriptions for planning efficiency reasons. HTN planning is dedicated to finding a solution within the representation extracted by DL, which is reduced by facts, operators, and events found irrelevant by the filtering process accomplished by the DL reasoner. Figure 2.7 illustrates what is represented in the DL formalism and how this information is processed by the DL reasoner to produce a filtered planning domain for the HTN planner.

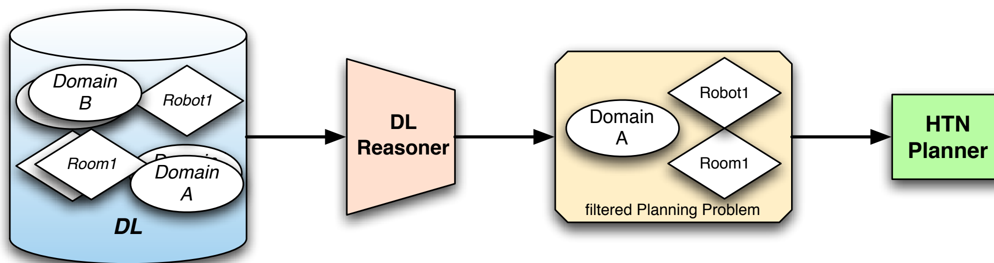


Figure 2.7.: A schematic diagram on how the planning problem is passed from the DL representation to the HTN planner.

2.4. Ontologies

The first step in designing a KB is choosing the ontological categories. [Sow00, Chapter 2] describes some ontological categories, two of which are *microworlds* and *Cyc categories*. The *microworlds* category contains a small number of concepts that are designed for a single application. The *Cyc categories*' ultimate goal is to accommodate all human knowledge. Its name

is taken from the syllable of the world *encyclopedia*. *microworlds* is related to the bottom up approach, that is usually practised by programmers, while the *Cyc categories* are related to the top down approach, that is usually followed by philosophers.

DL defines knowledge in two fractions, namely *TBox* and *ABox*. Hence, the planning domain needs to be modelled in these fractions as well. Figure 2.8 depicts two *ABoxes* and two *TBoxes* for representing planning knowledge in DL representation. The first pair of *TBox* and *ABox* represent the HTN planning ontologies while the other pair represent the states of the planning problem that are needed by the HTN planner. In the actual implementation, there is in fact only one pair of *TBox* and *ABox*. This distinction is made here in order to more clearly understand the concept of how the ontologies are described in HDL.

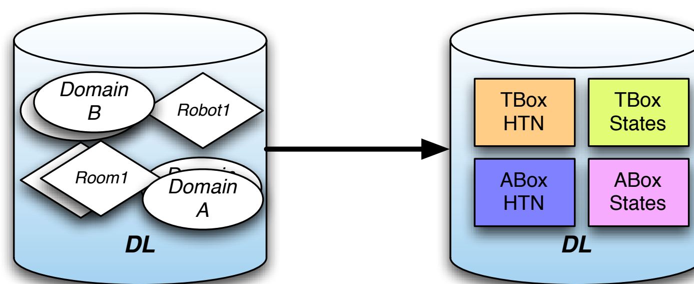


Figure 2.8.: Planning knowledge representation in DL *TBoxes* and *ABoxes*.

In the HDL system, the ontologies are implemented using the bottom up approach or the *microworlds* approach. The HTN planning concepts are currently implemented for the HTN planning system. Therefore, additional implementations of other planning systems might require changes in the planning concepts. The HDL system is applied to three different domains, namely the robotics domain in Chapter 3, the RoboCup domain in Chapter 4, and Blocks World domain in Chapter 5. If one wishes to use the *Cyc categories* instead, this can be accomplished by customising the $useState$ (see Section 2.5.2) in the method and operator instances of the respective planning domain.

2.4.1. HTN Planning Definition

In order to qualify the DL reasoner to generate an HTN planning problem, the DL ontologies must include knowledge about the HTN planning itself, i.e., how actions, methods, and tasks are defined in HTN. In this section, we recapitulate some HTN planning definitions, which are required to build a complete planning problem.

In HTN planning, a planning problem is defined in a slightly different way than in classical planning. In classical planning, the goal is modelled as condition of states. In HTN, instead of describing the planning problem in terms of states, it uses methods and operators. The goal is defined as method(s) that the system should perform.

The way a planning problem is decomposed in HTN planning is similar to how humans think. It decomposes the problem or task recursively into smaller tasks until atomic actions are reached. Consider a simple example, a task such as "bring a coke to the guest on the armchair". This task is decomposed into smaller subtasks such as "navigate to sideboard", "grasp coke", "navigate to armchair", and "release coke". Figure 2.9 shows how the task is decomposed into task networks. Every subtask may consist of several operators (e.g. move forward, move absolute, etc). The dashed arrows in the figure show the ordering of each subtask. The goal can be achieved by executing all operators with their ordering in the depth first manner.

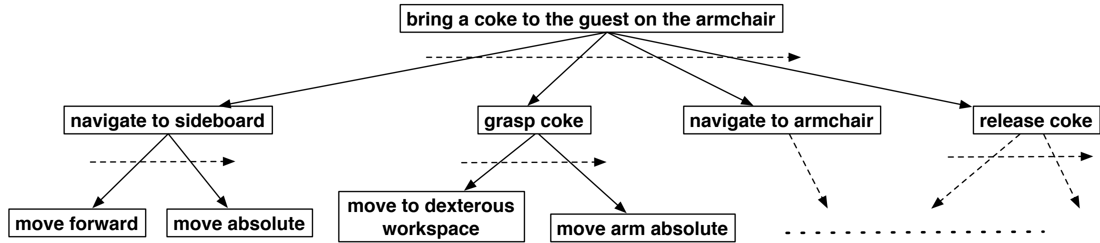


Figure 2.9.: Illustration on how HTN expands the planning problem into subtasks.

The HTN planner requires two inputs, one for describing the environment and the other for the objectives. The environment is represented in the planning domain $\mathcal{D}$ and the objectives are represented as a task network in the planning problem $\mathcal{P}$. Formally, HTN planning is defined as follows [GNT04, Ch. 11]:

Definition 2.1. An HTN *planning domain* is a pair

$$\mathcal{D} = (O, M),$$

where O is a set of operators and M is a set of methods. *[GNT04, Def. 11.11]*

Definition 2.2. An HTN *planning problem* is a four-tuple

$$\mathcal{P} = (s_0, w, O, M),$$

where s_0 is the initial state, w is the initial task network and $\mathcal{D} = (O, M)$ is an HTN planning domain. *[GNT04, Def. 11.11]*

Definition 2.3. An HTN *method* is a four-tuple

$$m = (name(m), task(m), subtasks(m), constr(m)),$$

where $name(m)$ is a unique method name and its variable symbols, $task(m)$ is a nonprimitive task, and $(subtasks(m), constr(m))$ is a task network. *[GNT04, Def. 11.10]*

Definition 2.4. A *task network* is a pair

$$w = (U, C),$$

where U is a set of task nodes and C is a set of constraints. *[GNT04, Def. 11.9]*

A task network, stated simply, is a pre-defined sub-plan, consisting of a partial-ordered sub-tasks. See [GNT04, Ch. 11.5] for details.

An HTN planning algorithm is supposed to deliver a sequence of actions $\pi = \langle a_1, ..., a_n \rangle$ for a given planning problem $\mathcal{P} = (s_0, w, O, M)$ such that a_1 is applicable in s_0, and π accomplishes w in the sense that it is a properly instantiated version of w. The solution of an HTN planning problem is defined in *[GNT04, Def. 11.12]*.

2.4.2. HTN Planning *TBox*

There is no one "correct" way or methodology to develop ontologies [NM01]. There are always alternatives. A model that is good for one application might not be suitable for other applications. Developing an ontology is an iterative process, where each iteration might improve the ontology itself. Concepts in ontologies should be close to physical or logical objects of the domain of interest. It should also define the relationship between them. The ontologies that are modelled in this work are a result of this iterative process, however they may not be applicable to other similar domains. Nevertheless, the ontologies presented here are sufficient to capture the planning domain and states of an HTN planning problem into the HDL system.

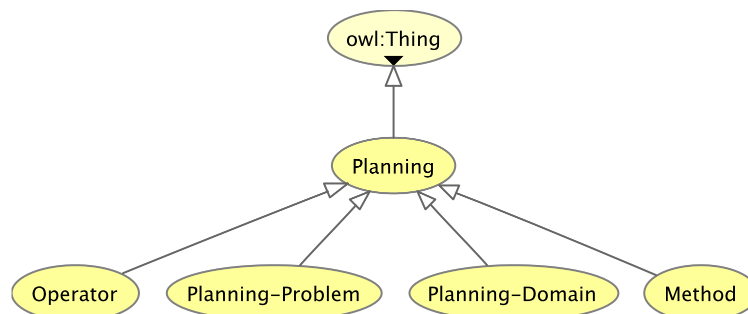


Figure 2.10.: HTN planning ontology.

According to the formal HTN planning definitions in the previous section, four concepts are used in the DL representation. These concepts are sufficient for representing HTN planning problems in DL. These four concepts are $Planning\text{-}Domain$, $Planning\text{-}Problem$, $Method$, and $Operator$. Figure 2.10 depicts them in relation to the $Thing$ concept.

Every concept in the DL ontology must be a sub-concept of "$Thing$", as in OOP where each class inherits directly or indirectly from the "object" class. The "$Planning$" concept is defined for organisational purposes. It distinguishes planning concepts from the other concepts which might coexists in the HDL system. It has four sub-concepts namely "$Planning\text{-}Domain$", "$Planning\text{-}Problem$", "$Method$" and "$Operator$". These concepts are pairwise disjoint or mutually disjoint.

Planning Domain

$Planning\text{-}Domain$ corresponds to Def. 2.1. $\mathcal{D} = (O, M)$ and is modelled in DL as follows:

$$\begin{aligned} Planning{-}Domain \sqsubseteq\ & Planning \sqcap \\ & \exists\ hasMethod.Method \sqcap \\ & \exists\ hasOperator.Operator \end{aligned}$$

Definition 2.5. An instance of an HDL *planning domain* is a triple

$$\mathbb{d} = (name(\mathbb{d}), hasOperator(\mathbb{O}), hasMethod(\mathbb{M}))$$

where $name(\mathbb{d})$ is a unique domain name, $\mathbb{O}$ is a set of operator instances in HDL, and $\mathbb{M}$ is a set of method instances in HDL.

Let us use an example where a planning domain $d_{domain1} = (\mathcal{O}, \mathcal{M})$ in HTN, where $\mathcal{O} = \{o_1, o_2, ..., o_i\}$ and $\mathcal{M} = \{m_1, m_2, ..., m_j\}$, is defined in HDL as follows:

$$\mathbb{d}_{\texttt{domain1}} = (\texttt{domain1}, hasOperator(\{\texttt{o1,o2,...,oi}\}), hasMethod(\{\texttt{m1,m2,...,mj}\}))$$

The {o1,o2,...,oi} and {m1,m2,...,mj} are HDL instances of operators $\mathcal{O}$ and methods $\mathcal{M}$. The $\mathbb{d}_{\texttt{domain1}}$ is asserted (or inserted into the *ABox* of the HDL system) either as depicted in Figure 2.11 or as follows:

```
Planning-Domain(domain1),
  hasMethod(domain1, m1),
  hasMethod(domain1, m2),
  ...,
  hasMethod(domain1, mj),
  hasOperator(domain1, o1),
  hasOperator(domain1, o2),
  ...,
  hasOperator(domain1, oi)
```

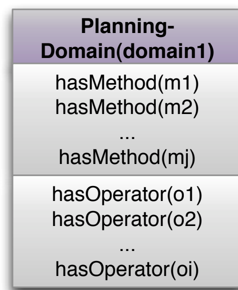


Figure 2.11.: An example of HDL planning domain instance for `domain1`.

Planning Problem

According to Def. 2.2, $Planning\text{-}Problem$ consists of four tuples $\mathcal{P} = (s_0, w, O, M)$. The $Planning\text{-}Problem$ is represented in DL as follows:

$$\begin{aligned} Planning{-}Problem \sqsubseteq\ & Planning \sqcap \\ & \exists\ hasDomain.Planning{-}Domain \end{aligned}$$

The domain $\mathcal{D}$ consists of operators O and methods M, hence, two of four elements in $\mathcal{P}$ are implicitly represented through the $Planning\text{-}Domain$. However, there are still two missing components, namely initial states s_0 and initial task network w. The trick here lies in the definitions of the $Method$ and $Operator$ concepts. Together, they make up the task network. Both concepts have an additional property, namely "$useState$", that has the purpose of generating the initial state s_0. The task network w, which is the goal of the planning problem is provided by the user, thereby completing the planning problem $\mathcal{P}$. Two algorithms for extracting a complete planning problem are presented in Section 2.5.1.

Methods

The HTN method is defined in Def. 2.3. In HDL, $Method$ is defined as follows:

$$\begin{aligned} Method \sqsubseteq\ & Planning \sqcap \\ & \exists\ hasMethod.Method \sqcap \\ & \exists\ hasOperator.Operator \sqcap \\ & \geqslant 1\ useState \sqcap \\ & \leqslant 1\ shop2code \end{aligned}$$

As one might already notice, there is one additional property besides $useState$, namely $shop2\text{-}code$. Each method can have at most one $shop2code$ property which contains the planning specific syntax (in this case SHOP2) for the given method. The $useState$ properties contain the states that are needed by the corresponding method.

Definition 2.6. Let m be a method in HTN and $w = (U, E)$ be a graph that represents a task network of m. The HDL system's immediate successors of m are

$$Succ_1(m) = \{u' | u' \in U \wedge u' \neq m \wedge (m, u') \in E\}$$

where u' can either be a method or an operator in the HTN.

$succ(m)$ is the set of all immediate successors of m. Let us take an example method $m1$, whose task network is depicted in Figure 2.12. The immediate successor of $m1$ is $succ(m1) = \{m1, m2, m3, o1, o2\}$. Although $succ(m1)$ contains method $m1$, the HDL system's immediate

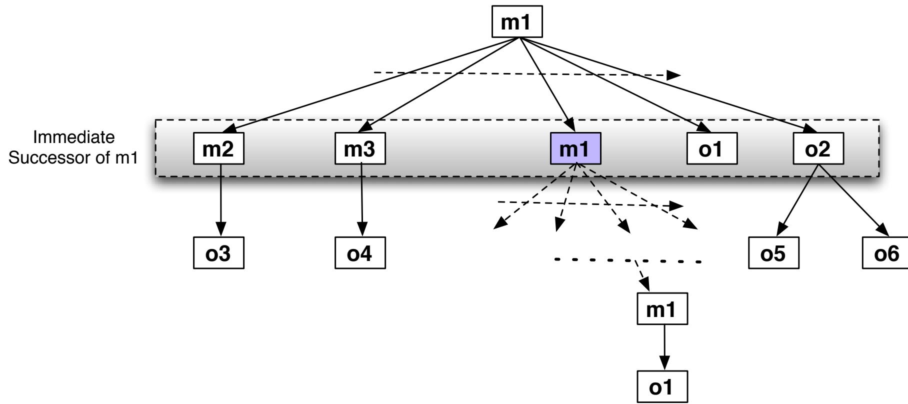


Figure 2.12.: Task network of method $m1$.

successor, $Succ_1(m1)$, does not. To better understand Definition 2.6, let us redraw the task network, but without its ordering, as shown in Figure 2.13. This figure shows the interconnections between method and operator in the task network. The graph is defined as $G = (U, E)$ where:
$U = \{m1, m2, m3, o1, o2, o3, o4, o5, o6\}$
$E = \{\{m1, m1\}, \{m1, m2\}, \{m1, m3\}, \{m1, o1\}, \{m1, o2\}, \{m2, o3\}, \{m3, o4\}, \{o2, o5\}, \{o2, o6\}\}$
Applying Definition 2.6 to G produces $Succ_1(m1) = \{m2, m3, o1, o2\}$.

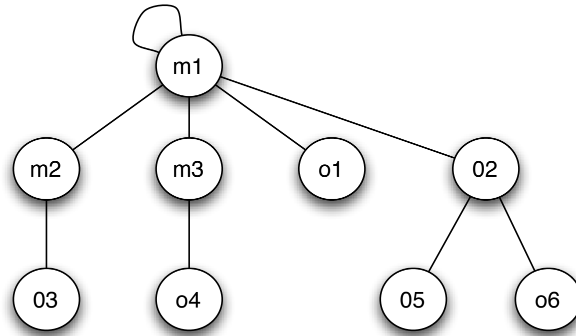


Figure 2.13.: Graph representing the task network of method $m1$.

Definition 2.7. An instance of an HDL *method* is a five-tuple

$$\mathbb{m} = (name(\mathbb{m}), hasOperator(\mathbb{O}), hasMethod(\mathbb{M}), useState(\mathbb{S}), shop2code(shop_2))$$

where $name(\mathbb{m})$ is a unique method name, $\mathbb{M}$ is a set of method instances in $Succ_1(\mathbb{m})$ ($\mathbb{M} \subset Succ_1(\mathbb{m})$), $\mathbb{O}$ is a set of operator instances in $Succ_1(\mathbb{m})$ ($\mathbb{O} \subset Succ_1(\mathbb{m})$), $\mathbb{S}$ is a set of preconditions states in m, and $shop_2$ is a JSHOP2 representation of method m.

Definition 2.8. Let m be a method in an HTN and $precond(m) = \{p_1, p_2, ..., p_i\}$ is the set of all preconditions in m. The set of preconditions in the HDL system is

$$\mathbb{S} = HDLize(precond(m))$$

HDLize is a process for translating the preconditions from the HTN representation into the one used by the HDL system. The details of this process are presented in section 2.5.2.

Using the example shown in Figure 2.12, assume that $\mathbb{S}_m = \{s1, s2, s3\}$ and $shop_2 = sc2$, method $m1$ is then asserted either as depicted in Figure 2.14 or as follows:

```
Method(m1),
  hasMethod(m1, m2),
  hasMethod(m1, m3),
  hasOperator(m1, o1),
  hasOperator(m1, o2),
  useState(m1,s1),
  useState(m1,s2),
  useState(m1,s3),
  shop2code(m1, sc2)
```

Method(m1)
hasMethod(m2) hasMethod(m3)
hasOperator(o1) hasOperator(o2)
useState(s1) useState(s2) useState(s3)
shop2code(sc2)

Figure 2.14.: An example of the HDL method instance for `m1`.

Operators

Just like methods, operators are also defined with two additional properties. In SHOP2, operators can be non-primitive or primitive. A non-primitive operator might consist of other operators whereas primitive operators are distinguished by the exclamation mark before the first letter of the operator's name, e.g. (!visit). In order to make the $Operator$ generic, it is described as follows:

$$\begin{aligned} Operator \sqsubseteq\ & Planning\ \sqcap \\ & \exists\ hasOperator.Operator\ \sqcap \\ & \geqslant 1\ useState\ \sqcap \\ & \leqslant 1\ shop2code \end{aligned}$$

Definition 2.9. Let o be a non-primitive operator in an HTN and $w = (U, E)$ be a graph that represents a task network of o. The HDL system's immediate successors of o are

$$Succ_1(o) = \{u' | u' \in U \wedge u' \neq o \wedge (o, u') \in E\}$$

where u' is an operator in the HTN.

Definition 2.10. An instance of the HDL *operator* is a four-tuple

$$\mathbb{o} = (name(\mathbb{o}), hasOperator(\mathbb{O}), useState(\mathbb{S}), shop2code(shop_2))$$

where $name(\mathbb{o})$ is a unique operator name, $\mathbb{O}$ is a set of operator instances in $Succ_1(\mathbb{o})$ ($\mathbb{O} \subset Succ_1(\mathbb{o})$), $\mathbb{S}$ is a set of preconditions states in o, and $shop_2$ is a JSHOP2 representation of method o.

Definition 2.11. Let o be a non-primitive operator in an HTN and $precond(o) = \{p_1, p_2, ..., p_i\}$ be the set of all preconditions in o. The set of preconditions in the HDL system is

$$\mathbb{S} = HDLize(precond(o))$$

Using the operator $o2$ in Figure 2.12 as example, the $Succ_1(o2) = \{o5, o6\}$. Assuming the $\mathbb{S}_{\mathbb{o}} = \{s1, s2, s3\}$ and $shop2 = sc2$, the operator $o2$ is asserted either as depicted in Figure 2.15 or as follows:

```
Operator(o2),
  hasOperator(o2, o5),
  hasOperator(o2, o6),
  useState(o2,s1),
  useState(o2,s2),
  useState(o2,s3),
  shop2code(o2, sc2)
```

Figure 2.15.: An example of the HDL operator instance for o2.

2.5. Reasoning

In the previous section, HTN planning has been defined as DL terminology concepts. Now, assume that these concepts are instantiated in the DL's *ABox*. How would this knowledge then be extracted from the DL system back into a valid HTN planning representation? A reasoner is needed to perform this action. Figure 2.16 illustrates how the reasoning process is applied to the DL system in order to extract an HTN planning problem and an HTN planning domain.

HDL uses two reasoning phases. The first phase is the DL reasoner, which is implemented using Pellet. However, any other DL reasoner would also be suitable. The output of this phase is fed to algorithms responsible for generating a planner specific syntax (phase 2). In this implementation, the planner that is used is SHOP2. The DL reasoner infers instances in the *ABox* according to the defined concepts in the *TBox*. This information is then processed by the algorithms which are presented below.

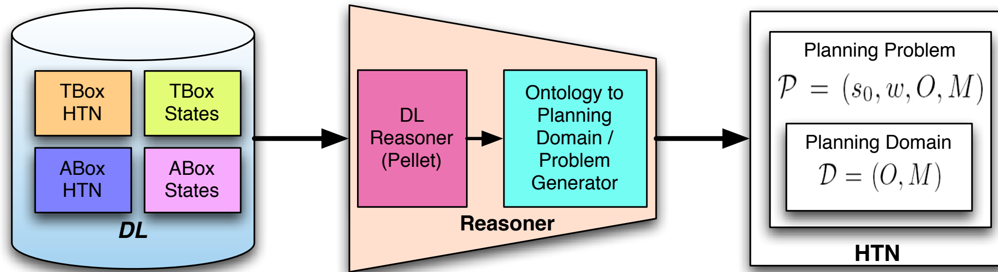


Figure 2.16.: A reasoning process over DL representation to extract a concrete HTN planning problem.

2.5.1. Algorithm for HTN Planning Domain/Problem Generator

The planning concepts that are described in Section 2.4.1, are contained in the *TBox* in the DL formalism. Instances of these concepts must be asserted in the *ABox* before a usable planning problem can be deduced from the DL system.

Let $\mathbb{D}$ denote $Planning\text{-}Domain$, $\mathbb{O}$ $Operator$, and $\mathbb{M}$ $Method$. Let d_i represent domain instances, so $\mathbb{D} = \{d_1, ..., d_n\}$ denotes all domain instances in the knowledge base. The d_i are independent domains, but a domain could be a subset of another one.

A domain d consists of operators $\mathbb{O}$ and methods $\mathbb{M}$, such that $d = \{\mathbb{O}, \mathbb{M}\}$. A non-primitive operator is an operator that is defined in terms of other operator(s). This is visible in the $Operator$ concept, which uses the $hasOperator$ property. Therefore, an operator is formalised as $o_x = \{o_1, ..., o_n | o_x \neq o_1, ...o_x \neq o_n\}$. Analogously, a method is formalised as $m_x = \{m_1, ..., m_n, o_1, ...o_p | m_x \neq m_1, ..., m_x \neq m_n\}$.

One could define the planning goal or *initial task network* in two ways: first, by choosing a domain $d \in \mathbb{D}$ and then selecting the goal from d; second, by choosing from $\mathbb{M}$. Goals are represented as methods in the knowledge base.

Using the first technique, three of the four planning problem elements in Def. 2.2 (page 28) are given, namely, $d_x = \{m_1, ..., m_n, o_1, ...o_p\}$ and w. Algorithm 2.1, *extractInitialState*, takes domain d as parameter and assembles the required state s_0 from the knowledge base. Its complexity is linear in the size of the sets of methods/operators and the ontology. Specifically, let $m = (|\mathbb{M}| + |\mathbb{O}|)$ and $n = (|Thing| - |Planning|)$, then the run time is $O(mn)$.

Using the second technique, only the goal or *initial task network* w is defined. It may

Algorithm 2.1: extractInitialState($input$)

```
Require: input = d
Ensure: output = s_0 (hashtable)
foreach m in input do
    forall state in m.useState do
        output ⇐ +state
    end
end
foreach o in input do
    forall state in o.useState do
        output ⇐ +state
    end
end
```

consist of several methods, hence $w = \{m_1, .., m_n\}$. Algorithm 2.2, *generateDomain*, recursively derives the planning domain from w, starting from the methods in w and unfolding them recursively. However, the complexity of this algorithm is linear in $n = (|\mathbb{M}| + |\mathbb{O}|)$, since it is called only once for each method and operator. Calling Algorithm 2.1 with the generated planning domain will return the *initial state* s_0.

Either way, a complete planning problem for the HTN planner can be assembled from the knowledge base. Our system ensures that the generated planning problem is a valid HTN planning problem. Note that, in addition to running the two algorithms, the *ABox* needs to be filled with the recent domain facts.

Algorithm 2.2: generateDomain($input$)

```
Require: input = m or o
Ensure: output = d
output ⇐ +input
if input is a method then
    foreach m from input.hasMethod do
        if output does not contain m then
            output ⇐ +generateDomain(m)
        end
    end
end
foreach o from input.hasOperator do
    if output does not contain o then
        output ⇐ +generateDomain(o)
    end
end
```

Let us use a simple example shown in Figure 2.17 which depicts instances of a planning domain for a simple navigation domain. Details of this domain are presented in Chapter 3. The user can either choose an instance of the $Planning\text{-}Domain$ or $Method$. If the user chooses

the first option, the planning domain contains all the needed methods and operators. However, if the user chooses an instance of $Method$, e.g. `navigate2`. Then applying Algorithm 2.2 produces a planning domain $\mathbb{d}_{\texttt{navigate2}} = \{\mathbb{m}_{\texttt{navigate2}}, \mathbb{m}_{\texttt{navigate}}, \mathbb{o}_{\texttt{drive-robot}}, \mathbb{o}_{\texttt{visit}}, \mathbb{o}_{\texttt{unvisit}}\}$. Once the domain is defined, Algorithm 2.1 is applied and produces $\mathbb{S}_{\texttt{navigate2}} = \{\ Robot, Room\ \}$. The HDL system uses the DL reasoners to infer the model and automatically produce the planning problem from them.

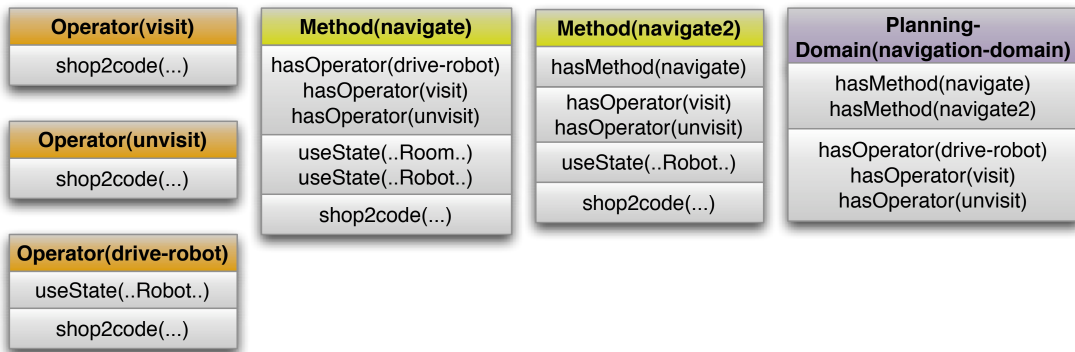


Figure 2.17.: Simple navigation domain example.

2.5.2. Algorithm for Generating SHOP2 Code

The algorithms shown in the previous section generate lists of tuples of the HTN planning problem. For the HTN planning, JSHOP2 [Ilg06] is used. JSHOP2 is a JAVA implementation of SHOP2. SHOP2 requires two inputs in order to decompose a plan, namely the planning-domain description and the planning-problem description. In SHOP2, the planning domain descriptions are presented as follows [Ilg06]:

```
(defdomain domain-name (d1 d2 .  .  .  dn))
```

where d_j is either a method, an operator or an axiom. However, we do not use axioms in our DL terminology. Nevertheless, an axiom can be represented as an instance of a ground operator. Listing 2.1 shows an example of the basic domain which is included in the JSHOP2 distribution. The domain has two operators (pickup and drop) and a method (swap).

Listing 2.1: SHOP2 basic domain description.

```
; This extremely simple example shows some of the most essential
; features of SHOP2.

(defdomain basic (
  (:operator (!pickup ?a) () () ((have ?a)))
  (:operator (!drop ?a) ((have ?a)) ((have ?a)) ())

  (:method (swap ?x ?y)
    ((have ?x) (not (have ?y)))
```

```
    ((!drop ?x) (!pickup ?y))
    ((have ?y) (not (have ?x)))
    ((!drop ?y) (!pickup ?x)))
  )
)
```

The basic domain is represented as the following tuple $d_{basic-domain} = \{m_{swap}, o_{pickup}, o_{drop}\}$. Each method and operator has a property *shop2code* as defined in the *TBox* concept. The SHOP2 planning domain description is generated by concatenating the *shop2code* property that is stored with each instance of $Method$ or $Operator$ in a valid planning domain $d_x = \{m_1, ..., m_n, o_1, ...o_p\}$. This domain is either generated from Algorithm 2.2, or from an instance in the $Planning$-$Domain$. According to the defined *TBox* concept, each instance of a method or operator has at most one SHOP2 syntax. Consider an example from the basic domain above, o_{pickup}. *shop2code* is then described as "`(:operator (!pickup ?a) () () ((have ?a)))`". The proper header and footer are added to the concatenated results, hence the generated planning domain description is a valid planning domain in SHOP2 syntax.

The planning problem description is defined as the following [Ilg06]:

```
(defproblem problem-name domain-name
      ([a1,1 a1,2 ...  a1,n])T1 ...
      ([am,1 am,2 ...   am,n])Tm)
```

where each $a_{i,j}$ is a ground logical atom and each T_i is a task list. Listing 2.2 shows an example of a problem description for the basic domain description that is defined in the Listing 2.1. The planning problem is pretty simple in this example. It has only one ground logical atom and only one objective in the task network. However, generating the problem description from the DL representation is more complex than generating the domain description.

A planning problem is defined as the four tuple $\mathcal{P} = (s_0, w, O, M)$; O and M are represented implicitly in "*domain-name*" of the problem description. The ground logical atom represents a state in planning. In the HDL system, it is generated automatically from the $useState$ property of a method or an operator in the DL representation. Methods or operators can have more than one $useState$.

Listing 2.2: SHOP2 basic problem description.

```
(defproblem problem basic
  ((have kiwi))
  ((swap banjo kiwi))
)
```

The fundamental question is how the states are generated automatically from s_0. s_0 is the result of Algorithm 2.1. The $useState$ consists of a template in SHOP2 syntax. The *HDLize* process captures the preconditions of a given method or operator and writes them in HDL using a template. The template is defined as triples and described as follows:

(predicate <?val1> <?val2>);?val1=I:*Concept*<,?val2=P:*property*>

The template contains two parts which are separated by a semicolon. The first part is the SHOP2 syntax of the template. The ?val1 and ?val2 are replaced by the value of s_0. The second part defines which concept and which property are used for replacing the SHOP2 template. Let us look into the basic problem description, which has only one ground logical atom namely (have kiwi). The *shop2code* for this ground logical atom is written as follow (have ?val2);?val1=Actor,?val2=have. The term ?val1 is used for retrieving the instance of the concept *Actor*. However ?val1 does not appear in the SHOP2 syntax. ?val2 evaluates the property "have" from the instance that appears in ?val1. This value is then substituted with the SHOP2 syntax in the template.

The fourth tuple of the planning problem, namely the *task network*, is defined by the user as the planning objectives. Having all components from the algorithms and user input, a valid planning problem description for the SHOP2 planner can be built. As in the planning domain, it needs a suitable header and footer in addition to a list of ground logical atoms and a task network. Chapter 3 gives some examples on how these are assembled.

3. HDL Systems in the Robotics Domain

In the previous chapter, the terminological concept of HTN planning was defined. In order to generate a valid HTN planning problem for SHOP2, a terminological concept that models the environment needs to be defined. In addition, the HTN planning operators, actors and objects must be instantiated and inserted into the corresponding *ABox*. In this chapter, a generic method for modelling the environment in HDL and filling the HDL system is introduced. Two implementations from the robotics domain are also presented, namely the robot navigation domain and the pick-and-place domain.

3.1. Modelling the HTN Planning Problem in the HDL System

There is no fixed method for describing terminological concepts in the knowledge base. However, a rule of thumb is that such a method should be refined through several iterations. In the previous chapter, the terminologies for representing HTN planning in the DL representation were defined. However, the instances of the planning itself are not yet modelled.

Describing a planning problem for the planning system is not trivial. It requires a process involving use cases and testing similar to that followed in the software engineering discipline. Therefore, a method for modelling HTN planning problems in the HDL system has been defined and successfully tested over several use cases.
This method has 7 steps and is described below:

1. Define the actions and objectives:
   This first step reduces the size of the problem by clearly defining the objectives. They are modelled by simple actions which will achieve them. This step thus requires us to define the test domain where the actions should take place.

2. Define the task networks:
   As previously described in Section 2.2.3, HTN planning is a heuristic approach. The heuristic comes from human knowledge. The result is a plan that is very intuitive in that it solves the problem in a manner that a human being would. In the previous step, some actions and the objectives were defined. In this step the task networks for achieving the objectives is built.

3. Program the planning domain description:
   This step instantiates the network, defined in the previous step, into a particular planning-

understandable syntax, in this case SHOP2. The relations between methods and operators are connected through pre- and post-conditions. The post-conditions might change some particular states by removing or adding some facts.

4. Test the planning domain:
   It is necessary to test the planning domain description to validate its results. If it does not successfully extract a plan for a simple problem or if some error occurs, it is necessary to repeat the previous step(s). Only if a working planning domain description, that runs under the test cases, is found, it is possible to proceed to the next step.

5. Define the HTN *ABox* in the DL system:
   Once a valid planning domain is fixed, its description in DL representation can be inserted into the *ABox*. This can be done by describing parts of methods or operators. However, it is necessary to capture and model the relationship of each of them such that it can be derived later automatically using Algorithms 2.1 and 2.2. It is also possible to define a complete planning-domain that involves the particular methods and operators.

6. Modelling and instantiating the states in the HDL system:
   The states of the planning problem need to be conceptualised in the *TBox* of the DL system. The real problem can then be instantiated in the HDL system.

7. Testing the HDL system:
   The final step tests the HDL system on its ability to generate a correct planning domain and planning problem.

Steps 1 to 4 above quickly describe the process used in designing and writing planning problems for existing planning systems. Three additional steps are needed to translate the problem into DL representation such that it can be used by the HDL system. These steps provide a rough guide to successfully modelling a planning problem in the HDL system. One might be tempted to model the planning problem directly in DL and then try it. If a problem appears though, it might be more difficult to find the root of the error.

3.2. Navigation Domain

A mobile robot's basic operation is to move from its current position to the desired one. In order to perform this operation, planning is needed. However, the level of planning might be different from what has so far been discussed. One such level refers to path planning in which the task is to plan trajectories within a space which the robot should then follow in order to achieve the goal [Lat91, LaV06]. From the deliberative point of view, planning refers to the symbolic planning problem where the robot needs to move to the goal position by using a topological map. In this work, it is the deliberative layer and not path planning that is addressed. As proof

of concept from the method defined in the previous section, the navigation domain is explained chronologically by following the steps in that method.

3.2.1. Step 1: Define the Actions and Objectives

In order to define the actions and objectives, a test domain has to be defined first. One simple test domain for navigation is shown in Figure 3.1. It depicts a map of two buildings which are connected to each other by a corridor. Each building has six rooms and one corridor. A topological map can be built corresponding to this figure which contains the information on how one room is connected to the other. The map also shows the location of the robot, `room-1`, and the goal position in `room-6`.

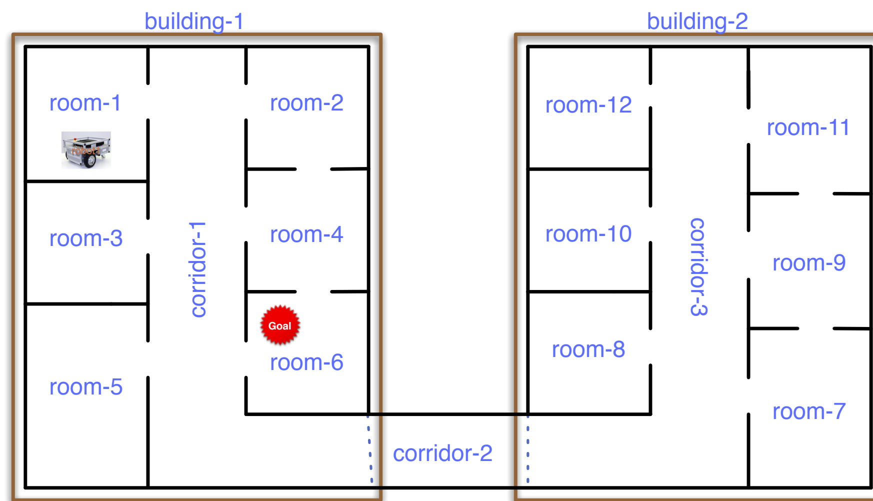


Figure 3.1.: HTN planning ontology.

The actions that a robot can perform limit the number of operators in the HTN planning domain. For a mobile robot, the basic operator would be `move`. In order to distinguish between the `move` command in the path-planning task, we define it as `drive-robot`. The objectives of the planning process are written as methods that can be decomposed into an HTN. In the navigation domain, the objective is to `navigate the robot from initial-location to goal-location`. Hence, the method would be `navigate`.

Each method and operator can have variables, for example to let the planner know which robot, or the destination of the goal. Hence, the operator `drive-robot` is written as `(!drive-robot ?robot ?loc-from ?loc-to)`. The syntax of SHOP2 is derived from the LISP syntax. This is due to the first implementation of the SHOP system which was implemented in LISP. The exclamation mark tells the system that the operator is an atomic operator [Ilg06]. It has three variables, namely `robot`, `loc-from`, and `loc-to`. Every variable is marked with a question mark in the front of its name [Ilg06]. This operator will perform the `drive-robot` action for `robot` from `loc-from` to `loc-to`.

A similar approach is applied for the `navigate` method. In the navigation domain, the `navigate` method will have similar variables to the `drive-robot` operator. It is defined as `(navigate ?robot ?from ?to)`. It has the same semantic meaning as the `drive-robot` operator. However, the difference is in their influence over states. An operator can have add and delete lists where it can add new states into the system and delete some of them. However, a method consists of a network of some other methods or operators. It does not influence the states directly.

The signature of the `drive-robot` operator is defined, however the implementation is not yet fixed. Below is the definition of the `drive-robot` operator:

`(!drive-robot ?robot ?loc-from ?loc-to)` *;; operator drive-robot*
preconds: $L_1 =$ `at(robot,loc-from)`
delete-list: $D_1 =$ `at(robot,loc-from)`
add-list: $A_1 =$ `at(robot,loc-to)`

The first list (L_1) contains the preconditions of the operator. In order to use an operator, the preconditions must first be fulfilled. The second list (D_1) is the delete-list which contains the states that will be deleted. The third list (A_1) is the add-list containing states that will be added to the search space. The `drive-robot` operator first checks whether the robot is located at the initial location as defined in the variable `loc-from`. Once this is confirmed, it will remove the state that tells the system that the robot is at `loc-from` and then add a new one, namely `loc-to`. Thus, using the `drive-robot` operator changes the state of the robot from the initial position (`loc-from`) to the destination position (`loc-to`).

One might ask the question, why do we need the method `navigate`? Can we just use the `drive-robot` operator instead? The answer is no. It is necessary to use the `navigate` method rather than the operator because the operator will only work under the specific circumstances where the `loc-from` location and the `loc-to` location are adjacent to or coincidental with each other. If this is not the case, the operator will only have the effect that the robot should be at the `loc-to` location without considering the path it takes. Hence, the method in HTN is the heuristic that tells the system how to solve a particular problem.

3.2.2. Step 2: Define the Task Networks

Defining a task networks in the HTN is done by defining the methods which were determined in the previous step. The `navigate` method is defined as follows:

`(navigate ?robot ?from ?to)` *;; the robot is already at destination*
task: `navigate(robot, from, to)`
subtasks: $\emptyset$
constr: `at(robot, to)`

`(navigate ?robot ?from ?to)` *;; from and to are adjacent to each other*

task: `navigate(robot, from, to)`

subtasks: $u_1 =$ `!drive-robot(robot, from, to)`

constr: `adjacentto(from, to)`

`(navigate ?robot ?from ?to)` *;; method to extract the path*

task: `navigate(robot, from, to)`

subtasks: $u_1 =$ `!drive-robot(robot, from, room)`

$u_2 =$ `!visit(room)`

$u_3 =$ `navigate(robot, room, to)`

$u_4 =$ `!unvisit(room)`

constr: $u_1 \prec u_2, u_2 \prec u_3, u_3 \prec u_4,$ `room(room), adjacentto(from,room),`

`not(visited(room))`

Methods in SHOP2 can be exemplified by cases. The `navigate` method, given here, has three cases. Each case has constraints that check whether the following task list is applicable or not. These constraints may also have information about the sub-tasks, ordering, e.g. as shown in third case of `navigate` method. The first case of the `navigate` method checks whether the current robot location is coincidental with the destination location. If this is the case, the plan will successfully return and do nothing. The second case checks whether the initial position and the destination are adjacent to each other. If so, then it will execute the `drive-robot` operator. This case can be handled directly by the `drive-robot` operator. However, the third case has the real heuristic of how the search should be performed. It checks a new state, `room`, to see if it is adjacent with the initial location. It also checks whether it has been visited or not. Figure 3.2 depicts the task network of the navigation domain where the start location and destination location are neither coincidental nor adjacent. The `navigate` method is a recursive one, it calls itself with the new initial location (`?room`). That is also the reason why this method has cases to check for the break condition for the recursion.

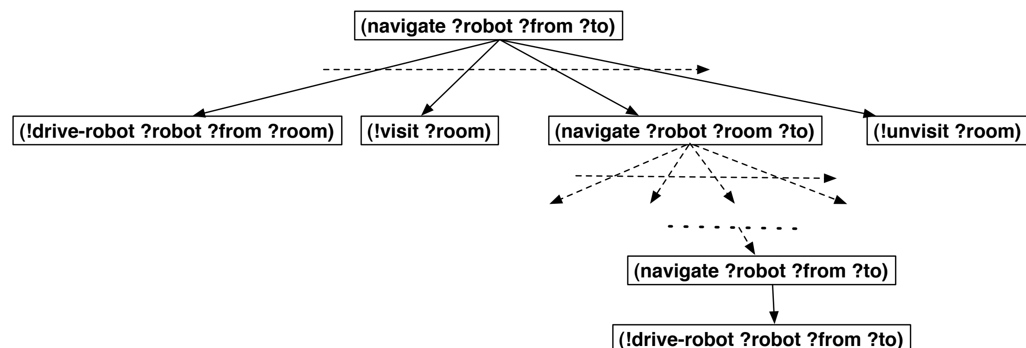


Figure 3.2.: Navigation task network.

Note that the `navigate` method has two additional operators, namely `visit` and `unvi-`

sit. These operators are needed for the search heuristic. The function of these operators is to mark the places which have been visited by the robot in the search space to prevent it from embarking on a cyclic search. These operators use a marker, which is represented as a state, namely (visited ?room).

The method navigate shows how a simple problem is decomposed into smaller subtasks in the task network in order to solve a particular problem. It shows a hierarchy of the operators and methods. However, HTN can do abstraction over some methods too. This is shown here through an analysis of the navigate method once again. It has three variables, one of which could be removed. The goal is to let the robot navigate into a certain room. As the initial state of the robot is already modelled in the planning-domain, it is not necessary to explicitly mention it in the planning problem. This is also expected by the users as the robot should know where it is through the use of a map within its memory. Hence, a new method can be built over navigate as shown below:

(navigate ?robot ?to)
task: navigate(robot, to)
subtasks: $u_1 =$!visit(from)
$u_2 =$ navigate(robot, from, to)
$u_3 =$!unvisit(from)
constr: $u_1 \prec u_2, u_2 \prec u_3,$ at(robot, from)

The result is two methods with the name, navigate. The difference between them is the number of parameters. One has three variables and the other has two. SHOP2 can automatically distinguish which method to use based on the number of variables of the method or operator. The abstract navigate method checks the current location of the robot and uses this information within its task network after marking the location and calling the navigate method with three parameters.

3.2.3. Step 3: Program the Planning Domain

The methods and operators have been defined in the previous steps. However, the previous definitions are not in any specific HTN language. Therefore, in this step the methods and operators have to be implemented in one of HTN language, in this case SHOP2.

Listing 3.1: Navigation domain.

```
(defdomain navigation_domain (
    (:operator (!drive-robot ?robot ?loc-from ?loc-to)
      ((at ?robot ?loc-from))
      ((at ?robot ?loc-from))
      ((at ?robot ?loc-to))
    )
```

```
    (:operator (!unvisit ?waypoint)
      ()
      ((visited ?waypoint))
      ()
    )

    (:operator (!visit ?waypoint)
      ()
      ()
      ((visited ?waypoint))
    )

    (:method (navigate ?robot ?from ?to)
      Case1 ((at ?robot ?to))
        ()
      Case2 ((adjacentto ?from ?to))
        ((!drive-robot ?robot ?from ?to))
      Case3 ((room ?room)(adjacentto ?from ?room)(not (visited ?...
          room)))
        ((!drive-robot ?robot ?from ?room)
         (!visit ?room)
         (navigate ?robot ?room ?to)
         (!unvisit ?room))
    )

    (:method (navigate ?robot ?to)
      ((at ?robot ?from))
      ((!visit ?from)(navigate ?robot ?from ?to)(!unvisit ?from))
    )
  )
)
```

Listing 3.1 shows the navigation domain written in SHOP2 syntax. There are three operators and two methods. The main operator `drive-robot` is defined previously in the first step. Writing this operator in SHOP2 syntax is a straight forward. The first line is the method signature with its parameters. Its contents are the preconditions, delete lists, and add lists [Ilg06]. The interesting part is the `navigate` with three parameters which is defined in the second step. In the definition, it is represented by three definitions with their sub-tasks. However, in the SHOP2 syntax, it is represented as one method with three cases. The first list in each case is the constraints and the second list is the sub-tasks. The “Case1”, “Case2”, and “Case3” are merely labels for each case.

3.2.4. Step 4: Test the Planning Domain

Once the planning domain is defined, we need to model the environment in the form of states which conform to the preconditions of all methods and operators. Hence, in this case, four kinds of states are needed:

```
(at ?robot ?location)
(room ?room)
(adjacentto ?room1 ?room2)
(visited ?waypoint)
```

The state `(at ?robot ?location)` represents the location of any robot in the environment. In the current example, this would be {`(at robot1 room-1)`}. The state `(room ?room)` represents any room in the environment, namely {`(room room-1)`,`(room room-2)`, ...`(room room-12)`,`(room corridor-1)`,`(room corridor-2)`,`(room corridor-3)`}. The corridors are modelled as rooms as they have the same functionality. The state `(adjacent-to ?room1 ?room2)` represents the connectivity between rooms. In this case, {`(adjacentto room-1 corridor-1)`, `(adjacentto corridor-1 room-1)`, `(adjacentto room-2 corridor-1)`, `(adjacentto corridor-1 room-2)`, ... `(adjacentto corridor-3 room-12)`}. The `adjacentto` state must be represented symmetrically in order to allow the search to proceed in both directions. The last state `(visited ?waypoint)` does not appear in the environment. It only appears internally in the search space of the planning system.

In the problem shown in Figure 3.1, the objective of the planning problem is written as either `(navigate robot1 room-1 room-6)` or `(navigate robot1 room-6)`. The planning problem of the `(navigate robot1 room-6)` is shown in Listing 3.2.

Listing 3.2: Planning problem description for "`navigation-domain`".

```
(defproblem problem_navigation_domain navigation_domain
  (
    (at robot1 room-1)
    (room room-1)
    (room room-2)
    ... omitted ...
    (room room-12)
    (room corridor-1)
    (room corridor-2)
    (room corridor-3)
    (adjacentto room-1 corridor-1)
    (adjacentto corridor-1 room-1)
    (adjacentto room-2 corridor-1)
    (adjacentto corridor-1 room-2)
    ... omitted ...
    (adjacentto corridor-3 room-12)
  )
  (
    (navigate robot1 room-6)
  )
)
```

The result of this problem description is shown below:

```
(!visit room-1)
(!drive-robot robot1 room-1 corridor-1)
(!visit corridor-1)
(!drive-robot robot2 corridor-1 room-6)
(!unvisit corridor-1)
(!unvisit room-1)
```

If one uses the `(navigate robot1 room-1 room-6)` instead, the result will differ slightly from the one presented above. The difference lies in the operators `(!visit room-1)` and `(!unvisit room-1)`. However, this is not a problem, since these two operators have no real impact on the motion of the robot itself. This solution plan confirms that the navigation domain works.

3.2.5. Step 5: Define the HTN *ABox* in the HDL System

As the planning domain descriptions are successfully modelled, one needs to model them in the DL representation. In the navigation domain, there are three operators {`drive-robot`, `visit`, `unvisit`} and two forms of the `navigate` method. In DL each instance must have a unique name, thus two forms of the method `navigate` must be written with indices. Hence, we define a planning domain `navigation_domain` with three operators and two methods: $d_{navigation_domain} = \{m_{navigate}, m_{navigate2}, o_{drive\text{-}robot}, o_{visit}, o_{unvisit}\}$. Figure 3.3 shows the asserted instances in the HDL. The operators are shown in orange-coloured boxes (see Listing B.1 for detail), the methods in green (see Listing B.2 for detail), and the planning domain in purple (see Listing B.3 for detail).

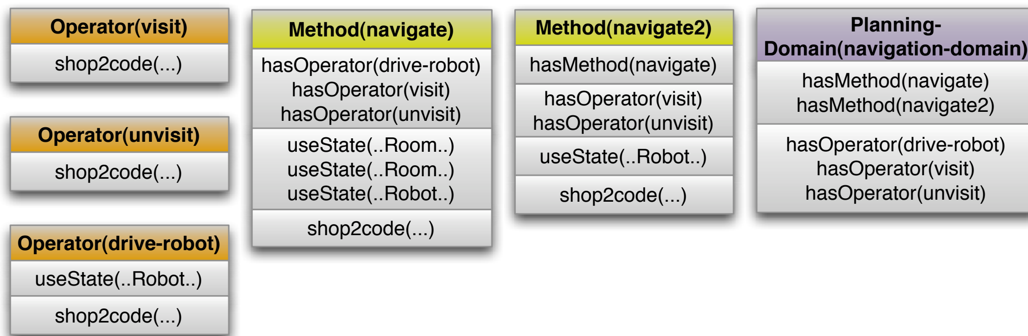


Figure 3.3.: Navigation domain's assertions in the HDL.

The Algorithms 2.1 and 2.2 can then be applied to the DL system to generate the planning problem. The users can choose an instance from either the $Planning\text{-}Domain$ or the $Method$. In the navigation example, the same domain is generated whether the user chooses $d_{navigation_domain}$ or $m_{navigate2}$. If $m_{navigate}$ is chosen, the generated planning domain will have one method less than the other two mentioned before; it will be generated without the method $m_{navigate2}$.

3.2.6. Step 6: Modelling and Instantiating the States in the HDL System

Once the assertions in the HTN's *ABox* are defined, the states must be inserted into the *ABox* too. To enable the DL reasoner to reason about the knowledge and generate the states automatically, the states need to be modelled in DL representation in the form of *TBox* and *ABox*. The *TBox* for the navigation domain is shown in Figure 3.4. The robot is represented as a sub-concept of "*Actor*". The corridors are modelled as part of or as a sub-concept of "*Room*", because they are treated as ordinary rooms. The buildings are defined in the DL too. The concept "$\mathit{Fixed\text{-}Object}$" is needed for organisational purposes.

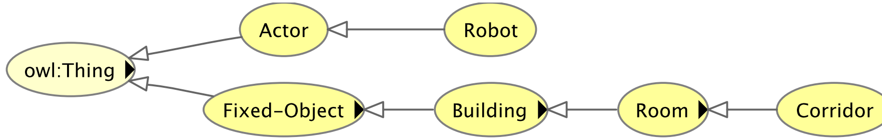


Figure 3.4.: Navigation domain's states concepts.

The concepts are sufficient to capture the domain that is shown in Figure 3.1. Three concepts have direct influence on the planning states, namely Robot, Room, and $\mathit{Corridor}$. The $\mathit{Corridor}$ is treated as a Room. The Robot is defined as follows:

$$\begin{aligned} \mathit{Robot} \sqsubseteq\ & \mathit{Actor} \sqcap \\ & \exists\, \mathit{at}.\mathit{Room} \sqcap \\ & = 1\ \mathit{at} \end{aligned}$$

The DL syntax describes the Robot as a sub-concept of Actor and shows that it has a property, at, whose range is some member of Room. In addition, this property has exactly one member. Similarly, Room is defined as follows:

$$\begin{aligned} \mathit{Room} \sqsubseteq\ & \mathit{Building} \sqcap \\ & \exists\, \mathit{adjacentto}.\mathit{Room} \end{aligned}$$

The $\mathit{Corridor}$ inherits the conditions of Room, because it is a sub-concept of Room.

The terminological concepts of the planning states are now defined in the DL system. Hence, the instances of the planning domain can be inserted into the *ABox* of the DL system. As shown in Figure 3.1, it has three corridors $\{\mathit{Corridor}(\texttt{corridor-1}), \mathit{Corridor}(\texttt{corridor-2}), \mathit{Corridor}(\texttt{corridor-3})\}$; 12 rooms $\{\mathit{Room}(\texttt{room-1}), \mathit{Room}(\texttt{room-2}), \dots\ \mathit{Room}(\texttt{room-12})\}$ and a robot $\{\mathit{Robot}(\texttt{robot1})\}$. In addition, the properties need to be inserted into the DL system too: $\{\mathit{at}(\texttt{robot1}, \texttt{room-1}), \mathit{adjacentto}(\texttt{room-1}, \texttt{corridor-1}), \mathit{adjacentto}(\texttt{corridor-1}, \texttt{room-1}), \dots\ \mathit{adjacentto}(\texttt{corridor-3}, \texttt{room-12})\}$.

3.2.7. Step 7: Testing the HDL System

By asserting the instances of the planning domain, the HDL system has a complete definition of the navigation domain. The user can now define the destination that the robot should navigate to. The HDL system will then extract the planning domain and planning problem from the DL representation. The generated planning domain for navigation is shown in Listing A.1. The HDL system produces a correct planning domain which is similar to the one defined in the third step (Section 3.2.3). It produces a viable solution plan for the problem shown in Figure 3.1 that is similar to the result of the fourth step (Section 3.2.4). Appendix A.1 shows some generated planning problems and their solutions.

3.3. Exploiting the HDL system

In this section, we discuss the benefit of modelling HTN in DL. Modelling HTN planning problems in the HDL system requires some effort and time. What is the benefit of an HDL system? There are advantages to having the HTN planning domain modelled in the HDL system. In the previous sections, a proof of concept for the HDL system being capable of capturing and modelling the HTN planning problem was shown. In this section, one advantage of the HDL system is elaborated.

The navigation domain, presented in the previous section, is quite simple. It discards information that does not have direct influence on the planning process. Nevertheless, the actions generated by the planning system are enough to navigate the robot from the initial position to the destination in this particular domain. What happens if the robot should work in the other building where the topological information is different from the current one? In this case, new states are needed to model the new environment. The current planning domain is still usable for this problem. Is it possible to model all the possible topological information (states) of the environment in the planning problem? Theoretically, yes, but in practise, this is not done. As the planning description gets larger, it may cause the planning problem to become intractable. This is because the search space grows too large.

Imagine that the robot should work in a six-storey building with a large number of rooms. It is clear that a mobile robot can only be on one of the six available floors at any given time. Hence, some of the floors and rooms are not accessible to the robot. Thus, the topological information of the other floors is irrelevant for the planning problem. Even on a given floor, some rooms might not be accessible to the robot. For example, the door may be closed or some rooms might have a smaller footprint than the robot. In this case, there are some irrelevant states which should be removed from the planning problem. Here, the realisation of this particular case is presented.

Assume that the robot can acquire the states of the doors (whether they are open or closed). This adds new information to our domain. This is depicted in Figure 3.5. The domain contains the same topological map as shown in Figure 3.1, but with the added information of the doors.

In Figure 3.5, the robot can only access several rooms, specifically, those which have an open door, such as `room-1`, `room-3`, `corridor-1` and so on.

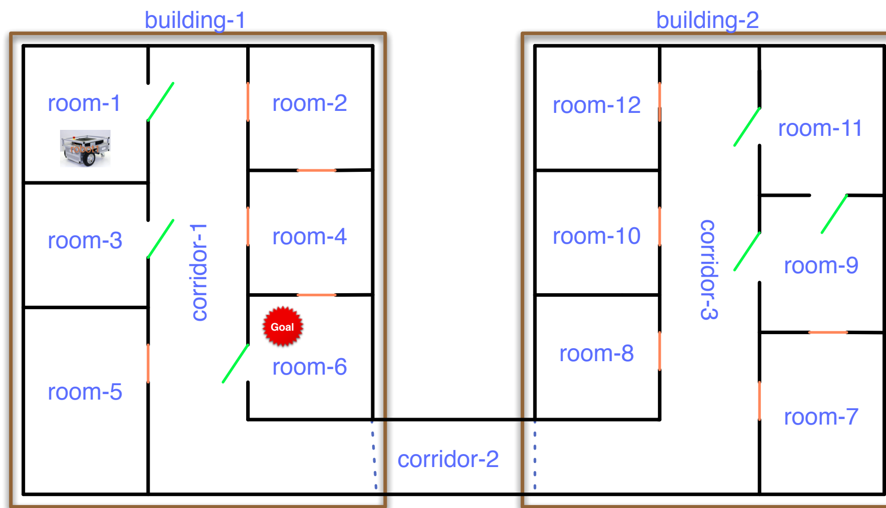


Figure 3.5.: Simple navigation domain with door(s) connecting between rooms, which can be either open or closed.

The question now is how to successfully navigate around the new environment with an ordinary planning model, i.e. using the HTN approach. In this case, the door information also needs to be modelled in the planning domain. Hence, the planning domain, introduced in Section 3.2, cannot be used directly. A new planning domain which considers the state of the doors needs to be modelled. In addition, the door states also have to be written into the planning problem description. As a result, the size of the the planning problem will increase linearly in the number of rooms, assuming that each room has only one door. Hence, the planning process would eventually become intractable. The planner might be able to evaluate whether a room is accessible or not, but only after it has finished the extraction of the plan. By then, it may be too late if the problem is big enough to make the plan intractable. A human taking the role of the planning designer might help with this problem as he or she will only model the accessible rooms. However, this process would not be performed automatically. The proposed system, HDL, is able to automated this action.

In the HDL system, the planning problem is extracted automatically from the DL representation. The environment or domain is modelled in DL instead. DL reasoning is tractable and can handle a much larger amount of information than HTN planning. There are two advantages of using the HDL system here. Firstly, the previous planning domain can be used without any modification. Secondly, the planning problem description can be reduced to a smaller one, depending on the number of open doors and this is done automatically. The filter criteria are defined as DL concepts in the *TBox*.

One more benefit of using the HDL system is the model that has just been defined for the simple navigation domain can still be used and extended to support doors. Figure 3.6 shows

the navigation domain with the extended concepts. Six additional concepts are defined in order to model the domain. Additionally, there are the three concepts in orange, which instances are inferred by the DL reasoner from the asserted instances (*ABox*).

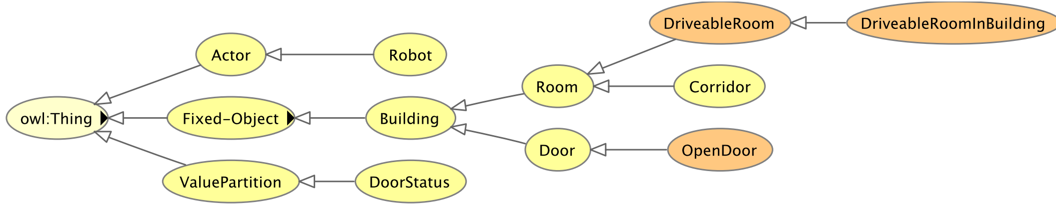


Figure 3.6.: Extended navigation domain's states concepts.

The concepts $Door$, $ValuePartition$, and $DoorStatus$ represent doors and their status. $Door$ is defined as:

$$\begin{aligned} Door \sqsubseteq\ & Building\ \sqcap \\ & \exists\ hasState.DoorStatus \sqcap \\ & = 1\ hasState \end{aligned}$$

Each instance of $Door$ has exactly one state either "open" or "closed". This property is defined by the instances of the $DoorStatus$, namely $\{DoorStatus(\texttt{isOpen}), DoorStatus(\texttt{isClose})\}$. The doors are inserted into the *ABox* as follows: $\{Door(\texttt{door_1}), Door(\texttt{door_2}), Door(\texttt{door_2_4}), \ldots\ Door(\texttt{door_12})\}$. In addition, the state of every door is defined as $\{hasState(\texttt{door_1}, \texttt{isOpen}), hasState(\texttt{door_2}, \texttt{isClose}), hasState(\texttt{door_2_4}, \texttt{isClose}), \ldots\ hasState(\texttt{door_12}, \texttt{isClose})\}$.

The doors are modelled and asserted in the HDL system. However, the $Room$ concept is not yet included. In order to accommodate additional information like $Door$ and $Building$, $Room$ is redefined as follows:

$$\begin{aligned} Room \sqsubseteq\ & Building\ \sqcap \\ & \exists\ adjacentto.Room \sqcap \\ & \exists\ hasDoor.Door \sqcap \\ & \exists\ inBuilding.Building \end{aligned}$$

Two instances of $Building$ are asserted in the model, namely $\{Building(\texttt{building-1}), Building\ (\texttt{building-2})\}$. The properties $hasDoor$ and $inBuilding$ are filled with this knowledge: $\{inBuilding(\texttt{room-1}, \texttt{building-1}), inBuilding(\texttt{room-2}, \texttt{building-1}), \ldots\ hasDoor(\texttt{room- 1}, \texttt{door_1}), hasDoor(\texttt{building-1}, \texttt{door_1}), \ldots\ hasDoor(\texttt{room-12}, \texttt{door_12})\}$.

The navigation domain, as shown in Figure 3.5, is now modelled in the HDL system. This knowledge is just inserted into the *ABox*. Retrieving an instance of a concept is done merely by reading the asserted knowledge from the *ABox*. The DL reasoner can do more than just return

asserted information. It can infer new knowledge from current knowledge. This is precisely what is needed of the system. It should filter out irrelevant knowledge such as rooms with a closed door.

Additional concepts can be defined over the current ones. These concepts are shown in Figure 3.6 as orange coloured ellipses. The first one is the $OpenDoor$ concept that is defined as follows:

$$\begin{aligned} OpenDoor \equiv\ & Door \sqcap \\ & \ni\ hasState(isOpen) \end{aligned}$$

The DL reasoner infers the knowledge and fills the $OpenDoor$ with instances of $Door$ which have $hasState$ value "isOpen". The second concept is $DriveableRoom$, which is defined as:

$$\begin{aligned} DriveableRoom \equiv\ & Room \sqcap \\ & \exists\ hasDoor.OpenDoor \end{aligned}$$

The $DriveableRoom$ concept uses the $OpenDoor$ concept as a condition for being part of the concept. Hence, any instance of $Room$ with an open door will automatically become part of this concept. The third concept $DriveableRoomInBuilding$ shows how the concept can be refined into a more specific one, namely for retrieving a room with any open door in a particular building. This is defined as follows:

$$\begin{aligned} DriveableRoomInBuilding \equiv\ & DriveableRoom \sqcap \\ & \ni\ inBuilding(building\text{-}1) \end{aligned}$$

The defined concept will return all $DriveableRoom$ in `building-1`. These are some examples of how the concepts are defined to filter the states for the planning problem description. In the following section the results of using the DL model, as shown in Figure 3.6, on the navigation domain with doors, as shown in Figure 3.5, are presented.

3.3.1. Results

The extended navigation domain is defined in the HDL system. The rooms and their doors are modelled in such a way that the DL reasoner is able to reason about them. However, the question remains of how the HDL system would know which states should be generated for the planning problem description. As mentioned previously in Section 2.4.2, the operators and methods contain the information which states are needed. Thus, the $useState$ property of the methods and operators needs to be customised. In the navigation domain, the $useState$ properties of the method `navigate` must be customised. Especially for the instance query of the concept $Room$. It can be replaced with any other sub-concept of $Room$.

Table 3.1 shows the number of generated states for different specifications of the $Fixed$-

Table 3.1.: Generated states for different specialisations of the $Fixed\text{-}Object$ concept.

Condition	Number of states
s_{room}	16
$s_{room\text{-}in\text{-}building}$	8
$s_{driveable\text{-}room}$	8
$s_{driveable\text{-}room\text{-}in\text{-}building1}$	5
$s_{driveable\text{-}room\text{-}in\text{-}building2}$	3

$Object$ concept [HH08]. It gives an impression of how specialisation alters the number of generated states s_0, based on the number of rooms to consider in some concrete planning problem. As expected, using a more specific concept reduces the number of generated states s_0, which can improve planning efficiency by reducing the size of the search space.

3.4. Pick-and-Place Domain

In Section 3.2, a planning domain for mobile robotics is presented. The HDL system is able to reduce the number of states by refining the ontologies of the domain (see Section 3.3). In that domain, the robot's capability determines the possible operators in the planning domain. Nowadays, the field of robotics research and engineering is growing rapidly. Robotic manipulators, traditionally used by manufacturers for producing products requiring high precision, such as cars, are currently being produced in lightweight forms, for example the KuKA light weight arm (LBR) and the Katana arm. KuKA, Motoman, Neuronics and Schunk are some such robot manipulator manufacturers. The development of lightweight arms enables roboticists to integrate them with mobile robots. Hence, such robots are also called mobile manipulator robots. In addition to the actions related to their mobility, the mobile manipulator is capable of performing other actions relating to the manipulation of an object. The manipulation depends on the kind of mounted arm; it can be grasping, lifting, pushing, etc. In this example, we use a manipulator with grasping capability.

In the navigation domain example, the planning domain can be reduced by adding additional information used for the filtering process. The HDL system can work with the same planning domain and generate a filtered planning problem. In the pick-and-place domain, some additional advantages of the HDL systems can be seen. Hence, the pick-and-place domain will extend the previous environment by adding several objects. It also shows how the planning domain can be extended and used for the pick-and-place domain. Hence, the HDL system solves the re-usability requirement as mentioned in Section 1.2.

3.4.1. Problem Specification

The pick-and-place domain extends the navigation domain map with some objects. Figure 3.7 depicts this domain. An additional robot, namely `robot2`, can be seen in this extended domain. `robot1` is located in `room-1` and `robot2` is located in `room-4`. Some objects are also added into the domain, namely five containers and six fruits. The containers are introduced here to provide the operators with more specific parameters. Instead of telling the planner that `apple1` is in `room-1`, it gives a more precise location of the apple within that room, e.g. in `basket-1` and the basket is in `room-1`.

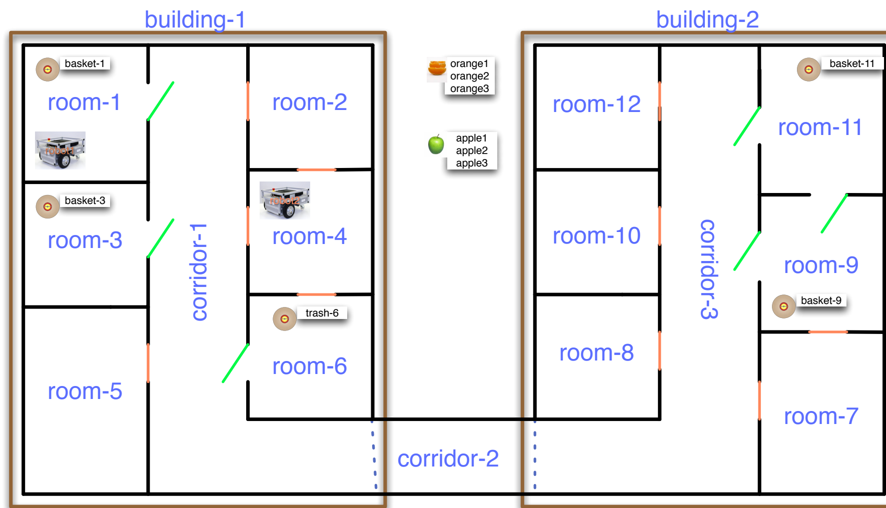


Figure 3.7.: Pick-and-place domain with two robots and six manipulable objects.

A typical problem for a mobile manipulator is to manipulate an object. A complex task would be to bring an object from an initial location to a destination location. It involves several sequential actions, for example navigating to the initial position, grasping the object, navigating the robot to the destination location, and releasing the object.

3.4.2. Modelling Actors and Objects

The pick-and-place domain is an extension of the navigation domain. Hence, for the sake of clarity, we will start by modelling additional actors and objects in the HDL system. Most of the concepts and instances from the navigation domain can be used directly in the new domain. However, some additional concepts that are necessary in order to model the manipulable objects are defined. These can be seen in Figure 3.8.

The concept for containers is defined as a sub-concept of $\mathit{Fixed\text{-}Object}$. The $\mathit{Container}$ is described in DL as follows:

$$\begin{aligned} \mathit{Container} \sqsubseteq\ & \mathit{Fixed\text{-}Object} \sqcap \\ & \exists\, at.\mathit{Room} \end{aligned}$$

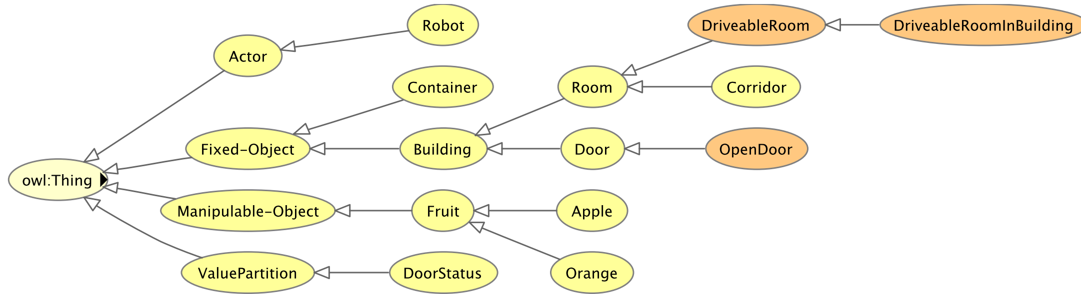


Figure 3.8.: Extended pick-and-place domain's states concepts.

The instances of the containers are asserted in the HDL system, $\{Container($`container-1`$)$, $at($`container-1`, `room-1`$)$, ... $Container($`trash-6`$)$, $at($`trash-6`, `room-6`$)\}$. The $Fruit$ is modelled as part of $Manipulable\text{-}Object$ which is defined as:

$$Manipulable\text{-}Object \sqsubseteq Thing \sqcap \exists\, at.Container$$

$Apple$ and $Orange$ are sub-concepts of $Fruit$. Hence, $Apple$ and $Orange$ inherit the property of $Manipulable\text{-}Object$. Thus, the assertions of the fruits are described as follows: $\{Apple($`apple1`$)$, $at($`apple1`, `basket-1`$)$, ... $Orange($`orange3`$)$, $at($`orange3`, `basket-11`$)\}$.

Having the actors and objects modelled and asserted in the HDL system, the environment states are automatically captured in the system. Hence, the next step would be to define the planning domain with new operators and methods for a mobile manipulator.

3.4.3. Defining the HTN Planning Domain for Pick-and-Place Tasks

The mobile manipulator has more functionality than a simple mobile robot. Thus, it also has additional planning operators. Operators can either be compared with low level control skills or in a more abstract manner. In the abstract operator, the executor needs to translate an operator into more specific commands which match the low level controller.

The HTN planner provides hierarchies in its network which have been presented in the navigation domain through the use of two navigate methods with different signatures. The HDL system provides another abstraction at the ontology level. This abstraction is presented in this section through the pick-and-place domain.

Two possible basic manipulation actions which the mobile manipulator can perform are "pick-up an object" and "put an object". However, in order to perform these actions, the robot needs to be as close as possible to the object. A manipulator has physical limitations which impact the positions that it can reach. These limitations are defined formally as the manipulator's workspace. There are two kinds of workspaces, namely reachable workspaces and dexterous workspaces. A reachable workspace is the area in which the tool tip of the manipulator can reach. The dexterous workspace is the area in which the manipulator can reach a point in the work-

space in all possible poses [Cra05]. Two additional operators are introduced according to the workspace, namely "move the robot to dexterous workspace" and "move away from dexterous workspace". Why do we need these two new operators in addition to the `pick-up` and `put` operators?

The mobile manipulator might have collision avoidance integrated in its low level controller. Hence, telling the robot to move around in the environment would cause no problem due to the collision avoidance skill. However, when the robot should grasp an object, it needs to move as close to the object as possible. In the environment, the object might be located on top of a table. Having collision avoidance active might prevent the robot from approaching the object. The move to dexterous workspace has the purpose of allowing the robot to reduce the boundary of its collision avoidance such that it can move as close to the object as possible to grasp it.

3.4.3.1. Partial Pick-and-Place Domain

In the partial pick-and-place domain, the `navigate` method from the navigation domain is considered as an operator for the pick-and-place domain. Having the `navigate` method as an operator makes the planning problem much simpler than the complete one. This is explained in more detail in the following section. Once the operators are fixed, the methods need to be defined. Methods represent the plan objectives. A complex method for a mobile manipulation task might be "move object from one container to a destination container". This task might be decomposed into smaller subtasks, such as "`get-object`" and "`put-object`".

The HTN for the partial pick-and-place domain for the task "move an object to destination container" is shown in Figure 3.9. The process is similar to how humans think to solve the problem. The first action would be to first go to the location where the object is located and then to try to grasp it. In order to grasp the object, the robot needs to get near to the object's container and then perform the grasping action. Before the robot navigates to the destination room, it might first need to move away from the container. A similar approach is used when the robot is in the destination room. It will approach the container and put the object into the container.

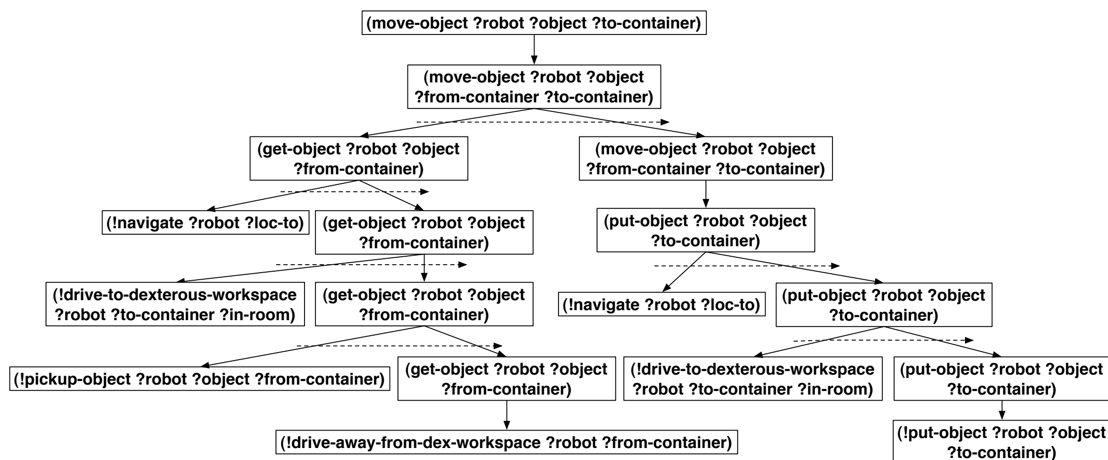


Figure 3.9.: Partial pick-and-place domain task network.

As shown in the Figure 3.9, the `move-object` task is divided into two subtasks namely `get- object` and `put-object`. The main objectives are represented using two methods which represent a hierarchy in the planning domain. As in the navigation domain, they have different parameters. The first method of `move-object` requires three parameters, shown in the first level of the task network (see Figure 3.9). These parameters are the robot itself, the object, and the destination. However, the location of the object is not a parameter, but will be automatically filled in from the robot's world model. This method is defined as follows:

`(move-object ?robot ?object ?to-container)`
task: `move-object(robot, object, to-container)`
subtasks: $u_1 =$ `move-object(robot, object, from-container, to-container)`
constr: `at(object, from-container)`

The second method of `move-object` has one more parameter, namely the object location (`from-container`). This method consists of two cases. The first case decomposes into the `get-object` task and the second one decomposes into the `put-object` task. Below is the definition of this method:

`(move-object ?robot ?object ?from-container ?to-container)`
task: `move-object(robot, object, from-container, to-container)`
subtasks: $u_1 =$ `get-object(robot, object, from-container)`
$u_2 =$ `move-object(robot, object, from-container, to-container)`
constr: $u_1 \prec u_2$, `not(at(object, robot))`, `at(object, from-container)`

`(move-object ?robot ?object ?from-container ?to-container)`
task: `move-object(robot, object, from-container, to-container)`
subtasks: $u_1 =$ `put-object(robot, object, to-container)`
constr: `at(object, robot)`

The cases are represented as recursion in the network. Figure 3.10 shows the assertions for method `moveobject_p` and `moveobject2_p`. Detail of these assertions is shown in Listing B.4.

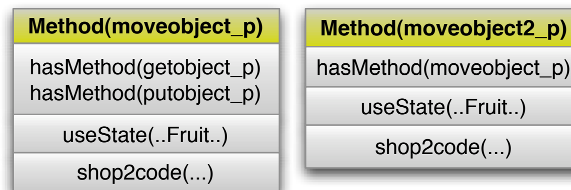


Figure 3.10.: Method "`moveobject_p`" and "`moveobject2_p`" assertions in the HDL.

The method `get-object` is also represented in two forms. The first one has two parameters, namely `robot` and `object`. As in the `move-object` task with three parameters, the location of the object is retrieved automatically from the robot's world model. This method is defined as follows:

(get-object ?robot ?object)
task: get-object(robot, object)
subtasks: u_1 = get-object(robot, object, from-container)
constr: at(object, from-container)

The second form consists of four cases. The first case tests whether the `object` is at the `robot`. It moves the robot away from the grasping position. Thus, the robot can safely navigate to the next location without collision with any object, e.g. furniture. This case terminates the recursion too. The second case calls the `pick-up` operator if the object is located in the dexterous workspace of the robot. The third case moves the robot to approach the object if the object is not in the dexterous space and the robot is in the same room as the object. The fourth case calls the `navigate` operator if the robot is located in another room. This method is defined as follows:

(get-object ?robot ?object ?from-container) *;; case 1*
task: get-object(robot, object, from-container)
subtasks: u_1 = !drive-away-from-dex-workspace(robot, from-container)
constr: at(object, robot)

(get-object ?robot ?object ?from-container) *;; case 2*
task: get-object(robot, object, from-container)
subtasks: u_1 = !pickup-object(robot, object, from-container)
u_2 = get-object(robot, object, from-container)
constr: $u_1 \prec u_2$, at-dexterous-workspace(robot, from-container)

(get-object ?robot ?object ?from-container) *;; case 3*
task: get-object(robot, object, from-container)
subtasks: u_1 = !drive-to-dexterous-workspace(robot, from-container, room)
u_2 = get-object(robot, object, from-container)
constr: $u_1 \prec u_2$, not(at-dexterous-workspace(robot, from-container)), at(from-container, room), at(robot, room)

(get-object ?robot ?object ?from-container) *;; case 4*
task: get-object(robot, object, from-container)
subtasks: u_1 = !navigate(robot, room)
u_2 = get-object(robot, object, from-container)

constr: $u_1 \prec u_2$, `not(at-dexterous-workspace(robot, from-container)),`
`at(from-container, room),not(at(robot, room))`

Figure 3.11 shows the assertions for the methods `getobject_p` and `getobject2_p`. The details of these assertions are shown in Listing B.5.

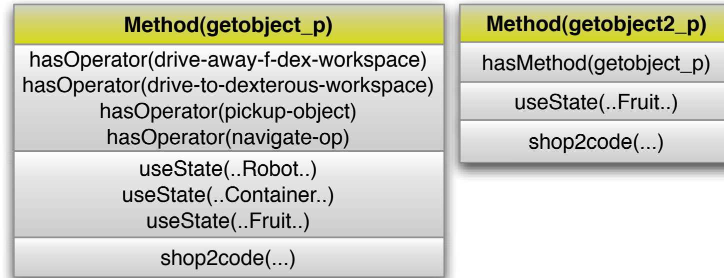


Figure 3.11.: Method "`getobject_p`" and "`getobject2_p`" assertions in the HDL.

Unlike the `move-object` or `get-object` methods, it has no redundant information in its parameters. Hence, the `put-object` method has only one implementation with four cases. The first case terminates the recursion if the robot does not have the object anymore, which assumes that the object was successfully placed on the desired location. The second case calls the `put-object` operator if the destination location is within the robot's dexterous workspace. If the robot is not there yet, the third case performs the action to move the robot to its dexterous workspace for placing the object. The previous action can only be performed if the robot is located in the same room as the destination location. The fourth case calls the `navigate` operator if the robot is not there yet. The `move-object` method is defined as follows:

`(put-object ?robot ?object ?to-container)` *;; case 1*
task: `put-object(robot, object, to-container)`
subtasks: $\emptyset$
constr: `not(at(object, robot))`

`(put-object ?robot ?object ?to-container)` *;; case 2*
task: `put-object(robot, object, to-container)`
subtasks: $u_1 =$ `!put-object(robot, object, to-container)`
constr: `at(object, robot),at-dexterous-workspace(robot, to-container)`

`(put-object ?robot ?object ?to-container)` *;; case 3*
task: `put-object(robot, object, to-container)`
subtasks: $u_1 =$ `!drive-to-dexterous-workspace(robot, from-container,`
`room)`
$u_2 =$ `put-object(robot, object, to-container)`
constr: $u_1 \prec u_2$, `at(object, robot),not(at-dexterous-workspace(robot,`

`to-container)),at(to-container, room),at(robot, room)`

`(put-object ?robot ?object ?to-container)` *;; case 4*

task: `put-object(robot, object, to-container)`

subtasks: $u_1 =$ `!navigate(robot, room)`

$u_2 =$ `put-object(robot, object, to-container)`

constr: $u_1 \prec u_2$, `at(object, robot),not(at-dexterous-workspace(robot, to-container)),at(to-container, room),not(at(robot, room))`

The method `putobject_p` assertion is shown in Figure 3.12. Detail of this assertion is in Listing B.6.

Method(putobject_p)
hasOperator(put-object) hasOperator(drive-to-dexterous-workspace) hasOperator(navigate-op)
useState(..Robot..) useState(..Container..)
shop2code(...)

Figure 3.12.: Method "`putobject_p`" assertion in the HDL.

These methods use some operators, namely `navigate-op`, `drive-to-dexterous-workspace`, `drive-away-f-dex-workspace`, `pickup-object`, and `put-object`. These operators are defined as follows:

`(!navigate ?robot ?loc-to)`

preconds: $L_1 =$ `at(robot,loc-from)`

delete-list: $D_1 =$ `at(robot,loc-from)`

add-list: $A_1 =$ `at(robot,loc-to)`

`(!drive-to-dexterous-workspace ?robot ?to-container ?in-room)`

preconds: $L_1 =$ `at(robot,in-room)` $\wedge$ `at(to-container, in-room)`

delete-list: –

add-list: $A_1 =$ `at-dexterous-workspace(robot, to-container)`

`(!drive-away-from-dex-workspace ?robot ?from-container)`

preconds: –

delete-list: $D_1 =$ `at-dexterous-workspace(robot, from-container)`

add-list: –

`(!pickup-object ?robot ?object ?from-container)`

preconds: $L_1 =$ `at-dexterous-workspace(robot,from-container)` $\wedge$ `not(has-object(robot))`

delete-list: $D_1 =$ `at(object,from-container)`

$D_2 =$ `protection(at-dexterous-workspace(robot, container))`

add-list: $A_1 =$ `has-object(robot)`

$A_2 =$ `at(object, robot)`

`(!put-object ?robot ?object ?to-container)`

preconds: $L_1 =$ `at-dexterous-workspace(robot, to-container)` $\wedge$ `has-object(robot)` $\wedge$ `at(object, robot)`

delete-list: $D_1 =$ `has-object(robot)`

$D_2 =$ `at(object, robot)`

add-list: $A_1 =$ `at(object,to-container)`

These operator assertions are shown in Figure 3.13, where the detail is listed in B.7.

Figure 3.13.: Operators assertion in the HDL for pick-and-place domain.

In addition, an insertion into the *ABox* of the $Planning\text{-}Domain$ concept is done for the `pick-and-place_domain_partial`. The domain is defined as $d_{pick-and-place_domain_partial} = \{m_{moveobject_p}, m_{moveobject2_p}, m_{getobject_p}, m_{getobject2_p}, m_{putobject_p}, o_{navigate\text{-}op}, o_{pickup\text{-}object}, o_{drive\text{-}to\text{-}dexterous\text{-}workspace}, o_{drive\text{-}away\text{-}f\text{-}dex\text{-}workspace}, o_{put\text{-}object}\}$. The domain is asserted in the HDL system as shown in Figure 3.14 and detailed in Listing B.8.

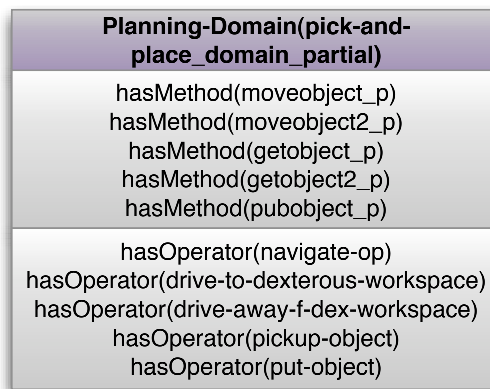


Figure 3.14.: Planning domain assertion in the HDL for pick-and-place domain.

An analysis of how the domain is generated from the HDL system for solving certain problems is shown later in Section 3.4.4. The difference between the partial domain and the complete one is described in the following section.

3.4.3.2. Complete Pick-and-Place Domain

The complete pick-and-place domain combines the navigation domain and the partial pick-and-place domain. It uses a complete network of the navigation domain. Therefore, a benefit of the HDL system is the enhanced re-usability of the planning domain. No new definition is needed for the navigation part. However, for the manipulation part, only the operators are applied exactly for both domains, partial and complete. The methods in the complete pick-and-place domain are very similar to the partial one, except for the method that involves the `navigate` task.

Figure 3.15 shows the task network for the complete pick-and-place domain. The navigation operator is replaced by the navigation method from the navigation domain. Hence, the generated plan is more granular than the partial one. The properties which have been defined in the navigation domain are also applied to the complete pick-and-place domain. For example, instance of *Room* is being used.

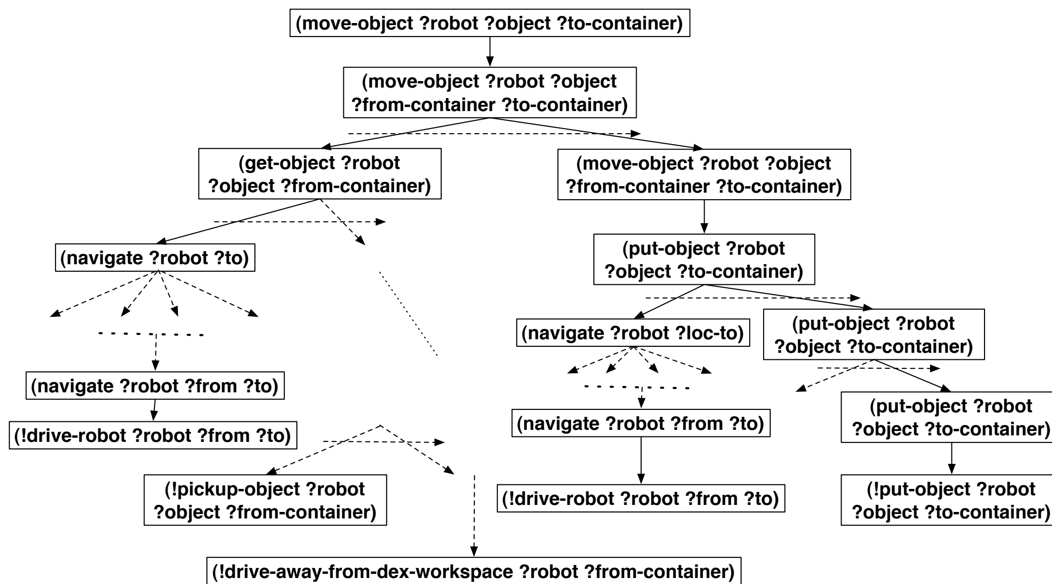


Figure 3.15.: Complete pick-and-place domain task network.

The complete pick-and-place domain has seven methods, of which five are from the partial pick-and-place domain and two from the navigation domain. The `get-object` method with three variables has four cases as in the partial pick-and-place domain (see Section 3.4.3.1). The first three cases are identical to those in the partial one, but the fourth one is not. Below is the definition of the fourth case of the `get-object` method in the complete pick-and-place domain:

`(get-object ?robot ?object ?from-container)` *;; case 4*

task: `get-object(robot, object, from-container)`

subtasks: $u_1 =$ `navigate(robot, room)` *;; ** method calls ***

$u_2 =$ `get-object(robot, object, from-container)`

constr: $u_1 \prec u_2$, `not(at-dexterous-workspace(robot, from-container))`, `at(from-container, room)`, `not(at(robot, room))`

Similarly, the `put-object`'s fourth case is redefined as follows:

`(put-object ?robot ?object ?to-container)` *;; case 4*

task: `put-object(robot, object, to-container)`

subtasks: $u_1 =$ `navigate(robot, room)` *;; ** method calls ***

$u_2 =$ `put-object(robot, object, to-container)`

constr: $u_1 \prec u_2$, `at(object, robot)`, `not(at-dexterous-workspace(robot, to-container))`, `at(to-container, room)`, `not(at(robot, room))`

These methods are almost identical to the ones in the partial pick-and-place domain. Instead of calling the `navigate` operator, it is now calling the `navigate` method. Hence, these methods can be copied from the partial pick-and-place domain. However, they should be represented as instances in the HDL system. Due to the uniqueness requirement in the instance of the *ABox* in DL, the introduction of new instance names for these methods is necessary. The complete pick-and-place domain has the suffix '`c`' at the end of the method names instead of '`p`', which are used by the partial pick-and-place domain.

The method assertions are depicted in Figure 3.16 and the detail is in Listing B.9. In addition to these methods, a planning domain instance for the complete pick-and-place one is inserted into the *ABox* as shown in Figure 3.17 and detailed in Listing B.10.

3.4.4. Results

In the HDL system, the domains can coexist with each other. A domain does not conflict with another but instead contributes to the increase in re-usability. In this section, two issues of the HDL system will be considered. The first issue is the number of stored planning domains and their effect on the generated ones. The second issue is the difference between the two pick-and-place domains, namely the partial pick-and-place domain and the complete one.

In the experiment, the HDL system has three instances of $Planning\text{-}Domain$, 12 instances of $Method$, and eight instances of $Operator$. As previously explained in Section 2.5.1, from an instance of $Method$ a valid planning domain can be generated. Hence, in this experiment, 15 possible planning domains can be generated by the system. Table 3.2 shows the number of involved methods and operators for different objectives. Regardless of which method the user chooses, the HDL system generates a valid HTN planning problem for the planner to extract.

Let us now look into more concrete objectives of the pick-and-place task by referring

Figure 3.16.: Method assertions in the HDL for complete pick-and-place domain.

to the problem shown in Figure 3.7. Two robots are in the environment, namely `robot1` in `room-1` and `robot2` in `room-4`. Three different tasks will be proposed within the environment to differentiate between the partial pick-and-place domain and the complete one.

The first task is to let the robots move `orange1` from `room-11` into `trash-6` in `room-6`. In the first approach, two objectives will be provided separately to the HDL system. The first objective is (move robot1 orange1 trash-6) and the second is (move robot2 orange1 trash-6). These objectives will be run on both the partial and complete domains.

The first run for `robot1` in the partial domain gives the following output:

```
Plan cost: 7.0
____________________
(!navigate robot1 room-11)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange1 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!navigate robot1 room-6)
(!drive-to-dexterous-workspace robot1 trash-6 room-6)
(!put-object robot1 orange1 trash-6)
```

The HDL system extracts 13 states in the planning problem description which is enough for producing the output shown above. In the output, the first action is to navigate the robot into `room-11` where `basket-11` is located. The user never defined or told the system where `orange1` is located in the environment. However, the system could extract this necessary information, for example in which container `orange1` is located, and provided this information in the planning problem. Hence, the sequence of actions which is produced is a valid one. The following output is produced by the complete planning domain (the dummy operators are omitted):

Figure 3.17.: Planning domain assertions in the HDL for complete pick-and-place domain.

```
Plan cost: 32.0
_______________
(!drive-robot robot1 room-1 corridor-1)
(!drive-robot robot1 corridor-1 corridor-2)
(!drive-robot robot1 corridor-2 corridor-3)
(!drive-robot robot1 corridor-3 room-11)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange1 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!drive-robot robot1 room-11 room-9)
(!drive-robot robot1 room-9 corridor-3)
(!drive-robot robot1 corridor-3 corridor-2)
(!drive-robot robot1 corridor-2 corridor-1)
(!drive-robot robot1 corridor-1 room-6)
(!drive-to-dexterous-workspace robot1 trash-6 room-6)
(!put-object robot1 orange1 trash-6)
```

The complete planning problem has 46 states which are generated and deduced automatically by the HDL system. These states are produced by applying the $DriveableRoom$ filter on the navigation domain. The number of actions increases from seven to 32 and includes the real navigation of the robot from one location to another in order to achieve the goal. Figure 3.18 shows the sequence of actions excluding the dummy operators from the navigation domain. From this figure, it is clear that the complete pick-and-place domain is a combination of the partial pick-and-place domain and the navigation domain. The HDL system provides the user with the freedom to choose to which level of the solution the system should extract. This may be reasonable in a very dynamic environment or in a very large one. In these cases, it may be useful to extract only partial solutions. Due to the dynamic nature of the environment, the generated plans might be obsolete during plan execution anyway. Hence, solving some of the problem up to certain level and another part during the execution might save time and provide

Table 3.2.: Generated planning domain description from different instances of $Method$ or $Planning\text{-}Domain$.

Instance Name	# of Generated Methods	# of Generated Operators
$d_{navigation_domain}$	2	3
$m_{navigate}$	1	3
$m_{navigate2}$	2	3
$d_{pick\text{-}and\text{-}place_domain_partial}$	5	5
$m_{moveobject_p}$	3	5
$m_{moveobject2_p}$	4	5
$m_{getobject_p}$	1	4
$m_{getobject2_p}$	2	4
$m_{putobject_p}$	1	3
$d_{pick\text{-}and\text{-}place_domain_complete}$	7	7
$m_{moveobject_c}$	5	7
$m_{moveobject2_c}$	6	7
$m_{getobject_c}$	3	6
$m_{getobject2_c}$	4	6
$m_{putobject_c}$	3	5

the robot with the most recent information regarding the environment.

The problem becomes more interesting when one of the involved planning domains is not solvable. An example of this is the task of `robot2` which is trapped in `room-4`. The generated plan for this task with the partial pick-and-place domain is shown below:

```
Plan cost: 7.0
____________________
(!navigate robot2 room-11)
(!drive-to-dexterous-workspace robot2 basket-11 room-11)
(!pickup-object robot2 orange1 basket-11)
(!drive-away-from-dex-workspace robot2 basket-11)
(!navigate robot2 room-6)
(!drive-to-dexterous-workspace robot2 trash-6 room-6)
(!put-object robot2 orange1 trash-6)
```

The planner has basically produced the same amount of states in the planning problem description. Even the generated plans are identical for `robot2` by replacing the actor with itself. Up to now, the plans are still valid. However, the complete pick-and-place domain produces no solution. Figure 3.19 illustrates the sequence of actions for `robot2`. When the system tries to extract (navigate robot2 room-11), it produces no solution because the robot is trapped in the room. The partial problem is still able to generate an output as if the robot is not trapped. However, the complete one is a combination of both where the “and” operand is applied to both domains. Hence, the final result is “no-solution”. This is of course true for the current state. The robot would not be able the execute the rest of the plan anyway.

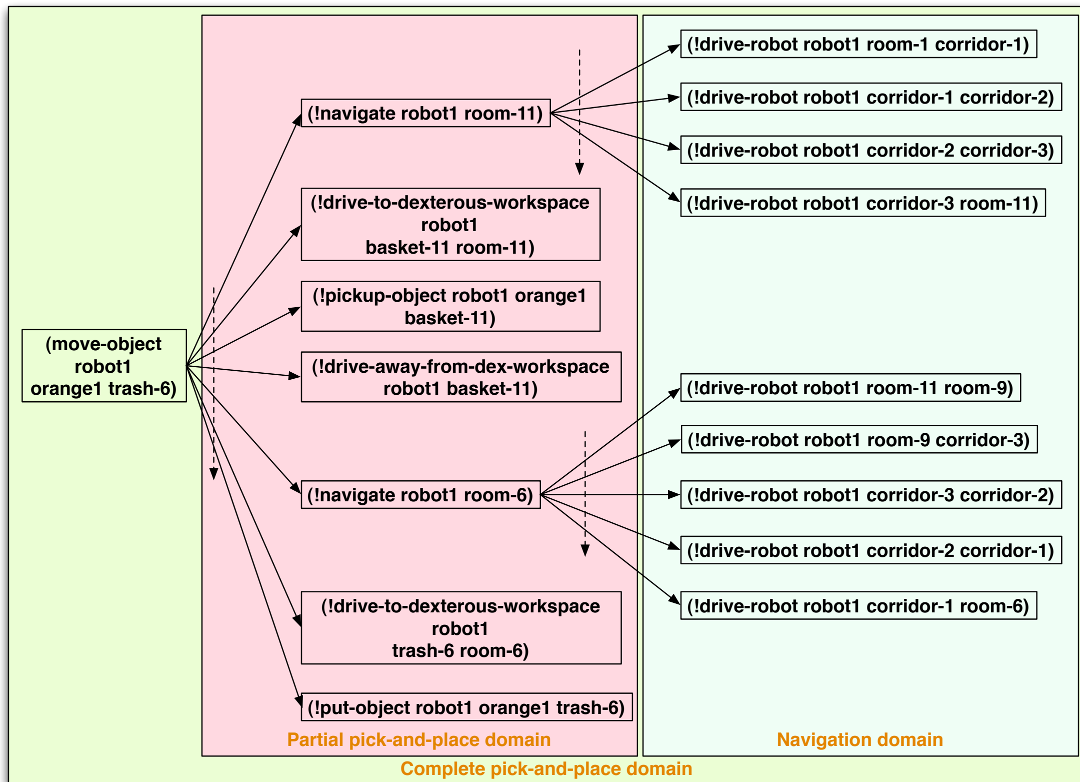


Figure 3.18.: The complete pick-and-place plan and its relation to the partial pick-and-place plan and the navigation domain.

The second task has been chosen to show the effect of combining both objectives into one planning problem. The planner can have multiple objectives, all of which are connected to each other conjunctively. In other words, all the objectives must be fulfilled. In this task two objectives, namely (move robot1 orange1 trash-6) and (move robot2 orange1 trash-6), are given to the HDL system. The following is the output of the partial pick-and-place domain:

```
Plan cost: 13.0
________________________
(!navigate robot1 room-11)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange1 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!navigate robot1 room-6)
(!drive-to-dexterous-workspace robot1 trash-6 room-6)
(!put-object robot1 orange1 trash-6)
(!navigate robot2 room-6)
(!drive-to-dexterous-workspace robot2 trash-6 room-6)
(!pickup-object robot2 orange1 trash-6)
(!drive-away-from-dex-workspace robot2 trash-6)
(!drive-to-dexterous-workspace robot2 trash-6 room-6)
(!put-object robot2 orange1 trash-6)
```

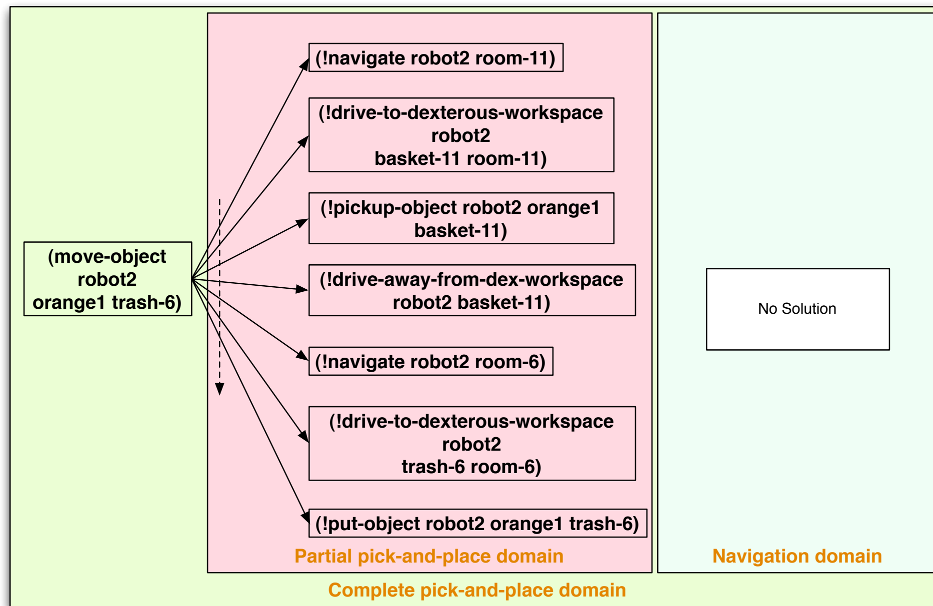


Figure 3.19.: Illustration the solution plan for `robot2`.

The output shows which actions need to be performed by which robot in order to fulfil the tasks. The generated planning problem description for the partial domain has the same number of states as the previous example, namely 13. In the complete planning domain, the number of states remain the same as in the previous one, 46 states. However, the planner returns no result, because one of the objectives can not be achieved.

The third task is to enter multiple objectives into the HDL system where these objectives are achievable. The new task would be to let the robot move every orange into the trash. Therefore, three objectives are given to the system: (move robot1 orange1 trash-6), (move robot1 orange2 trash-6), and (move robot1 orange3 trash-6). The following is the output of the planner:

```
Plan cost: 16.0
______________
(!navigate robot1 room-11)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange1 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!navigate robot1 room-6)
(!drive-to-dexterous-workspace robot1 trash-6 room-6)
(!put-object robot1 orange1 trash-6)
(!navigate robot1 room-11)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange2 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!put-object robot1 orange2 trash-6)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange3 basket-11)
```

```
(!drive-away-from-dex-workspace robot1 basket-11)
(!put-object robot1 orange3 trash-6)
```

The output of the complete pick-and-place domain is shown below (without the dummy operators):

```
Plan cost: 52.0
--------------------
(!drive-robot robot1 room-1 corridor-1)
(!drive-robot robot1 corridor-1 corridor-2)
(!drive-robot robot1 corridor-2 corridor-3)
(!drive-robot robot1 corridor-3 room-11)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange1 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!drive-robot robot1 room-11 room-9)
(!drive-robot robot1 room-9 corridor-3)
(!drive-robot robot1 corridor-3 corridor-2)
(!drive-robot robot1 corridor-2 corridor-1)
(!drive-robot robot1 corridor-1 room-6)
(!drive-to-dexterous-workspace robot1 trash-6 room-6)
(!put-object robot1 orange1 trash-6)
(!drive-robot robot1 room-6 corridor-1)
(!drive-robot robot1 corridor-1 corridor-2)
(!drive-robot robot1 corridor-2 corridor-3)
(!drive-robot robot1 corridor-3 room-11)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange2 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!put-object robot1 orange2 trash-6)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange3 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!put-object robot1 orange3 trash-6)
--------------------
Plan cost: 49.0
--------------------
(!drive-robot robot1 room-1 corridor-1)
(!drive-robot robot1 corridor-1 corridor-2)
(!drive-robot robot1 corridor-2 corridor-3)
(!drive-robot robot1 corridor-3 room-11)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange1 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!drive-robot robot1 room-11 corridor-3)
(!drive-robot robot1 corridor-3 corridor-2)
(!drive-robot robot1 corridor-2 corridor-1)
(!drive-robot robot1 corridor-1 room-6)
(!drive-to-dexterous-workspace robot1 trash-6 room-6)
```

```
(!put-object robot1 orange1 trash-6)
(!drive-robot robot1 room-6 corridor-1)
(!drive-robot robot1 corridor-1 corridor-2)
(!drive-robot robot1 corridor-2 corridor-3)
(!drive-robot robot1 corridor-3 room-11)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange2 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!put-object robot1 orange2 trash-6)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange3 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!put-object robot1 orange3 trash-6)
```

In this case, the planner returns two sequences which both fulfill the planning objectives. The difference is in the number of actions that is needed by each one. One can see the number of actions in the planning cost. In this case, no specific planning cost is defined in the planning domain. Hence, every single operation costs one unit. However, one can also use the cost function in order to optimise the plan in such a way. This extension will be discussed in Chapter 7. In addition, some extension to the current example is presented in the discussion.

4. Case Study: "Johnny Jackanapes"

In the previous chapter, two robotics domains were implemented using the HDL system. This chapter presents a case study that uses the HDL system in a mobile robot. This example demonstrates how the HDL system may be integrated within an existing robotic system.

In this case study, the robot "Johnny Jackanapes" has participated thrice in the RoboCup@-Home competition as part of the b-it-bots team [HPB+08, HPB+09]. We did quite well in the competition coming in second place at the RoboCup@Home world championship 2008 in SuZhou, China, and in first place at the RoboCup@Home German Open 2009 in Hannover, Germany, and in first place at the RoboCup@Home world championship 2009 in Graz, Austria. The system presented in this chapter is based on the state of the system at the RoboCup@Home world championship 2008. We were one of few teams that could perform mobile manipulation during this competition.

First, a brief introduction to the RoboCup@Home competition is given. The tasks and challenges of this competition are presented. A description of the robot and its components is then provided. One of the tests is detailed in this section to provide an example of the integration of the HDL system. Finally, the HDL system implementation within this robot is explained.

4.1. RoboCup@Home

RoboCup, initially called "The Robot World Cup Initiative", is an endeavour for promoting a wide problem in AI into the AI and intelligent robotics research community that lead to integration of a wide range of technologies [KAK+95]. Originally, the standard test was the robot soccer competition. The first competition was held at the International Joint Conference on Artificial Intelligent (IJCAI) 1997 in Nagoya, Japan [KAK+97]. The research interest within the community grew and, as a result, more leagues were established. These leagues are Humanoid League, RoboCup Junior, RoboCup Rescue and RoboCup@Home.

RoboCup@Home is one of the youngest leagues in the RoboCup competition. The first competition was held in 2006. The main purpose of the RoboCup@Home competition is to test the robot's capability in the mobile robotics domain. It needs methodical approaches to integrate several different technologies and functionalities into a robust service robot. The teams must test their robots in dynamic environments outside of their labs.

4.1.1. Test Scenarios

According to the 2008 rule book, the competition consists of three stages. The first stage consists of five tests and one *open challenge*. Each team must participate in three tests within this stage. The second stage consists of four tests and one *demo challenge*. In this stage, each team has three test slots. The team can perform three tests from the four possibilities. If the team is not satisfied with the results from the first or second slots, it can decide to redo the same test as previously performed in the next slot. If a team decided to do the same test twice or even thrice in the second stage, the score is taken from the best score for this test. If the tests consist of two or three different tests, the score of the second stage is the sum of the distinctive tests. The third stage is the *final* that allows the teams to showcase the robot’s capabilities to the jury. Figure

Figure 4.1.: (a) Introduce (RoboCup 2008), (b) Fetch & Carry (RoboCup 2009), (c) Fast Follow (German Open 2009), (d) Supermarket (German Open 2009), (e) Party Bot (RoboCup 2009), (f) Final (RoboCup 2008).

4.1 shows snapshots of Johnny in some tests.

The tests in the first stage are *introduce*, *fast follow*, *fetch & carry*, *who's who*, and *competitive lost and found*. Every team must perform the *introduce* test, in which the robot should introduce itself to the audience. *Fast follow* tests the robot's mobility as the robot should follow a person leading it. *Fetch & carry* tests the robot capability to understand commands, detecting objects and manipulating them. *Who's who* is a test of the person-identification capabilities of the robot. *Competitive lost and found* is the test of the robot's abilities to navigate and detect objects in the environment.

The second stage's tests are more difficult than those of the first one as they either combine some tests into one or add more tasks to some tests. These tests are *partybot*, *supermarket*, *walk & talk*, and *cleaning up*. *Partybot*, or who's who reloaded, is a more advanced test of the person identification, navigation and mapping capabilities. The difference between it and stage one's *who's who* is that the robot must search for a person, identify her/him, remember her/him and offer a drink to the person. Hence, the robot must not only remember the person but also the location where it encountered this person before. *Supermarket*, or lost & found reloaded, is a test where the robot receives a request from a person and delivers the requested objects. In this test, the robot has to identify the correct object among other objects before grasping it and handing it over to the person. *Walk & talk* is a more advanced *fast follow* test where the robot not only follows a person but also remembers some locations that are shown by the followed person in the first phase. The second phase of the test is the navigation phase, where the robot should receive a sequence of "learnt" places that it should navigate to. *Cleaning up* is a test where the robot should collect objects lying around in the environment and bring them to a defined place.

4.1.2. Challenges

In RoboCup@Home, the robot must perform all the tests autonomously. The team is only allowed to touch the robot at the beginning of each test. The purpose of having autonomous systems in each test is to test the robot's performance out of the box. As service robots, they are designed to serve human beings and in the future can be purchased just as any household appliance. The robot and human should interact in a natural manner, for example using a speech recognition system or gesture commands. Thus, another challenge in this competition is to have a robust HRI (Human Robot Interaction) system [GS07]. As such, HRI is not limited to speech only, but also includes vision for recognising face or gestures.

The basic feature of service robots is their mobility. Hence, each robot must have a good navigation system. Some tests require the capability of the robot to create a map and localise itself within the environment. This is the motivation for one research area in mobile robotics that deals with maps such as SLAM (Simultaneous Localisation And Mapping). In order to be able to perform all tests, the robot should be equipped with a manipulator. Hence, mobile manipulation is another challenge in RoboCup@Home. Another important challenge is integration. The robots are composed of various software and hardware which must function as a single, efficient,

robust working system.

4.2. Johnny Jackanapes, The Robot

Our robot "Johnny Jackanapes" is built as a mobile manipulator such that it can perform most of the tests in the RoboCup@Home scheme. Its built on the VolksBot [SBC+08, WNB06] under-carriage and is equipped with a Katana manipulator. Figure 4.2 shows the robot, "Johnny Jackanapes". Details of its hardware and software systems are presented in the following sections.

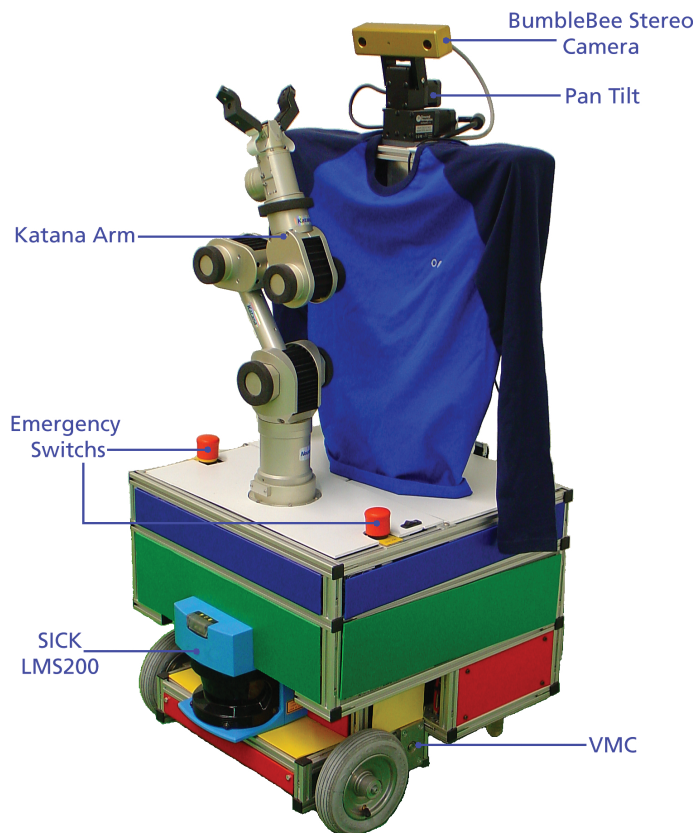


Figure 4.2.: Hardware specification of Johnny Jackanapes.

4.2.1. Hardware Components

Johnny Jackanapes's components are mounted on the Volksbot platform. The overall platform dimensions are $51\,cm$ wide, $51\,cm$ long and $120\,cm$ high. Its weight is around $50\,kg$. Its maximum velocity is $2\,m/s$. The following are the detailed specifications of the robot's components as shown in Figure 4.2:

- Volksbot Motor Controller (VMC)
  VMC is the drive unit used for locomotion. It controls the two actively driven wheels of

the differential drive. These motors have $150\,Watts$ of power. Besides these two wheels, two castor wheels are attached to the robot to enhance the robot's rotation and stability under load.

- Neuronics Katana 6M180 robot arm
  The Katana 6M180 is a light weight manipulator which is mounted on the robot in such a way as to provide good manoeuvrability and reachability for object grasping. It can lift an object weighting up to $500\,grams$. It has six motors giving five degrees of freedom and gripping capability. Its workspace radius is $60\,cm$. The gripper is equipped with infrared and force sensors. Thus, it can detect whether an object is present within the gripper and also whether it has an object grasped.

- SICK LMS 200 laser range finder
  The laser range finder is the primary sensor of the navigation and localisation module. It is used for perceiving environmental structures. It can accurately measure the surrounding environment within an angle of $180\,^{\circ}$ within a 2D scan plane.

- Pan-Tilt system
  In order to track an object in the front of the robot, a pan tilt system is mounted underneath the cameras. It is used to control the camera view, especially during the object detection or person identification processes. During object grasping, the camera can be pointed toward the object using the pan-tilt system such that a more accurate distance approximation can be obtained from the camera images.

- Bumble Bee camera
  This is a stereo camera that is used mainly by the manipulation module. It is used for detecting objects and measuring their distance from the robot. The disparity images can then be calculated to obtain the pose of the object relative to the camera pose. Its pose is then transposed into the robot pose and used to control the robot and its manipulator to grasp the object.

- Logitech web camera
  This camera is not shown in Figure 4.2. It is usually mounted on top of the Bumble Bee camera. It is an ordinary web camera that delivers high resolution images for the person identification process.

4.2.2. Software Components

The robot consists of several software components. These components are organised based on their functionality. These are described here.

- Navigation and Localisation module
  This module provides the navigation and locomotion functionality of the robot. It receives

the sensor readings from the laser range finder and odometry for its localisation and mapping tasks. It uses SLAM based on Iterative Closest Point (ICP) techniques [HKR09]. The VMC controller is connected and controlled by this module too. It provides basic navigation commands to the sequencer.

- Object Detection and Manipulation module
  Object detection and manipulation are organised as one module due to their need to closely cooperate with each other. In order to grasp an object, the robot needs to recognise the object. It then needs to compute the relative pose of the object to the robot. The pose is then obtained through inverse kinematics which and is used to move the robot arm.

- Speech Recognition module
  The speech recognition module is used for recognising user commands. It parses the speech commands into robot-understandable commands. It should be speaker independent. In order to avoid false recognition, each command must start with the robot's name, "Johnny".

- Face Recognition module
  The face recognition module is used for recognising a person. It can distinguish whether a person is in front of the camera or not. It uses facial landmarks from eyes, mouth and nose. Each person has a special constellation of these properties, from distance between eyes to the locations of the mouth and nose. It distinguishes eye colour and also facial colour. Thus, this module not only distinguishes person from object, but also identifies her/him.

- Speech Synthesis module
  The robot communicates with the human in a natural way. Therefore, it not only listens to the speech commands but it also responds audibly to the human. This module is used as text to speech by synthesising it. It is implemented using Mac OS X speech API, which is currently known to be the best natural speech synthesis program that freely available within its operating system. It uses the predefined profile, "Alex".

- Sequencer module
  The sequencer module is responsible for executing pre-programmed sequences of actions. These sequences of actions represent certain tasks for the robot. It reads states from other component modules and sends the given commands to these modules.

As mentioned in Section 2.1 the robot system is usually built with a given architecture. Johnny Jackanapes's architecture is implemented using component based approaches [Ore04]. Thus, each component or module has its own structural architecture and provides a certain functionality to the overall system.

Johnny's components are implemented using the best of approaches regardless of which operating system the components run on. Therefore, Johnny is equipped with three notebooks with three operating systems, namely Windows, Mac OS X, and Linux. These components need to communicate with each other. They need to exchange information or send commands to each other.

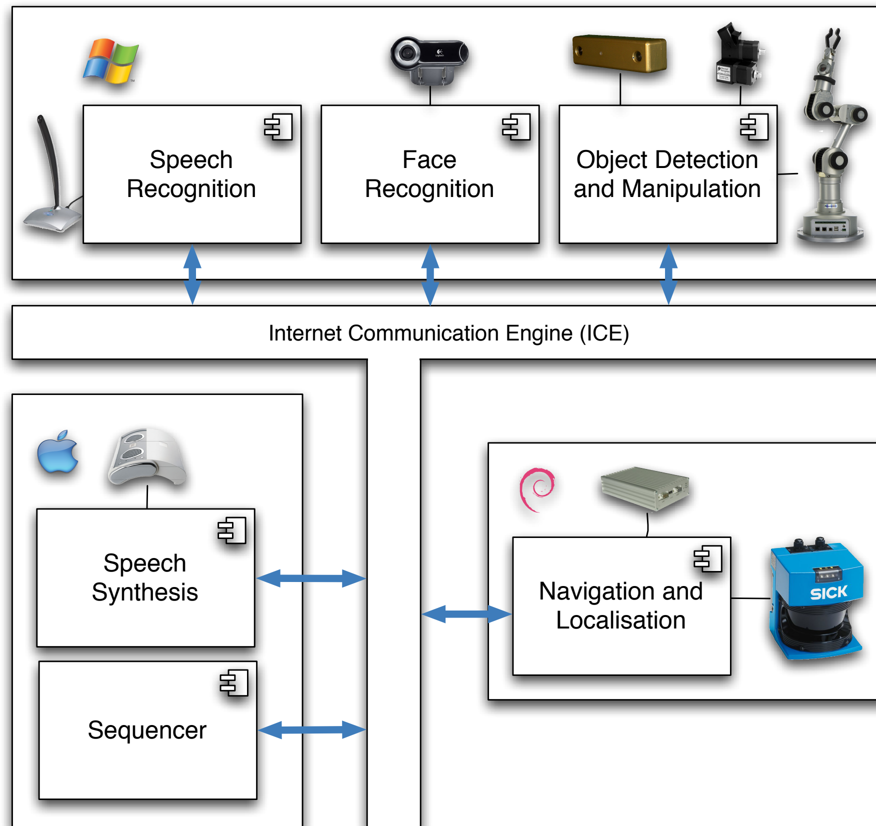


Figure 4.3.: Components and operating systems platforms.

These modules are connected to each other through a communication middleware. Example of available middleware are the Common Object Resource Broker Architecture (CORBA) [Gro04], Internet Communication Engine (ICE) [Hen04, HS07], and Distributed Component Object Model (DCOM) [GG97]. Johnny uses ICE as its middleware due to its wide support in term of operating systems and programming languages. In addition, the ICE installation procedure is more transparent than CORBA's.

The components are connected with each other through a client-server or point-to-point communication protocol. Each component defines its own interface in an Interface Definition Language (IDL). In ICE, the IDL is SLICE. A component implements the skeleton of any other component with which it should communicate. Each function call is sent using a Remote Procedure Call (RPC).

Figure 4.3 shows how the hardware components are connected with the software components. Three components are implemented in the Windows operating system, namely “speech recognition”, “face recognition”, and “object detection and manipulation”. Two components are implemented using Mac OS X, namely “speech synthesis” and “sequencer”. Finally, the “navigation and localisation” is implemented in the Debian Linux platform. These components are implemented in such way as to guarantee the real time requirement for controlling the robot. For example, the laser scanner is used directly by the navigation and localisation component and also has direct access to the motor controller. High level communication is used for exchanging the pose of the robot or sending move commands to it. In a similar manner, the manipulator, pan-tilt mechanism, and bumble bee camera are directly connected to the object detection and manipulation component. Hence, the component can get raw images from the camera and process them for retrieving the object’s pose with the required speed. In some cases, one component might need access to other hardware components. For example, the face recognition component has to control the pan tilt unit in order to change the camera’s viewing angle. The “object detection and manipulation” component provides some ICE interfaces for controlling the pan tilt unit. The face recognition module uses these ICE interfaces in order to change the angle.

4.2.3. Applications

As previously mentioned, the work presented here is based on the state of the robot system during the 2008 RoboCup World Championship in China. Johnny was not equipped with any deliberative layer, hence, it had no planner for performing the tasks. The sequencer is programmed in such a way as to solve particular scenarios. In most of the test cases, a planner is not really necessary. Particularly, tasks that have a fixed ordering of their actions. For example, *introduce*, *fast follow*, *who’s who*. Although, the other tasks can be solved without any planner, the sequencer does not allow flexibility. Any changes of the sequence ordering will require reprogramming.

The *fetch & carry* task is more complicated than the other tasks. It is similar to the pick-and-place domain that is presented in the previous chapter. Unfortunately, Johnny was only able to complete half the task. It did not go any further due to its collision avoidance mechanism that prevented Johnny from moving close enough to grasp the object. In the *open challenge*, Johnny was performing a task similar to the *fetch & carry* one. He successfully completed this task. Hence, the *open challenge* scenario is used here as an example of how the HDL system could be integrated into an existing system.

In the *open challenge*, Johnny’s task was to bring a coke to a guest in the armchair. Below is the dialogue between the user and Johnny, in which the user tells Johnny to bring a coke to the guest in the armchair. In this situation, Johnny knows where all the drinks are placed. However, he did not know where the guest was sitting. In this dialogue, Johnny’s ability to understand commands and request missing information from the user is displayed. Due to the noise in the

arena, it is necessary to add confirmation questions for each successfully recognised command in order to minimise false positives.

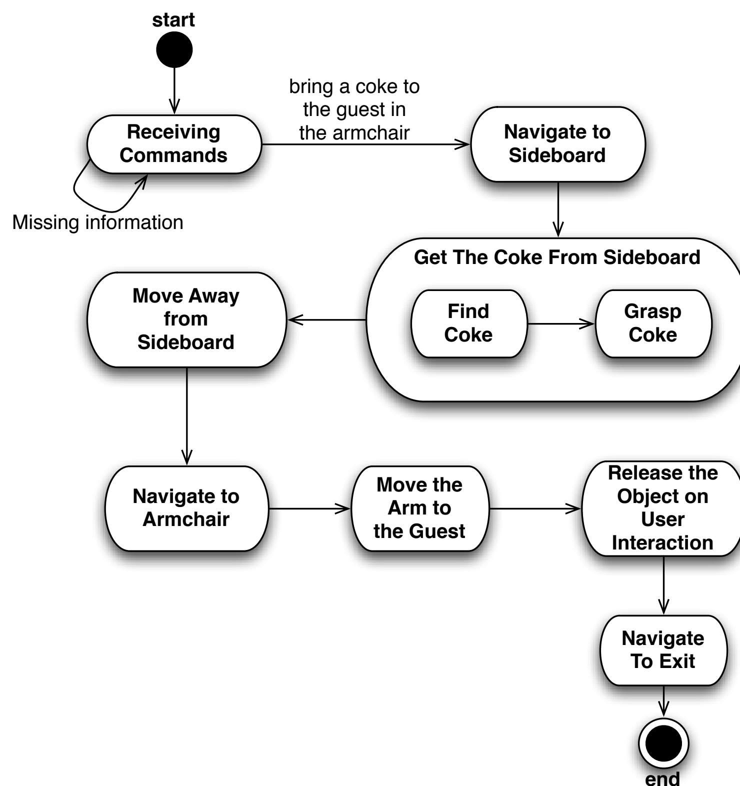


Figure 4.4.: State machine diagram of the open challenge scenario.

User: "Johnny, bring our guest a drink please!"

Johnny: "I understood, you want me to bring a drink to the guest.
Is this correct?"

User: "Johnny yes."

Johnny: "What kind of drink should I bring to the guest?"

User: "Johnny, a coke."

Johnny: "I understood, you want me to bring a coke to the guest.
Is this correct?"

User: "Johnny yes."

Johnny: "You said yes.
Okay I will bring the coke to the guest.
Please tell me where the guest is?"

User: "Johnny, in the armchair."

Johnny: "I understood you, the guest is sitting on the armchair.
Is this correct?"

User: "Johnny yes."

Johnny: “You said yes.
Okay I will bring the coke to the armchair now. ”

The dialogue above was completed, because the command matched the pattern that Johnny has, namely “bring <an object> from <location> to <location>”. In this case, Johnny had three objects in his memory, namely “green tea”, “yellow tea”, and “coke”. All of these objects are located on the sideboard, thus it was enough to tell Johnny what object to grasp. The destination is where the guest is sitting, namely “armchair”. This dialogue was the first state in the open challenge state machine, as depicted in Figure 4.4. After all necessary information was gathered the command was executed.

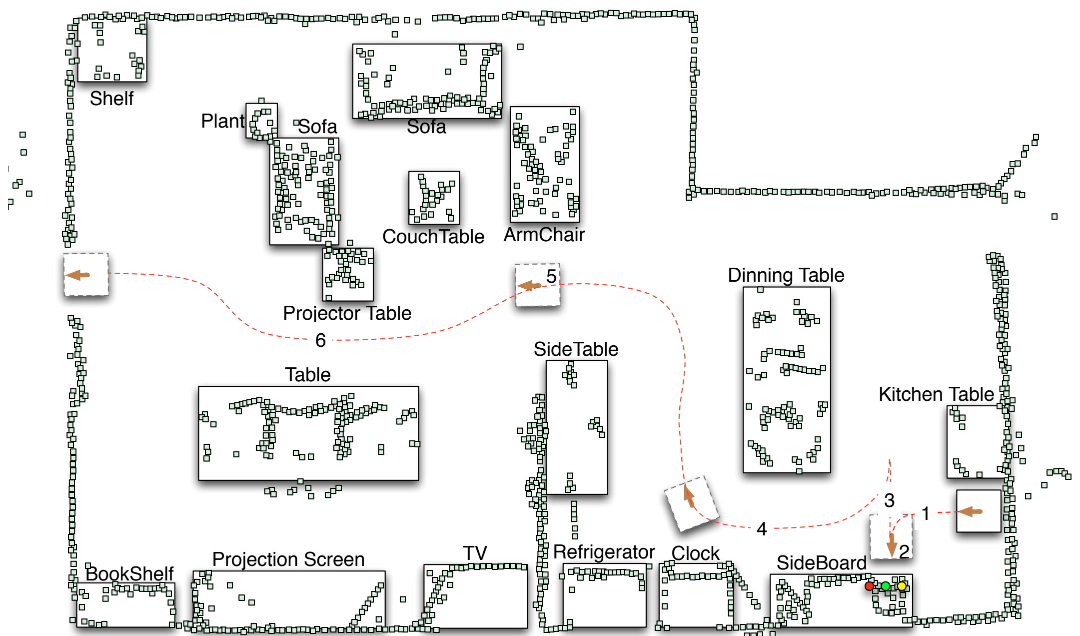
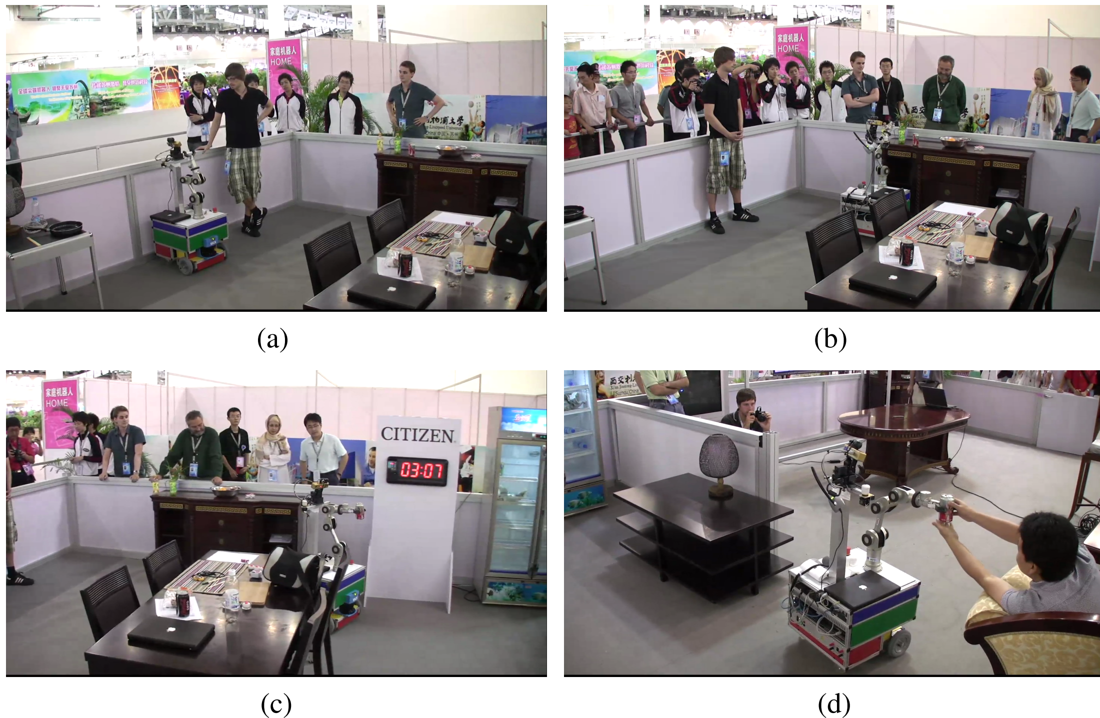


Figure 4.5.: RoboCup@Home arena points of cloud from mapping module with Johnny trajectories during open challenge scenario.

Figure 4.5 depicts the map of the RoboCup@Home arena that was captured through the SICK laser range finder. Some boxes are drawn manually to show what objects are represented by which points. In this figure, Johnny is depicted as a small box with an arrow showing his direction. In addition, the paths that Johnny travelled is also depicted in the map.

By referring to both Figures, 4.4 and 4.5, the actions and trajectories can be assigned. The first action, corresponding to trajectory 1, is the “navigate to sideboard” action. The second action is the “get the coke from sideboard” action which consists of “find coke” and “grasp coke”. After successfully grasping the coke, Johnny performed the third action which is “move away from sideboard”. Next the “navigate to armchair” action, shown by trajectory 4 was

performed. Action 5 consists of the two actions “move the arm to the guest” and “release the object on user interaction”. Finally, action 6, “navigate to exit” was performed. The last action is actually not part of the command, however, Johnny sensed his battery power was depleted and he needed to recharge. Thus, he decided to leave the arena after successfully completing the given task. In the RoboCup@Home scenarios, the robot gets additional points if it can leave the arena autonomously.



(a) (b) (c) (d)

Figure 4.6.: (a) Johnny receiving a command, (b) grasping the coke, (c) bringing the coke to the guest, (d) handing the coke to the guest.

Figure 4.6 shows a snapshot of the open challenge scenario. The first snippet (a) shows the user giving the command to Johnny. Snippet (b) shows Johnny grasping the coke on the sideboard. Snippet (c) shows Johnny holding the coke while moving to the armchair and snippet (d) shows Johnny passing the coke to the guest, who received the coke from him. Video footage of Johnny’s performance at the RoboCup@Home competition in SuZhou, China can be watched at the following location: “*http://www.b-it-bots.de/Media/Media.html*” under the section “RoboCup@Home WM 2008”.

4.3. The HDL system in Johnny Jackanapes

In the previous section, the details of Johnny’s components and one of the tasks in RoboCup@ Home are presented. This task was performed without any deliberative layer, but with simple pre-programmed state machines. In this section, the HDL system is implemented in Johnny to solve the flexibility problem mentioned above. The HDL system enhances Johnny’s system

in two ways. Firstly, it is used as a knowledge base component. Secondly, it is used as a deliberative layer for planning Johnny’s actions. Before the details of these two implementations are presented, a description of how Johnny’s hardware and software components, as presented in Section 4.2, fit into the HDL architecture, which is shown in Figure 2.2 is given.

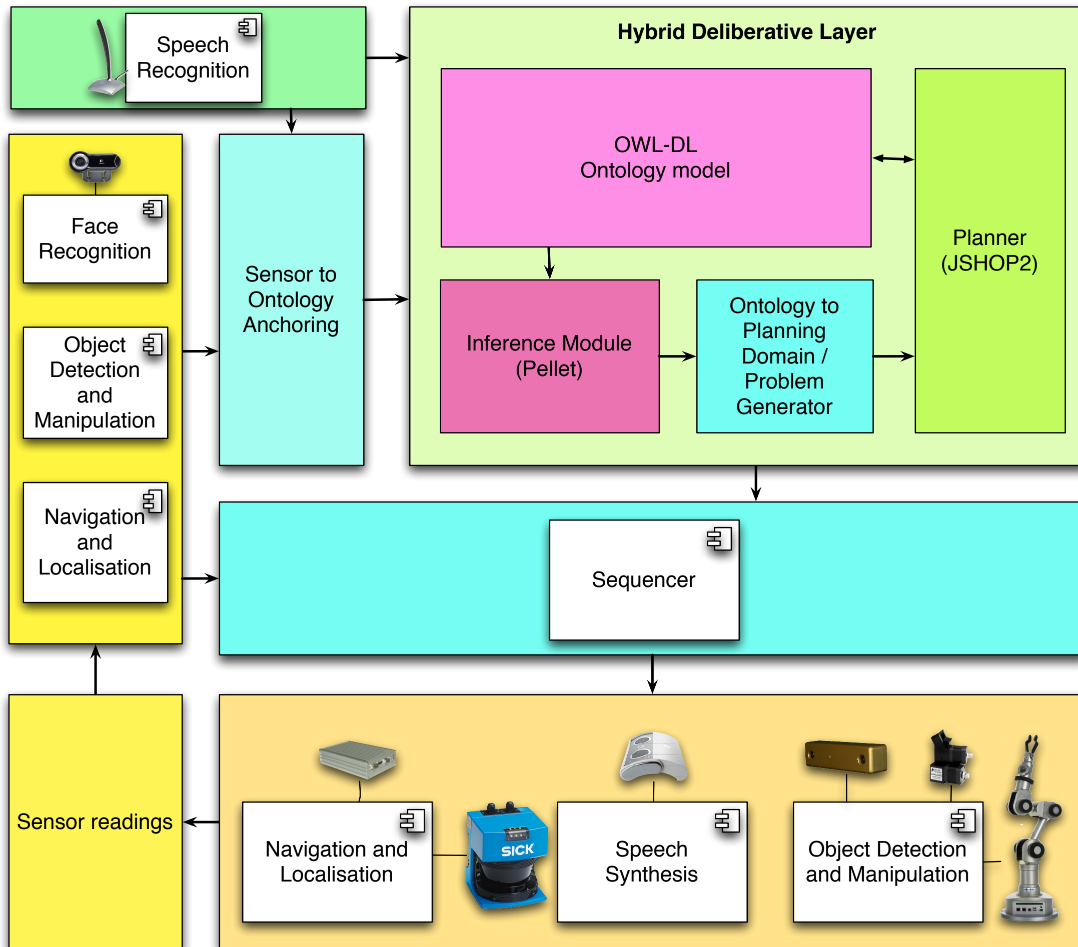


Figure 4.7.: HDL system architecture in Johnny Jackanapes.

Figure 4.7 shows the HDL architecture in Johnny. Johnny’s components are drawn inside the HDL architecture to show which components are responsible for which block in this architecture. Two components, namely “object detection and manipulation” and “navigation and localisation”, appear in two different blocks because these components serve as both actuators as well as sensor processing. Speech recognition component provides the input to the HDL system. It communicates with the HDL system in order to check the completeness of the input, see Section 4.3.1.3 for details. It has access to the speech synthesis module to provide the speech output to the user, in case additional information is required.

The output of the speech recognition is the goal input for the HDL system. The HDL system produces some possible solution plans for the sequencer modules. The plan extraction

is detailed in Section 4.3.2. The sequencer component sorts the solution plans by their costs and then selects the solution plan with the smallest cost. It executes and monitors the sequence of actions by calling the appropriate ICE interfaces of the other software components.

Three software components are placed in the bottom layer as the actuators. These components are “navigation and localisation”, “speech synthesis”, and “object detection and manipulation”. The functions of these components are described previously in Section 4.2.2.

The perception layer has three components, namely “face recognition”, “object detection and manipulation”, and “navigation and localisation”. These components process the sensor reading from the hardware components that connect to them. The sensor readings layer in this architecture represents processes that happen inside those software components. The processed information is fed into the sequencer and to the sensor-to-ontology anchoring layer. The sequencer needs this information as feedback from the current action, such that it can monitor the execution process by deciding whether the current action was successfully performed or not. The sensor-to-ontology anchoring is the interface that translates the information from these components to update the HDL model. Section 4.3.1 shows how the ontology is modelled and which components are using or updating it.

4.3.1. Using the HDL System as Knowledge Base Component

The HDL system can be used in several different ways as a KB component in Johnny. In this section some examples on how it is implemented are explained for each of the software modules in Johnny. While there may certainly be other ways of implementing the system, the following sections serve as a proof of concept that the HDL system can be used as a common KB for a robotics system.

4.3.1.1. Navigation and Localisation

The navigation and localisation module shares some knowledge with the planning system. It works in the low level controller where it commands the robot directly and computes trajectories for moving the robot to given poses. At a higher level, the planning level, the topological information of the robot’s environment is needed for planning the robot’s movement in terms of actions. However, the information on both levels is semantically the same but with different representations. Some approaches where this information is stored in some layers, are described in [ZOMMJ$^+$08, OMMJZ$^+$07, GSC$^+$05, BMM$^+$07].

In the HDL system, the topological information is stored in the DL model as this information is needed for extracting the planning actions in both the navigation and pick-and-place domains. Therefore, integration of the information would only make sense if the information is merged with the existing model. Thus, it is still usable for the available model and also improves the model’s usability.

In Figure 3.8, the ontology of the pick-and-place domain is shown. The RoboCup ontology

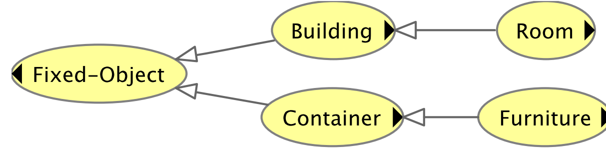


Figure 4.8.: Ontology for a RoboCup@Home environment that is relevant for the navigation and localisation module.

for navigation and localisation is based on this model. Figure 4.8 shows the enhanced ontology of the pick-and-place domain. In this figure, the class *Door* and sub-classes of *Room* are omitted. However, these classes are still in the HDL model.

The properties of the class *Container* are enhanced with three properties. *Container* is described in DL as follows:

$$
\begin{aligned}
Container \sqsubseteq\ & Fixed{-}Object \sqcap \\
& \exists\ at.Room \sqcap \\
& \leqslant 1\ locationPointClouds \sqcap \\
& \leqslant 1\ locationPolygons \sqcap \\
& \leqslant 1\ objectPose
\end{aligned}
$$

The first property, *locationPointClouds*, stores either the points of the container instance or links to this information. Thus, the localisation module can use this information to match the laser scan reading with the corresponding object. The second property, *locationPolygons*, stores the points that build the object. This is useful for drawing the map for the user view, such as the one shown in Figure 4.5. The third property, *objectPose*, stores the pose of the object's instance relative to its parent's instance. This information is important for dynamically changing the map. As mentioned previously, one of the challenges of the RoboCup@Home tests is the dynamic environment. Thus, the furniture might be placed in different positions for each test. Hence, having the property *objectPose* enables the updating of the object's pose dynamically. These three properties are also added to the class *Room*.

The class *Furniture* is added into the model as sub-class of *Container*, which is defined as follows:

$$
\begin{aligned}
Furniture \sqsubseteq\ & Container \sqcap \\
& \exists\ hasProperty.ObjectProperty
\end{aligned}
$$

This class is used for describing the furniture in the RoboCup environment. A piece of furniture can be categorised as a container because some manipulable objects can be placed on a piece furniture. The property *hasProperty* is described in more detail in the Section 4.3.2.

Lets us take an example where the "`dining-table`" is rotated 90° from the position shown in Figure 4.5. The "`dining-table`" is inserted into the *ABox* as an instance of class *Furniture*

as follows:

```
Furniture(dining-table),
at(dining-table, kitchen),
locationPolygons(dining-table, '[0,0],[0,2136.30],[1005.83,
  2136.30],[1005.83,0]'),
locationPointClouds(dining-table, '... [528.05,49.09],[849.26,
  93.53],[254.27,659.03],...'),
objectPose(dining-table,'[0,1,1776.14; -1,0,3428.00;0,0,1]')
```

The assertion above tells us that the "`dining-table`" is located in the "`kitchen`". It also represents the polygons that make up the table in the property *locationPolygons* and the point clouds of the laser scan that are defined in the property *locationPointClouds*. The polygons and point clouds represent the "dining-table" itself, in which (0,0) is the bottom left corner of the table (see Figure 4.5). The property *objectPose* contains the homogeneous transform matrix for representing the "dining-table" relative to the "kitchen". In this case, it uses for 2D transformation, which can be written as:

$$objectPose = \begin{bmatrix} 0 & 1 & 1776.14 \\ -1 & 0 & 3428.00 \\ 0 & 0 & 0 \end{bmatrix}$$

This pose describes the $90°$ rotation and the translation $(1776.14\,mm, 3428.00\,mm)$ from the object's origin. The final polygons are calculated as $objectPose * locationPolygons$ and the final point clouds are calculated as $objectPose * locationPointClouds$. The final map is shown in Figure 4.9.

The map is represented in the HDL system and helps the localisation system to draw its initial map. However, these points might not exactly be the same as the actual sensor readings. The differences are minimised by the navigation and localisation component as the robot moves within the environment. The polygons are helpful for drawing the boundary over the points so that the user can identify what object is being represented by these points. Any change of the environment can be actualised by the user by updating the property *objectPose*.

4.3.1.2. Object Detection and Manipulation

In a mobile manipulator, the manipulation component depends on the navigation and on the object detection components. The navigation component should control the approach to the place where the object is located. Once the robot arrives at that location, it needs to detect the object itself. The HDL system needs this kind of information such as where the object is located. From this information it should be able to derive the container of the object and where the container is placed. Thus, the object is the information that is being shared with the object detection component.

In order to enable the manipulator to grasp the object, it should be able to recognise the ob-

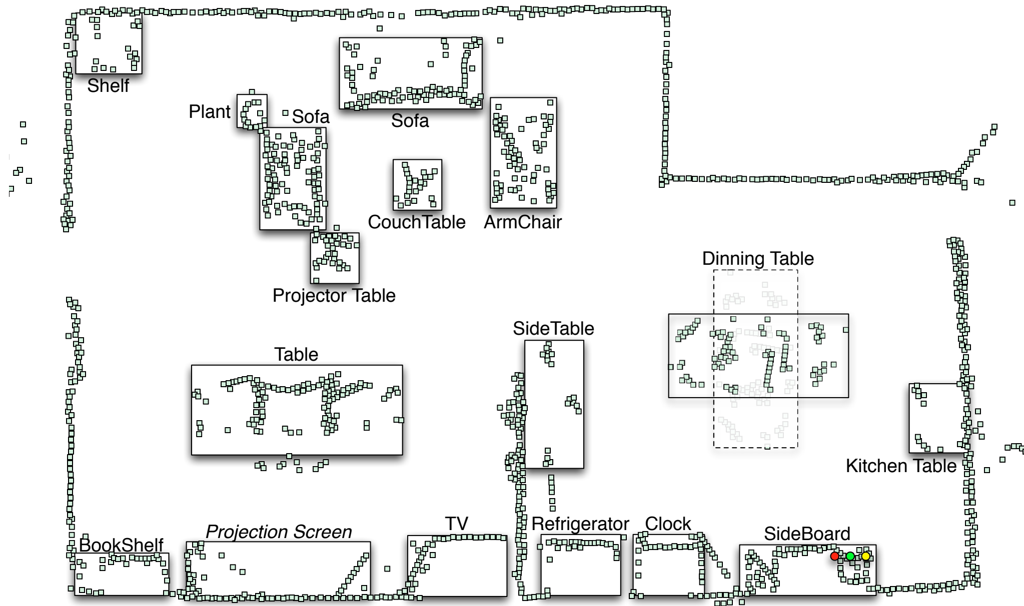


Figure 4.9.: The RoboCup@Home environment with the dining table rotated $90°$.

ject and extract its pose relative to the robot. The object detection component stores the known objects as a collection of features such as SIFT and the object's colour. Therefore, this information needs to be linked with the information stored in the HDL system. A class *RoboCup-Object* is added into the HDL model to define the objects that are used in the competition. This class is defined as follows:

$$\begin{aligned} RoboCup\text{-}Object \sqsubseteq\ & Manipulable\text{-}Object\ \sqcap \\ & \exists\ hasProperty.ObjectProperty\ \sqcap \\ & \leqslant 1\ objectFeature\ \sqcap \\ & \leqslant 1\ objectPose \end{aligned}$$

The class *RoboCup-Object* extends the class *Manipulable-Object* with three additional properties, which are relevant for the RoboCup domain. The first property is *hasProperty* that is described in detail in Section 4.3.2. The second property is *objectFeature* that is used for storing the required features for detecting the object. The value of this property is a link to the directory or file that holds the features. The third property is *objectPose* that stores the pose of the object. This property is not always available, but in some cases the value can be determined by the robot itself. For example, after putting the object in a certain position, the robot will know the pose of the object. It inherits the property of its super-class property *at* that stores the location information of the object.

The HDL system stores the objects in its model which contains the needed information to recognise the object. Thus, an object can be distinguished from the others through its unique features. It also has the symbolic information that enables the planner to extract necessary actions that involve object manipulation.

4.3.1.3. Human Robot Interaction

The HRI in Johnny consists of the speech recognition and the face recognition components. The HDL system can enhance both components in the HRI. It enhances the speech recognition component indirectly by enabling the overall system to communicate with the users more naturally. For example, in response to the "Bring a drink to the guest" command, Johnny asked "What kind of drink should I bring to the guest?". In this situation, a user might answer Johnny with another question such as "What drinks do we have?". Thus, Johnny would be able to query the HDL system for any instances of the class *RoboCup-Object* with property *hasProperty* `drinkable` and answer the user with a list of available drinks. This also works in the other direction. For example, Johnny might ask the user to fill the missing information which he does not have. In the previous dialogue, he asked the user "Where is the guest sitting?", because he needs this information in order to perform the requested task. The answer is then used to complete the missing information in the HDL system.

In the face recognition component, the HDL system stores the biometric information of the people. It also enables Johnny to distinguish the owner from the guest by defining these as DL concepts as shown in Figure 4.10. Thus, a person that belongs to class $Admin$ can give advanced commands to Johnny as an ordinary user. This can also restrict guests from giving commands to Johnny.

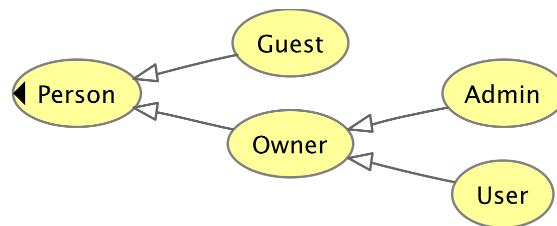


Figure 4.10.: The concept $Person$ and its subclasses.

The class $Person$ is defined as follows:

$$
\begin{aligned}
Person \sqsubseteq\ & Thing\ \sqcap \\
& \exists\, at.Furniture\ \sqcap \\
& \leqslant 1\ locationName\ \sqcap \\
& \leqslant 1\ robotPose\ \sqcap \\
& \leqslant 1\ faceFeature
\end{aligned}
$$

The property at stores the location when the person is sitting on a piece of furniture. In case the person is standing at a location, two additional properties, $locationName$ and $robotPose$, are used to store the location information. The $robotPose$ property is needed by the robot to navigate to this location later. The navigation and localisation component stores pairs of this location information ($locationName$,$robotPose$) internally. Thus, the sequencer can tell the robot to move to a given pose by providing the name of the stored location instead of the desired pose. Besides that, the $robotPose$ helps the robot to navigate to the defined pose if the piece of furniture where the person is sitting is large. For example, if the person is sitting at a dining table with four chairs, the $robotPose$ gives information where the robot should arrive in order to deliver or interact with the person. This information is necessary in some of the tests, such as *partybot*, where the robot needs to recognise a human and remember his/her name and offer him/her a drink. Thus, the robot needs to go back to the position where this person was sitting or standing in order to give the drink to him/her.

The HDL system provides "nice to have" features for the HRI in the RoboCup domain. The planning system is not directly affected by improving the HRI component as long as the information in the HDL system is sufficient for the current problem. However, it helps users to interact with Johnny in a more natural way by improving the way in which they communicate with each other.

4.3.2. Solving RoboCup@Home Tasks with the HDL System

In this section, the *open challenge* scenario described previously, is solved using the HDL system. Two approaches are presented in this section. This scenario has a similar task to the one in the pick-and-place domain which was presented in the previous chapter. Thus, the first approach for solving the task uses the pick-and-place domain. The second approach extends the pick-and-place domain for the RoboCup domain. The modelling of the environment is presented first, as it is required by both approaches.

4.3.2.1. Modelling the *ABox* of RoboCup@Home Environment

The RoboCup@Home environment consists of two rooms, several pieces of furniture and some objects. These have been previously described in Section 4.3.1. However, some of the instances might contain irrelevant facts for the planning system. For example, among the furniture only some with objects in or on them are relevant for the planning problem. Even the objects themselves may only be relevant if they can be manipulated by Johnny. Thus, having all of the objects in the planning problem is not necessary.

Figure 4.11 shows the ontology of the RoboCup@Home environment. The new concepts are depicted with orange-coloured ellipses. Initially the HDL system contains the concepts and instances from the previous experiments. Thus, the$RoboCup$-$Room$ concept is the first one to

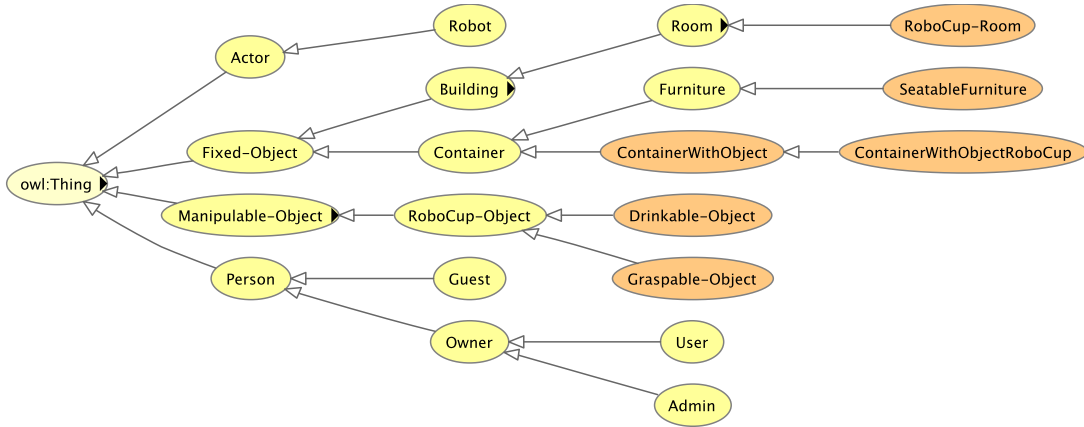


Figure 4.11.: RoboCup@Home ontology.

filter out instances of the non-RoboCup environment. This concept is defined as follows:

$$\begin{aligned} RoboCup\text{-}Room \equiv\ & Room\ \sqcap \\ & \ni\ inBuilding(robocup\text{-}arena) \end{aligned}$$

A property *hasProperty* is used by some of the concepts to add semantic function of the instances. For example, someone can sit on a chair, thus it will contain the property "`hasSeat`". This can be used to define the affordances of some objects. For example, objects that contain liquid for drinking are labelled with "`drinkable`". Let us start with the first concept, namely $SeatableFurniture$, which is defined as follows:

$$\begin{aligned} SeatableFurniture \equiv\ & Furniture\ \sqcap \\ & \ni\ hasProperty(hasSeat) \end{aligned}$$

Any instance of $Furniture$ that has property `hasSeat` is deduced to be a member of this concept. This is quite useful in the RoboCup@Home scenario, in particular in the context where a guest might sit on some pieces of furniture. Thus, instead of having all instances of furniture in the planning problem only those that are members of $SeatableFurniture$ are included.

In order to filter out containers which contain objects from those which do not, a concept $ContainerWithObject$ is defined as follows:

$$\begin{aligned} ContainerWithObject \equiv\ & Container\ \sqcap \\ & \exists\ hasObject.Manipulable\text{-}Object \end{aligned}$$

The property *hasObject* is a symmetrical property to the property at. Thus, the DL reasoner will be able to deduce the container that has some object within it. However, these are not restricted to the RoboCup containers only. In order to limit them more strictly, a sub-concept of

$ContainerWithObject$ is defined as follows:

$$ContainerWithObjectRoboCup \equiv ContainerWithObject \sqcap \\ (\ni\ at(kitchen) \sqcup \ni at(living\text{-}room))$$

This concept checks where the containers are placed. If they are located either in the `kitchen` or the `living-room`, they are part of this concept. The `kitchen` and `living-room` are two rooms in the RoboCup@Home environment.

The previous concepts deal with the location, furniture and containers. However, the object itself is not yet filtered. In the scenario, there are at least 15 objects used for the competition. Some of these objects are too big or too heavy for Johnny's manipulator. Therefore, two additional concepts are shown here as examples of how to filter those objects. The first concept is $Drinkable\text{-}Object$, which is defined as follows:

$$Drinkable\text{-}Object \equiv RoboCup\text{-}Object \sqcap \\ \ni\ hasProperty(drinkable)$$

It uses the property $hasProperty$ with the value `drinkable`. In this scenario, the instances of the $Drinkable\text{-}Object$ class are `yellow-tea`, `green-tea`, and `coke`. Similarly, the concept $Graspable\text{-}\ Object$ is defined as follows:

$$Graspable\text{-}Object \equiv RoboCup\text{-}Object \sqcap \\ \ni\ hasProperty(small) \sqcap \\ \ni\ hasProperty(lessThan500g)$$

It checks whether an object is `small` and `lessThan500g`.

In RoboCup@Home, there are two rooms, an exit room, 17 pieces of furniture, and 15 objects. Table 4.1 shows instances of the concepts defined in Figure 4.11. It shows that the newly defined concepts can reduce the number of instances, thus the planning problem will also be smaller.

4.3.2.2. Solving the Scenario with Pick-and-Place Domain

In the *open challenge* scenario, the main task is to bring a cola to the guest. This task is quite similar to the one in the pick-and-place domain. In the pick-and-place domain, the robot moves the object from one location to another, whereas in the open challenge scenario the robot moves the cola from one location and gives it to the guest. Therefore, it is possible to solve the task using the pick-and-place domain.

The *open challenge* consists of two objectives; the first objective is (move-object johnny coke armchair) and the second objective is (navigate johnny exit). The property $useState$ is adjusted to accommodate the RoboCup@Home environment. The involved concepts for this problem are

Table 4.1.: RoboCup@Home concepts and their instances.

Concept	# of Asserted Instances	# of Inferred Instances
$Fixed\text{-}Object$	0	62
$Building$	3	40
$Room$	16	19
$RoboCup\text{-}Room$	0	3
$Container$	5	22
$ContainerWithObject$	0	6
$ContainerWithObjectRoboCup$	0	4
$Furniture$	17	17
$SeatableFurniture$	0	4
$Manipulable\text{-}Object$	0	22
$RoboCup\text{-}Object$	15	15
$Drinkable\text{-}Object$	0	3
$Graspable\text{-}Object$	0	7

$ContainerWitObjectRoboCup$, $SeatableFurniture$, and $Drinkable\text{-}Object$.

The HDL system generates as planning problem shown in Listing A.13. The planning domain is the same one as for the pick-and-place domain. The complete solution plan is shown in Listing A.14. Below is the solution plan without dummy operators:

```
(!drive-to-dexterous-workspace johnny sideboard kitchen)
(!pickup-object johnny coke sideboard)
(!drive-away-from-dex-workspace johnny sideboard)
(!drive-robot johnny kitchen living-room)
(!drive-to-dexterous-workspace johnny armchair living-room)
(!put-object johnny coke armchair)
(!drive-robot johnny living-room exit)
```

The solution plan generates the necessary states for performing the *open challenge* task. The correctness of this solution can be validated against the state machine depicted in Figure 4.4 (see also Figure 4.5). However, in this solution Johnny puts the cola in the armchair instead of giving it to the guest in the armchair. The re-usability of the stored planning domain is again shown. This is another instance of the pick-and-place domain.

4.3.2.3. Adding New Methods in the *ABox*

In this section, some new methods and operators are defined over the pick-and-place domain in order to solve the open challenge scenario. One of the advantages of the HDL system is that new methods can be modelled in such a way that enables them to use available methods in the model.

In the previous section, the open challenge task was solved using the pick-and-place domain, however, the second objective, where the robot should give the object to the user, is not

really tackled by the pick-and-place domain. It is performed by the `put-object` operator. Therefore, a new operator `give-object` is defined as follows:

`(!give-object ?robot ?object ?to-person)`

preconds: $L_1 =$ `on(to-person, to-container)` $\wedge$
`at-dexterous-workspace(robot, to-container)` $\wedge$
`has-object(robot)` $\wedge$ `at(object, robot)`

delete-list: $D_1 =$ `has-object(robot)`
$D_2 =$ `at(object, robot)`

add-list: $A_1 =$ `has-object(to-person)`
$A_2 =$ `at(object, to-person)`

In addition to this new operator, a method `give-object` with the heuristic on when to execute the operator is defined as follows:

`(give-object ?robot ?object ?to-person)` *;; case 1*
task: `give-object(robot, object, to-person)`
subtasks: $\emptyset$
constr: `not(at(object, robot))`

`(give-object ?robot ?object ?to-person)` *;; case 2*
task: `give-object(robot, object, to-person)`
subtasks: $u_1 =$ `!give-object(robot, object, to-person)`
constr: `at(object, robot)`, `on(to-person,to-container)`,
`at-dexterous-workspace(robot, to-container)`

`(give-object ?robot ?object ?to-person)` *;; case 3*
task: `give-object(robot, object, to-person)`
subtasks: $u_1 =$ `!drive-to-dexterous-workspace(robot, to-container, room)`
$u_2 =$ `give-object(robot, object, to-person)`
constr: $u_1 \prec u_2$, `at(object, robot)`, `on(to-person,to-container)`,
`not(at-dexterous-workspace(robot,to-container))` ,
`at(to-container, room)`, `at(robot, room)`

`(give-object ?robot ?object ?to-person)` *;; case 4*
task: `give-object(robot, object, to-person)`
subtasks: $u_1 =$ `navigate(robot, room)` *;; ** method calls ***
$u_2 =$ `give-object(robot, object, to-person)`
constr: $u_1 \prec u_2$, `at(object, robot)`, `on(to-person,to-container)`,
`not(at-dexterous-workspace(robot,to-container))`,
`at(to-container, room)`, `not(at(robot, room))`

Finally, a method for bringing an object to a person is defined as follows:

(bring-object ?robot ?object ?to-person)
task: bring-object(robot, object, to-person)
subtasks: $u_1 =$ bring-object(robot, object, from-container, to-person)
constr: at(object, from-container)

(bring-object ?robot ?object ?from-container ?to-person) *;; case 1*
task: bring-object(robot, object, from-container, to-container)
subtasks: $u_1 =$ get-object(robot, object, from-container)
$u_2 =$ bring-object(robot, object, from-container, to-person)
constr: $u_1 \prec u_2$, not(at(object, robot)), at(object, from-container)

(bring-object ?robot ?object ?from-container ?to-person) *;; case 2*
task: bring-object(robot, object, from-container, to-person)
subtasks: $u_1 =$ give-object(robot, object, to-person)
constr: at(object, robot)

The method `bring-object` is the main objective in the open challenge task. Besides the newly defined method, it also uses the methods in the pick-and-place and navigation domains, such as `get-object` and `navigate`. The goal of the open challenge task is then defined as (bring-object johnny coke guest) and (navigate johnny exit). The generated planning domain and planning problem is shown in Listings A.15 and A.16. The solution plan without dummy operators is shown below (complete solution plan is shown in Listing A.17):

```
Plan cost: 11.0
(!drive-to-dexterous-workspace johnny sideboard kitchen)
(!pickup-object johnny coke sideboard)
(!drive-away-from-dex-workspace johnny sideboard)
(!drive-robot johnny kitchen living-room)
(!drive-to-dexterous-workspace johnny armchair living-room)
(!give-object johnny coke guest)
(!drive-robot johnny living-room exit)
```

The final sequence is now giving the object, the coke, to the guest. The previous step was to move to the dexterous workspace, i.e. armchair, where the guest is sitting. Thus, the solution plan represents similar sequences as the state machine that was created using fixed programmed. Therefore, the state machine can easily be replaced by the sequence of actions generated by the HDL system.

5. HDL Systems in the AI Domain

The HDL system has been developed for solving planning problems in robotics. Chapter 3 presented its application in solving problems in two robotics domains, namely those of navigation and pick-and-place. As the HDL extends the HTN planner, it can be used for solving any planning problem that can be solved by an HTN planner. In this chapter, HDL is used for solving the well-known *Blocks-world* problem in the AI domain.

5.1. Blocks-World

The blocks world was originally developed by Terry Winograd as a test bed for his program [Win72]. It has become increasingly popular and widely used as a test bed for planning algorithms. It is the well-known planning problem that causes difficulty for the STRIPS planner with the *Sussman* anomaly. The container-stacking problem that is used as an example domain in the *Automated Planning* book is a *Dock Worker Robot* (DWR) adaptation of the blocks world domain [GNT04, Chapter 4]. An example of [GS05] presented a multi-robot planning approach that uses the blocks world domain.

In the planning community, a common domain is required in order to compare performance of one planning algorithm with another. The *International Planning Competition* (IPC) has influenced the way planning is compared and evaluated. The blocks world domain has been included in the evaluation because it is well-known. It can easily be modified into increasingly more complex problems. The blocks world is one of benchmark problems for planning evaluation. Other benchmark problems are simplified versions of realistic planning problems, such as the logistics or refrigerator repair domains [HD02]. While the blocks world problem appears to be a simple "toy problem" at first, finding an optimal plan is NP-Hard. The NP-Hardness is caused by a deadlock situation [GN92].

5.1.1. Problem Statement

The blocks world is basically a stacking problem. In this example, a simplified variation of the blocks world problem is used. Originally, the blocks world problem has blocks of different sizes and included complications such as the use of pyramids [Win72, GN92]. Figure 5.1 depicts a blocks world domain that involves four blocks. The planning objective is to arrange the blocks as in the given goal state.

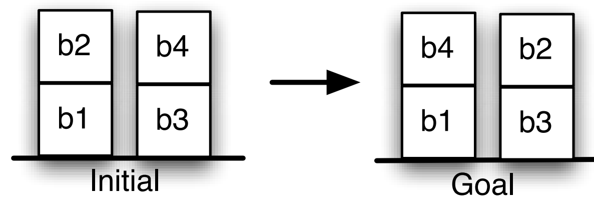


Figure 5.1.: Blocks world domain with four blocks.

The blocks world domain has four operators, namely `pickup`, `putdown`, `stack`, and `unstack`. The `pickup` and `unstack` operators have the same purpose, namely to take a block from the domain. If the block is on the table then the `pickup` operator is used, otherwise `unstack` is used. In a similar manner, returning the object back to the domain is achieved through the use of the two operators `putdown` and `stack`. When the intention is to put the block directly on the table the `putdown` operator is executed, otherwise, the `stack` operator is used.

The information on the blocks in the domain are defined in the initial state I and goal state G. These facts are defined as follows:

$$I = \{(on{-}table\ b1), (on\ b2\ b1), (clear\ b2), (on{-}table\ b3), (on\ b4\ b3), (clear\ b4)\}$$
$$G = \{(on{-}table\ b1), (on\ b4\ b1), (clear\ b2), (on{-}table\ b3), (on\ b2\ b3), (clear\ b2)\}$$

In addition to the facts above, the blocks are defined explicitly in the state I, $(block\ b1)$, $(block\ b2)$, $(block\ b3)$, $(block\ b4)$.

5.1.2. The HTN Planning Domain

One reason for using the blocks world domain as an example here is due to the availability of the domain as an example of the source distribution in JSHOP2. In the example, two problem sets are provided, a small problem with four blocks, as shown in Figure 5.1, and a large problem with 300 blocks. In this case, the small problem is sufficient to show the planning problem in the HDL system.

Though the blocks world domain has four operators as previously mentioned, the HTN needs several additional methods to solve the problem. These methods are the heuristic that the programmers put into the planning domain. They provide the HTN planner with information on how to decompose the planning problem. In the HTN, the blocks world domain has six operators, eleven methods, and two axioms. The two additional operators are `assert` and `remove`. These operators are needed to keep track of what needs to be done. The `assert` operator adds the planning objective to the facts. The `remove` operator deletes the objective from the list of facts. In HTN, the planning domain may have axioms. The axioms are used as theorem provers, as subroutines of the planning procedure [GNT04, Chapter 11]. In the HDL, the axioms are not explicitly defined in the *TBox*. Rather, they are modelled as methods without

immediate successors. The blocks world planning domain is presented in A.18.

In SHOP2, the axioms are defined syntactically as follows:

```
(:- a [name1] L1 [name2] L2 .  .  .  [namen] Ln)
```

where `a` is the axiom's head and can be treated as a logical atom and its tail is the list of logical preconditions `L`. Each logical precondition may have a name, but this is not mandatory. A logical precondition is either a logical expression or one of two special precondition forms in SHOP2, namely *first satisfier precondition* or *sorted precondition*. Details about these preconditions are given in [Ilg06]. The value of `a` is expressed in the following definition.

Definition 5.1. Let a be an axiom in HTN planning and L_i be a logical precondition that is defined through HORN clauses, such that $a = \{L_1, L_2, ..., L_n\}$. a will return true iff any logical expression in a holds true: $a \equiv L_1 \vee L_2 \vee ... \vee L_n$

The main method in the blocks world domain is `achieve-goals`. It receives the final state from the user and decomposes it into subtasks. It is defined as follows:

`(achieve-goals ?goals)` *;; method to set the planning goals*
task: `achieve-goals(goals)`
subtasks: $u_1 =$ `assert-goals(goals)`
$u_2 =$ `find-nomove()`
$u_3 =$ `add-new-goals()`
$u_4 =$ `find-moveable()`
$u_5 =$ `move-block()`
constr.: $u_1 \prec u_2, u_2 \prec u_3, u_3 \prec u_4, u_4 \prec u_5$

The task `assert-goals` extracts the facts in the `goals` and inserts them into the HTN to track the facts that need to be achieved. It is implemented in two methods. The first one is a recursive method and the second one is the method for terminating the recursion if the given parameter is `nil`. The first method receives `goals` as its parameter. The `goals` are separated into two parameters, namely `goal` and `goals`. The second parameter, `goals`, is different than the first one. `goals` (g) contains a set of facts or terms t_i, such that $g = \{t_1, t_2, ..., t_i\}$. The SHOP2 syntax `(?goal . ?goals)` extracts g and decomposes it into g_1 and g' where $g_1 = \{t_1\}$ and $g' = \{g - \{g_1\}\}$. If g contains only one term ($g = \{t_1\}$) then g' will contain `nil`. The `assert-goals` methods are defined as follows:

`(assert-goals (?goal . ?goals))` *;; method to insert goal facts into the system*
task: `assert-goals(goals)`
subtasks: $u_1 =$ `!assert((goal ?goal))`
$u_2 =$ `assert-goals(goals)`
constr.: $u_1 \prec u_2$

`(assert-goals nil)` *;; method to terminate the recursion*

task: `assert-goals('nil')`
subtasks: $\emptyset$
constr.: $\emptyset$

The second subtask of `achieve-goals` is `find-nomove`. Its task is to mark all the blocks which do not need to be moved. It uses the axiom `need-to-move` to check whether a block is involved in the goal or in the decomposition process, e.g. it is on top of another block that needs to be moved. It is a recursive method that terminates when there are no blocks left to be marked. The method is defined as follows:

`(find-nomove)` *;; method to mark blocks that do not move*
task: `find-nomove()`
subtasks: $u_1 =$ `!assert((dont-move ?x))`
$u_2 =$ `find-nomove()`
constr.: $u_1 \prec u_2$, `block(x)`, `not(dont-move(x))`, `not(need-to-move(x))`

The axiom `need-to-move` is defined as follows:

`(need-to-move ?x)` *;; axiom to test block that need to be moved*
preconds: $L_1 =$ `on(x,y)` $\wedge$ `goal(on(x,z))` $\wedge$ `not(same(x,z))`
$L_2 =$ `on-table(x)` $\wedge$ `goal(on(y,z))`
$L_3 =$ `on(x,y)` $\wedge$ `goal(on-table(x))`
$L_4 =$ `on(x,y)` $\wedge$ `goal(clear(y))`
$L_5 =$ `on(x,z)` $\wedge$ `goal(on(y,z))` $\wedge$ `not(same(x,y))`
$L_6 =$ `on(x,w)` $\wedge$ `need-to-move(w)`

This axiom calls axiom `same` in L_1, where `same` returns true iff the given parameters are identical. In L_6, it calls itself to test whether another block w also needs to be moved.

The `add-new-goals` method has the function of asserting new goals to the facts in HTN. The assertion is done if the blocks have to be moved and are not associated with goals already. This method is defined as follows:

`(add-new-goals)` *;; method to implicit goals*
task: `add-new-goals()`
subtasks: $u_1 =$ `!assert((goal (on ?x ?y)))`
$u_2 =$ `add-new-goals()`
constr.: $u_1 \prec u_2$, `block(x)`, `not(dont-move(x))`,
`not(goal(on-table(x))`, `not(goal(on(x,y)))`

The `find-movable` method asserts new facts in the system when a block can be moved immediately to its final position under the current state of the world. These blocks are marked with either a `put-on-table` fact or `stack-on-block` fact, depending on their associated

goal. This method has two cases such that it splits into two possible branches depending on its constraints. This method is defined as follows:

`(find-moveable)` *;; method to find blocks to put on table*
task: `find-moveable()`
subtasks: $u_1 =$ `!assert((put-on-table ?x))`
$u_2 =$ `find-moveable()`
constr.: $u_1 \prec u_2$, `clear(x),not(dont-move(x)),goal(on-table(x), not(put-on-table(x))`

`(find-moveable)` *;; method to find blocks to stack on another block*
task: `find-moveable()`
subtasks: $u_1 =$ `!assert((stack-on-block ?x ?y))`
$u_2 =$ `find-moveable()`
constr.: $u_1 \prec u_2$, `clear(x),not(dont-move(x)),goal(on(x,y), not(stack-on-block(x,y))),dont-move(y),clear(y)`

Although the `find-moveable` methods are described above as two different methods, they are implemented as a single method in HTN. In addition, this method has a third case, in which its constraints are `nil` and the subtasks are `nil`. The third case is for terminating the recursion.

The last subtask of method `achieve-goals` is `move-block`. It is the main method in this domain. It moves the blocks that were previously marked by `find-moveable`. It then moves the rest of the blocks. It has four cases of which the fourth is the terminator for the recursion. This method is defined as follows:

`(move-block)` *;; method to stack block x on y*
task: `move-block()`
subtasks: $u_1 =$ `move-block1(x,y)`
$u_2 =$ `move-block()`
constr.: $u_1 \prec u_2$, `stack-on-block(x,y)`

`(move-block)` *;; method for moving x from on-top of y to table*
task: `move-block()`
subtasks: $u_1 =$ `!unstack(x,y)`
$u_2 =$ `!putdown(x)`
$u_3 =$ `!assert((dont-move ?x))`
$u_4 =$ `!remove((put-on-table ?x))`
$u_5 =$ `check(x)`
$u_6 =$ `check2(y)`
$u_7 =$ `check3(y)`
$u_8 =$ `move-block()`

constr.: $u_1 \prec u_2$, $u_2 \prec u_3$,$u_3 \prec u_4$, $u_4 \prec u_5$, $u_5 \prec u_6$, $u_6 \prec u_7$, $u_7 \prec u_8$,
`put-on-table(x)`, `on(x,y)`

`(move-block)` *;; method for moving x out of the way*
task: `move-block()`
subtasks: $u_1 =$ `!unstack(x,y)`
$u_2 =$ `!putdown(x)`
$u_3 =$ `check2(y)`
$u_4 =$ `check3(y)`
$u_5 =$ `move-block()`
constr.: $u_1 \prec u_2$, $u_2 \prec u_3$,$u_3 \prec u_4$, $u_4 \prec u_5$, `clear(x)`, `not(dont-move(x))`,
`on(x,y)`

The `move-block` method uses four helper methods, namely `move-block1`, `check`, `check2`, and `check3`. These helper methods are described in detail in Appendix B.3.

The blocks world domain is a simple planning problem with four operators. The HTN implementation involves several methods to solve this domain. It uses heuristics to avoid known problems and to optimise some of the internal methods. Nevertheless, these methods cannot guarantee that the domain is fit for any problem in the blocks world. We will see in section 6.3.2 what the effect is of adding irrelevant blocks to the HTN planner.

To summarise, the blocks world domain is defined as $\mathcal{D}_{bw} = (O_{bw}, M_{bw})$, where $O_{bw} = \{pickup, unstack, putdown, stack, assert, remove\}$ and $M_{bw} = \{$ $achieve\text{-}goals$, $assert\text{-}goals1$, $assert\text{-}goals2$, $find\text{-}nomove$, $add\text{-}new\text{-}goals$, $check$, $check2$, $check3$, $find\text{-}moveable$, $move\text{-}block1$, $move\text{-}block$ $\}$. In addition, two axioms are also defined in the $\mathcal{D}_{bw}$, namely $same$ and $need\text{-}to\text{-}move$.

The blocks world problem for the problem shown in Figure 5.1 is defined as $\mathcal{P}_{bw} = (s_{0bw}, w_{bw}, O_{bw}, M_{bw}\}$, where O_{bw} and M_{bw} are the same as the ones defined for D_{bw}. The initial state s_{0bw} contains the facts as defined in I in Section 5.1.1 (page 97). The initial task network is defined as $w_{bw} = (\{u\}, \emptyset)$, where u is a node such that $t_u = achieve{-}goals(G)$ and G is the goal state as defined in Section 5.1.1.

The HTN planner decomposes the problem by expanding the initial task network into its subtasks. Initially, $w_1 = w_{bw} = (\{u\}, \emptyset)$. Applying method `achieve-goals` to u produces a task network $w_2 = (\{u_1, u_2, u_3, u_4, u_5\}, C_2)$, where the $C_2 = u_1 \prec u_2, u_2 \prec u_3, u_3 \prec u_4, u_4 \prec u_5$. u_1 to u_5 are the subtasks of the previously described `achieve-goals` method, such that $t_{u1} = assert{-}goals(G)$, $t_{u2} = find{-}nomove()$, $t_{u3} = add{-}new{-}goals()$, $t_{u4} = find{-}moveable()$, and $t_{u5} = move{-}block()$. The third task network is built by applying `assert-goal` method to u_1. Hence, $w_3 = (\{u_6, u_7, u_2, u_3, u_4, u_5\}, C_3)$, where $C_3 = u_6 \prec u_7, u_7 \prec u_2, u_2 \prec u_3, u_3 \prec u_4, u_4 \prec u_5$, $t_{u6} = assert((goal\ (on{-}table\ b1)))$ and $t_{u7} = assert - goals(G')$, $G' = \{G - \{(on - table\ b1)\}\}$. The rest of the decomposition task network can be done in a similar manner. Figure 5.2 illustrates the decomposition tree for the

blocks world domain.

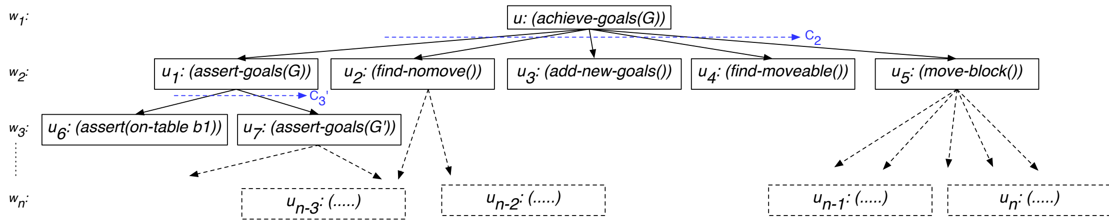


Figure 5.2.: Blocks world task network decomposition tree.

5.2. HDL Implementation of Blocks World

In the HDL implementation, the blocks world domain is implemented as it is described in the previous section. Hence, it shows how to transfer an existing HTN planning domain into an HDL domain. The transformation is done in two steps. First the domain transformation is performed and then the modelling of the preconditions or facts in the HDL system is carried out.

5.2.1. HDL's Blocks World Domain

As the blocks world domain already exists in the HTN, one can apply Definitions 2.5 to 2.11 to instantiate the *ABox* of the $Planning\text{-}Domain$, $Method$, and $Operator$ in the HDL system. The blocks world domain is defined in the $Planning\text{-}Domain$ as `blocks_domain` and its assertion is shown in Figure 5.3 (see Listing B.11 for detail of this assertion).

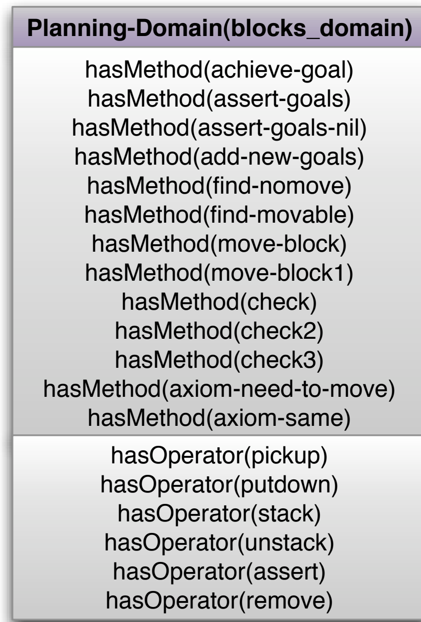


Figure 5.3.: An instance of $Planning\text{-}Domain$ represents the blocks world domain.

The `blocks_domain` methods have two additional methods to represent the axioms that are used in the blocks world domain. The method `achieve-goals` with its subgoals is asserted as follows:

```
Method(achieve-goal),
  hasMethod(achieve-goal, assert-goals),
  hasMethod(achieve-goal, assert-goals-nil),
  hasMethod(achieve-goal, add-new-goals),
  hasMethod(achieve-goal, find-nomove),
  hasMethod(achieve-goal, find-moveable),
  hasMethod(achieve-goal, move-block)
```

These assertions follow the Definition 2.7 and map the subtasks of `achieve-goals` into *hasMethod* or *hasOperator*. The method `assert-goals` gives an example of how both properties are assigned some values, as shown below:

```
Method(assert-goals),
  hasMethod(assert-goals, assert-goals-nil),
  hasOperator(assert-goals, assert)
```

The rest of the methods and operators are asserted analogously.

The blocks world domain is now modelled in the HDL system. This model is enough for generating $\mathcal{D}_{bw}$ by the system. However, these assertions are not sufficient for generating the planning problem $\mathcal{P}_{bw}$. The *useState* has not yet been defined for the methods and operators above. This property is discussed in detail in the following section.

5.2.2. Modelling Blocks World in HDL

The initial state s_0 is generated automatically by HDL. However, one needs to model this state in HDL as a concept in the *TBox* and instantiate the *ABox* afterwards. Definition 2.8 and 2.11 define some of the constraints on the methods and operators that might be defined as *useState* in HDL. This is exactly the case in the blocks world domain where some facts, required by some methods or operators, are generated during the decomposition process. Hence, those facts do not need any model or representation in the HDL system.

In the blocks world domain, the initial state represents the blocks and their arrangement on the table. For example, the state shown in Figure 5.1 is defined as I in Section 5.1.1. The facts can be categorised into four patterns, namely: `(block bn)`, `(on-table bn)`, `(on bn bm)`, and `(clear bn)`. Figure 5.6 shows the ontology of the blocks world domain in HDL.

The blocks world is grouped as the $BlocksWorld$ concept. Hence, it can be easily distinguished from concepts of other domains. However, the $ValuePartition$ concept is used to represent the status of some blocks in $BlockStatus$. This status is discussed later with other concepts depending on it. The main concept for representing the blocks world domain is the

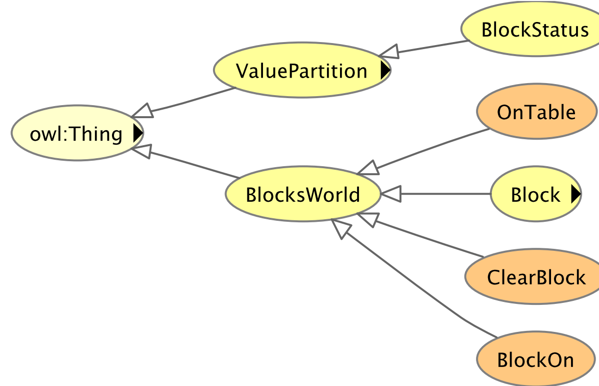


Figure 5.4.: Blocks world domain ontology.

$Block$ concept. It is defined as follows:

$$\begin{aligned} Block \sqsubseteq\ & BlocksWorld \sqcap \\ & \leqslant 1\ blockOn \sqcap \\ & \exists\ hasState.BlockStatus \sqcap \\ & \leqslant 1\ hasState \end{aligned}$$

This concept conveys the information that a block can have at most one $blockOn$ property and at most one $hasState$ property that is instance of $BlockStatus$. These properties are sufficient to represent four kinds of facts in the blocks world domain. The `(block bn)` is represented by an instance of $Block$. The property $blockOn$ represents either `(on-table bn)` or `(on bn bm)`. The last property `(clear bn)` is represented by the $hasState$ property.

$Block$ represents all four possible facts for the blocks world domain. However, the *useState* property is defined as a triple (see Section 2.5.2). As a triple, it needs one concept and optionally one property. For this purpose, $Block$ is too general. An instance of $Block$ can have only one property $blockOn$, denoting that that it is on top of another block or on top of the table. Hence, additional concepts are introduced. These new concepts are $OnTable$, $BlockOn$, and $ClearBlock$. The instances of these concepts are deduced automatically by the DL reasoner.

The concept $OnTable$ infers blocks that are put directly on the table. For this concept, a `Table` is defined as an instance of the $BlocksWorld$ concept. Thus, the table is part of the blocks world domain but it is not an instance of $Block$. The $OnTable$ concept is defined as follows:

$$\begin{aligned} OnTable \equiv\ & BlocksWorld \sqcap \\ & \ni\ blockOn(Table) \end{aligned}$$

The concept $BlockOn$ is used to deduce all blocks that are placed on top of other blocks. It is

defined as follows:

$$BlockOn \equiv BlocksWorld \sqcap \\ \exists\, blockOn.Block$$

The last concept, $ClearBlock$, is used for inferring all blocks which are located at the top. A helper instance, namely `isClear`, is defined as a member of $BlockStatus$. All top-most blocks are asserted with the property $hasState$ set with `isClear`. Hence, the $ClearBlock$ concept is defined as follows:

$$ClearBlock \equiv BlocksWorld \sqcap \\ \ni\ hasState(isClear)$$

Each of the concepts defined above provides mappings between HDL instances and HTN facts. These mappings are defined below:

$$\begin{aligned} &T_1 : (block\ ?val1); ?val1 = I{:}Block \\ &T_2 : (on{-}table\ ?val1); ?val1 = I{:}OnTable \\ &T_3 : (on\ ?val1\ ?val2); ?val1 = I{:}BlockOn, ?val2 = P{:}blockOn \\ &T_4 : (clear\ ?val1); ?val1 = I{:}ClearBlock \end{aligned} \tag{5.1}$$

The template T_1 maps each instance of a class $Block$, b_n, to the HTN representation in SHOP2 syntax, `(block bn)`. Template T_2 maps a block b_n that is located on the table to `(on-table bn)`. Template T_3 maps a block b_n that is placed on top of block b_m to `(on bn bm)`. Finally, template T_4 maps the topmost block b_n to `(clear bn)`. Details of this procedure are found in Section 2.5.2.

To complete the blocks world domain assertion in HDL, the following assertions need to be added:

```
useState(find-nomove, T1),
useState(add-new-goals, T1),
useState(find-moveable, T4),
useState(check, T4),
useState(check2, T4),
useState(check3, T4),
useState(move-block1, T3),
useState(move-block, T3),
useState(axiom-need-to-move, T2),
useState(axiom-need-to-move, T1)
```

In the assertion above, $T_1 ... T_4$ are used to minimise the text. However, in the HDL implementation the full representation as shown above (5.1) is asserted instead. The task network for "achieving goals" in the blocks world domain is composed of all the methods and operators defined above. Hence, the use state will be automatically deduced from its $useState$ node prop-

erty. This deduction is a complete one which includes T_1 to T_4. Thus, the HDL system can generate a valid planning problem for the blocks world domain. The result of the HDL system on the simple blocks world domain is presented in Section 5.3.1.

5.2.3. Enhanced Model of Blocks World Domain in HDL

One of the main idea of modelling planning domain in HDL is to enable the co-existence of domains and even several problems for those domains simultaneously within the system. The blocks world concept that is depicted in Figure 5.4 lacks this capability. A problem might arise if more than one blocks world domain is stored in the system. For example, in addition to the previously defined blocks world problem, shown in Figure 5.1, another domain is also stored in the system. Figure 5.5 shows eight blocks stored in the HDL system and two blocks world domain each with four blocks in HTN problem.

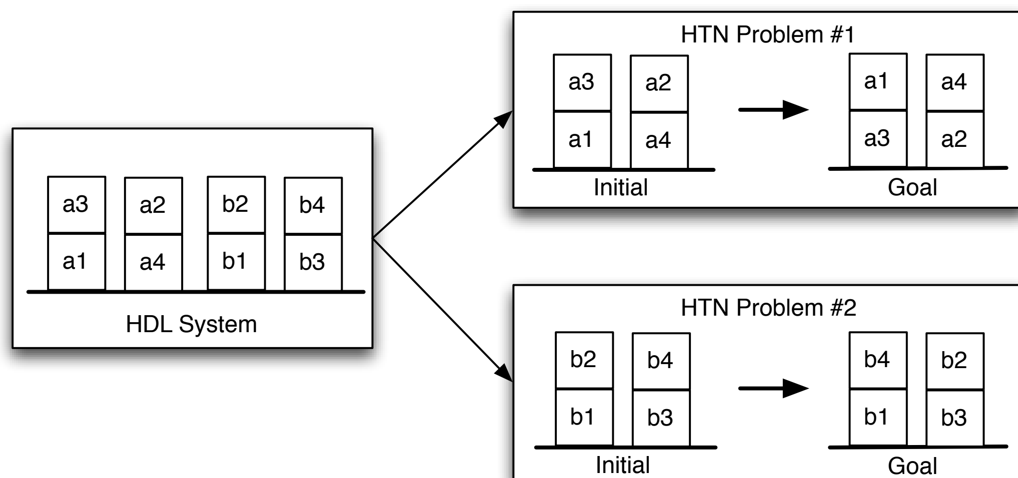


Figure 5.5.: Two blocks world domains with four blocks each.

The simple blocks world domain concept as shown in Figure 5.4 cannot distinguish which blocks are relevant for a particular problem. The result of the current problem by applying this concept is presented in Section 5.3.2. The HDL system generates all stored blocks in the system as part of the planning problem. This might overwhelm the planner.

Additional concepts are required in order to distinguish blocks that are involved in one particular problem from the others which are not. These concepts provide the system with filter parameters that refine the specification of the blocks. An enhanced blocks world ontology is shown in Figure 5.6.

Once again, this feature shows that the HDL system is flexible and reusable. The concepts that are modelled in the simple blocks world domain are supplemented with finer concepts. These are $UsedBlock$, $UsedBlockOnTable$, $UsedBlockClear$, $InvolvedBlock$, and $UsedBlockOn$. Among these concepts, only one is directly influenced by the user, namely

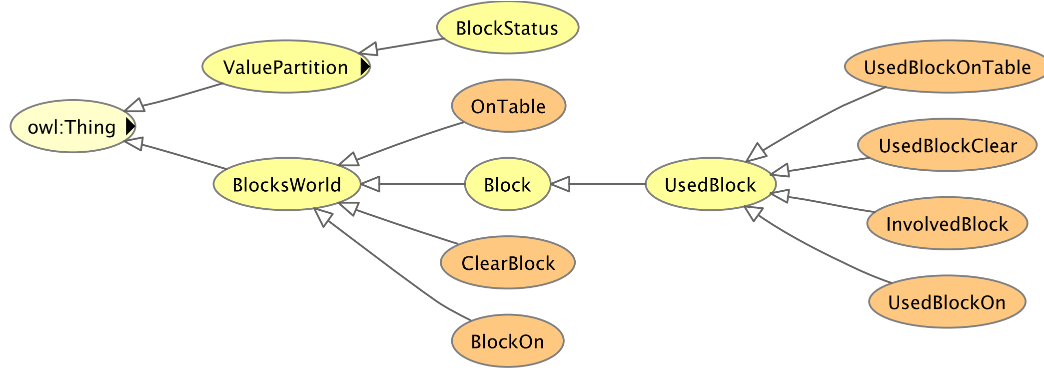


Figure 5.6.: Enhanced blocks world domain ontology.

$UsedBlock$. The others are automatically deduced by the DL reasoner.

The blocks that appear in the goal state G are asserted as instances of $UsedBlock$ instead of $Block$. The $UsedBlock$ concept is defined as follows:

$$UsedBlock \sqsubseteq Block$$

Hence, it inherits all the properties that are defined for $Block$. The blocks world domain needs four mappings as explained in the previous section. These mappings are defined as templates, T_1 to T_4. $UsedBlock$ is the replacement concept for $Block$. Hence, three more concepts are needed to represent used blocks with their specific properties as needed by each template. For representing used blocks on the table, the concept $UsedBlockOnTable$ is used. It is defined as follows:

$$UsedBlockOnTable \equiv UsedBlock \sqcap \ni blockOn(Table)$$

Thus, only blocks, that are instances of $UsedBlock$, are inferred as instances of this concept. In a similar manner, the concepts $UsedBlockClear$ and $UsedBlockOn$ are defined as follows:

$$UsedBlockClear \equiv UsedBlock \sqcap \ni hasState(isClear)$$

$$UsedBlockOn \equiv UsedBlock \sqcap \exists blockOn.Block$$

The templates T_1 to T_4 need to be updated with these new concepts. These templates are

redefined as follows:

$$T_1 : (block\ ?val1); ?val1 = I{:}UsedBlock$$
$$T_2 : (on{-}table\ ?val1); ?val1 = I{:}UsedBlockOnTable$$
$$T_3 : (on\ ?val1\ ?val2); ?val1 = I{:}UsedBlockOn, ?val2 = P{:}blockOn$$
$$T_4 : (clear\ ?val1); ?val1 = I{:}UsedBlockClear$$

The assertion for the blocks world in the HDL system uses these new templates instead. The resulting planning problem, shown in Figure 5.5, which has been defined using the new concepts is shown in Section 5.3.2.

One might notice that there is still one new concept in the enhanced blocks world ontology as shown in Figure 5.6, namely $InvolvedBlock$. What is the purpose of this concept when others defined before are sufficient to model the domain? In both of the previous examples, those four new concepts have been sufficient to model them. However, the HDL system can do more than just filter the blocks that appear in the goal state G. It can also reason about its knowledge base in order to infer blocks that are implicitly relevant for the particular problem although they do not appear explicitly in the goal state.

Figure 5.7 shows the combined blocks world problem in which three blocks are added to the HDL system and the goal state is a combination of these blocks. The blocks in the goal state are defined as instances of $UsedBlock$. These blocks are `a1`, `b2`, and `c3`. However, in order to pickup block `a1`, it is necessary to first unstack block `a3`. The same holds for block `c3`. By only applying the four new concepts in the enhanced blocks world ontology, these implicit blocks cannot be detected. Thus, the planning problem is not a valid one because it does not model the world properly.

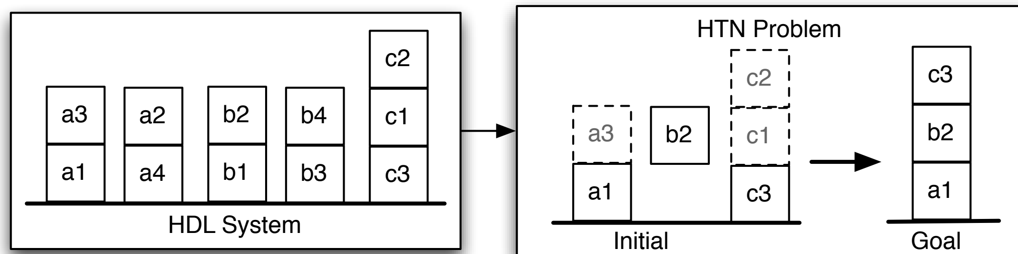


Figure 5.7.: Complex blocks world domains.

In this particular problem, the $InvolvedBlock$ concept plays its role. It has the purpose of deducing implicit blocks that are relevant for the current problem although they are not mentioned explicitly in the goal state. It is defined as follows:

$$InvolvedBlock \equiv \exists\ blockOn.UsedBlock$$

Hence, a block that is located on any block in $UsedBlock$, will be deduced as a member of $InvolvedBlock$. As a member of this concept, it is also a member of $UsedBlock$. Thus, any block on top of it will be inferred as a member of $InvolvedBlock$ too. Section 5.3.3 presents the resulting blocks world problem shown in Figure 5.7.

To summarise, the simple blocks world domain ontology (see Figure 5.4) is sufficient to model the blocks world model that is given as an example (see Figure 5.1). However, this concept cannot model the complicated blocks world domains, such as the ones presented in this section. Two simple blocks worlds and one complex blocks world has been introduced. Thus, an enhanced blocks world domain ontology (see Figure 5.6) is needed for solving these problems.

5.3. Experimental Results

In this section, the results of the examples presented in the previous sections are detailed. These examples are the simple blocks world (Figure 5.1 on page 98), two simple blocks worlds (Figure 5.5 on page 107), and complex blocks world (Figure 5.7 on page 109). The first example is exactly the same problem as in the distribution of the JSHOP HTN planner. However, all the results shown below are solved using the HDL system.

5.3.1. Simple Blocks World

The results presented in this section are based on the blocks world domain shown in Figure 5.1 using the concept definition shown in Figure 5.4. Although the blocks world domain is already modelled in the HDL system, the initial state itself is not yet. Hence, the first step is to model the initial state as shown in Figure 5.1 in the HDL system by asserting the blocks as follows:

```
Block(b1), blockOn(b1, Table),
Block(b2), blockOn(b2, b1), hasState(b2, isClear),
Block(b3), blockOn(b3, Table),
Block(b4), blockOn(b4, b3), hasState(b2, isClear)
```

The output of the DL reasoner over this model is shown in Table 5.2.

Table 5.2.: Results generated by the DL reasoner for the simple blocks world domain.

Concept	Asserted Instances	Inferred Instances
$Block$	b1,b2,b3,b4	b1,b2,b3,b4
$OnTable$	-	b1,b3
$BlockOn$	-	b2,b4
$ClearBlock$	-	b2,b4

The HDL system generates the blocks world planning domain that is shown in Appendix A.4. The planning problem with w = `(achieve-goals ((on-table b1) (on b4 b1) (clear b4) (on-table b3) (on b2 b3) (clear b2)))` is shown in Appendix A.4.1. The result of the HTN planner for the blocks world domain with this problem consists of four plans. Figure 5.8 shows the solution plans without dummy operators, namely `assert` and `remove`; a complete solution plans are shown in Appendix A.4.1:

Plan #1	Plan #2	Plan #3	Plan #4
(!unstack b4 b3)	(!unstack b2 b1)	(!unstack b4 b3)	(!unstack b2 b1)
(!putdown b4)	(!putdown b2)	(!putdown b4)	(!putdown b2)
(!unstack b2 b1)	(!unstack b4 b3)	(!unstack b2 b1)	(!unstack b4 b3)
(!stack b2 b3)	(!stack b4 b1)	(!stack b2 b3)	(!stack b4 b1)
(!pickup b4)	(!pickup b2)	(!pickup b4)	(!pickup b2)
(!stack b4 b1)	(!stack b2 b3)	(!stack b4 b1)	(!stack b2 b3)

Figure 5.8.: Solution plans for the simple blocks world problem (without dummy operators).

The simple blocks world problem, shown in Figure 5.1, can be solved in two ways. The first approach unstacks block `b4`, puts it on the table and then continues with block `b2`. This is shown in solution plans 1 and 3. The second approach is to unstack the block `b2` and then continue with block `b4` as given in solution plans 2 and 4. So why does the HTN planner generate four solution plans instead of two? The answer is due to the `assert` operator that adds the fact `dont-move` to the blocks which are already in the right place. Please refer to Listing A.20 in Appendix A.4.1 to see these differences. A detailed analysis of this phenomenon is referred to Section 6.3.2.

5.3.2. Two Simple Blocks World Problems

In this section, a similar blocks world problem with four blocks, as in the previous example, is added to the HDL system. Hence, two possible problems are stored in the system as shown in Figure 5.5. Two approaches are discussed; the first approach uses the simple blocks world concept shown in Figure 5.4 and the second one uses the enhanced blocks world concept depicted in Figure 5.6. The new blocks are stored in the system with the following assertions:

```
Block(a1), blockOn(a1, Table),
Block(a2), blockOn(a2, a4), hasState(a2, isClear),
Block(a3), blockOn(a3, a1), hasState(a3, isClear),
Block(a4), blockOn(a4, Table)
```

As mentioned in Section 5.2.3, the simple blocks world model cannot distinguish which blocks are involved in which problem. Hence, all stored blocks appear in the HTN planning problem though they are not part of the current one. The output of the DL reasoner for the first approach is shown in Table 5.3.

Table 5.3.: Result of DL reasoner over two blocks world domains.

Concept	Asserted Instances	Inferred Instances
$Block$	a1,a2,a3,a4,b1,b2,b3,b4	a1,a2,a3,a4,b1,b2,b3,b4
$OnTable$	-	a1,a4,b1,b3
$BlockOn$	-	a2,a3,b2,b4
$ClearBlock$	-	a2,a3,b2,b4

Problem number 2 is in fact trying to achieve the same goal as in the previous section. Hence, the initial task network w is `(achieve- goals ((on-table b1) (on b4 b1) (clear b4) (on-table b3) (clear b2) (on b2 b3)))`. The HDL system generates the same planning domain as shown in appendix A.4. However, the planning problem consists of eight blocks instead of four. This problem description is shown in Listing A.21. Thus, the planner returns more than the four plans. In fact, it returns exactly 1440 plans. Section 6.3.2 presents a detailed comparison and explanation of why additional blocks increase the number of plans in the blocks world domain.

Problem number 1 generates the same amount of facts in the problem description. Thus, it faces the same problem as solving problem number 2. Hence, the simple blocks world concepts could not solve this problem elegantly.

Let us now use the enhanced blocks world ontology to solve the problems. The assertions shown in the previous sections are still valid for use with this ontology. For problem number 1, some blocks need to be instances of $UsedBlock$ instead of $Block$. These blocks are a1, a2, a3, and a4.

Table 5.4.: Results from the DL reasoner for the two blocks world domains for first problem.

Concept	Asserted Instances	Inferred Instances
$Block$	b1,b2,b3,b4	a1,a2,a3,a4,b1,b2,b3,b4
$UsedBlock$	a1,a2,a3,a4	a1,a2,a3,a4
$OnTable$	-	a1,a4,b1,b3
$UsedBlockOnTable$	-	a1,a4
$BlockOn$	-	a2,a3,b2,b4
$UsedBlockOn$	-	a2,a3
$ClearBlock$	-	a2,a3,b2,b4
$UsedBlockClear$	-	a2,a3
$InvolvedBlock$	-	a2,a3

Table 5.4 shows the output of the DL reasoner for the asserted blocks. It proves that through the use of the enhanced model, the relevant blocks can be distinguished from the rest. These relevant blocks are instances of the concept $UsedBlock$. The initial network for problem num-

ber 1 is defined as $w =$ (achieve-goals ((on-table a3) (on a1 a3) (clear a1) (on-table a2) (on a4 a2) (clear a4))). The HDL system produces a smaller subset of planning problems as listed in A.22. The HTN planner returns four plans as shown in Figure 5.9. The complete planning result is shown in A.23.

Plan #1	Plan #2	Plan #3	Plan #4
(!unstack a2 a4)	(!unstack a3 a1)	(!unstack a3 a1)	(!unstack a2 a4)
(!putdown a2)	(!putdown a3)	(!putdown a3)	(!putdown a2)
(!pickup a4)	(!pickup a1)	(!pickup a1)	(!pickup a4)
(!stack a4 a2)	(!stack a1 a3)	(!stack a1 a3)	(!stack a4 a2)
(!unstack a3 a1)	(!unstack a2 a4)	(!unstack a2 a4)	(!unstack a3 a1)
(!putdown a3)	(!putdown a2)	(!putdown a2)	(!putdown a3)
(!pickup a1)	(!pickup a4)	(!pickup a4)	(!pickup a1)
(!stack a1 a3)	(!stack a4 a2)	(!stack a4 a2)	(!stack a1 a3)

Figure 5.9.: Solution plans for the two blocks world problem (without dummy operators).

With a similar approach for problem number 2, the blocks are adjusted to be either a member of the $Block$ concept or the $UsedBlock$ concept. Table 5.5 shows the asserted and inferred blocks for problem number 2. The number of facts in the problem descriptions is the same as the one produced in Section 5.3.1. Hence, requesting the same objective of the HTN planner produces similar (see Section 5.3.1).

Table 5.5.: Result of DL reasoner over two blocks world domains for problem number 2.

Concept	Asserted Instances	Inferred Instances
$Block$	a1,a2,a3,a4	a1,a2,a3,a4,b1,b2,b3,b4
$UsedBlock$	b1,b2,b3,b4	b1,b2,b3,b4
$OnTable$	-	a1,a4,b1,b3
$UsedBlockOnTable$	-	b1,b3
$BlockOn$	-	a2,a3,b2,b4
$UsedBlockOn$	-	b2,b4
$ClearBlock$	-	a2,a3,b2,b4
$UsedBlockClear$	-	b2,b4
$InvolvedBlock$	-	b2,b4

5.3.3. Complex Blocks World Problems

The complex blocks world problem is shown in Figure 5.7 (page 109). It adds three more blocks to the system as it is described in the previous example, namely c1, c2, and c3. They are asserted to the system as follows:

```
Block(c1), blockOn(c1, c3),
Block(c2), blockOn(c2, c1), hasState(c2, isClear),
Block(c3), blockOn(c3, Table)
```

The goal state is defined as $w =$ `(achieve-goals ((on-table a1) (on b2 a1) (on c3 b2) (clear c3)))`. Thus, only three blocks (a1, b2, and c3) are explicitly defined in the goal state. These blocks need to be instances of $UsedBlock$. In the previous examples, the number of blocks defined in the goal state is equal to the number of blocks described in the initial state. In this example, once again the use of the HDL approach proves beneficial in terms of the flexibility and automatic deduction of the environment that has been modelled directly in the HDL system.

Some planners may use a heuristic filter for the goal state, such that only those blocks that are explicitly defined in the goal state are relevant blocks. This, however, would invalidate the results because of the implicit blocks that prevent a block from being picked up, e.g. block a1 or c3. If all the blocks are included in the initial state, the number of blocks increases to seven blocks for the given example. Hence, it will bog the planner down with irrelevant blocks. This is shown in the following chapter.

Table 5.6.: Result of the DL reasoner for the complex blocks world domain.

Concept	Asserted Instances	Inferred Instances
$Block$	a2,a3,a4,b1,b3,b4,c1,c2	a1,a2,a3,a4,b1,b2,b3,b4,c1,c2,c3
$UsedBlock$	a1,b2,c3	a1,a3,b2,c1,c2,c3
$OnTable$	-	a1,a4,b1,b3,c3
$UsedBlockOnTable$	-	a1,c3
$BlockOn$	-	a2,a3,b2,b4,c1,c2
$UsedBlockOn$	-	a3,c1,c2
$ClearBlock$	-	a2,a3,b2,b4,c2
$UsedBlockClear$	-	a3,b2,c2
$InvolvedBlock$	-	a3,c1,c2

The $InvolvedBlock$ concept, which deduces the blocks put on top of $UsedBlock$, is used to define the relevant blocks for the given problem. The $InvolvedBlock$ concept is a subclass of the $UsedBlock$ concept. Thus, instances of $InvolvedBlock$ are also relevant blocks. Table 5.6 shows the deduced output from the DL reasoner for this blocks world problem. Although only three blocks are asserted as $UsedBlock$, the DL reasoner deduces that six blocks are needed for this problem. Hence, the HDL system can correctly produce the initial state for this problem. Considering the example in Figure 5.7, the system has to unstack block a3 before it can pickup block a1. The same must be done with block c3 where two more blocks are hindering its direct manipulation, namely block c1 and c2.

The HDL system generates a planning problem description with 14 facts as shown in A.24. The HTN planner generates 72 possible plans, four of these plans are shown in A.25. Two of these plans are shown in Figure 5.10 without their dummy operators.

Plan #1	Plan #2
(!unstack c2 c1)	(!unstack a3 a1)
(!putdown c2)	(!putdown a3)
(!unstack a3 a1)	(!pickup b2)
(!putdown a3)	(!stack b2 a1)
(!pickup b2)	(!unstack c2 c1)
(!stack b2 a1)	(!putdown c2)
(!unstack c1 c3)	(!unstack c1 c3)
(!putdown c1)	(!putdown c1)
(!pickup c3)	(!pickup c3)
(!stack c3 b2)	(!stack c3 b2)

Figure 5.10.: Solution plans for the complex blocks world problem (without dummy operators).

The resulting plans are valid because their actions include implicit blocks. For example, in order to stack block `b2` on block `a1`, one of the preceding actions should remove block `a3` from `a1`. This action will only be possible if those implicit blocks are modelled in the planning problem description. Anomalies and problems with the blocks world domain are discussed later in Chapter 7.

6. Results and Evaluation

In the previous chapters, a number of planning problems in robotics and AI are solved using the HDL system. They show the benefits of using the system. In this chapter, the system is benchmarked in comparison to the HTN planner. This is done for both the robotics domain and the AI domain. The main question is that of the HDL system's complexity. Firstly, a look into how the HDL system processes the user requests and what the activities are that produce the output. Secondly, the experiment setup is introduced. Finally, the empirical experiments are discussed.

6.1. Complexity of the HDL System

HDL is composed of a number of components. The main ones are the DL reasoner and the HTN planner. Analysing the complexity of such system is not trivial. It could be that a given planning problem is intractable. Hence, what is the complexity of the DL system combined with an HTN planner? The complexity of the system depends on two aspects. The first aspect is the DL reasoning part and the second is the definition of the planning problem. In order to analyse the complexity of the system, one needs to understand how these components interact with each other.

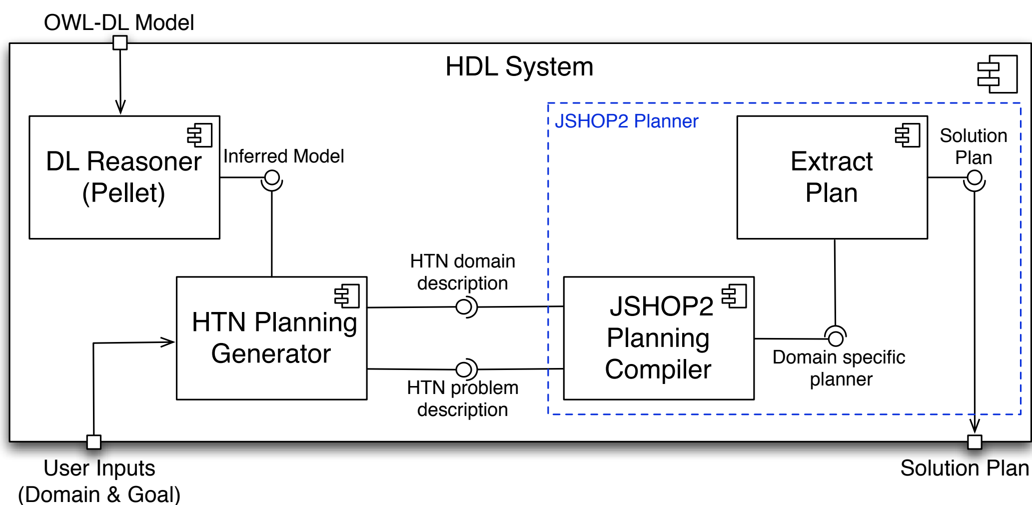


Figure 6.1.: The HDL system's internal component structure.

Figure 6.1 shows HDL's internal component structure. The HDL system needs two in-

puts: OWL-DL model and the user's inputs that consist of a goal and a domain. The output is the sequence of actions in the solution plan. The OWL-DL model is first processed by the DL reasoner and produces an inferred model. This product and the user's inputs are the sources for the HTN planning generator. The generator produces the HTN planning problem and the HTN planning domain for the planner. In this work, the planner is implemented in JSHOP2, which uses planning compilation to optimise the planner. This is discussed later in this section.

The complexity of the DL reasoner depends on the supported DL language and the reasoning algorithms. As mentioned previously in Chapter 2, there are reasoners available that can be used with the HDL system presented here. However, we will concentrate on one particular reasoner that is used in our implementation, namely "*Pellet*" [SPG+07]. HDL uses the *Jena* API [McB01] and the OWL API [HM03, HBN07] for communicating with *Pellet*. Figure 6.2 shows Pellet's main components. Besides the two interfaces mentioned before, Pellet also supports DIG (DL Implementation Group) interface, a standardised XML interface for DL systems [BMC03, Dic04, TBK+06], and the SPARQL Parser [PS07]. The details of the Pellet reasoner are beyond the scope of this work. More information is found in [SPG+07].

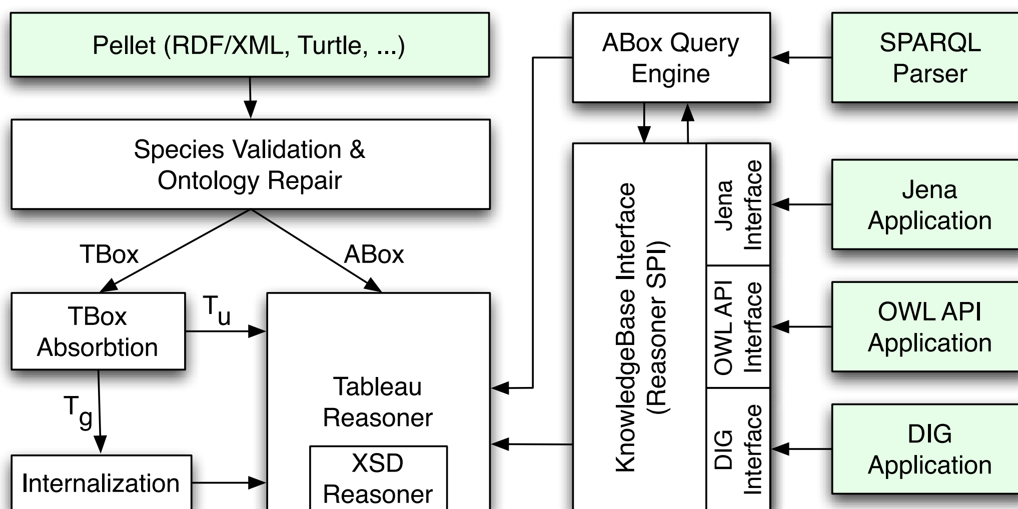


Figure 6.2.: Main components of the Pellet reasoner [SPG+07].

Pellet is a complete OWL-DL and a very incomplete OWL-FULL consistency checker. Hence, it can prove whether an OWL model is consistent, inconsistent or unknown. Pellet uses state of the art optimisation techniques provided in DL literature which have also been implemented in other reasoners such as FACT++ and RACER. Expressive DLs, for example $\mathcal{SHOIN}(\mathcal{D})$, have a very high worst-case complexity [SPG+07].

DL $\mathcal{ALC}$ is the basic language of DL and mostly used by DL reasoning services. [BHS08] proves the complexity of DL $\mathcal{ALC}$. Two of the complexity theorems found there are repeated below:

Theorem 6.1. *Satisfiability and subsumption of $\mathcal{ALC}$ concepts and consistency of $\mathcal{ALC}$ ABoxes*

are PSpace-complete problems [BHS08, Theorem 3.4].

Theorem 6.2. *Satisfiability in $\mathcal{ALC}$ with respect to general TBoxes is* EXPTIME*-complete [BHS08, Theorem 3.6].*

This high worst case complexity (EXPTIME-complete) of $\mathcal{ALC}$ with respect to an arbitrary KB is artificial and rarely occurs in practise [HKNjP94, Hor98, Neb90, BHS08]. Modern DL reasoning systems, such as Pellet, FACT++ and RACER, use a wide range of optimisation techniques to improve the *typical case* performance by several orders of magnitude [HPS99, BHS08].

The next component in Figure 6.1 is the HTN *planning generator*. Figure 6.3 depicts the components involved in the planning generation process. This process consists of four steps. The first step is carried out by the "*domain generator*" which processes the domain selected by the user. The user can choose either an instance of a $Planning\text{-}Domain$ or an instance of a $Method$. The worst case complexity of this step is $O(n)$ as described in Section 2.5.1 for Algorithm 2.2. The second step is that carried out by the problem generator that composes the planning problem from user defined goals and extracts the initial state s_0 from the domain d. The worst case complexity of this second step is $O(mn)$, where $m = (|\mathbb{M}| + |\mathbb{O}|)$ and $n = (|Thing| - |Planning|)$, as described in Section 2.5.1 for Algorithm 2.1. The third and fourth steps are accomplished by "java object to planner syntax". Its algorithm is straight forward through the input list. Hence, its complexity is $O(n)$.

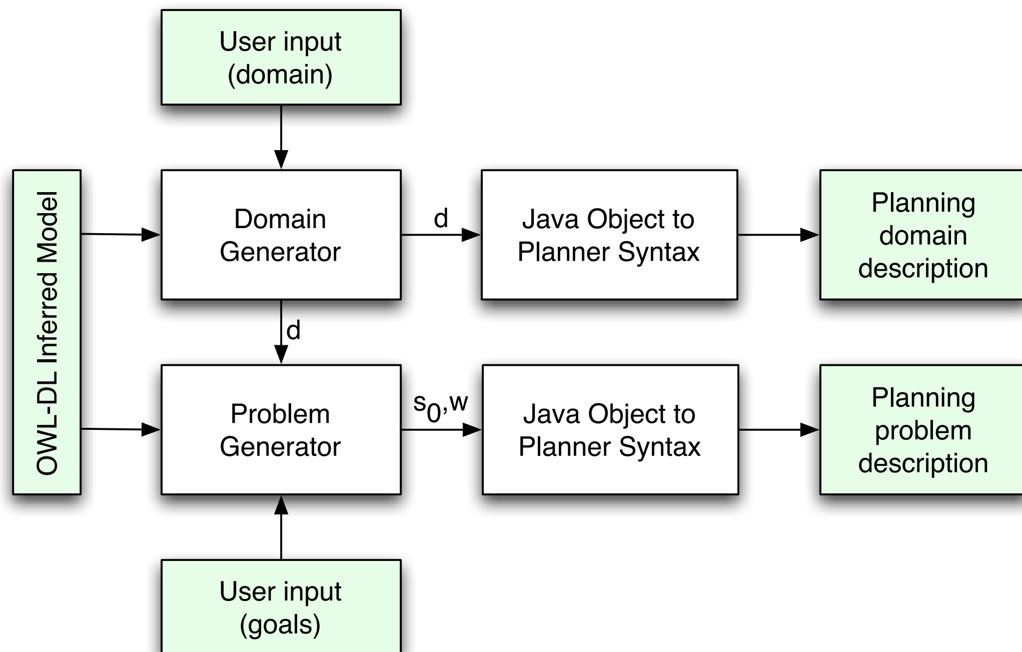


Figure 6.3.: HTN planning generator components.

The complexity of the HTN planning generator is composed of these four steps. In most

cases, the planning domain is invariant or constant. On the other hand, the planning problem might grow according to the amount of stored knowledge. Hence, the first and third steps' complexity can be replaced by a constant. The complexity of the problem generator's step is $O(mn)$. By assuming the planning domain to be constant, ($m = c$), the complexity is $O(n)$. The fourth step also depends on the problem generator. Hence, the complexity will also be $O(n)$. The overall complexity of the HTN planning generator is $O(n)$.

The last component in Figure 6.1 is the HTN planner, in this case JSHOP2. Although in the figure it is shown as two components, these components are in fact part of JSHOP2 and are described later in this section. The complexity of totally-ordered HTN planning is ExpSpace-hard and in Double-ExpSpace. The computability of HTN planning without this restriction is semi-decidable [NSE98]. Table 6.1 shows a comparison of the complexity of HTN planning with different settings.

Table 6.1.: Complexity and computability of HTN planning [EHN94a].

Restrictions on non-primitive tasks	Must every HTN be totally ordered?	Are variables allowed?	
		no	**yes**
none	no	Undecidable	Undecidable [2]
	yes	in ExpTime; PSpace-hard	in DExpTime; ExpSpace-hard
"regularity" [1]	doesn't matter	PSpace-complete	ExpSpace-complete
no non-primitive tasks	no	NP-complete	NP-complete
	yes	Polynomial time	NP-complete

[1] At most one non-primitive task, which must follow all primitive tasks.
[2] Even if the planning domain is fixed in advance.

The SHOP system is a domain-independent HTN planning system, which uses a sound and complete HTN planning algorithm [NCLMA99]. SHOP2 extends the SHOP planning algorithm enabling the decomposition of each method into a partially ordered set of subtasks and allowing the creation of plans that interleave subtasks from different tasks [NMAC+01]. The planning procedure is Turing-complete as well as sound and complete over a large class of planning problems [NMAC+01, NIK+03]. SHOP and SHOP2 are written in Lisp. However, JSHOP and JSHOP2 are written in *Java*. JSHOP2 compiles its domain description into a domain-specific planner and then runs that planner to solve the planning problem in that domain. Hence, some optimisation of that domain can be performed by considering the information in advance. Another reason for the compilations is to enable external code calls [Ilg06]. Figure 6.4 shows this compilation process. Thus, it explains why in Figure 6.1 JSHOP2 is represented as two components.

The complexity of the HDL system's components has been presented. The most complex component in the system is the HTN planning system. Its complexity depends on the planning domain and the planning problem for that domain. The HDL system neither improves the planning algorithm nor the DL reasoning algorithm. However, the HDL system enables the filtering

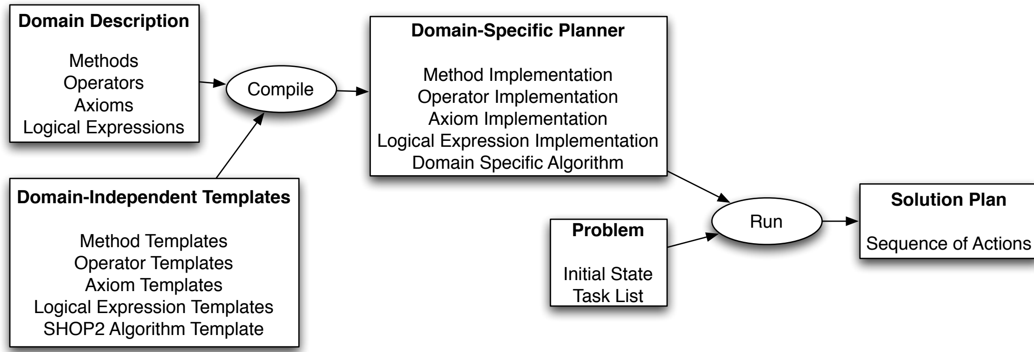


Figure 6.4.: JSHOP2 compilation process [Ilg06].

of the planning problem which increases the performance of the overall system. Empirical experiments which support this claim are presented in Section 6.3. The factor by which the HDL system improves the overall performance up to the planning problem generation is presented next.

Definition 6.3. Let $\pi = \{a_1, a_2, ..., a_n\}$ be the optimal solution of a planning problem $\mathcal{P} = \{s_0, w, M, O\}$, where $s_0 = \{s_1, s_2, ..., s_n\}$ is the set of the initial states. The planning problem $\mathcal{P}$ contains only relevant states (s_0) iff:

- The optimal solution plan will remain $\pi = \{a_1, a_2, ..., a_n\}$ although additional states s_{irr_i} are added into s_0, such that $s_0 = \{s_1, s_2, ..., s_n, s_{irr_0}, ..., s_{irr_i}\}$.
- The solution plan will be invalid or missing if any of the states in s_0 is omitted.

The complexity of the planning system depends on the domain and the planning problem. In one domain, the complexity is defined by the number of involved states, which is s_0 in HTN. Hence, having only relevant states in the planning problem will reduce the planning complexity. Figure 6.5 illustrates the amount of states or information which are stored in the DL KB on the left side and possible generated planning domains on the right side. It shows that the generated planning problem might vary, depending on how the *UseState* property is defined. In the following section, an experiment design for empirical experiments is presented.

6.2. Experiment Design

The purpose of the experiment is to compare or benchmark the effect of having irrelevant states in the planning domain on the HDL planning system and the pure HTN planning system. The benchmark is measured by the four values, namely number of states in the model, number of states in the planning problem, number of solution plans, and time to plan. The benchmark is performed fairly in that the irrelevant states are still part of the domain, although the HDL

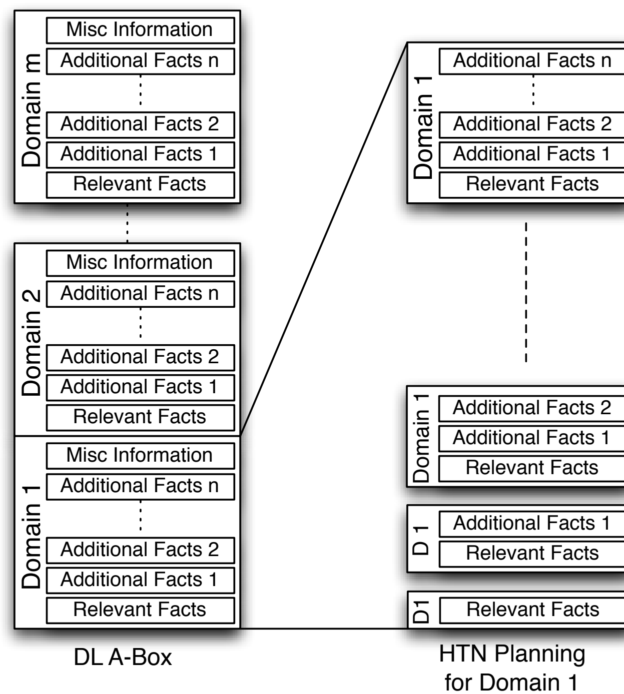


Figure 6.5.: Stored states in DL and possible generated HTN planning domain.

system can contain states from different domains. Although the benchmark is intended to compare pure HTN planning and the HDL system, the pure HTN problem is generated by the HDL system too. As shown in Figure 6.5, the planning problem generation depends on the user who decides which instances of a concept should be involved in the problem description. Thus, the HDL system might generate a relevant planning problem and also some planning problem with additional facts, which could be irrelevant for this particular problem.

Figure 6.6 shows the benchmark scenario. The preparation step generates two sets of OWL-DL models, one for the pure HTN approach and the other for the HDL approach. In both approaches, the same procedures are performed. The upper part is for the pure HTN approach and the lower part is for the HDL approach. These procedures can be seen as a sequence of two steps; the first step is the DL reasoning and the second one is the planning step. In the first step, the OWL-DL model is processed and its results are the HTN planning domain description and problem description. The planning step is the decomposition of the planning problem to find the solution plan. The inputs of this step are the planning problem description and the planning domain description. This step is the usual process in the planning system. Hence, it represents exactly how the pure HTN approach is executed.

In the experiment, the number of states increases until one of the systems can not handle this anymore. The other system still runs for some additional states. As mentioned before, four criteria are considered for this benchmark. The notions for the measurements are defined as

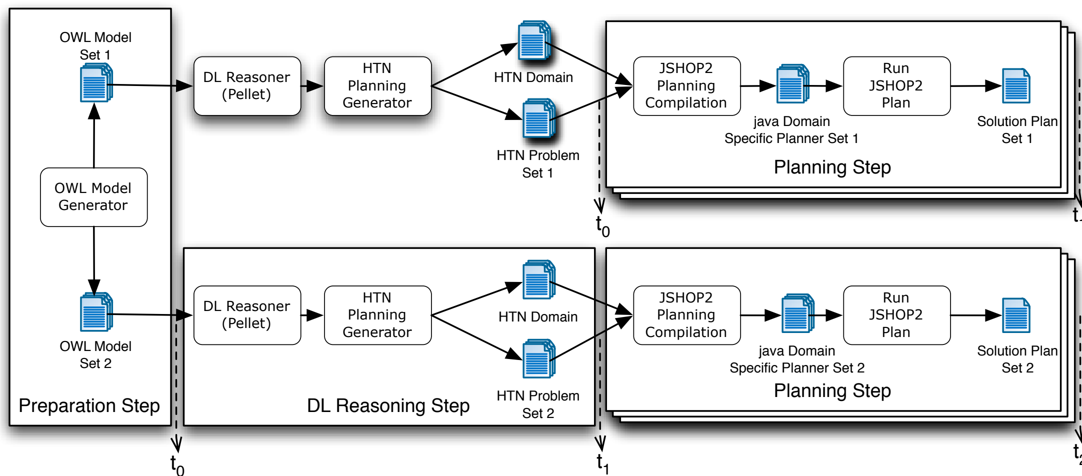


Figure 6.6.: Experiment scenario in navigation domain.

follows:

- Σn_{HTN}, Σn_{HDL} represent the number of states in the model.
- $\Sigma s0_{HTN}$, $\Sigma s0_{HDL}$ represent the number of states in the planning problem.
- $\Sigma \pi_{HTN}$, $\Sigma \pi_{HDL}$ represent the number of solution plans.
- t_{HTN}, t_{HDL} represent the required time to extract the plan.

The first three parameters are measured immediately for each run. The fourth parameter, however, is computed as follows (see Figure 6.6):

$$\begin{aligned} t_{HTN} &= t_{PlanningStep} = t_1 - t_0 \\ t_{HDL} &= t_{ReasoningStep} + t_{PlanningStep} = (t_1 - t_0) + (t_2 - t_1) \end{aligned}$$

As shown in the equations above, the HDL approach consists of time to reason and time to plan. Hence, mathematically for the same problem it will follow this equation $t_{HDL} \geqslant t_{HTN}$.

Figure 6.6 also shows the internal processes of the experiments. Each step is executed in a separate process, however the planning step is executed in a new process for each single problem. Why does the planning step need special treatment in this case? As explained in the previous section, JSHOP2 uses a special compilation process to optimise the plan. Thus, every problem has its own java classes. Due to the problem generation scheme in the previous two steps, the generated java classes where the main function is placed have the same name for every problem in the same domain. The nature of the java class loader is such that it will not load classes that have been loaded into its memory [Chr]. Hence, running the experiment with one process for all the generated problems will produce the result for the first problem only. The following problem will not be loaded and executed properly. Therefore, every problem has to run in a new process.

6.3. Experiments

Two experiments are performed to benchmark the effect of additional facts in the planning problem between pure HTN planning and HDL planning. These experiments are "*navigation domain*" and "*blocks world domain*". The benchmark procedure is performed as described in the previous section. A detailed analysis of the results are presented and discussed in the following sections.

6.3.1. Navigation Domain

The navigation domain has been explained in detail in Section 3.2. The *TBox* is defined as depicted in Figure 3.6 on page 53. Two actors are defined in the domain, namely *robot1* and *robot2*. The goal is to navigate them from *room-start* to *room-goal*. Figure 6.7 shows the test scenario map for this navigation domain.

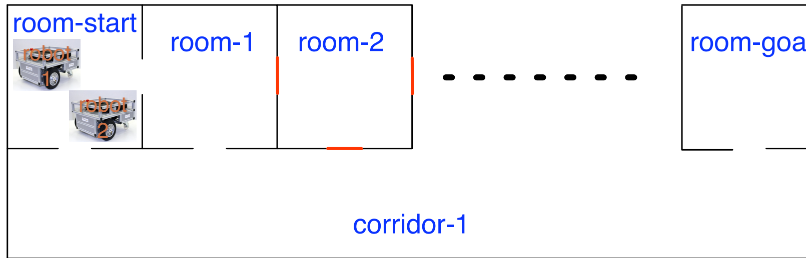


Figure 6.7.: Test scenario in navigation domain.

The planning domain $\mathcal{D}$ is the same for both approaches. It does not change during the test. It is detailed in Section 3.2.3 and the HDL model for this domain is covered in Section 3.2.5. The planning domain description is shown in Listing A.1.

The task network w is also the same for the whole test, namely $w = \{$ `(navigate robot1 room-goal),(navigate robot2 room-goal)` $\}$. The initial states s_0 varies in each test; it is one of the measurement parameters in the benchmarking process, namely $\Sigma {s_0}_{HTN}$ and $\Sigma {s_0}_{HDL}$. However, s_0 is a dependent variable and is generated automatically in the reasoning step. The parameter that is changed incrementally in each test is the number of states in the model (Σn_{HTN} and Σn_{HDL}).

In the test scenario shown in Figure 6.7, the optimal solution plan is to drive the robots to `room-goal` through `corridor-1`. Hence, these three states (`room-start`, `corridor-1` and `room-goal`) are the relevant ones. However, the first room (`room-1`) has two open doors, one of which is connected with `room-start` and the other is connected with `corridor-1`. The intention of having the first room with two open doors is to test that the given JSHOP2 parameters are set to return all solution plans. In every test, an additional room with three "closed" doors is added between `room-start` and `room-goal` as shown in Figure 6.7. This additional room with three closed doors is intended to maximise the difference between the two approaches. Thus,

the number of states (rooms) in the model is formalised as follows:

$$\Sigma_{i=1}^{p} n_{HTN} = \Sigma_{i=1}^{p} n_{HDL} = 3 + i$$

p is determined either by the first approach that fails to generate the plan or when it reaches the upper limit which is 1000 in these experiments.

The nth and m^{th} room (where $m = n - 1$) are defined in OWL-DL as follows:

```
Room(room-n),
adjacentto(room-n, corridor-1), adjacentto(corridor-1, room-n),
adjacentto(room-n, room-m), adjacentto(room-m, room-n),
Door(door-n), hasState(door-n, isClose),
Door(door-n-m), hasState(door-n-m, isClose)
```

In each incremental process, three instances and six properties are added to the model. However, the planning problems contain some of this information due to the restriction on the expressivity of the planning system (see Chapter 3). Nevertheless, the HDL system generates valid planning problems automatically. The difference is in the $useState$ property. The pure HTN approach uses instances of $Room$ and the HDL approach uses instances of $DriveableRoom$. Hence, each added room into the model will also appear in the planning problem for the pure-HTN approach. However, it is not the case for the HDL approach because the room is not accessible for the robot. Each additional room will add five more facts into the planning problem. For n^{th} and m^{th} room (where $m = n - 1$) will add the following facts:

```
(room room-n)
(adjacentto room-n corridor-1)
(adjacentto corridor-1 room-n)
(adjacentto room-n room-m)
(adjacentto room-m room-n)
```

6.3.1.1. Analysing the Result

The benchmarking was done on a 2 GHz Intel Core Duo MacBook 13" (version 1,1) with 2 GB RAM and Java version 1.5. All tests were done with default Java settings, no tweaking on the heaps or memory allocation for the Java VM was carried out.

Number of Facts in the Planning Problem This is the result of the DL reasoning step in which the planning problem is generated from the DL model. Figure 6.8 shows the plots of the number of facts for both approaches. Let m_{HTN} and m_{HDL} represent the number for states in the planning problem ($\Sigma s_{0_{HTN}}$ and $\Sigma s_{0_{HDL}}$) and n represents the number of rooms in the DL

model. Thus, the number of facts are expressed by the following functions:

$$m_{HTN}(n) = 5n + 9$$
$$m_{HDL}(n) = \begin{cases} 14 & \text{if } n = 1, \\ n + 14 & \text{if } n > 1 \end{cases}$$

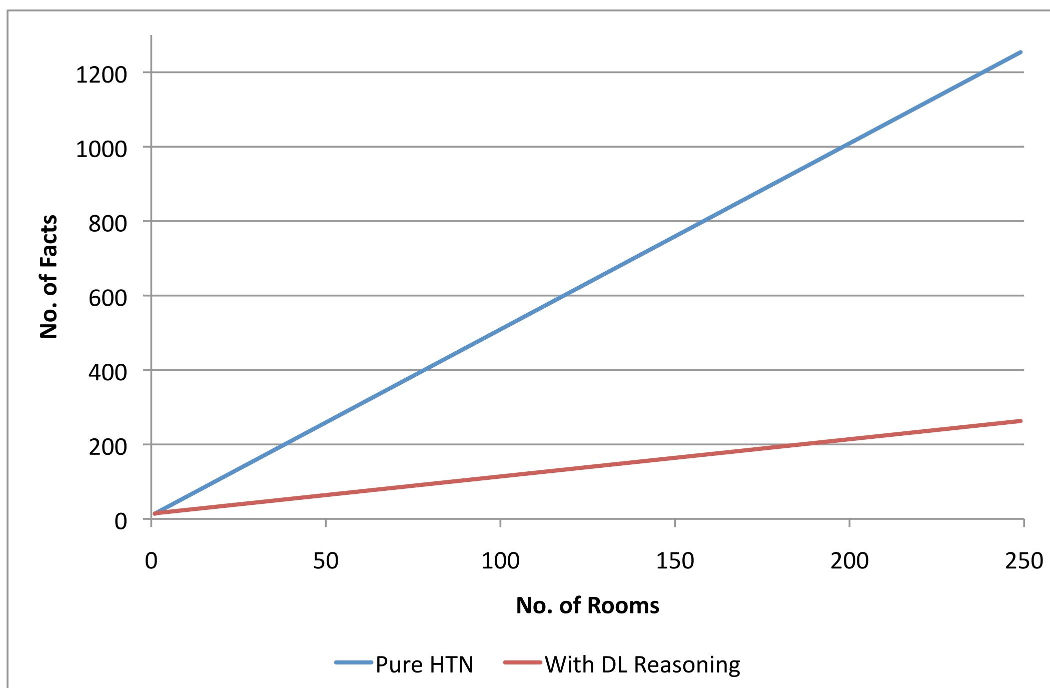


Figure 6.8.: Number of facts in the planning problem for navigation domain.

The number of facts in both approaches increases linearly with the number of added rooms. However, the pure-HTN approach grows five times faster than the HDL approach. It is clear for the pure-HTN approach that each additional room will add five facts to the planning problem. However, why does the HDL approach also increase with the number of rooms? The DL reasoner reasons about the DL model and should return only instances of $DriveableRoom$. Hence, the number of facts for the HDL approach should be constant. The answer to this question is found in the modelling process as described in Section 3.3. Each $Room$ has a property $adjacentto$, that defines the topological connection between rooms. The property $adjacentto$ is a *symmetric relation*. Thus:

$$\forall\, n, m \in Room, n\ adjacentto\ m \Rightarrow m\ adjacentto\ n$$

Each added room has a topological connection with `corridor-1`. While `room-n` is not included in the planning problem, `corridor-1` is always included. Thus, the fact `(adjacentto corridor-1 room-n)` is added incrementally to the planning problem. For the same reason, the second test ($n = 2$) added two additional facts from the first test. One of the facts is due to the connection between `corridor-1` and `room-2` and the second one is due to the connection

between `room-1` and `room-2`.

Number of solution plans This is one outputs of the planning system that depends on the planning problem. Figure 6.9 depicts the number of generated plans with relation to the number of added rooms for both approaches. It shows that the number of generated plans is constant in the HDL approach, which is four. In contrast, in the pure HTN approach the number of generated plans increases quadratically with the number of rooms. The following equations represent the relation between number of rooms n and number of solution plans π:

$$\begin{aligned}\pi_{HTN}(n) &= n^2 + 2n + 1 = (n+1)^2 \\ \pi_{HDL}(n) &= 4\end{aligned}$$

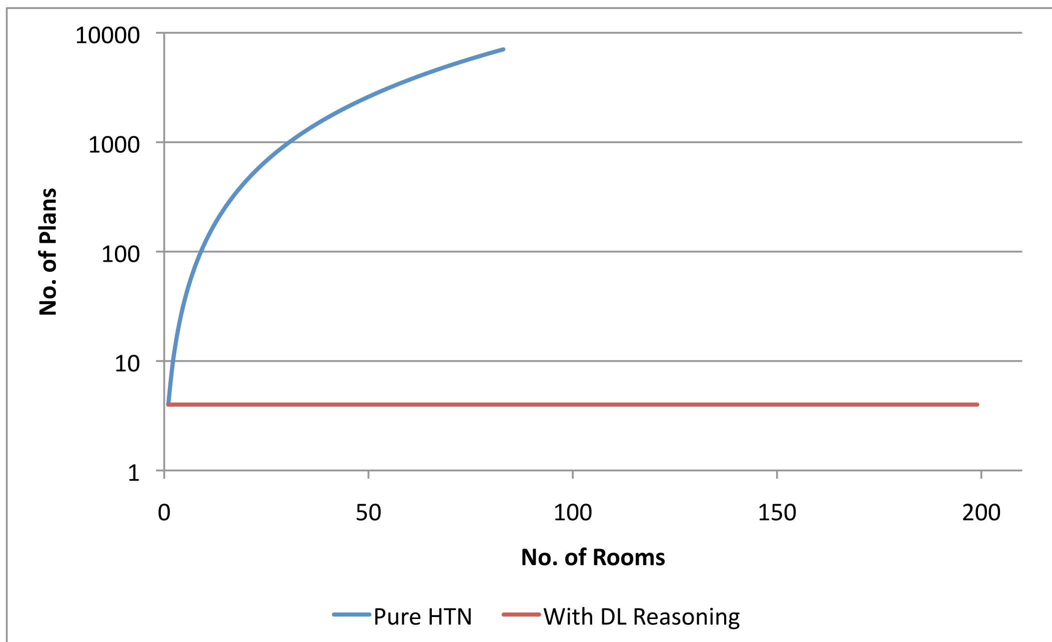


Figure 6.9.: Number of generated plans (logarithmic scale) in the navigation domain.

The upper bound of n in the pure-HTN approach is 83, because the planner failed to return any plan for $n > 83$.

The reason why the number of solution plans increases quadratically is due to the number of robots in the model. The planning goals are to navigate those robots into `room−goal`. Hence, each added room provides a new path to reach the goal (see Figure 6.7). The planning system found $(n + 1)$ possible paths to reach the goal for one robot. For each path, the planning system will try to fulfil the second objective, which has the same number of possible paths. Therefore, the number of plans is $(n + 1)(n + 1)$. Additional robots will increase the number of solution plans with the factor of possible paths. In this navigation benchmark domain, the relation between number of rooms n and number of robots r is described in the following

equation:

$$\pi_{HTN} = (n+1)^r$$

Time to plan This is the total amount of time needed to generate a solution plan. The formula to measure the time to plan is described in Section 6.2. Figure 6.10 shows the time for extracting the plans with relation to the number of rooms. In the pure HTN approach the planning time increases commensurately with the number of rooms in the problem description. The planning time is less than two seconds for rooms less than 25. It is 193 seconds for $n = 83$. For $n \geq 84$, the planner returned no plans. The elapsed time before the planner gave up is shown as a dashed line in green.

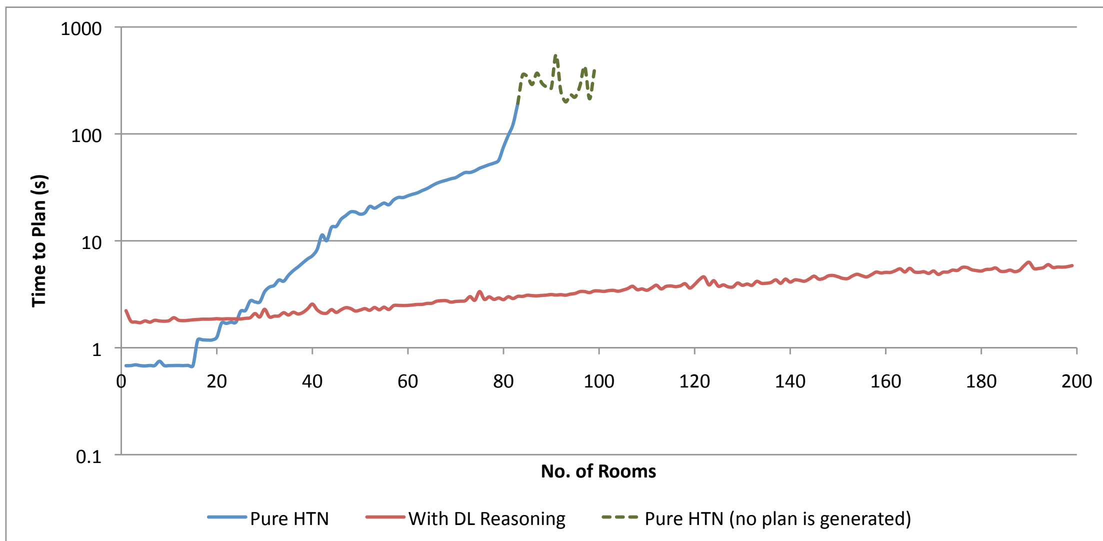


Figure 6.10.: Computation time (logarithmic scale) in the navigation domain.

In the HDL approach, the overall planning time consists of DL-reasoning time and HTN-planning time. Therefore, for $n < 25$ it requires more time than the pure HTN approach. However, the average overall time for up to 200 rooms is 3.5 seconds. The overall time increases in average around 17 ms for each added room into the DL model. This is less than 0.5% of the overall time.

In the Figure 6.10, the t_{HTN} increases slightly quadratically before it rises erratically. Applying regression on both curves with n as number of rooms produces the following equations:

$$\begin{aligned} t_{HTN}(n) &= 0.012n^2 - 0.33n + 2.3, \sigma_\epsilon = 9.94, (0 < n < 80) \\ t_{HDL}(n) &= 0.023n + 1.2, \sigma_\epsilon = 4.05, (0 < n < 250) \end{aligned}$$

In contrast, the HDL approach increases linearly with a very gentle slope. Hence, it does not suffer much with the increasing number of rooms.

Figure 6.11 shows the computation time for extracting one solution plan only in relation to the number of rooms. As previously mentioned, the HDL approach increases slightly for

each added room which is clearly shown in the figure. However, the computation time for the planning only in the HDL approach is steady. This is shown as the dotted line in the figure. It is obvious that the number of added rooms does not influence the computation time of the planning part. This is because the planning problems contain relevant states only. In the pure HTN approach, the computation time for planning one solution is smaller than that in the HDL approach where the reasoning time remains the same although it should compute one solution plan only. However, the computation time in the pure HTN approach increases slightly as the number of rooms increases. This is shown in the figure by comparing the pure HTN approach with the planning part of the HDL approach. The pure HTN approach can still extract a solution plan for $n \geq 84$. However, it has another limitation due to the JSHOP2 compilation process (see Figure 6.4). The java compiler produced the error "*code too large*" in the "`createState0(State s)`" method while it processed the planning problem with 331 additional rooms. Hence, the pure HTN approach can only process a planning problem where $n < 331$.

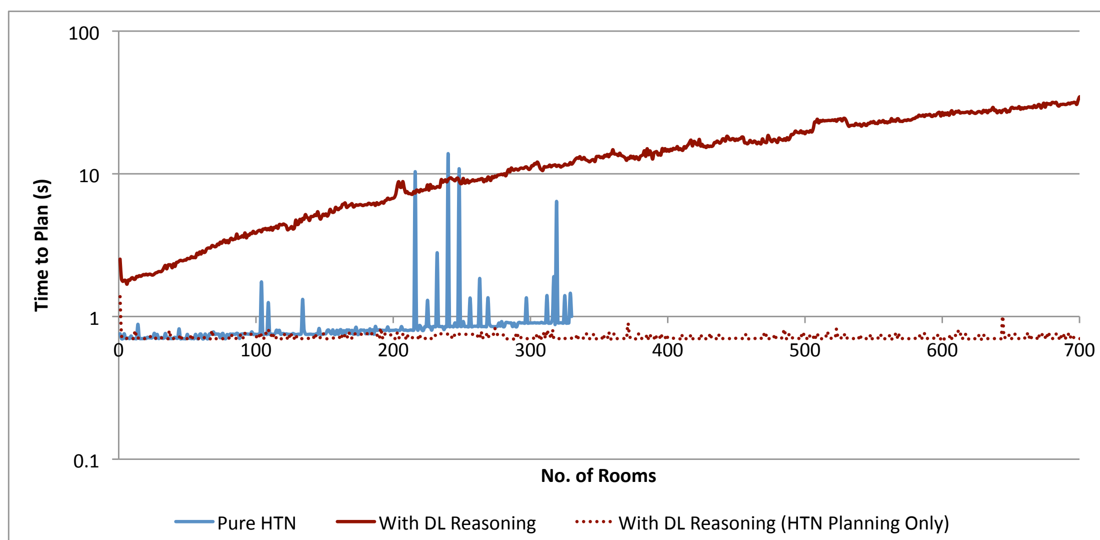


Figure 6.11.: Computation time (logarithmic scale) in the navigation domain for planning one solution.

6.3.2. The Blocks World Domain

The blocks world domain is described in-depth in Chapter 5. In this experiment, the four blocks problem as shown in Figure 5.1 on page 98 is used. The planning domain $\mathcal{D}$ is the same for both approaches and can be seen in Listing A.18. The HDL system uses the enhanced blocks world domain ontology as shown in Figure 5.6 (page 108) in its *TBox*. An additional block b_n is added into the model incrementally as shown in Figure 6.12.

The planning goal defined in the *initial task network* w is same for both approaches, namely $w = \{$`(achieve-goals ((on-table b1) (on b4 b1) (clear b4) (on-table b3) (on b2 b3) (clear b2)))`$\}$. The initial state s_0 is generated automatically dur-

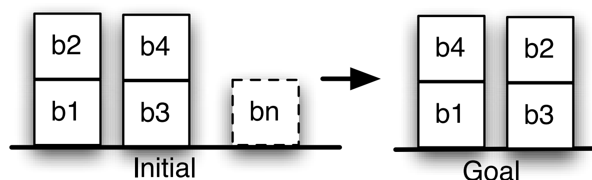


Figure 6.12.: Experiment scenario for the blocks world domain.

ing the reasoning step. The added blocks increase the number of states in the model (Σn_{HTN} and Σn_{HDL}). In Figure 6.12, the relevant blocks are `b1`, `b2`, `b3`, and `b4`. These blocks are asserted in the *ABox* of the HDL system as follows:

```
UsedBlock(b1), blockOn(b1, Table),
UsedBlock(b2), blockOn(b2, b1), hasState(b2, isClear),
UsedBlock(b3), blockOn(b3, Table),
UsedBlock(b4), blockOn(b4, b3), hasState(b2, isClear)
```

The number of states (blocks) in the model is formalised as follows:

$$\Sigma_{i=1}^{p} n_{HTN} = \Sigma_{i=1}^{p} n_{HDL} = 4 + i$$

As in the navigation domain experiment, the value p is determined by the first approach that fails to generate a plan or when it reaches the upper limit which is 250 for this experiments.

The nth block is defined in the OWL-DL as follows:

```
Block(bn), blockOn(bn, Table), hasState(bn, isClear)
```

This information maps directly into the planning problem, because these are the facts that describe the block `bn` in the environment. The involved blocks in the HTN approach's $useState$ are defined as instances of $Block$. However, the HDL approach uses instances of $UsedBlock$ instead. Thus, any newly-added block will immediately appear in the HTN approach, as though those blocks had been directly inserted into the planning problem. The differs from the navigation domain in that the model of block `bn` and its properties will be translated one to one into the planning problem. The planning problem's facts for block `bn` are shown below:

```
(block bn)
(on-table bn)
(clear bn)
```

6.3.2.1. Analysing the Results

The benchmarking was carried out using the same machine and specifications as mentioned in Section 6.3.1.1.

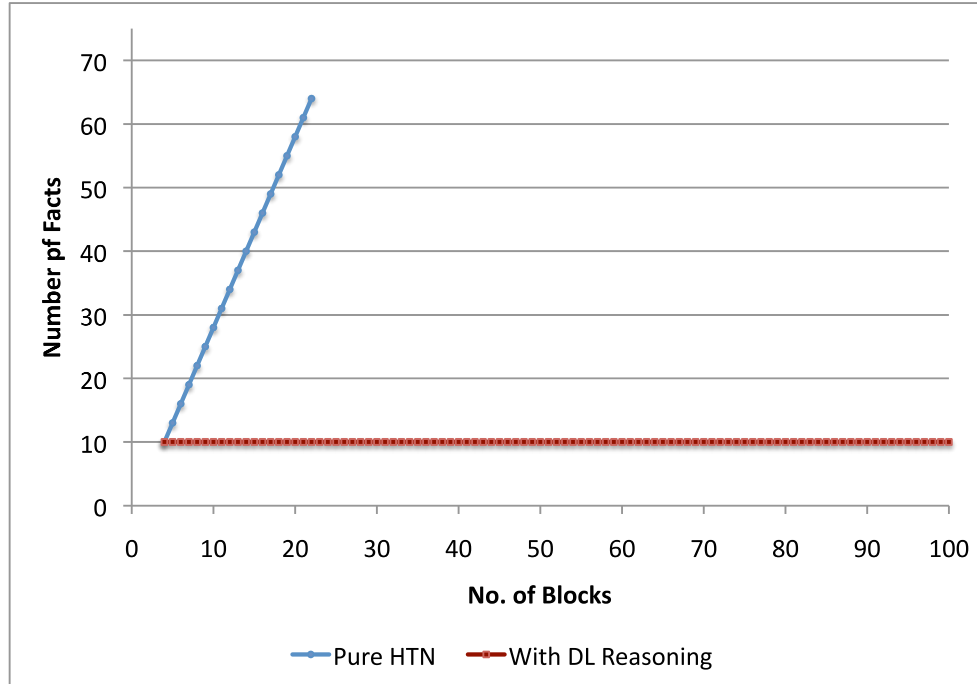


Figure 6.13.: Number of facts in the planning problem for the blocks world domain.

Number of facts in the planning problem These are shown in Figure 6.13. Let m_{HTN} and m_{HDL} represent the number of states in the planning problems ($\Sigma s_{0_{HTN}}$ and $\Sigma s_{0_{HDL}}$) and n represent the number of blocks in the DL model. Hence, the number of facts are expressed as follows:

$$m_{HTN}(n) = 3n - 2$$
$$m_{HDL}(n) = 10$$

In these equations, n is the number of all blocks in the DL model, not only the additional blocks. Therefore, n starts from four upwards. In the HDL approach, the facts in the planning problem stay constant at 10 facts regardless of the number of the blocks in the DL model. The complete planning problem is shown in Listing A.19 in Appendix A.4.1. In the HTN approach, the number of facts increases to three times the number of added blocks. However, the initial state s_0 in the HTN approach is also generated automatically by the HDL system. Although the relevant blocks are defined as instances of $UsedBlock$ and the HTN uses instances of $Block$ for generating the planning problem. Those relevant blocks are also included in the planning problem because $Block$ subsumes $UsedBlock$ ($Block \sqsubseteq UsedBlock$).

Number of solution plans These are shown in relation to the number of blocks in DL model in Figure 6.14. The number of solution plans for the HDL approach remains constant at four in this experiment. However, in the HTN approach, it increases along a power series. The

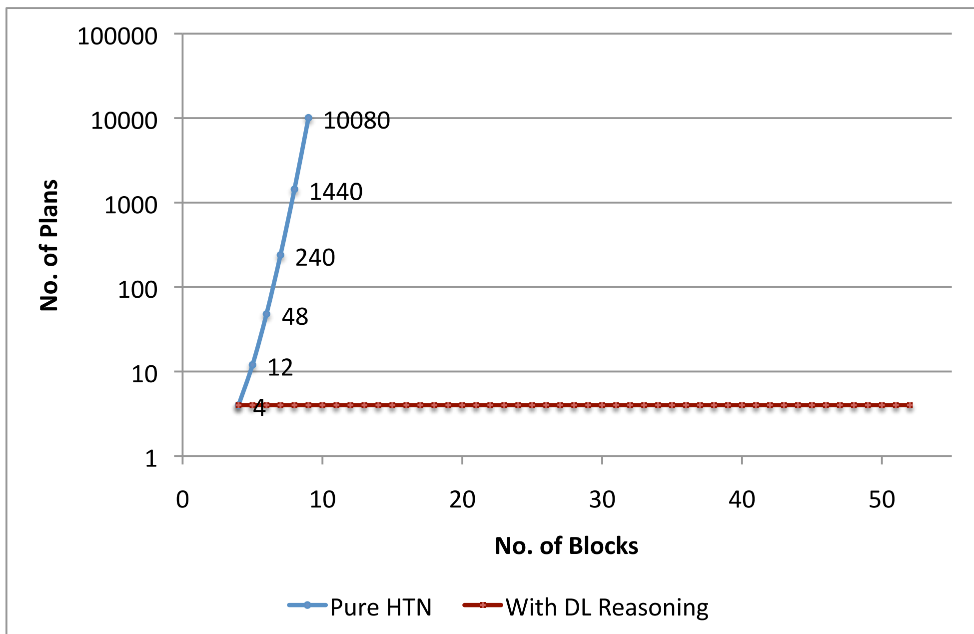


Figure 6.14.: The number of generated plans (logarithmic scale) in the blocks world domain.

following equations show the relation between the number of blocks n and number of solution plans π:

$$\begin{aligned}\pi_{HTN}(n) &= 2(n-2)! \\ \pi_{HDL}(n) &= 4\end{aligned}$$

The HTN approach can decompose the plan to nine blocks in the DL model or five extra blocks in addition to the relevant blocks. Why does this happen in the HTN approach? Is the planning domain not properly modelled? The blocks world domain was carefully defined as described in Section 5.1.2, it detects unused blocks and marks them with `(dont-move(x))`. However, this is where the flaw of this domain lies. Let us analyse the problem as shown in Figure 6.12 carefully. Two of the relevant blocks (`b1` and `b3`) are already in the right place. Hence, the solution plans should either start by picking block `b2` or `b4` as its first move. Before these actions are performed, the domain tells the planning system to mark all blocks that do not need to be moved, i.e. block `b1` and `b3`. However, the ordering of this labelling is not determined. As a result, the system will label them in any possible ordering ($x!$). This explains the equation π_{HTN} above. The factor two, in the equation above, is the number of possible solution plans and the factor $(n-2)$ is the total number of blocks that need to be labelled as `dont-move`.

Time to plan For the blocks world problem, the time to plan is shown in Figure 6.15. As mentioned previously in Section 6.2, the HDL approach needs extra computation time for reas-

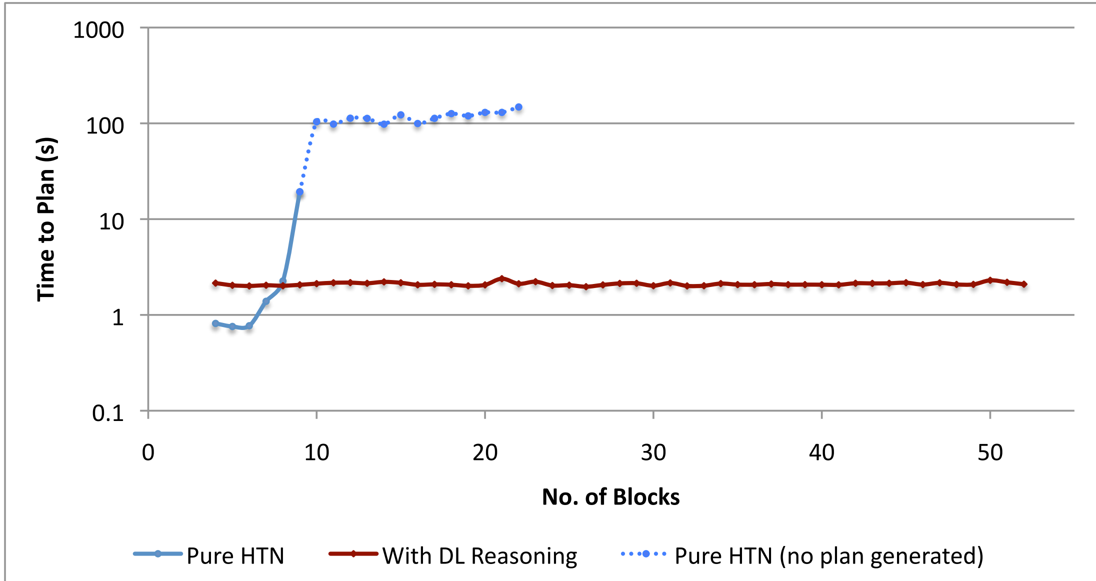


Figure 6.15.: Computation time (logarithmic scale) in blocks world domain.

oning about the DL model. This can be clearly seen in this figure; for $n < 8$, or four additional blocks, the computation time in the HDL approach is larger than that in the pure HTN approach. Adding new blocks slightly increases the overall computation time. However, this time is relatively small: given up to 50 blocks, the overall time is still two seconds. The average overall time is 2.1 seconds. The average added time for each additional block in the model is $1\ ms$. This is on average less than 0.2% of the overall computation time. Below are the equations that relate the time to plan to the number of blocks in the model by applying regression:

$$\begin{aligned} t_{HTN}(n) &= 0.32n^4 - 7.6n^3 + 66n^2 - 250n + 350, \sigma_\epsilon = 1.08, (3 < n < 10) \\ t_{HDL}(n) &= 0.0043n + 2, \sigma_\epsilon = 1.08, (3 < n < 151) \end{aligned}$$

The pure HTN approach needs $19\ s$ to extract the plan when five additional blocks are added to the problem. It needs $104\ s$ to plan with six additional blocks, however, it returns no plan due to memory failure. The dotted line shows the pure HTN approach for $10 \leq n \leq 22$, where the planner did not return any solution plan. For $n > 22$, the JSHOP2 planner gave up completely.

Figure 6.16 shows the computation time for extracting one solution plan only in relation to the number of blocks. The HDL computation time increases linearly with the number of blocks. However, the computation time for the planning part in the HDL approach is steady, which is shown by the dotted line in the figure. Thus, the additional blocks affect only the HDL reasoning part. In the pure HTN approach, the computation time for $n < 218$ is steady similar to the computation time for the planning part in the HDL approach. However, for $n \geq 218$ the computation time jumps from $0.75\ s$ to $1.25\ s$. This is due to memory allocation in Java, which allocates memory in blocks as needed by the program. This explains why the increase is not

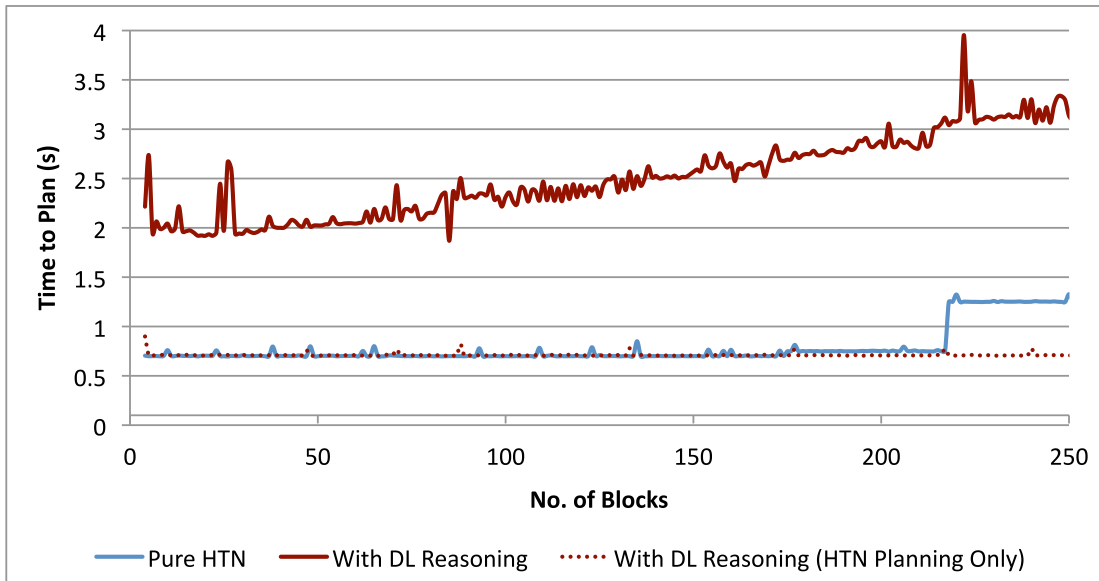


Figure 6.16.: Computation time (logarithmic scale) in the blocks world domain for planning one solution.

linear but jumps from one value to another.

6.4. Concluding Remarks

Both experiments, that of the navigation and of the blocks world domains, have shown that that HDL approach outperforms the HTN approach. The planning complexity depends on the planning domain that includes the number of operators and heuristics of that domain. In the blocks world domain the addition of a few additional blocks causes the blocks world problem to become intractable. However, in the navigation domain the planner continued to perform when several rooms were added. There is a trade off for using the HDL system, namely additional time to reason about the model. This is shown clearly in the blocks world domain. Nevertheless, the additional time is relatively small in comparison to the benefit of using the HDL system.

One should consider that in the experiments above, both approaches were generated by the HDL system. It shows that the features and properties of the HTN planning system are inherited by the HDL system. Thus, the problems that were faced by the HTN planning approach could also appear in the HDL approach's generated planning problem when the features are not chosen carefully. However, the HDL system is more expressive than the HTN planning system. It can capture the environment and model it comprehensively in DL and still produces a relatively small planning problem, with the same planning-domain.

7. Discussion

In this chapter, the HDL system is discussed. A number of open questions and possible improvements of some methods that have been presented in the previous chapters will be addressed.

7.1. HTN Blocks World Anomaly

HTN planning is a heuristic planning approach, where the knowledge for solving the problem is modelled in the planning domain. The blocks world domain, as described in Chapter 5, shows that, in the HTN planning's implementation for this domain, six operators, eleven methods, and two axioms are used. In contrast, the original STRIPS planning domain needs only four operators. These additional methods and operators provide the HTN planner with a way to decompose the planning problem.

One of the new methods, `find-nomove`, is defined for marking the blocks that are already located in the desired position. Therefore, these blocks do not change during the rest of the planning process. In a programmer's mind, this method is necessary. However, in this section it is shown that some other problem might arise due to this method. One might expect to get an empty solution plan $\pi = \{\}$ if $w_0 = s_0$, or in words the goal states are equal to the initial states. Unfortunately, this is not the case in HTN planning, where the solution plan is not empty. Therefore, we call this the HTN blocks world anomaly.

In Section 6.3.2.1, the results showed that one factor which impacts the number of solution plans in the blocks world domain is the number of "non-moveable" blocks. If the initial state is the same as the goal state, the number of solution plans in not empty but $x!$ where x is the number of non-moveable blocks. Therefore, the planner can easily get overwhelmed by the increasing number of blocks in the planning problem.

This case shows that even though the HTN planning domain uses heuristics to decompose the plan, it might not be free from flaws in the domain description. The method `find-nomove` has the purpose to mark the blocks such that the planning algorithm does not use or move these blocks. However, this method causes the problem resulting from the marking process.

The question now is how to solve this anomaly. At least three possible solutions are described here. Firstly, change the heuristic used for the blocks world domain. However, this means that either the methods, operators or axioms will have to change in order to avoid this anomaly. Nevertheless, another problem might also arise with the new changes. Secondly, limit the number of solution plans generated by HTN planner. JSHOP2 has an option to decompose only one or n number of plans. In this way, the planning domain remains the same. However,

an optimal plan may not be produced. Thirdly, using the HDL system. This approach is shown in Section 6.3.2.1, where the risk of this anomaly can be reduced by filtering irrelevant blocks. However, this anomaly may remain if the initial state and the goal state are the same. Some new concepts to overcome this problem can be defined in the HDL system. For example, a concept that checks whether some objects are already in the desired positions so that they may be excluded from the planning problem. In this way, the planning domain can still remain the same. Of course, another problem-specific approach can always be defined. Especially for this anomaly, a naive solution would be to simply check the initial state and the goal state. If both are identical, then the plan would immediately return without any further decomposition.

7.2. Inconsistencies in the Model

The consistency of the HDL model is necessary in order to produce a better solution plan for the given problem. In the robotics domain, the environment may change from time to time. For example, at one point in time a door is closed and at another point it may open. In a home or office environment as well as in environments where other agents operate, the environment can be changed by humans or by other agents. Thus, the inconsistency problem may appear more often.

The main question is how to deal with inconsistency in the model. In HDL, there are two kinds of inconsistencies. The first one is an inconsistency in the HDL model and the second one is an inconsistency within the environment states as describes in the previous paragraph. The first inconsistency should be handled by the DL reasoner as it provides the capability to perform consistency checks on the model. If the model is inconsistent, the HDL system might not be able to produce any planning problem for the planners.

The second inconsistency depends on the time. For a given moment in time, all the states in the HDL model are valid. However, as the plan is generated and the robot starts to execute the actions of the plan some states might become invalid. This problem is a difficult one where it remains an open problem in our daily life. For example, we want to travel from one city to another using several connecting trains. Although the connections are generated according to the actual time tables, some events may transpire that could not be predicted in advance. As we are already at the train station waiting for one train, the train could be delayed for some reason, e.g. machine breakdown or doors jam. This event could make the travel plan invalid as we could not achieve the goal of boarding the connecting train.

This problem could be minimised if the environment where the robot works is equipped with an advanced sensory system that are connected to the HDL system. Any changes on the environment would be reflected in the update of the HDL model. Thus, it minimises the inconsistencies in our model. Of course, the generated plan might still become invalid because it was generated at time t where some states might change during plan execution. This situation is unavoidable. However, due to the filtering capability of the HDL system, a new planning prob-

lem could be generated with the more current states. Thus, an update of the solution plan could replace the previous invalid plan. As the filtering process could limit the number of the states, the planner will be able to compute the new plan in reasonable time.

The robot can also collect information from its own sensors. However, this sensory information usually contains noise, that can result in misinterpretation of the readings. One well-developed approach, used to model the uncertainty of sensory input in the robotics domain, is the probabilistic approach. Each reading in the model is accompanied by a probabilistic value. Thus providing a level of certainty for each piece of information. The challenge is how to combine this model of uncertainty with the DL model as DL reasoning is based on logic. A work on extending Markov Logic to model a probabilistic distribution in relational domains is presented in [JKB07]. This shows ongoing research towards combining logic reasoning and the probabilistic approach.

Another approach to avoid this situation is by limiting the size of the planning problem. As discussed in Section 3.4, two pick-and-place domains with different granularity at the level of the plan actions could produce different plans for the same objectives. The partial pick-and-place domain plans the actions at a higher level of abstraction than the complete pick-and-place one. It neglects the real navigation of the robot and assumes that the robot can go to the place where the object is located. Therefore, the generated plan gives a hint of the next step. As the robot starts executing a navigate operator, it will ask the HDL system to generate plans for navigating itself to the destination location. With this approach, the inconsistency is minimised by expanding the plan as needed. Section 7.4 describes another possibility of handling this problem in a plan-based-control approach for robotics.

7.3. Defining Usable Objects

Let us recall the pick-and-place domain in Section 3.4. The manipulable object could be an edible object. Thus, the object could be used and subsequently disappear from the environment. This could lead to the inconsistency problem, mentioned above. However, the question arises of how this kind of objects should be modelled in the HDL. One possible way to avoid inconsistencies stemming from usable objects, is by not modelling these objects in the HDL system but rather by instantiating them during the plan's execution.

JSHOP2 can have a method or operator that calls outside functions through "call term" [Ilg06]. Thus, a new operator "`find-object(object)`" could be defined in the HDL. This method contains a call to the real search function on the robot. In the planning extraction process it will instantiate the searched object in the planning states. In the execution process, it will return true if the object is found and false otherwise. Therefore, the robot will know whether the object is there or not.

However, this function does not explicitly mention where the object should be searched for. In the pick-and-place domain, the concept of $Container$ is defined as the place holder of

some objects. This concept can be used for improving the search heuristic of the `find-object` operator by adding the container as its parameter. The operator call is defined as `find-object (object, container)`. Thus, the search space is limited heuristically by searching only certain places where the given objects are usually placed. E.g. apples or oranges are usually placed in a fruit-basket.

Having object and container as parameters of `find-object` provides us with the possibility to model the ontology in such a way that can benefit from these definitions. In addition, a more advanced search that involves more than one container can also be defined, either as an HTN planning method or operator, or as HDL concepts.

7.4. HDL and Plan-Based-Control Approaches for Robotics

Another planning approach for robotics is shown in [Bee02b], which is also called plan-based-control for robotics. In this approach, the planner, executor and robot control mechanism are tightly coupled into one coherent system. Thus, they share the same information between all components. One major advantage with this system is how the executer communicates with the planning part during the plan execution process. Any failure during this process can be handled with the same system or in other words the plan recovery heuristic can be built into the system.

The HDL system extends the deliberative layer on the multi-layered control architecture. Thus, it does not explicitly defined the executive component. Some efforts have to be taken in order to map the method and operators to the executive components. These mappings can also be done in the HDL models, where each of the methods and operators contain information for the executive layer. It tells the executive layer, which robot's command or function has to be called in order to execute the corresponding operators or methods. One might also add a robot's specific information in these mapping. For example, the `drive` command on robot A is performed by the function `move` and on robot B by the function `move-robot`. Hence, the generated plans are not bound to one specific robot but may be used for other robot types. The same efforts are also needed in order to recover from a plan execution failure, This is discussed in more detail in Section 7.5.

There is a trade-off between the HDL and plan-based-control approach. In the plan-based-control approach, the system is a coherent one that knows in advanced what are the methods or operators performing and how to recover from a plan execution failure. In the HDL system, these are not given by default but can be implemented for a specific executor or robot. Thus, additional efforts have to be invested for this functionality. However, one major advantage is that the HDL system is more generic. It can be used by any system where the generated plans can be mapped to that specific system. Besides that, the HDL system can also be integrated into the plan-based-control system as the highest deliberative layer. Hence, the system can benefit from both systems by having a robust robot control system from the plan-based-control architecture and an advanced deliberative layer from the HDL system.

7.5. Plan Recovery from Failures

The execution of a plan by a mobile robot is not an easy task. Sometimes failures can arise during execution. For example, while grasping an object, the robot might accidentally miss the object, or the object may accidentally be dropped. The question is how to handle this failure. Can this failure be modelled in the HDL system?

One might add information in some methods or operators where the probability of execution failure is quite high. For example, the method `grasp-object` has a high likelihood to fail. Therefore, one might put this information as one of the method's properties in addition to information on how many tries the robot should attempt before it should give up in case of failure. The recovery procedure might also be programmed in the method's property. This can be done in a similar way to programming techniques by having "try" and "catch" clauses. With these clauses, the programmer adds some code to be executed in the case that some known exception occurs. Similarly, one might provide exception handling for the `grasp-object` method. E.g. if the object slips, then the robot should move the arm to its home pose and then retry the object detection procedure to test whether the object still graspable. This may be done in advance to handle other such failure events.

An executive language like PLEXIL [BDE+07, DMP07] can have exception-handling code for its method or operator descriptions. Thus, one might combine the information from the solution plan and the exception-handling procedures in the HDL system into the PLEXIL language. This shows that although plan recovery is not originally supported by HTN planning, one might use an executive language like PLEXIL and add the exception handling in some methods or operators in the HDL system.

7.6. Operator Cost and Plan Optimisation

The JSHOP2 planner can return a number of possible solution plans. For each plan the cost of the actions are shown. This cost function can be used to find the shortest plan. In the examples shown in this work, the cost of each operator is not defined. Hence, the JSHOP2 planner assigns the default value "1" for each operator. JSHOP2 does give the possibility to define the cost of each operator explicitly.

In the HDL system, one can also use the cost value for the operators. This value can help to optimise the generated plans. For example, in the partial pick-and-place domain the operator `navigate-op` can benefit from the cost value. As the generated plans for the partial pick-and-place domain currently assume that the `navigate-op` action costs only one, although it can in-fact take several steps on the complete pick-and-place domain. So, changing the cost value with a fixed number can provide a better result on the plan cost. Additional efforts to compute the cost value dynamically might also increase the precision of the plan cost result. For this problem, the cost value depends on the distance of the current robot position and the destination position.

Another benefit of having cost functions is shown in the following example. Assume the robot has a basket that can have two or more objects in it. Thus, the different cost values for each operator produce different optimisation of the solution plans. If the goal is to bring some objects (a,b,c) that are located in an initial place to a destination place, the planner can determine that a result that grasped two objects and put them in a robot's basket before moving to the destination place has less cost than separately grasping one object and moving to the destination.

7.7. HDL Versus Other DL-Planning Approaches

In other approaches presented in [Gil05], the planning system is integrated with the DL system. What are the trade-offs between those approaches and the HDL system? Why does the HDL approach separate the planner and the DL reasoner? There are several reasons for the design of the HDL system and trade-offs between the HDL approach and other DL-planning approaches.

The first reason is performance. Moving or defining the whole planning system into the DL reasoning system could increase the time needed for inferring the model. As shown in Section 6.1, the complexity of the DL reasoning system is better than the HTN planner's which is semi-intractable (see Table 6.1). Thus, integrating the planning system with the DL reasoning system will increase the reasoning complexity of the overall systems. As the main purpose of DL is to deduce facts based on its model, and the planning system's purpose is to search for the actions which lead to the goal state. Having both approaches in one system means that the reasoning system has to deduce the current facts and also expand those facts for plan search purposes. As a result, the overall system will no longer be tractable. In the HDL system, there are two reasoning parts that work independently and separately. The DL reasoning system is responsible for generating the current states for the planning problem. The HTN planner then decomposes the planning problem into solution plans. Thus, the overall worst case complexity is not worse than the HTN planning system's. It has already been shown in the example that choosing the right concept in the DL model for the planning problem can reduce planning complexity.

The performance reason is also a reason why the planning domain is modelled down to the methods and operators only. The DL reasoners would also be able to deduce the atomic literals of the planning operators and then reason the planning taxonomies. However, this approach will increase the reasoning complexity for the DL reasoner while gaining fewer advantages. With our approach, the generated planning domains are valid and the initial states can also be reasoned based on these domains. Besides that, modelling the atomic literals of a planning operator will bind the planning domain to a specific kind of planning system. With the current approach of the HDL system, one might use another planning system by changing the actual planning code in the model.

The second reason is compatibility and extensibility. Although the surveyed works in [Gil05] have used DL, they have not been incorporated in state-of-the-art planning algorithms. This shows that integrating the most recent planning algorithms in the DL reasoning system

is not a trivial one. As previously mentioned, it might cause the complexity of the system to become intractable. In the HDL approach, the separation between DL reasoning and HTN planning gives the flexibility to incorporate any other planning systems (in this case HTN planning). The HDL system produces a planning problem in planner specific syntax, which is defined in the methods or operators property. If another planning system which uses different syntax than SHOP2 is used, one can add properties for supporting the new system. Therefore, regardless the planning system the DL model is still the same and will produce a valid planning problem for the new planners. Similar to the DL reasoners, HDL can use several reasoning systems as mentioned in Section 2.2.2.

8. Conclusions

This chapter concludes this work and provides a view into possible future works.

8.1. Summary

In this work, a novel approach for integrating DL reasoning and HTN planning is presented. The Hybrid Deliberative Layer (HDL) solves the problem that an intelligent agent faces in dealing with a large amount of information which may or may not be useful in generating a plan to achieve a goal. The information, that an agent may need, is acquired and stored in the DL model. Thus, the HDL is used as the main knowledge base system for the agent.

HTN planning is modelled as concepts in DL terminologies (details in Section 2.4.2). The HDL system uses the knowledge to generate the planning domain automatically for the given problem. The DL reasoner reasons about the consistency of the planning domain. Thus, the generated planning problem is valid and filled with the actual information from the DL model.

Algorithms, developed by the author and presented in Section 2.5.1, for generating the HTN planning problems are applied to the inferred instances of the *ABox*. Thus, only the relevant instances for the given planning problem are included as planning facts. Advanced concepts of the environment can also be modelled for filtering the planning problem further. These concepts reduce the size of the planning problem because these exclude irrelevant instances.

In this work, the HDL system is designed and implemented using an off-the-shelf HTN planner, namely JSHOP2, as its planning system and *Pellet* as its DL reasoner. Other reasoning systems and planning systems can also be incorporated within the HDL system. The HTN planner and DL reasoner are fused into a coherent system, such that the overall system gains from the benefits of both systems.

Some robotics domains are modelled and implemented using the HDL system (see Chapter 3). These domains are "navigation domain" and "pick-and-place domain". Two possible implementations for the pick-and-place domain are presented, namely the partial and complete pick-and-place domains. These domains show that the HDL system can have additional hierarchy on its model thus providing additional abstraction levels than the HTN planning system. A case study of "Johnny Jackanapes", a mobile manipulator, for solving one task in the RoboCup@Home competition is also discussed in Chapter 4. As the HDL system is useable not only in the robotics domain, but also in the AI domain, a blocks world domain is modelled and solved using the HDL system (Chapter 5). Several different blocks world problems are discussed to highlight the advantages of the HDL system.

Two case studies are presented to benchmark the complexity of the HDL system and the HTN planning system in Section 6.3. These studies are the navigation domain with an increasing number of rooms and the blocks world domain with an increasing number of blocks. The HDL system generates a smaller planning problem than the pure HTN approach, due to the limitation on its expressivity. In the HDL approach the additional rooms or blocks are modelled in DL but these are excluded from the planning problem. The complexity of the HDL system is analysed and discussed in Chapter 6. The overall complexity will not be worse than that of the HTN planning system as it is the most computationally complex component in the HDL system.

8.2. Strengths and Limitations

The strengths of the HDL system are its expressiveness, compatibility and expandability. It enables the robotic system to have a deliberative layer that can model knowledge of a rich environment. In addition, it can still have state-of-the-art planning systems as its planner. Thus, it can improve its overall performance. The HDL system is compatible with the other DL reasoning systems and HTN planning systems. One can change these components with the others if necessary. This also improves the expandability of the HDL system, where multiple reasoners or planners can be used for solving the same problems.

The limitations of the HDL system is the worst case complexity it inherits from the most computationally complex component, in this implementation, the HTN planner. However, it can limit the complexity by defining advanced concepts in DL such that fewer instances are involved in the planning problem. Besides that, the HDL system generates sequences of actions symbolically. Thus, it serves only as a hint for the robots. In order to execute these actions, an executive layer has to translate the actions into robot commands and monitor their execution.

8.3. Future Work

The HDL system can be further enhanced. In this section, some methods to do just that are presented.

8.3.1. Affordance-Based Planning

In the case study "Johnny Jackanapes", the concepts using affordance are shown as examples. For example, an object with a small fingerprint and less than $500\,g$ is a graspable and lift-able object, thus it can be manipulated by the robot. Affordance depends on the actor or robot, so an object might be graspable for one manipulator but not for another, e.g. a robot which can lift metallic objects, because it uses magnet to lift up the object would not be able to grasp an object made only from plastic.

Although in the case study, affordance-based planning is shown, it still needs further research. How the affordance is modelled in DL is an example of questions to be answered by

such research. In an EU funded project, MACS [RPS+08], affordance was the main research topic. They tried to figure out how the robot can learn the affordances and use them to plan further actions. Another aspect is how the affordance is usable by the other robots. For example, in the case study some objects are defined as "drinkable", although the robot cannot drink but it can use the "drinkable" (affordance for human being) to filter out irrelevant objects. The task was to bring some drink to the guest, thus, it should deduce that "some drink" means instances of $ManipulableObject$ which has the "drinkable" as its affordance property.

8.3.2. Using Other HTN Planning Implementations

In the implementation presented here, the HTN planner is JSHOP2. There are some other HTN planners available. Thus, implementations which use other HTN planners could be explored as future work. In order to support other HTN planners, an additional property can be added to the methods or operators' instances. This property contains the code for the given methods or operators in the new planner syntax.

Similarly, a planning system other than HTN planning can be implemented, either by defining a new planning concept in the DL model or translating the current HTN planning into another planning syntax. For example, an approach to translate STRIPS planning to HTN is presented in [EHN94b].

8.3.3. Collocate the Planner Using Web-Service

OWL-DL was developed for semantic web applications. Thus, it supports the standard syntax that is used by the world wide web protocol. The model written in this form can be transferred through the HTTP protocol without any problem. Thus, the HDL system can be implemented on a dedicated machine that has more than enough resources for serving more than one client. Thus, multiple robots can work with the same model and share information with the others using a centralised HDL system as some robots might have limited resources for high computational load.

In [HH05], the author presented a planner web-services. The system runs several planning systems on its machine and receives requests from the clients. The clients' requests are not only for one particular planning system but can be for multiple planning systems that are offered by the service. This service can also be integrated with the HDL system to improve the scalability of the system.

8.3.4. Application in Plan-Based Robot Control

The main motivation for this work is the mobile manipulation domain. In Chapters 3 and 4, the robotics domains are presented. The HDL system can use the information from the environment and generate solution plans for these domains. However, there is still some work to be done in integrating the results with the real robots. Recall Figure 2.2, the HDL system it the light

green box on the top right corner. There are two main connections to the robot in order to have the system integrated. These are the sensor to ontology anchoring and the scheduler/executor module.

Automatic insertion of the perception data of a robot into the KB has to be addressed. This ensures that the knowledge within the KB remains current and up to date. The sensory information has to be processed and asserted into the HDL model. This is not a trivial task, because assertion means to insert instances of objects in the real world into the model. Two main problems are faced here; firstly, how to capture the model of the environment to the conceptual model in DL *TBox*, and secondly, how to group or recognise an object and ground it to the model in the DL *ABox*.

The sequencing layer which controls plan execution has to be implemented. The sequencing layer must not only coordinate the execution of actions within a plan, but it must handle exceptions and failures encountered during the plan's execution. In the event that a failure occurs, a heuristic such as trying to execute the next best plan or re-planning may be used. In addition, the scheduler might also be needed for controlling mobile manipulator because it has several resources that can run in parallel. For example, the robot can move its arm while it moves with its locomotion drive. Therefore, a scheduler has to make sure that simply moving the arm will not hit any obstacle. This can be seen as coordination problem.

8.3.5. Using DL Inference Engine for Plan Repair

The HDL system uses the DL inference engine to reason about the plan taxonomies in order to generate smaller planning problems. However, the DL inference engine can be used for other purposes in order to improve the HDL system. One of the important aspects is plan repair in case of any failure during execution. In Section 7.5, another approach for handling plan failure is proposed by implementing the exception handling during plan execution. However, this approach might not be successfully handled by the system if the current action is not repairable. For example, the robot is supposed to grasp a cola can, but it accidentally misses the object and makes the object fall down. In the case a simple gripper, like that of our robot Johnny Jackanapes, the grasp action cannot be performed due to the displacement of the can. After several tries, the execution layer will give up and ask for a new plan. However, in such a technical failure, the planner might come up with other plans which might still contain the action to grasp the same cola can.

We can use the DL inference engine for reasoning about the model in order to repair plan failure. In some cases, the methods and operators can be modelled redundantly. Thus, it is possible to achieve the same task using different methods or operators. Having this information modelled in the DL, the HDL system can then use its reasoner to generate new plans with alternative methods or operators.

Bibliography

[AGPC04] Mariano Fernández-López Asunción Gómez-Pérez and Oscar Corcho. *Ontological Engineering: With Examples from the Areas of Knowledge Management, e-Commerce and the Semantic Web.* Advanced Information and Knowledge Processing. Springer Berlin Heidelberg, 2004.

[Ark87] Ronald C. Arkin. Motor Schema Based Navigation for a Mobile Robot: An Approach to Programming by Behavior. *Robotics and Automation. Proceedings. 1987 IEEE International Conference on*, 4:264–271, March 1987.

[Ark98] Ronald C. Arkin. *Behavior-Based Robotics*. MIT Press, Cambridge, MA, USA, 1998.

[AS89] Minoru Asada and Yoshiaki Shirai. Building a World Model for a Mobile Robot Using Dynamic Semantic Constraints. In *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 1629–1634, 1989.

[ATS03] Kai O. Arras, Nicola Tomatis, and Roland Siegwart. Robox, a Remarkable Mobile Robot for the Real World. *Experimental Robotics VIII*, 5:178–187, 2003.

[BAB+01] Michael Beetz, Tom Arbuckle, Thorsten Belker, Armin B. Cremers, Dirk Schulz, Maren Bennewitz, Wolfram Burgard, Dirk Hähnel, Dieter Fox, and Henrik Grosskreutz. Integrated Plan-Based Control of Autonomous Robots in Human Environments. *IEEE Intelligent Systems*, 16(5):56–65, 2001.

[BCF+99] Wolfram Burgard, Armin B. Cremers, Dieter Fox, Dirk Hähnel, Gerhard Lakemeyer, Dirk Schulz, Walter Steiner, and Sebastian Thrun. Experiences with an Interactive Museum Tour-Guide Robot. *Artificial Intelligence*, 114(1-2):3–55, 1999.

[BCM+03] Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider. *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press, 2003.

[BDE+07] Vijay Baskaran, Michael Dalal, Tara Estlin, Chuck Fry, Robert Harris, Michael Iatauro, Ari Jónsson, Corina Pasareanu, Reid Simmons, and Vandi Verma. Plan Execution Interchance Language (PLEXIL) Version 1.0. Nasa technical memorandum, NASA, Nov 2007.

[Bee02a] Michael Beetz. *Plan-Based Control of Robotic Agents: Improving the Capabilities of Autonomous Robots*, volume 2554 of *Lecture Notes in Computer Science*. Springer, 2002.

[Bee02b] Michael Beetz. Plan Representation for Robotic Agents. In *Proceedings of the AI Planning and Scheduling (AIPS)*, pages 223–232, 2002.

[BFL83] Ronald J. Brachman, Richard E. Fikes, and Hector J. Levesque. Krypton: A Functional Approach to Knowledge Representation. *Computer*, 16(10):67–73, Oct. 1983.

[BFT95] Paolo Bresciani, Enrico Franconi, and Sergio Tessaris. Implementing and Testing Expressive Description Logics: Preliminary Report. In *Proceedings of the First International Knowledge Retrieval, Use and Storage for Efficiency (KRUSE) Symposium*, pages 131–139, 1995.

[BH91] Franz Baader and Bernhard Hollunder. KRIS: Knowledge Representation and Inference System. *ACM SIGART Bulletin*, 2(3):8–14, 1991.

[BHS08] Franz Baader, Ian Horrocks, and Ulrike Sattler. Description Logics. *Handbook of Knowledge Representation*, 2008.

[BKRG01] Jim Blythe, Jihie Kim, Surya Ramachandran, and Yolanda Gil. An Integrated Environment for Knowledge Acquisition. In *Proceedings of the 6th International Conference on Intelligent User Interfaces*, pages 13–20, New York, NY, USA, 2001. ACM.

[BL04] Ronald J. Brachman and Hector J. Levesque. *Knowledge Representation and Reasoning*. Elsevier, 2004.

[BMC03] Sean Bechhofer, Ralf Möller, and Peter Crowther. The DIG Description Logic Interface (2003). In *Proceedings of the International Workshop on Description Logics*, 2003.

[BMM+07] Patrick Beeson, Matt MacMahon, Joseph Modayil, Aniket Murarka, Benjamin Kuipers, and Brian Stankiewicz. Integrating Multiple Representations of Spatial Knowledge for Mapping, Navigation, and Communication. In *Proceedings of the Interaction Challenges for Intelligent Assistants*, pages 1–9, 2007.

[BMPS+91] Ronald J. Brachman, Deborah L. McGuinness, Peter F. Patel-Schneider, Lori A. Resnick, Lori Alperin Resnick, and Alexander Borgida. Living with CLASSIC: When and How to Use a KL-ONE-Like Language. *Principles of Semantic Networks*, pages 401–456, 1991.

[BN03] Franz Baader and Werner Nutt. Basic Description Logics. *The Description Logic Handbook: Theory, implementation, and applications*, 2003.

[Bro86] Rodney A. Brooks. A Robust Layered Control System for a Mobile Robot. *Robotics and Automation, IEEE Journal of [legacy, pre - 1988]*, 2(1):14–23, Mar 1986.

[BS85] Ronald J. Brachman and James G. Schmolze. An Overview of The KL-ONE Knowledge Representation System. *Cognitive Science*, 1985.

[CFL02] Stephen Cresswell, Maria Fox, and Derek Long. Extending TIM Domain Analysis to Handle ADL Constructs. In *Proceedings of the Knowledge Engineering Tools and Techniques for AI Planning: AIPS'02 Workshop*, 2002.

[CGK+02] Hans Chalupsky, Yolanda Gil, Craig A. Knoblock, Kristina Lerman, Jean Oh, David V. Pynadath, Thomas A. Russ, and Milind Tambe. Electric Elves: Agent Technology for Supporting Human Organizations. *AI Magazine*, 2002.

[Chr] Binildas Christudas. Internals of Java Class Loading. http://www.onjava.com/pub/a/onjava/2005/01/26/classloading.html.

[Cra05] John J. Craig. *Introduction to Robotics: Mechanics and Control*. Pearson Education International, 2005.

[DBSB91] Premkumar Devanbu, Ronald J. Brachman, Peter G. Selfridge, and Bruce W. Ballard. LaSSIE: a Knowledge-based Software Information System. *Communications of the ACM*, 34(5):34–49, 1991.

[Den87] Daniel C Dennett. *The Intentional Stance*, chapter 3 (Three Kinds of Intentional Psychology), pages 43–82. MIT Press, 1987.

[DGINR96] Giuseppe De Giacomo, Luca Iocchi, Daniele Nardi, and Riccardo Rosati. Moving a Robot: the KR&R Approach at Work. In *Proceedings of the Fifth International Conference on the Principles of Knowledge Representation and Reasoning (KR'96)*, pages 198–209, 1996.

[Dic04] Ian Dickinson. Implementation experience with the DIG 1.1 specification. Technical Report HPL-2004-85, HP Labs, 2004.

[DL96] Premkumar T. Devanbu and Diane J. Litman. Taxonomic plan reasoning. *Artificial Intelligence*, 84(1-2):1–35, 1996.

[DMP07] Gilles Dowek, César Muñoz, and Corina Pasareanu. Formal Semantics of a Synchronous Plan Execution Language. In *Proceedings of the Workshop on*

Planning and Plan Execution for Real-World Systems: Principles and Practices for Planning in Execution at the International Conference on Automated Planning and Scheduling (ICAPS), 2007.

[DSB+04] Mike Dean, Guus Schreiber, Sean Bechhofer, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. OWL Web Ontology Language Reference. W3C Recommendation. http://www.w3.org/TR/owl-ref/, 2004.

[EHN94a] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN Planning: Complexity and Expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 2, pages 1123–1128, Menlo Park, CA, USA, 1994. AAAI Press.

[EHN94b] Kutluhan Erol, James Hendler, and Dana S. Nau. Semantics for Hierarchical Task-Network Planning. Technical report, UMIACS University of Maryland, 1994.

[Fir87] R. James Firby. An Investigation into Reactive Planning in Complex Domains. In *Proceedings of the Sixth National Conference on Artificial Intelligence*, pages 202–206, 1987.

[Fir94] R. James Firby. Task Networks for Controlling Continuous Processes. In *Proceedings of the Artificial Intelligence Planning Systems (AIPS)*, pages 49–54, 1994.

[FL98] Maria Fox and Derek Long. The Automatic Inference of State Invariants in TIM. *Journal of Artificial Intelligence Research*, 9:367 – 421, 1998.

[GFMGS07] C. Galindo, J.-A. Fernández-Madrigal, J. González, and A. Saffiotti. Using Semantic Information for Improving Efficiency of Robot Task Planning. In *Proceedings of the ICRA-07 Workshop on Semantic Information in Robotics*, pages 27 – 32, 2007.

[GG97] R. Grimes and Dr R. Grimes. *Professional DCOM Programming*. Wrox Press Ltd., Birmingham, UK, UK, 1997.

[GGFM04] Cipriano Galindo, Javier Gonzalez, and Juan-Antonio Fernandez-Madrigal. Interactive Task Planning through Multiple Abstraction: Application to Assistant Robotics. In *Proceedings of the European Conference on AI (ECAI)*, pages 1015–1016, 2004.

[Gil94] Yolanda Gil. Knowledge Refinement in a Reflective Architecture. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, volume 1,

pages 520–526, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[Gil05] Yolanda Gil. Description Logics and Planning. *AI Magazine*, 26(2):73–84, 2005.

[GM96] Yolanda Gil and Eric Melz. Explicit Representations of Problem-Solving Strategies to Support Knowledge Acquisition. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 469–476. AAAI Press, 1996.

[GN92] Naresh Gupta and Dana S. Nau. On the Complexity of Blocks-World Planning. *Artificial Intelligence*, 56(2-3):223–254, 1992.

[GNT04] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[GR01] Yolanda Gil and Surya Ramachandran. PHOSPHORUS: a Task-based Agent Matchmaker. In *Proceedings of the Fifth International Conference on Autonomous Agents*, pages 110–111, New York, NY, USA, 2001. ACM.

[Gro04] Object Management Group. *Common Object Request Broker Architecture (CORBA/IIOP) Version 3.03*. The Object Management Group, March 2004.

[GS05] Adam Galuszka and Andrzej Swierniak. Non–Cooperative Game Approach To Multi-Robot Planning. *International Journal of Applied Mathematics and Computer Science*, 15(3):359–367, 2005.

[GS07] Michael A. Goodrich and Alan C. Schultz. Human-Robot Interaction: A Survey. *Foundations and Trends in Human-Computer Interaction*, 1:203–275, 2007.

[GSC+05] C. Galindo, A. Saffiotti, S. Coradeschi, P. Buschka, J. A. Fernandez-Madrigal, and J. Gonzalez. Multi-hierarchical Semantic Maps for Mobile Robotics. In *Proceedings of the Intelligent Robots and Systems (IROS)*, pages 2278–2283, 2005.

[HBN07] Matthew Horridge, Sean Bechhofer, and Olaf Noppens. Igniting the OWL 1.1 Touch Paper: The OWL API. In *Proceedings of the 3rd OWL Experienced and Directions Workshop (OWLED)*, Inssbruck, Austria, June 2007.

[HC08] Joachim Hertzberg and Raja Chatila. AI Reasoning Methods for Robotics. *Springer Handbook of Robotics*, pages 207–223, 2008.

[HD02] Adele E Howe and Eric Dahlman. A Critical Assessment of Benchmark Comparison in Planning. *Journal of Artificial Intelligence Research*, pages 1–33, 2002.

[Hen04] Michi Henning. A New Approach to Object-Oriented Middleware. *IEEE Internet Computing*, 8(1):66–75, 2004.

[HH05] Ronny Hartanto and Joachim Hertzberg. Offering Existing AI Planners as Web Services. In *Proceedings of the 19th Workshop "Planen, Scheduling und Konfigurieren, Entwerfen"*, 2005.

[HH07] Ronny Hartanto and Joachim Hertzberg. Augmenting JSHOP2 Planning with OWL-DL. In *Proceedings of the 21st Workshop "Planen, Scheduling und Konfigurieren, Entwerfen"*, 2007.

[HH08] Ronny Hartanto and Joachim Hertzberg. Fusing DL Reasoning with HTN Planning. *KI 2008: Advances in Artificial Intelligence*, pages 62–69, 2008.

[HH09] Ronny Hartanto and Joachim Hertzberg. On the Benefit of Fusing DL-Reasoning with HTN-Planning. *KI 2009: Advances in Artificial Intelligence*, pages 41–48, 2009.

[HKNjP94] Jochen Heinsohn, Daniel Kudenko, Bernhard Nebel, and Hans jurgen Profitlich. An Empirical Analysis of Terminological Representation Systems. *Artificial Intelligence*, 68:767–773, 1994.

[HKR+04] Matthew Horridge, Holger Knublauch, Alan Rector, Robert Stevens, and Chris Wroe. *A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools Edition 1.0*. The University of Manchester and Stanford University, 2004.

[HKR09] Dirk Holz, Gerhard K. Kraetzschmar, and Erich Rome. Robust and Computationally Efficient Navigation in Domestic Environments. In *Proceedings of the RoboCup 2009 Symposium*, 2009.

[HM99] Volker Haarslev and Ralf Möller. RACE System Description. In *Proceedings of the International Workshop on Description Logics*, pages 130–132, 1999.

[HM01] Volker Haarslev and Ralf Möller. RACER System Description. In *Proceedings of the First International Joint Conference on Automated Reasoning (IJCAR '01)*, pages 701–706, London, UK, 2001. Springer-Verlag.

[HM03] Volker Haarslev and Ralf Möller. Racer: An OWL Reasoning Agent for the Semantic Web. In *Proceedings of the International Workshop on Applications, Products and Services of Web-based Support Systems*, pages 91–95, 2003.

[Hor98] Ian R. Horrocks. Using an Expressive Description Logic: FaCT or Fiction. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 636–647, 1998.

[HPB+08] Dirk Holz, Jan Paulus, Thomas Breuer, Geovanny Giorgana, Ronny Hartanto, Walter Nowak, Johnny Jackanapes, Paul Ploeger, and Gerhard Kraetzschmar. The b-it-bots RoboCup@Home 2008 Team Description Paper. Technical report, Bonn-Rhein-Sieg University of Applied Sciences, 2008.

[HPB+09] Dirk Holz, Jan Paulus, Thomas Breuer, Geovanny Giorgana, Michael Reckhaus, Frederik Hegger, Christian Müller, Zha Jin, Ronny Hartanto, Paul Ploeger, and Gerhard Kraetzschmar. The b-it-bots RoboCup@Home 2009 Team Description Paper. Technical report, Bonn-Rhein-Sieg University of Applied Sciences, 2009.

[HPS99] Ian Horrocks and Peter F. Patel-Schneider. Optimizing Description Logic Subsumption. *Journal of Logic and Computation*, 9(3):267–293, 1999.

[HS07] Michi Henning and Mark Spruiell. *Distributed Programming with ICE*. ZeroC Inc., revision 3.2 edition, March 2007.

[HSMH04] Ronny Hartanto, Frank Schönherr, Michael Mock, and Joachim Hertzberg. Target-oriented mobile robot behaviors for office navigation tasks. In *Proceedings of the Second IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*, pages 104–108, 2004.

[Ilg06] Okhtay Ilghami. Documentation for JSHOP2. Technical report, Department of Computer Science University of Maryland, 2006.

[JC97] Herbert Jaeger and Thomas Christaller. Dual Dynamics: Designing Behavior Systems for Autonomous Robots. *Artificial Life and Robotics*, 1997.

[JKB07] Dominik Jain, Bernhard Kirchlechner, and Michael Beetz. Extending Markov Logic to Model Probability Distributions in Relational Domains. In *Proceedings of the 30th annual German conference on Advances in Artificial Intelligence*, pages 129–143, Berlin, Heidelberg, 2007. Springer-Verlag.

[KAK+95] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The Robot World Cup Initiative. In *Proceedings of the IJCAI-95 Workshop on Entertainment and AI/Alife*, Montreal, 1995.

[KAK+97] Hiroaki Kitano, Minoru Asada, Yasuo Kuniyoshi, Itsuki Noda, and Eiichi Osawa. RoboCup: The Robot World Cup Initiative. In *Proceedings of the First International Conference on Autonomous Agent (Agents-97)*. The ACM Press, 1997.

[KBG+00] Benjamin Kuipers, Rob Browning, Bill Gribble, Mike Hewett, and Emilio Remolina. The Spatial Semantic Hierarchy. *Artificial Intelligence*, 119:191–233, 2000.

[KBR86] Thomas S. Kaczmarek, Raymond Bates, and Gabriel Robins. Recent Developments in NIKL. In *Proceedings the Fifth National Conference on Artificial Intelligence (AAAI)*, pages 978–985, 1986.

[Kha86] Oussama Khatib. Real-Time Obstacle Avoidance for Manipulators and Mobile Robots. *The International Journal of Robotics Research*, 5(1):90–98, 1986.

[KMRS97] Kurt Konolige, Karen Myers, Enrique Ruspini, and Alessandro Saffiotti. The Saphira Architecture: a Design for Autonomy. *Journal of Experimental and Theoretical Artificial Intelligence*, 9(2-3):215–235, 1997. Online at http://www.aass.oru.se/˜asaffio/.

[KS08] David Kortenkamp and Reid Simmons. Robotic Systems Architectures and Programming. *Springer Handbook of Robotics*, pages 187–206, 2008.

[Lat91] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[LaV06] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, New York, NY, USA, 2006.

[LF99] Derek Long and Maria Fox. Efficient Implementation of the Plan Graph in STAN. *Journal of Artificial Intelligence Research*, 10:87–115, 1999.

[LMP08] Vladimir Lifschitz, Leora Morgenstern, and David Plaisted. Knowledge Representation and Classical Logic. *Handbook of Knowledge Representation*, 2008.

[Low04] David G. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60:91–110, 2004.

[Mac91] Robert M. Macgregor. Using a Description Classifier to Enhance Deductive Inference. In *Proceedings of the Seventh IEEE Conference on AI Applications*, pages 141–147, 1991.

[MB87] Robert MacGregor and Raymond Bates. The LOOM Knowledge Representation Language. In *Proceedings of the Knowledge-Based Systems Workshop*, 1987.

[McB01] Brian McBride. Jena: Implementing the RDF Model and Syntax Specification. In *Proceedings of the Semantic Web Workshop, WWW2001*, 2001.

[McD91] Drew McDermott. A Reactive Plan Language. Technical Report YALE/DCS/TR864, Department of Computer Science Yale University, 1991.

[MDW91] Eric Mays, Robert Dionne, and Robert Weida. K-Rep System Overview. *ACM SIGART Bulletin*, 2(3):93–97, 1991.

[Mur00] Robin Murphy. *Introduction to AI Robotics*. A Bradford Book, MIT Press, Cambridge, New York, NY, USA, 2000.

[MvH04] Deborah L. McGuinness and Frank van Harmelen. OWL Web Ontology Language Overview. W3C website, 2004.

[Mya08] Adam Myatt. *Pro Netbeans IDE 6 Rich Client Platform Edition*. APress, 2008.

[NCLMA99] Dana S. Nau, Yue Cao, Amnon Lotem, and Héctor Muñoz-Avila. SHOP: Simple Hierarchical Ordered Planner. In Thomas Dean, editor, *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 968–975. Morgan Kaufmann, 1999.

[Neb90] Bernhard Nebel. Terminological Reasoning is Inherently Intractable. *Artificial Intelligence*, 43:235–249, 1990.

[NIK+03] Dana Nau, Okhtay Ilghami, Ugur Kuter, J. William Murdock, Dan Wu, and Fusun Yaman. SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research*, 20:379–404, 2003.

[Nil84] Nils J. Nilsson. Shakey The Robot. Technical Note 323, SRI International, 1984.

[NM01] Natalya F. Noy and Deborah L. McGuinness. Ontology Development 101 : A Guide to Creating Your First Ontology. Technical report, Stanford Knowledge Systems Laboratory, 2001.

[NMAC+01] Dana S. Nau, Héctor Muñoz-Avila, Yue Cao, Amnon Lotem, and Steven Mitchell. Total-Order Planning with Partially Ordered Subtasks. In Bernhard Nebel, editor, *Proceedings of the International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 425–430. Morgan Kaufmann, 2001.

[NSE98] Dana S. Nau, Stephen J. J. Smith, and Kutluhan Erol. Control Strategies in HTN Planning: Theory Versus Practice. In *Proceedings of AAAI-98/IAAI-98*, pages 1127–1133. AAAI Press, 1998.

[NWL+05] Andreas Nüchter, Oliver Wulf, Kai Lingemann, Joachim Hertzberg, Bernardo Wagner, and Hartmut Surmann. 3D Mapping with Semantic Knowledge. *RoboCup 2005: Robot Soccer World Cup IX*, pages 335–346, 2005.

[OMMJZ+07] Óscar Martínez Mozos, Patric Jensfelt, Hendrik Zender, Geert-Jan M. Kruijff, and Wolfram Burgard. From Labels to Semantics: An Integrated System for Conceptual Spatial Representations of Indoor Environments for Mobile Robots. In *Proceedings of the ICRA-07 Workshop on Semantic Information in Robotics (SIR)*, pages 33–40, Rome, Italy, April 2007.

[Ore04] Anders Orebäck. *A Component Framework for Autonomous Mobile Robots*. PhD thesis, KTH, Sweden, 2004.

[PHK04] Sang-Hyun Park, Jung-Hoon Hwang, and Dong-Soo Kwon. A New Human-Robot Interaction Method Using Semantic Symbols. In *Proceedings of the International Conference on Control, Automation and Systems*, pages 2005–2010, August 2004.

[Pow03] Shelley Powers. *Practical RDF: Solving Problems with the Resource Description Framework*. O'Reilly, 2003.

[PS07] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Candidate Recommendation. http://www.w3.org/TR/rdf-sparql-query/, 2007.

[PSKQ89] Christof Peltason, Albrecht Schmiedel, Carsten Kindermann, and Joachim Quantz. The BACK System Revisited. Technical report, Technische Universitaet Berlin, Berlin, Germany, Germany, 1989.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, Inc., second edition, 2003.

[RPS+08] E. Rome, L. Paletta, E. Sahin, G. Dorffner, J. Hertzberg, G. Fritz, J. Irran, F. Kintzler, C. Lörken, S. May, E. Ugur, and R. Breithaupt. *Towards Affordance-based Robot Control, Proceedings of Dagstuhl Seminar 06231*, chapter The MACS project: An approach to affordance-based robot control, pages 173–210. Springer-Verlag, 2008.

[SAB+03] Roland Siegwart, Kai O. Arras, Samir Bouabdhalla, Daniel Burnier, Gilles Froidevaux, Xavier Greppin, Björn Jensen, Antoine Lorotte, Laetitia Mayor, Mathieu Meisser, Roland Philippsen, Ralph Piguet, Guy Ramel, Gregoire Terrien, and Nicola Tomatis. Robox at Expo.02: A Large Scale Installation of Personal Robots. *Robotics and Autonomous Systems*, 42(3):203–222, 2003.

[SBC+08] Hartmut Surmann, Ansgar Bredenfeld, Thomas Christaller, Reiner Frings, Ulrike Petersen, and Thomas Wisspeintner. The Volksbot. In Emanuele Menegatti, editor, *Workshop Proceedings of International Conference on Simulation, Modelling and Programming for Autonomous Robots (SIMPAR)*, pages 551–561, November 2008. ISBN 978-88-95872-01-8.

[SG95] Bill Swartout and Yolanda Gil. EXPECT: Explicit Representations for Flexible Acquisition. In *Proceedings of the Ninth Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1995.

[SGH+97] Reid Simmons, R. Goodwin, K. Haigh, S. Koenig, Joseph O'Sullivan, and Maria Manuela Veloso. Xavier: Experience with a Layered Robot Architecture. In *Proceedings of the First International Conference on Autonomous Agent (Agents-97)*, 1997.

[SH02] Frank Schönherr and Joachim Hertzberg. The DD&P Robot Control Architecture. *Advances in Plan-Based Control of Robotic Agents*, pages 249–269, 2002.

[Sim94] Reid G. Simmons. Structured Control for Autonomous Robots. *IEEE Transactions on Robotics and Automation*, 10(1):34–43, 1994.

[Sow00] John F. Sowa. *Knowledge Representation – Logical, Philosophical, and Computational Foundations*. Thomson Learning, 2000.

[SPG+07] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A Practical OWL-DL Reasoner. *Journal of Web Semantics*, 5(2), 2007.

[TBB+00] S. Thrun, M. Beetz, M. Bennewitz, W. Burgard, A.B. Cremers, F. Dellaert, D. Fox, D. Hähnel, C. Rosenberg, N. Roy, J. Schulte, and D. Schulz. Probabilistic Algorithms and the Interactive Museum Tour-Guide Robot Minerva. *International Journal of Robotics Research*, 19(11):972–999, 2000.

[TBK+06] Anni-Yasmin Turhan, Sean Bechhofer, Alissa Kaplunova, Thorsten Liebig, Marko Luther, Ralf Möller, Olaf Noppens, Peter Patel-Schneider, Boontawee Suntrisrivaraporn, and Timo Weithöner. DIG 2.0 - Toward a Flexible Interface for Description Logic Reasoners. Technical report, TU Dresden and University of Ulm and University of Manchester and DoCoMo Euro-Labs and TU Hamburg-Harburg and Bell Labs Research, 2006.

[TH06] Dmitry Tsarkov and Ian Horrocks. FaCT++ Description Logic Reasoner: System Description. In *Proceedings of the International Joint Conference on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 292–297. Springer, 2006.

[VHH+05] Denny Vrandečić, Peter Haase, Pascal Hitzler, York Sure, and Rudi Studer. DLP - An Introduction. Technical report, Institute AIFB, Universität Karlsruhe, 2005.

[Wel99] Daniel S. Weld. Recent Advances in AI Planning. *AI Magazine*, 20(2):93–123, 1999.

[Win72] Terry Winograd. *Understanding Natural Language*. Academic Press, 1972.

[WNB06] Thomas Wisspeintner, Walter Nowak, and Ansgar Bredenfeld. VolksBot - A Flexible Component-Based Mobile Robot System. *RoboCup 2005: Robot Soccer World Cup IX*, Volume 4020/2006:716–723, 2006.

[ZJOMM+07] Hendrik Zender, Patric Jensfelt, Óscar Martínez Mozos, Geert-Jan M. Kruijff, and Wolfram Burgard. An Integrated Robotic System for Spatial Understanding and Situated Interaction in Indoor Environments. In *Proceedings of the Twenty-Second Conference on Artificial Intelligence (AAAI-07)*, pages 1584–1589, Vancouver, British Columbia, Canada, July 2007.

[ZOMMJ+08] Hendrik Zender, Óscar Martínez Mozos, Patric Jensfelt, Geert-Jan M. Kruijff, and Wolfram Burgard. Conceptual Spatial Representations for Indoor Mobile Robots. *Robotics and Autonomous Systems*, 56(6):493–502, June 2008.

Appendices

A. Generated Planning Domain

In this appendix the generated planning domain and problem descriptions in SHOP2 syntax are listed for the discussed problems. These outputs are generated automatically by the HDL system. In addition, the solution plans for selected problems are also shown.

A.1. Navigation Domain

The details of the navigation domain are presented in Section 3.2 on page 42. Two examples are presented in this section; the first example is from an instance of a *Planning-Domain* concept and the second example is from an instance of a *Method* concept.

A.1.1. An Example from the *Planning-Domain* Instance

Listing A.1 shows the generated planning domain description by choosing "`navigation-domain`" from the DL model. The `navigation-domain` is an instance of *Planning-Domain* concept.

Listing A.1: Planning domain description for "`navigation-domain`".

```
 1 ; JSHOP2 Planning Domain Description
   ; created on: Tue Dec 09 11:20:15 CET 2008
 3 ; generated from: http://mas.b-it-center.de/ontologies/planner....
     owl#

 5 (defdomain navigation_domain (
     ; http://mas.b-it-center.de/ontologies/planner.owl#drive-...
       robot
 7     (:operator (!drive-robot ?robot ?loc-from ?loc-to)
         ((at ?robot ?loc-from))
 9        ((at ?robot ?loc-from))
          ((at ?robot ?loc-to))
11     )
     ; http://mas.b-it-center.de/ontologies/planner.owl#unvisit
13     (:operator (!unvisit ?waypoint)
         ()
15       ((visited ?waypoint))
         ()
17     )
     ; http://mas.b-it-center.de/ontologies/planner.owl#visit
19     (:operator (!visit ?waypoint)
```

```
        ()
21      ()
        ((visited ?waypoint))
23    )

25  ; http://mas.b-it-center.de/ontologies/planner.owl#navigate
    (:method  (navigate ?robot ?from ?to)
27    Case1  ((at ?robot ?to))
      ()
29    Case2  ((adjacentto ?from ?to))
      ((!drive-robot ?robot ?from ?to))
31    Case3  ((room ?room)(adjacentto ?from ?room)(not (visited...
          ?room)))
      ((!drive-robot ?robot ?from ?room)
33    (!visit ?room)
      (navigate ?robot ?room ?to)
35    (!unvisit ?room))
    )
37  ; http://mas.b-it-center.de/ontologies/planner.owl#navigate2
    (:method (navigate ?robot ?to)
39    ((at ?robot ?from))
      ((!visit ?from)(navigate ?robot ?from ?to)(!unvisit ?from...
          ))
41  )
  )
43 )
```

Listing A.2 shows the generated planning problem description for the planning domain "navigation-domain" shown in Listing A.1. The planning objective is "(navigate robot1 room-6)".

Listing A.2: Planning problem description for "navigation-domain".

```
1  ; JSHOP2 Planning Problem Description
   ; created on: Tue Dec 09 11:20:36 CET 2008
3  ; generated for: navigation_domain

5  (defproblem problem_navigation_domain_1228818036430 ...
     navigation_domain
   (
7    (adjacentto corridor-1 corridor-2)
     (at robot1 room-1)
9    (adjacentto room-11 corridor-3)
     (room room-6)
11   (adjacentto corridor-1 room-2)
     (adjacentto corridor-3 corridor-2)
13   (adjacentto room-1 corridor-1)
     (room room-9)
15   (room room-11)
     (room room-1)
```

```
    (adjacentto corridor-1 room-5)
    (adjacentto corridor-2 corridor-3)
    (adjacentto room-6 corridor-1)
    (adjacentto corridor-1 room-3)
    (at robot2 room-4)
    (adjacentto corridor-3 room-10)
    (adjacentto room-11 room-9)
    (adjacentto corridor-1 room-6)
    (adjacentto corridor-1 room-4)
    (adjacentto room-6 room-4)
    (room corridor-1)
    (adjacentto room-9 room-11)
    (adjacentto corridor-2 corridor-1)
    (adjacentto corridor-3 room-7)
    (room corridor-2)
    (adjacentto corridor-3 room-11)
    (adjacentto corridor-3 room-8)
    (adjacentto corridor-1 room-1)
    (adjacentto corridor-3 room-12)
    (room room-3)
    (room corridor-3)
    (adjacentto room-9 room-7)
    (adjacentto corridor-3 room-9)
    (adjacentto room-9 corridor-3)
    (adjacentto room-3 corridor-1)
  )
  (
    (navigate robot1 room-6)
  )
)
```

Listing A.3 shows the planning solution for the planning domain in Listing A.1 and problem in Listing A.2.

Listing A.3: Solution plan for "`navigation-domain`".

```
Plan cost: 6.0

(!visit room-1)
(!drive-robot robot1 room-1 corridor-1)
(!visit corridor-1)
(!drive-robot robot1 corridor-1 room-6)
(!unvisit corridor-1)
(!unvisit room-1)
--------------------

Time Used = 0.0090
```

A.1.2. An Example from the $Method$ Instance

Listing A.4 shows the generated planning domain description by choosing "navigate", an instance of the $Method$ concept, from the DL model.

Listing A.4: Planning domain description for "navigate".

```
; JSHOP2 Planning Domain Description
; created on: Tue Dec 09 11:24:09 CET 2008
; generated from: http://mas.b-it-center.de/ontologies/planner....
    owl#

(defdomain navigate_domain (
  ; http://mas.b-it-center.de/ontologies/planner.owl#drive-...
      robot
    (:operator (!drive-robot ?robot ?loc-from ?loc-to)
      ((at ?robot ?loc-from))
       ((at ?robot ?loc-from))
       ((at ?robot ?loc-to))
    )
  ; http://mas.b-it-center.de/ontologies/planner.owl#visit
    (:operator (!visit ?waypoint)
      ()
      ()
      ((visited ?waypoint))
    )
  ; http://mas.b-it-center.de/ontologies/planner.owl#unvisit
    (:operator (!unvisit ?waypoint)
      ()
      ((visited ?waypoint))
      ()
    )

  ; http://mas.b-it-center.de/ontologies/planner.owl#navigate
    (:method  (navigate ?robot ?from ?to)
      Case1  ((at ?robot ?to))
      ()
      Case2  ((adjacentto ?from ?to))
      ((!drive-robot ?robot ?from ?to))
      Case3  ((room ?room)(adjacentto ?from ?room)(not (visited...
           ?room)))
      ((!drive-robot ?robot ?from ?room)
      (!visit ?room)
      (navigate ?robot ?room ?to)
      (!unvisit ?room))
    )
  )
)
```

Listing A.5 shows the generated planning problem for the planning domain "navigate" shown in Listing A.4. The planning objective is "(navigate robot1 room-1 room-6)".

Listing A.5: Planning problem description for "navigate".

```
; JSHOP2 Planning Problem Description
; created on: Tue Dec 09 11:24:28 CET 2008
; generated for: navigate_domain

(defproblem problem_navigate_domain_1228818268952 ...
    navigate_domain
  (
    (adjacentto corridor-3 room-10)
    (adjacentto corridor-1 room-6)
    (at robot1 room-1)
    (room room-6)
    (adjacentto room-3 corridor-1)
    (room room-3)
    (room room-11)
    (room room-1)
    (at robot2 room-4)
    (adjacentto corridor-3 corridor-2)
    (adjacentto corridor-3 room-9)
    (adjacentto room-1 corridor-1)
    (room corridor-2)
    (adjacentto corridor-1 corridor-2)
    (room corridor-3)
    (adjacentto corridor-3 room-11)
    (adjacentto corridor-1 room-2)
    (adjacentto room-11 room-9)
    (adjacentto room-9 corridor-3)
    (adjacentto corridor-1 room-1)
    (adjacentto room-11 corridor-3)
    (adjacentto room-6 corridor-1)
    (adjacentto corridor-3 room-8)
    (adjacentto room-9 room-11)
    (adjacentto corridor-2 corridor-3)
    (adjacentto room-6 room-4)
    (room corridor-1)
    (adjacentto corridor-1 room-3)
    (adjacentto corridor-3 room-12)
    (adjacentto corridor-2 corridor-1)
    (adjacentto corridor-1 room-4)
    (room room-9)
    (adjacentto corridor-1 room-5)
    (adjacentto corridor-3 room-7)
    (adjacentto room-9 room-7)
  )
```

```
    (
44      (navigate robot1 room-1 room-6)
    )
46 )
```

Listing A.6 shows the planning solution for the planning domain in Listing A.4 and problem in Listing A.5.

Listing A.6: Solution plan for "`navigate`".

```
   Plan cost: 4.0
2
   (!drive-robot robot1 room-1 corridor-1)
4  (!visit corridor-1)
   (!drive-robot robot1 corridor-1 room-6)
6  (!unvisit corridor-1)
   --------------------
8
   Time Used = 0.0090
```

A.2. Pick-and-Place Domain

The details of the pick-and-place domain are presented in Section 3.4 on page 55. Two domains are presented in this section; the first domain shows the partial pick-and-place domain and the second one shows the complete pick-and-place domain.

A.2.1. Partial Pick-and-Place Domain

Listing A.7 shows the planning domain description generated by choosing "`moveobject2_p`" from the DL model. The `moveobject2_p` is an instance of the $Method$ concept.

Listing A.7: Planning domain description for "`moveobject2_p`".

```
1  ; JSHOP2 Planning Domain Description
   ; created on: Tue Dec 09 17:30:46 CET 2008
3  ; generated from: http://mas.b-it-center.de/ontologies/planner....
       owl#

5  (defdomain moveobject2_p_domain (
     ; http://mas.b-it-center.de/ontologies/planner.owl#navigate-...
         op
7      (:operator (!navigate ?robot ?loc-to)
         ((at ?robot ?loc-from))
9                   ((at ?robot ?loc-from))
                    ((at ?robot ?loc-to))
11     )
```

```
    ; http://mas.b-it-center.de/ontologies/planner.owl#put-object
13    (:operator (!put-object ?robot ?object ?to-container)
        ((at-dexterous-workspace ?robot ?to-container)
15      (has-object ?robot)(at ?object ?robot))
          ((has-object ?robot)(at ?object ?robot))
17        ((at ?object ?to-container))
      )
19  ; http://mas.b-it-center.de/ontologies/planner.owl#drive-to-...
        dexterous-workspace
      (:operator (!drive-to-dexterous-workspace ?robot ?to-...
         container ?in-room)
21      ((at ?robot ?in-room)(at ?to-container ?in-room))
          ()
23        ((at-dexterous-workspace ?robot ?to-container))
      )
25  ; http://mas.b-it-center.de/ontologies/planner.owl#drive-away...
        -f-dex-workspace
      (:operator (!drive-away-from-dex-workspace ?robot ?from-...
         container)
27      ()
          ((at-dexterous-workspace ?robot ?from-container))
29        ()
      )
31  ; http://mas.b-it-center.de/ontologies/planner.owl#pickup-...
        object
      (:operator (!pickup-object ?robot ?object ?from-container)
33      ((at-dexterous-workspace ?robot ?from-container)
        (not (has-object ?robot)))
35        ((at ?object ?from-container)(:protection (at-dexterous...
              -workspace ?robot ?from-container)))
          ((has-object ?robot)(at ?object ?robot))
37    )

39  ; http://mas.b-it-center.de/ontologies/planner.owl#...
        moveobject2_p
      (:method (move-object ?robot ?object ?to-container)
41      ((at ?object ?from-container))
                ((move-object ?robot ?object ?from-container ?to-...
                   container))
43    )
    ; http://mas.b-it-center.de/ontologies/planner.owl#...
        getobject_p
45    (:method (get-object ?robot ?object ?from-container)
        Case1 ((at ?object ?robot))
47        ((!drive-away-from-dex-workspace ?robot ?from-container...
              ))
         Case2 ((at-dexterous-workspace ?robot ?from-container))
```

```
49          ((!pickup-object ?robot ?object ?from-container)(get-...
                object ?robot ?object ?from-container))
            Case3 ((not (at-dexterous-workspace ?robot ?from-...
                container))(at ?from-container ?room)(at ?robot ?...
                room))
51          ((!drive-to-dexterous-workspace ?robot ?from-container ...
                ?room)(get-object ?robot ?object ?from-container))
            Case4 ((not (at-dexterous-workspace ?robot ?from-...
                container))(at ?from-container ?room)(not (at ?...
                robot ?room)))
53          ((!navigate ?robot ?room)(get-object ?robot ?object ?...
                from-container))
      )
55  ; http://mas.b-it-center.de/ontologies/planner.owl#...
        moveobject_p
      (:method (move-object ?robot ?object ?from-container ?to-...
          container)
57      Case1 ((not (at ?object ?robot))(at ?object ?from-...
            container))
          ((get-object ?robot ?object ?from-container)(move-...
              object ?robot ?object ?from-container ?to-container...
              ))
59        Case2 ((at ?object ?robot))
          ((put-object ?robot ?object ?to-container))
61    )
    ; http://mas.b-it-center.de/ontologies/planner.owl#...
        putobject_p
63    (:method (put-object ?robot ?object ?to-container)
        Case1 ((not (at ?object ?robot))
65      )           ()
                    Case2 ((at ?object ?robot)(at-dexterous-...
                        workspace ?robot ?to-container))
67                  ((!put-object ?robot ?object ?to-container))
                    Case3 ((at ?object ?robot)(not (at-dexterous-...
                        workspace ?robot ?to-container))(at ?to-...
                        container ?room)(at ?robot ?room))
69                  ((!drive-to-dexterous-workspace ?robot ?to-...
                        container ?room)(put-object ?robot ?object...
                         ?to-container))
                    Case4 ((at ?object ?robot)(not (at-dexterous-...
                        workspace ?robot ?to-container))(at ?to-...
                        container ?room)(not (at ?robot ?room)))
71                  ((!navigate ?robot ?room)(put-object ?robot ?...
                        object ?to-container))
      )
73  )
  )
```

Listing A.8 shows the generated planning problem for the planning domain "moveobject2_p" shown in Listing A.7. The planning objective is "(move-object robot1 orange1 basket-9)".

Listing A.8: Planning problem description for "moveobject2_p".

```
; JSHOP2 Planning Problem Description
; created on: Tue Dec 09 17:31:05 CET 2008
; generated for: moveobject2_p_domain

(defproblem problem_moveobject2_p_domain_1228840265868 ...
    moveobject2_p_domain
  (
    (at basket-1 room-1)
    (at basket-9 room-9)
    (at apple1 basket-1)
    (at orange2 basket-11)
    (at orange3 basket-11)
    (at basket-3 room-3)
    (at trash-6 room-6)
    (at robot1 room-1)
    (at apple3 basket-1)
    (at apple2 basket-1)
    (at basket-11 room-11)
    (at orange1 basket-11)
    (at robot2 room-4)
  )
  (
    (move-object robot1 orange1 basket-9)
  )
)
```

Listing A.9 shows the planning solution for the planning domain in Listing A.7 and problem in Listing A.8.

Listing A.9: Solution plan for "moveobject2_p".

```
Plan cost: 7.0

(!navigate robot1 room-11)
(!drive-to-dexterous-workspace robot1 basket-11 room-11)
(!pickup-object robot1 orange1 basket-11)
(!drive-away-from-dex-workspace robot1 basket-11)
(!navigate robot1 room-9)
(!drive-to-dexterous-workspace robot1 basket-9 room-9)
(!put-object robot1 orange1 basket-9)
--------------------
```

```
12 Time Used = 0.019
```

A.2.2. Complete Pick-and-Place Domain

Listing A.10 shows the planning domain description generated by choosing "moveobject2_c" from the DL model. The moveobject2_c is an instance of the $Method$ concept.

Listing A.10: Planning domain description for "moveobject2_c".

```
; JSHOP2 Planning Domain Description
; created on: Tue Dec 09 21:02:45 CET 2008
; generated from: http://mas.b-it-center.de/ontologies/planner....
   owl#

(defdomain moveobject2_c_domain (
  ; http://mas.b-it-center.de/ontologies/planner.owl#drive-...
     robot
    (:operator (!drive-robot ?robot ?loc-from ?loc-to)
      ((at ?robot ?loc-from))
       ((at ?robot ?loc-from))
       ((at ?robot ?loc-to))
    )
  ; http://mas.b-it-center.de/ontologies/planner.owl#visit
    (:operator (!visit ?waypoint)
      ()
      ()
      ((visited ?waypoint))
    )
  ; http://mas.b-it-center.de/ontologies/planner.owl#put-object
    (:operator (!put-object ?robot ?object ?to-container)
      ((at-dexterous-workspace ?robot ?to-container)
      (has-object ?robot)(at ?object ?robot))
        ((has-object ?robot)(at ?object ?robot))
        ((at ?object ?to-container))
    )
  ; http://mas.b-it-center.de/ontologies/planner.owl#drive-to-...
     dexterous-workspace
    (:operator (!drive-to-dexterous-workspace ?robot ?to-...
       container ?in-room)
      ((at ?robot ?in-room)(at ?to-container ?in-room))
        ()
        ((at-dexterous-workspace ?robot ?to-container))
    )
  ; http://mas.b-it-center.de/ontologies/planner.owl#unvisit
    (:operator (!unvisit ?waypoint)
      ()
      ((visited ?waypoint))
      ()
```

```
36     )
     ; http://mas.b-it-center.de/ontologies/planner.owl#drive-away...
         -f-dex-workspace
38     (:operator (!drive-away-from-dex-workspace ?robot ?from-...
           container)
         ()
40         ((at-dexterous-workspace ?robot ?from-container))
           ()
42     )
     ; http://mas.b-it-center.de/ontologies/planner.owl#pickup-...
         object
44     (:operator (!pickup-object ?robot ?object ?from-container)
         ((at-dexterous-workspace ?robot ?from-container)
46       (not (has-object ?robot)))
           ((at ?object ?from-container)(:protection (at-dexterous...
               -workspace ?robot ?from-container)))
48         ((has-object ?robot)(at ?object ?robot))
       )
50
     ; http://mas.b-it-center.de/ontologies/planner.owl#...
         getobject_c
52     (:method (get-object ?robot ?object ?from-container)
         Case1 ((at ?object ?robot))
54         ((!drive-away-from-dex-workspace ?robot ?from-container...
               ))
           Case2 ((at-dexterous-workspace ?robot ?from-container))
56         ((!pickup-object ?robot ?object ?from-container)(get-...
               object ?robot ?object ?from-container))
           Case3 ((not (at-dexterous-workspace ?robot ?from-...
               container))(at ?from-container ?room)(at ?robot ?...
               room))
58         ((!drive-to-dexterous-workspace ?robot ?from-container ...
               ?room)(get-object ?robot ?object ?from-container))
           Case4 ((not (at-dexterous-workspace ?robot ?from-...
               container))(at ?from-container ?room)(not (at ?...
               robot ?room)))
60         ((navigate ?robot ?room)(get-object ?robot ?object ?...
               from-container))
       )
62   ; http://mas.b-it-center.de/ontologies/planner.owl#navigate
       (:method (navigate ?robot ?from ?to)
64       Case1  ((at ?robot ?to))
         ()
66       Case2  ((adjacentto ?from ?to))
         ((!drive-robot ?robot ?from ?to))
68       Case3  ((room ?room)(adjacentto ?from ?room)(not (visited...
              ?room)))
         ((!drive-robot ?robot ?from ?room)
```

```
70        (!visit ?room)
          (navigate ?robot ?room ?to)
72        (!unvisit ?room))
        )
74    ; http://mas.b-it-center.de/ontologies/planner.owl#...
        moveobject_c
        (:method (move-object ?robot ?object ?from-container ?to-...
            container)
76        Case1 ((not (at ?object ?robot))(at ?object ?from-...
              container))
            ((get-object ?robot ?object ?from-container)(move-...
                object ?robot ?object ?from-container ?to-container...
                ))
78          Case2 ((at ?object ?robot))
            ((put-object ?robot ?object ?to-container))
80      )
      ; http://mas.b-it-center.de/ontologies/planner.owl#...
        moveobject2_c
82      (:method (move-object ?robot ?object ?to-container)
          ((at ?object ?from-container))
84              ((move-object ?robot ?object ?from-container ?to-...
                    container))
        )
86    ; http://mas.b-it-center.de/ontologies/planner.owl#navigate2
        (:method (navigate ?robot ?to)
88        ((at ?robot ?from))
          ((!visit ?from)(navigate ?robot ?from ?to)(!unvisit ?from...
              ))
90      )
      ; http://mas.b-it-center.de/ontologies/planner.owl#...
        putobject_c
92      (:method (put-object ?robot ?object ?to-container)
          Case1 ((not (at ?object ?robot))
94        )           ()
                      Case2 ((at ?object ?robot)(at-dexterous-...
                          workspace ?robot ?to-container))
96                    ((!put-object ?robot ?object ?to-container))
                      Case3 ((at ?object ?robot)(not (at-dexterous-...
                          workspace ?robot ?to-container))(at ?to-...
                          container ?room)(at ?robot ?room))
98                    ((!drive-to-dexterous-workspace ?robot ?to-...
                          container ?room)(put-object ?robot ?object...
                           ?to-container))
                      Case4 ((at ?object ?robot)(not (at-dexterous-...
                          workspace ?robot ?to-container))(at ?to-...
                          container ?room)(not (at ?robot ?room)))
100                   ((navigate ?robot ?room)(put-object ?robot ?...
                          object ?to-container))
```

```
        )
102   )
)
```

Listing A.11 shows the planning problem generated for the planning domain "`moveobject2_c`" shown in Listing A.10. The planning objective is "`(move-object robot1 orange1 basket-9)`".

Listing A.11: Planning problem description for "`moveobject2_c`".

```
1  ; JSHOP2 Planning Problem Description
   ; created on: Tue Dec 09 21:03:16 CET 2008
3  ; generated for: moveobject2_c_domain

5  (defproblem problem_moveobject2_c_domain_1228852996332 ...
       moveobject2_c_domain
     (
7      (adjacentto corridor-3 room-7)
       (at trash-6 room-6)
9      (adjacentto room-3 corridor-1)
       (adjacentto corridor-1 room-3)
11     (adjacentto room-9 room-11)
       (room room-9)
13     (at apple3 basket-1)
       (room corridor-3)
15     (room room-1)
       (adjacentto room-11 room-9)
17     (room corridor-1)
       (adjacentto room-6 corridor-1)
19     (adjacentto corridor-2 corridor-3)
       (adjacentto room-9 corridor-3)
21     (at basket-1 room-1)
       (adjacentto room-9 room-7)
23     (adjacentto corridor-2 corridor-1)
       (adjacentto corridor-1 corridor-2)
25     (at orange1 basket-11)
       (adjacentto corridor-3 room-9)
27     (adjacentto room-11 corridor-3)
       (at orange3 basket-11)
29     (adjacentto corridor-3 corridor-2)
       (adjacentto corridor-1 room-4)
31     (at apple1 basket-1)
       (at robot2 room-4)
33     (adjacentto room-6 room-4)
       (adjacentto corridor-1 room-5)
35     (room corridor-2)
       (at robot1 room-1)
37     (adjacentto room-1 corridor-1)
```

```
    (at basket-9 room-9)
39  (adjacentto corridor-1 room-2)
    (at apple2 basket-1)
41  (at orange2 basket-11)
    (adjacentto corridor-3 room-11)
43  (adjacentto corridor-3 room-10)
    (room room-6)
45  (at basket-11 room-11)
    (adjacentto corridor-1 room-6)
47  (adjacentto corridor-1 room-1)
    (adjacentto corridor-3 room-8)
49  (at basket-3 room-3)
    (adjacentto corridor-3 room-12)
51  (room room-11)
    (room room-3)
53 )
   (
55  (move-object robot1 orange1 basket-9)
   )
57 )
```

Listing A.12 shows the planning solution for the planning domain in Listing A.10 and problem in Listing A.11.

Listing A.12: Solution plan for "moveobject2_c".

```
1  Plan cost: 20.0

3  (!visit room-1)
   (!drive-robot robot1 room-1 corridor-1)
5  (!visit corridor-1)
   (!drive-robot robot1 corridor-1 corridor-2)
7  (!visit corridor-2)
   (!drive-robot robot1 corridor-2 corridor-3)
9  (!visit corridor-3)
   (!drive-robot robot1 corridor-3 room-11)
11 (!unvisit corridor-3)
   (!unvisit corridor-2)
13 (!unvisit corridor-1)
   (!unvisit room-1)
15 (!drive-to-dexterous-workspace robot1 basket-11 room-11)
   (!pickup-object robot1 orange1 basket-11)
17 (!drive-away-from-dex-workspace robot1 basket-11)
   (!visit room-11)
19 (!drive-robot robot1 room-11 room-9)
   (!unvisit room-11)
21 (!drive-to-dexterous-workspace robot1 basket-9 room-9)
   (!put-object robot1 orange1 basket-9)
23 --------------------
```

```
25 Time Used = 0.051
```

A.3. Johnny Jackanapes Domain

The details of the Johnny Jackanapes domain are presented in Section 4.3.2 on page 90. Two possible solutions are presented in this work; the first solution is extracted using the complete pick-and-place domain and the second one comes about by customising the pick-and-place domain for Johnny's tasks.

A.3.1. Solution Using Pick-and-Place Domain

Listing A.13 shows the planning problem generated for Johnny's task using the planning domain "`moveobject2_c`" shown in Listing A.10. The planning objectives are "`(move-object johnny coke armchair)`" and "`(navigate johnny exit)`".

Listing A.13: Planning problem description for "`moveobject2_c`" in Johnny's example.

```
1  ; JSHOP2 Planning Problem Description
   ; created on: Mon Apr 13 19:13:54 CEST 2009
3  ; generated for: moveobject2_c_domain

5  (defproblem problem_moveobject2_c_domain_1239642834168 ...
     moveobject2_c_domain
     (
7      (at sofa-1 living-room)
       (room living-room)
9      (room exit)
       (adjacentto living-room kitchen)
11     (at green-tea sideboard)
       (at sofa-1 living-room)
13     (at sofa-2 living-room)
       (at bookshelf living-room)
15     (at dining-table kitchen)
       (adjacentto exit living-room)
17     (at johnny kitchen)
       (at robot1 room-1)
19     (room kitchen)
       (at armchair living-room)
21     (adjacentto kitchen living-room)
       (adjacentto living-room exit)
23     (at robot2 room-4)
       (at coke sideboard)
25     (at shelf living-room)
       (at yellow-tea sideboard)
27     (at sideboard kitchen)
```

```
  )
  (
    (move-object johnny coke armchair)
    (navigate johnny exit)
  )
)
```

Listing A.14 shows the planning solution for the planning domain in Listing A.10 and problem in Listing A.13.

Listing A.14: Solution plan for "moveobject2_c" in Johnny's move object planning task.

```
Plan cost: 11.0

(!drive-to-dexterous-workspace johnny sideboard kitchen)
(!pickup-object johnny coke sideboard)
(!drive-away-from-dex-workspace johnny sideboard)
(!visit kitchen)
(!drive-robot johnny kitchen living-room)
(!unvisit kitchen)
(!drive-to-dexterous-workspace johnny armchair living-room)
(!put-object johnny coke armchair)
(!visit living-room)
(!drive-robot johnny living-room exit)
(!unvisit living-room)
--------------------

Time Used = 0.014
```

A.3.2. Bring an Object Planning Domain

Listing A.15 shows the planning domain description generated by choosing "bringobject2", an instance of the $Method$ concept which was modified from the complete pick-and-place domain, from the DL model.

Listing A.15: Planning domain description for "bringobject2".

```
; JSHOP2 Planning Domain Description
; created on: Mon Apr 13 20:37:20 CEST 2009
; generated from: http://mas.b-it-center.de/ontologies/planner....
    owl#

(defdomain bringobject2_domain (
  ; http://mas.b-it-center.de/ontologies/planner.owl#drive-...
      robot
    (:operator (!drive-robot ?robot ?loc-from ?loc-to)
      ((at ?robot ?loc-from))
```

```
        ((at ?robot ?loc-from))
10      ((at ?robot ?loc-to))
    )
12  ; http://mas.b-it-center.de/ontologies/planner.owl#visit
    (:operator (!visit ?waypoint)
14    ()
      ()
16    ((visited ?waypoint))
    )
18  ; http://mas.b-it-center.de/ontologies/planner.owl#give-...
      object
    (:operator (!give-object ?robot ?object ?to-person)
20    ((on ?to-person ?to-container)
             (at-dexterous-workspace ?robot ?to-container)
22           (has-object ?robot)(at ?object ?robot))
              ((has-object ?robot)(at ?object ?robot))
24            ((has-object ?to-person)(at ?object ?to-person))
    )
26  ; http://mas.b-it-center.de/ontologies/planner.owl#drive-to-...
      dexterous-workspace
    (:operator (!drive-to-dexterous-workspace ?robot ?to-...
        container ?in-room)
28    ((at ?robot ?in-room)(at ?to-container ?in-room))
        ()
30      ((at-dexterous-workspace ?robot ?to-container))
    )
32  ; http://mas.b-it-center.de/ontologies/planner.owl#unvisit
    (:operator (!unvisit ?waypoint)
34    ()
      ((visited ?waypoint))
36    ()
    )
38  ; http://mas.b-it-center.de/ontologies/planner.owl#drive-away...
      -f-dex-workspace
    (:operator (!drive-away-from-dex-workspace ?robot ?from-...
        container)
40    ()
        ((at-dexterous-workspace ?robot ?from-container))
42      ()
    )
44  ; http://mas.b-it-center.de/ontologies/planner.owl#pickup-...
      object
    (:operator (!pickup-object ?robot ?object ?from-container)
46    ((at-dexterous-workspace ?robot ?from-container)
      (not (has-object ?robot)))
48      ((at ?object ?from-container)(:protection (at-dexterous...
            -workspace ?robot ?from-container)))
        ((has-object ?robot)(at ?object ?robot))
```

```
50    )

52  ; http://mas.b-it-center.de/ontologies/planner.owl#...
      getobject_c
    (:method (get-object ?robot ?object ?from-container)
54    Case1 ((at ?object ?robot))
        ((!drive-away-from-dex-workspace ?robot ?from-container...
          ))
56      Case2 ((at-dexterous-workspace ?robot ?from-container))
        ((!pickup-object ?robot ?object ?from-container)(get-...
          object ?robot ?object ?from-container))
58      Case3 ((not (at-dexterous-workspace ?robot ?from-...
          container))(at ?from-container ?room)(at ?robot ?...
          room))
        ((!drive-to-dexterous-workspace ?robot ?from-container ...
          ?room)(get-object ?robot ?object ?from-container))
60      Case4 ((not (at-dexterous-workspace ?robot ?from-...
          container))(at ?from-container ?room)(not (at ?...
          robot ?room)))
        ((navigate ?robot ?room)(get-object ?robot ?object ?...
          from-container))
62    )
    ; http://mas.b-it-center.de/ontologies/planner.owl#...
      bringobject2
64    (:method (bring-object ?robot ?object ?to-person)
      ((at ?object ?from-container))
66                ((bring-object ?robot ?object ?from-...
                    container ?to-person))
    )
68  ; http://mas.b-it-center.de/ontologies/planner.owl#navigate
    (:method  (navigate ?robot ?from ?to)
70    Case1  ((at ?robot ?to))
      ()
72    Case2  ((adjacentto ?from ?to))
      ((!drive-robot ?robot ?from ?to))
74    Case3  ((room ?room)(adjacentto ?from ?room)(not (visited...
          ?room)))
      ((!drive-robot ?robot ?from ?room)
76    (!visit ?room)
      (navigate ?robot ?room ?to)
78    (!unvisit ?room))
    )
80  ; http://mas.b-it-center.de/ontologies/planner.owl#...
      bringobject
    (:method (bring-object ?robot ?object ?from-container ?to-...
       person)
82    Case1 ((not (at ?object ?robot))(at ?object ?from-...
          container))
```

```
                ((get-object ?robot ?object ?from-container)(...
                    bring-object ?robot ?object ?from-container ?...
                    to-person))
84            Case2 ((at ?object ?robot))
                ((give-object ?robot ?object ?to-person))
86    )
    ; http://mas.b-it-center.de/ontologies/planner.owl#navigate2
88    (:method (navigate ?robot ?to)
      ((at ?robot ?from))
90    ((!visit ?from)(navigate ?robot ?from ?to)(!unvisit ?from...
          ))
    )
92  ; http://mas.b-it-center.de/ontologies/planner.owl#giveobject
    (:method (give-object ?robot ?object ?to-person)
94    Case1 ((not (at ?object ?robot)))          ()
            Case2 ((at ?object ?robot)(on ?to-person ?to-...
                container)(at-dexterous-workspace ?robot ?to-...
                container))
96              ((!give-object ?robot ?object ?to-person))
            Case3 ((at ?object ?robot)(on ?to-person ?to-...
                container)(not (at-dexterous-workspace ?robot ?...
                to-container))(at ?to-container ?room)(at ?...
                robot ?room))
98              ((!drive-to-dexterous-workspace ?robot ?to-...
                    container ?room)(give-object ?robot ?object ?...
                    to-person))
            Case4 ((at ?object ?robot)(on ?to-person ?to-...
                container)(not (at-dexterous-workspace ?robot ?...
                to-container))(at ?to-container ?room)(not (at ...
                ?robot ?room)))
100             ((navigate ?robot ?room)(give-object ?robot ?...
                    object ?to-person))
    )
102 )
)
```

Listing A.16 shows the planning problem generated for the planning domain “`bringobject2`” shown in Listing A.15. The planning objectives are “`(bring-object johnny coke guest)`” and “`(navigate johnny exit)`”.

Listing A.16: Planning problem description for “`bringobject2`”.

```
; JSHOP2 Planning Problem Description
2 ; created on: Mon Apr 13 20:37:57 CEST 2009
; generated for: bringobject2_domain
4
(defproblem problem_bringobject2_domain_1239647877985 ...
    bringobject2_domain
```

```
  (
    (at shelf living-room)
    (at sofa-1 living-room)
    (room living-room)
    (at yellow-tea sideboard)
    (at sofa-2 living-room)
    (at robot2 room-4)
    (on guest armchair)
    (room exit)
    (at green-tea sideboard)
    (adjacentto exit living-room)
    (at coke sideboard)
    (adjacentto living-room kitchen)
    (at bookshelf living-room)
    (at johnny kitchen)
    (at dining-table kitchen)
    (at armchair living-room)
    (adjacentto kitchen living-room)
    (at robot1 room-1)
    (at sideboard kitchen)
    (adjacentto living-room exit)
    (room kitchen)
    (at sofa-1 living-room)
  )
  (
    (bring-object johnny coke guest)
    (navigate johnny exit)
  )
)
```

Listing A.17 shows the planning solution for the planning domain in Listing A.15 and problem in Listing A.16.

Listing A.17: Solution plan for "`bringobject2`" in Johnny's bring object planning task.

```
Plan cost: 11.0

(!drive-to-dexterous-workspace johnny sideboard kitchen)
(!pickup-object johnny coke sideboard)
(!drive-away-from-dex-workspace johnny sideboard)
(!visit kitchen)
(!drive-robot johnny kitchen living-room)
(!unvisit kitchen)
(!drive-to-dexterous-workspace johnny armchair living-room)
(!give-object johnny coke guest)
(!visit living-room)
(!drive-robot johnny living-room exit)
(!unvisit living-room)
--------------------
```

```
16 Time Used = 0.014
```

A.4. Blocks World Domain

The details of the Blocks World domain are presented in Chapter 5 on page 97. Three Blocks World problems are presented in this section; the first problem is the simple blocks world domain, the second one is the two blocks worlds domain, and the third one is the complex blocks world domain. All these problems use the same planning domain, included in the JSHOP2 source distribution. Listing A.18 shows this domain, which was generated by choosing "`block_domain`" from the DL model. The `block_domain` is an instance of *Planning-Domain* concept.

Listing A.18: Planning domain description for "`blocks_domain`".

```
; JSHOP2 Planning Domain Description
; created on: Tue Dec 30 17:51:48 CET 2008
; generated from: http://mas.b-it-center.de/ontologies/planner....
   owl#

(defdomain blocks_domain (
  ; http://mas.b-it-center.de/ontologies/planner.owl#assert
    (:operator (!!assert ?g)
      ()
      ()
      (?g)
      ;; Since !!ASSERT isn't a real blocks-world operator, ...
          make its cost
      0
      0
    )
  ; http://mas.b-it-center.de/ontologies/planner.owl#unstack
    (:operator (!unstack ?e ?f)
      ()
    ((clear ?e)
      (on ?e ?f))
      ((holding ?e)
      (clear ?f))
    )
  ; http://mas.b-it-center.de/ontologies/planner.owl#putdown
    (:operator (!putdown ?b)
      ()
      ((holding ?b))
      ((on-table ?b)
      (clear ?b))
    )
```

```
      ; http://mas.b-it-center.de/ontologies/planner.owl#pickup
31      (:operator (!pickup ?a)
          ()
33        ((clear ?a)
          (on-table ?a))
35        ((holding ?a))
        )
37    ; http://mas.b-it-center.de/ontologies/planner.owl#stack
        (:operator (!stack ?c ?d)
39        ()
          ((holding ?c)
41        (clear ?d))
          ((on ?c ?d)
43        (clear ?c))
        )
45    ; http://mas.b-it-center.de/ontologies/planner.owl#remove
        (:operator (!!remove ?g)
47        ()
          (?g)
49        ()
          0
51      )

53    ; http://mas.b-it-center.de/ontologies/planner.owl#check2
        (:method (check2 ?x)
55        ((dont-move ?x)
          (goal (on ?y ?x))
57        (clear ?y))
                ((!!assert (stack-on-block ?y ?x)))
59              nil
                nil
61      )
      ; http://mas.b-it-center.de/ontologies/planner.owl#find-...
          movable
63      (:method (find-movable)
          ((clear ?x)
65        (not (dont-move ?x))
                (goal (on-table ?x))
67        (not (put-on-table ?x)))
                ((!!assert (put-on-table ?x))
69        (find-movable))
                 ((clear ?x)
71        (not (dont-move ?x))
          (goal (on ?x ?y))
73              (not (stack-on-block ?x ?y))
          (dont-move ?y)
75        (clear ?y))
                 ((!!assert (stack-on-block ?x ?y))
```

```
77        (find-movable))
                 nil
79               nil
     )
81   ; http://mas.b-it-center.de/ontologies/planner.owl#add-new-...
       goals
     (:method (add-new-goals)
83     ((block ?x)
       (not (dont-move ?x))
85     (not (goal (on-table ?x)))
             (not (goal (on ?x ?y))))
87           ((!!assert (goal (on-table ?x)))
       (add-new-goals))
89                                 nil
                                   nil
91   )
     ; http://mas.b-it-center.de/ontologies/planner.owl#achieve-...
       goal
93   (:method (achieve-goals ?goals)
       ()
95           ((assert-goals ?goals)
             (find-nomove)
97     (add-new-goals)
       (find-movable)
99     (move-block))
     )
101  ; http://mas.b-it-center.de/ontologies/planner.owl#axiom-same
     (:- (same ?x ?x)
103    nil
     )
105  ; http://mas.b-it-center.de/ontologies/planner.owl#axiom-need...
       -to-move
     (:- (need-to-move ?x)
107    ((on ?x ?y)
       (goal (on ?x ?z))
109    (not (same ?y ?z)))
                             ((on-table ?x)
111    (goal (on ?x ?z)))
               ((on ?x ?y)
113    (goal (on-table ?x)))
                      ((on ?x ?y)
115    (goal (clear ?y)))
                                 ((on ?x ?z)
117    (goal (on ?y ?z))
       (not (same ?x ?y)))
119           ((on ?x ?w)
       (need-to-move ?w))
121  )
```

```
    ; http://mas.b-it-center.de/ontologies/planner.owl#check3
123   (:method (check3 ?x)
        (dont-move ?x)
125           nil
              ((goal (on ?x ?y))
127     (clear ?y)
        (dont-move ?y))
129           ((!!assert (stack-on-block ?x ?y)))
              ((goal (on-table ?x)))
131           ((!!assert (put-on-table ?x)))
              nil
133           nil
      )
135 ; http://mas.b-it-center.de/ontologies/planner.owl#move-block
      (:method (move-block)
137     ((stack-on-block ?x ?y))
              ((move-block1 ?x ?y)
139     (move-block))
               method-for-moving-x-from-y-to-table
141            ((put-on-table ?x)
        (on ?x ?y))
143                       ((!unstack ?x ?y)
        (!putdown ?x)
145           (!!assert (dont-move ?x))
              (!!remove (put-on-table ?x))
147           (check ?x)
        (check2 ?y)
149     (check3 ?y)
        (move-block))
151            method-for-moving-x-out-of-the-way
               ((clear ?x)
153     (not (dont-move ?x))
        (on ?x ?y))
155            ((!unstack ?x ?y)
        (!putdown ?x)
157           (check2 ?y)
        (check3 ?y)
159     (move-block))
               termination-method-branch
161            nil
               nil
163   )
    ; http://mas.b-it-center.de/ontologies/planner.owl#check
165   (:method (check ?x)
        ((goal (on ?y ?x))
167     (clear ?y))
              ((!!assert (stack-on-block ?y ?x)))
169           nil
```

```
                nil
171     )
    ; http://mas.b-it-center.de/ontologies/planner.owl#move-...
       block1
173     (:method (move-block1 ?x ?z)
          method-for-moving-x-from-y-to-z  ((on ?x ?y))
175        ((!unstack ?x ?y)
           (!stack ?x ?z)
177              (!!assert (dont-move ?x))
                 (!!remove (stack-on-block ?x ?z))
179              (check ?x)
          (check2 ?y)
181       (check3 ?y))
                 method-for-moving-x-from-table-to-z
183              nil
                 ((!pickup ?x)
185       (!stack ?x ?z)
                 (!!assert (dont-move ?x))
187              (!!remove (stack-on-block ?x ?z))
                 (check ?x))
189     )
    ; http://mas.b-it-center.de/ontologies/planner.owl#assert-...
       goals-nil
191     (:method (assert-goals nil)
          ()
193              ()
        )
195 ; http://mas.b-it-center.de/ontologies/planner.owl#find-...
       nomove
        (:method (find-nomove)
197       ((block ?x)
          (not (dont-move ?x))
199       (not (need-to-move ?x)))
                 ((!!assert (dont-move ?x))
201       (find-nomove))
                nil
203             nil
        )
205 ; http://mas.b-it-center.de/ontologies/planner.owl#assert-...
       goals
        (:method (assert-goals (?goal . ?goals)
207       )
                  ()
209               ((!!assert (goal ?goal))
                  (assert-goals ?goals))
211     )
    )
213 )
```

A.4.1. Simple Blocks World Planning Example

Listing A.19 shows the planning problem generated for the planning domain "`blocks_domain`" shown in Listing A.18. The planning objective is "`(achieve-goals ((on-table b1) (on b4 b1) (clear b4) (on-table b3) (on b2 b3) (clear b2)))`" as shown in Figure 5.1 on page 98.

Listing A.19: Planning problem description for "`block_domain`" for solving simple blocks world problem.

```
; JSHOP2 Planning Problem Description
; created on: Tue Dec 30 17:51:58 CET 2008
; generated for: blocks_domain

(defproblem problem_blocks_domain_1230655918008 blocks_domain
  (
    (block b3)
    (clear b4)
    (on b2 b1)
    (on b4 b3)
    (on-table b3)
    (block b4)
    (on-table b1)
    (clear b2)
    (block b2)
    (block b1)
  )
  (
    (achieve-goals ((on-table b1) (on b4 b1) (clear b4) (on-...
        table b3) (on b2 b3) (clear b2)))
  )
)
```

Listing A.20 shows the planning solution for the planning domain in Listing A.18 and problem in Listing A.19.

Listing A.20: Solution plan for "`blocks_domain`" in simple blocks world example.

```
Plan cost: 18.0

(!!assert (goal (on-table b1)))
(!!assert (goal (on b4 b1)))
(!!assert (goal (clear b4)))
(!!assert (goal (on-table b3)))
(!!assert (goal (on b2 b3)))
(!!assert (goal (clear b2)))
(!!assert (dont-move b3))
(!!assert (dont-move b1))
```

```
11 (!unstack b4 b3)
   (!putdown b4)
13 (!!assert (stack-on-block b2 b3))
   (!unstack b2 b1)
15 (!stack b2 b3)
   (!!assert (dont-move b2))
17 (!!remove (stack-on-block b2 b3))
   (!!assert (stack-on-block b4 b1))
19 (!pickup b4)
   (!stack b4 b1)
21 (!!assert (dont-move b4))
   (!!remove (stack-on-block b4 b1))
23 --------------------

25 Plan cost: 18.0

27 (!!assert (goal (on-table b1)))
   (!!assert (goal (on b4 b1)))
29 (!!assert (goal (clear b4)))
   (!!assert (goal (on-table b3)))
31 (!!assert (goal (on b2 b3)))
   (!!assert (goal (clear b2)))
33 (!!assert (dont-move b3))
   (!!assert (dont-move b1))
35 (!unstack b2 b1)
   (!putdown b2)
37 (!!assert (stack-on-block b4 b1))
   (!unstack b4 b3)
39 (!stack b4 b1)
   (!!assert (dont-move b4))
41 (!!remove (stack-on-block b4 b1))
   (!!assert (stack-on-block b2 b3))
43 (!pickup b2)
   (!stack b2 b3)
45 (!!assert (dont-move b2))
   (!!remove (stack-on-block b2 b3))
47 --------------------

49 Plan cost: 18.0

51 (!!assert (goal (on-table b1)))
   (!!assert (goal (on b4 b1)))
53 (!!assert (goal (clear b4)))
   (!!assert (goal (on-table b3)))
55 (!!assert (goal (on b2 b3)))
   (!!assert (goal (clear b2)))
57 (!!assert (dont-move b1))
   (!!assert (dont-move b3))
```

```
59 (!unstack b4 b3)
   (!putdown b4)
61 (!!assert (stack-on-block b2 b3))
   (!unstack b2 b1)
63 (!stack b2 b3)
   (!!assert (dont-move b2))
65 (!!remove (stack-on-block b2 b3))
   (!!assert (stack-on-block b4 b1))
67 (!pickup b4)
   (!stack b4 b1)
69 (!!assert (dont-move b4))
   (!!remove (stack-on-block b4 b1))
71 --------------------

73 Plan cost: 18.0

75 (!!assert (goal (on-table b1)))
   (!!assert (goal (on b4 b1)))
77 (!!assert (goal (clear b4)))
   (!!assert (goal (on-table b3)))
79 (!!assert (goal (on b2 b3)))
   (!!assert (goal (clear b2)))
81 (!!assert (dont-move b1))
   (!!assert (dont-move b3))
83 (!unstack b2 b1)
   (!putdown b2)
85 (!!assert (stack-on-block b4 b1))
   (!unstack b4 b3)
87 (!stack b4 b1)
   (!!assert (dont-move b4))
89 (!!remove (stack-on-block b4 b1))
   (!!assert (stack-on-block b2 b3))
91 (!pickup b2)
   (!stack b2 b3)
93 (!!assert (dont-move b2))
   (!!remove (stack-on-block b2 b3))
95 --------------------

97 Time Used = 0.04
```

A.4.2. Two Blocks World Planning Example

Listing A.21 shows the generated planning problem for the planning domain “blocks_domain” shown in Listing A.18 for task two using the simple blocks world ontology. The planning objective is “(achieve-goals ((on-table b1) (on b4 b1) (clear b4) (on-table b3) (on b2 b3) (clear b2)))” as shown in Figure 5.5 on page 107.

Listing A.21: Planning problem description for "block_domain" for solving the two blocks worlds task two using the simple blocks world ontology.

```
; JSHOP2 Planning Problem Description
; created on: Fri Jan 02 16:17:41 CET 2009
; generated for: blocks_domain

(defproblem problem_blocks_domain_1230909461784 blocks_domain
  (
    (clear b2)
    (on-table b3)
    (block b1)
    (on b4 b3)
    (clear b4)
    (clear a2)
    (block b3)
    (block b4)
    (on-table b1)
    (block a4)
    (block b2)
    (block a3)
    (on a3 a1)
    (on b2 b1)
    (on a2 a4)
    (block a1)
    (on-table a4)
    (block a2)
    (on-table a1)
    (clear a3)
  )
  (
    (achieve-goals ((on-table b1) (on b4 b1) (clear b4) (on-...
        table b3) (on b2 b3) (clear b2)))
  )
)
```

Listing A.22 shows the generated planning problem for the planning domain "blocks_domain" shown in Listing A.18 for task one using the enhanced blocks world ontology. The planning objective is "(achieve-goals ((on-table a3)(on a1 a3)(clear a1)(on-table a2)(on a4 a2)(clear a4)))" as shown in Figure 5.5 on page 107.

Listing A.22: Planning problem description for "block_domain" for solving two blocks worlds task number one using the enhanced blocks world ontology.

```
; JSHOP2 Planning Problem Description
; created on: Tue Jan 06 17:26:18 CET 2009
; generated for: blocks_domain
```

```
5  (defproblem problem_blocks_domain_1231259178659 blocks_domain
     (
7      (clear a2)
       (on a3 a1)
9      (on-table a4)
       (block a3)
11     (on-table a1)
       (on a2 a4)
13     (block a4)
       (block a1)
15     (block a2)
       (clear a3)
17   )
     (
19     (achieve-goals ((on-table a3)(on a1 a3)(clear a1)(on-table ...
           a2)(on a4 a2)(clear a4)))
     )
21 )
```

Listing A.23 shows the planning solution for the planning domain in Listing A.18 and problem in Listing A.22.

Listing A.23: Solution plan for "`blocks_domain`" in the two blocks worlds example task number one using the enhanced blocks world ontology.

```
1  Plan cost: 24.0

3  (!!assert (goal (on-table a3)))
   (!!assert (goal (on a1 a3)))
5  (!!assert (goal (clear a1)))
   (!!assert (goal (on-table a2)))
7  (!!assert (goal (on a4 a2)))
   (!!assert (goal (clear a4)))
9  (!!assert (put-on-table a2))
   (!!assert (put-on-table a3))
11 (!unstack a2 a4)
   (!putdown a2)
13 (!!assert (dont-move a2))
   (!!remove (put-on-table a2))
15 (!!assert (stack-on-block a4 a2))
   (!!assert (stack-on-block a4 a2))
17 (!pickup a4)
   (!stack a4 a2)
19 (!!assert (dont-move a4))
   (!!remove (stack-on-block a4 a2))
21 (!unstack a3 a1)
   (!putdown a3)
23 (!!assert (dont-move a3))
   (!!remove (put-on-table a3))
```

```
25 (!!assert (stack-on-block a1 a3))
   (!!assert (stack-on-block a1 a3))
27 (!pickup a1)
   (!stack a1 a3)
29 (!!assert (dont-move a1))
   (!!remove (stack-on-block a1 a3))
31 --------------------

33 Plan cost: 24.0

35 (!!assert (goal (on-table a3)))
   (!!assert (goal (on a1 a3)))
37 (!!assert (goal (clear a1)))
   (!!assert (goal (on-table a2)))
39 (!!assert (goal (on a4 a2)))
   (!!assert (goal (clear a4)))
41 (!!assert (put-on-table a2))
   (!!assert (put-on-table a3))
43 (!unstack a3 a1)
   (!putdown a3)
45 (!!assert (dont-move a3))
   (!!remove (put-on-table a3))
47 (!!assert (stack-on-block a1 a3))
   (!!assert (stack-on-block a1 a3))
49 (!pickup a1)
   (!stack a1 a3)
51 (!!assert (dont-move a1))
   (!!remove (stack-on-block a1 a3))
53 (!unstack a2 a4)
   (!putdown a2)
55 (!!assert (dont-move a2))
   (!!remove (put-on-table a2))
57 (!!assert (stack-on-block a4 a2))
   (!!assert (stack-on-block a4 a2))
59 (!pickup a4)
   (!stack a4 a2)
61 (!!assert (dont-move a4))
   (!!remove (stack-on-block a4 a2))
63 --------------------

65 Plan cost: 24.0

67 (!!assert (goal (on-table a3)))
   (!!assert (goal (on a1 a3)))
69 (!!assert (goal (clear a1)))
   (!!assert (goal (on-table a2)))
71 (!!assert (goal (on a4 a2)))
   (!!assert (goal (clear a4)))
```

```
73 (!!assert (put-on-table a3))
   (!!assert (put-on-table a2))
75 (!unstack a3 a1)
   (!putdown a3)
77 (!!assert (dont-move a3))
   (!!remove (put-on-table a3))
79 (!!assert (stack-on-block a1 a3))
   (!!assert (stack-on-block a1 a3))
81 (!pickup a1)
   (!stack a1 a3)
83 (!!assert (dont-move a1))
   (!!remove (stack-on-block a1 a3))
85 (!unstack a2 a4)
   (!putdown a2)
87 (!!assert (dont-move a2))
   (!!remove (put-on-table a2))
89 (!!assert (stack-on-block a4 a2))
   (!!assert (stack-on-block a4 a2))
91 (!pickup a4)
   (!stack a4 a2)
93 (!!assert (dont-move a4))
   (!!remove (stack-on-block a4 a2))
95 --------------------

97 Plan cost: 24.0

99 (!!assert (goal (on-table a3)))
   (!!assert (goal (on a1 a3)))
101 (!!assert (goal (clear a1)))
   (!!assert (goal (on-table a2)))
103 (!!assert (goal (on a4 a2)))
   (!!assert (goal (clear a4)))
105 (!!assert (put-on-table a3))
   (!!assert (put-on-table a2))
107 (!unstack a2 a4)
   (!putdown a2)
109 (!!assert (dont-move a2))
   (!!remove (put-on-table a2))
111 (!!assert (stack-on-block a4 a2))
   (!!assert (stack-on-block a4 a2))
113 (!pickup a4)
   (!stack a4 a2)
115 (!!assert (dont-move a4))
   (!!remove (stack-on-block a4 a2))
117 (!unstack a3 a1)
   (!putdown a3)
119 (!!assert (dont-move a3))
   (!!remove (put-on-table a3))
```

```
121 (!!assert (stack-on-block a1 a3))
    (!!assert (stack-on-block a1 a3))
123 (!pickup a1)
    (!stack a1 a3)
125 (!!assert (dont-move a1))
    (!!remove (stack-on-block a1 a3))
127 --------------------

129 Time Used = 0.032
```

A.4.3. Complex Blocks World Example

Listing A.24 shows the generated planning problem for the planning domain "`blocks_domain`" shown in Listing A.18 for the complex blocks world problem. The planning objective is "`(achieve-goals ((on-table a1) (on b2 a1) (on c3 b2) (clear c3)))`" as shown in Figure 5.7 on page 109.

Listing A.24: Planning problem description for "`block_domain`" for solving the complex blocks world problem.

```
1  ; JSHOP2 Planning Problem Description
   ; created on: Wed Jan 07 22:46:19 CET 2009
3  ; generated for: blocks_domain

5  (defproblem problem_blocks_domain_1231364779743 blocks_domain
     (
7      (block a1)
       (block c2)
9      (clear b2)
       (block c1)
11     (on-table c3)
       (block a3)
13     (on c2 c1)
       (clear a3)
15     (block c3)
       (block b2)
17     (on-table a1)
       (clear c2)
19     (on a3 a1)
       (on c1 c3)
21   )
     (
23     (achieve-goals ((on-table a1) (on b2 a1) (on c3 b2) (clear ...
           c3)))
     )
25 )
```

Listing A.25 shows the planning solution for the planning domain in Listing A.18 and problem in Listing A.24.

Listing A.25: Solution plan for "`blocks_domain`" in complex blocks world example.

```
1
Plan cost: 29.0
3
(!!assert (goal (on-table a1)))
(!!assert (goal (on b2 a1)))
(!!assert (goal (on c3 b2)))
(!!assert (goal (clear c3)))
(!!assert (dont-move a1))
(!!assert (dont-move b2))
(!!assert (goal (on-table c2)))
(!!assert (goal (on-table c1)))
(!!assert (goal (on-table a3)))
(!!assert (put-on-table a3))
(!!assert (put-on-table c2))
(!unstack a3 a1)
(!putdown a3)
(!!assert (dont-move a3))
(!!remove (put-on-table a3))
(!!assert (stack-on-block b2 a1))
(!pickup b2)
(!stack b2 a1)
(!!assert (dont-move b2))
(!!remove (stack-on-block b2 a1))
(!unstack c2 c1)
(!putdown c2)
(!!assert (dont-move c2))
(!!remove (put-on-table c2))
(!!assert (put-on-table c1))
(!unstack c1 c3)
(!putdown c1)
(!!assert (dont-move c1))
(!!remove (put-on-table c1))
(!!assert (stack-on-block c3 b2))
(!pickup c3)
(!stack c3 b2)
(!!assert (dont-move c3))
(!!remove (stack-on-block c3 b2))
--------------------
39
Plan cost: 29.0
41
(!!assert (goal (on-table a1)))
(!!assert (goal (on b2 a1)))
```

```
(!!assert (goal (on c3 b2)))
(!!assert (goal (clear c3)))
(!!assert (dont-move a1))
(!!assert (dont-move b2))
(!!assert (goal (on-table c2)))
(!!assert (goal (on-table c1)))
(!!assert (goal (on-table a3)))
(!!assert (put-on-table a3))
(!!assert (put-on-table c2))
(!unstack c2 c1)
(!putdown c2)
(!!assert (dont-move c2))
(!!remove (put-on-table c2))
(!!assert (put-on-table c1))
(!unstack a3 a1)
(!putdown a3)
(!!assert (dont-move a3))
(!!remove (put-on-table a3))
(!!assert (stack-on-block b2 a1))
(!pickup b2)
(!stack b2 a1)
(!!assert (dont-move b2))
(!!remove (stack-on-block b2 a1))
(!unstack c1 c3)
(!putdown c1)
(!!assert (dont-move c1))
(!!remove (put-on-table c1))
(!!assert (stack-on-block c3 b2))
(!pickup c3)
(!stack c3 b2)
(!!assert (dont-move c3))
(!!remove (stack-on-block c3 b2))
--------------------

Plan cost: 25.0

(!!assert (goal (on-table a1)))
(!!assert (goal (on b2 a1)))
(!!assert (goal (on c3 b2)))
(!!assert (goal (clear c3)))
(!!assert (dont-move a1))
(!!assert (dont-move b2))
(!!assert (goal (on-table c2)))
(!!assert (goal (on-table c1)))
(!!assert (goal (on-table a3)))
(!!assert (put-on-table a3))
(!!assert (put-on-table c2))
(!unstack c2 c1)
```

```
    (!putdown c2)
93  (!!assert (dont-move c2))
    (!!remove (put-on-table c2))
95  (!!assert (put-on-table c1))
    (!unstack c1 c3)
97  (!putdown c1)
    (!!assert (dont-move c1))
99  (!!remove (put-on-table c1))
    (!!assert (stack-on-block c3 b2))
101 (!pickup c3)
    (!stack c3 b2)
103 (!!assert (dont-move c3))
    (!!remove (stack-on-block c3 b2))
105 (!unstack a3 a1)
    (!putdown a3)
107 (!!assert (dont-move a3))
    (!!remove (put-on-table a3))
109 --------------------

111 Plan cost: 29.0

113 (!!assert (goal (on-table a1)))
    (!!assert (goal (on b2 a1)))
115 (!!assert (goal (on c3 b2)))
    (!!assert (goal (clear c3)))
117 (!!assert (dont-move a1))
    (!!assert (dont-move b2))
119 (!!assert (goal (on-table c2)))
    (!!assert (goal (on-table c1)))
121 (!!assert (goal (on-table a3)))
    (!!assert (put-on-table c2))
123 (!!assert (put-on-table a3))
    (!unstack c2 c1)
125 (!putdown c2)
    (!!assert (dont-move c2))
127 (!!remove (put-on-table c2))
    (!!assert (put-on-table c1))
129 (!unstack a3 a1)
    (!putdown a3)
131 (!!assert (dont-move a3))
    (!!remove (put-on-table a3))
133 (!!assert (stack-on-block b2 a1))
    (!pickup b2)
135 (!stack b2 a1)
    (!!assert (dont-move b2))
137 (!!remove (stack-on-block b2 a1))
    (!unstack c1 c3)
139 (!putdown c1)
```

```
    (!!assert (dont-move c1))
141 (!!remove (put-on-table c1))
    (!!assert (stack-on-block c3 b2))
143 (!pickup c3)
    (!stack c3 b2)
145 (!!assert (dont-move c3))
    (!!remove (stack-on-block c3 b2))
147 --------------------

149 .... until 72 plans

151 Time Used = 0.117
```

B. HDL *ABox* Assertion

In this appendix, some of the HDL *ABox* assertions are presented. These assertions have to be made in order to put the knowledge in the HDL system. Corresponding representations are shown as figures within the chapters. The *shop2code* property is omitted, as this can be seen in the generated code in Appendix A.

B.1. Navigation Domain

The following lists show the assertions of the navigation domain. The navigation domain is detailed in Section 3.2 on page 42. Listing B.1 shows the operators' assertion. Listing B.2 shows the methods' assertion. Listing B.3 shows the planning-domain's assertion.

Listing B.1: Operators' assertion for "navigation_domain".

```
1 Operator(visit),
  Operator(unvisit),
3 Operator(drive-robot),
    useState(drive-robot,
5     '(at ?val1 ?val2);?val1=I:Robot,?val2=P:at')
```

Listing B.2: Methods' assertion for "navigation_domain".

```
1  Method(navigate),
     hasOperator(navigate, drive-robot),
3    hasOperator(navigate, visit),
     hasOperator(navigate, unvisit),
5    useState(navigate,
       '(room ?val1);?val1=I:Room')
7    useState(navigate,
       '(adjacentto ?val1 ?val2);?val1=I:Room,?val2=P:adjacentto')
9    useState(navigate,
       '(at ?val1 ?val2);?val1=I:Robot,?val2=P:at')
11 Method(navigate2),
     hasMethod(navigate2, navigate),
13   hasOperator(navigate2, visit),
     hasOperator(navigate2, unvisit),
15   useState(navigate2,
       '(at ?val1 ?val2);?val1=I:Robot,?val2=P:at')
```

Listing B.3: Planning domain's assertion for "navigation_domain".

```
Planning-Domain(navigation-domain),
2  hasMethod(navigation-domain, navigate),
   hasMethod(navigation-domain, navigate2),
4  hasOperator(navigation-domain, drive-robot),
   hasOperator(navigation-domain, visit),
6  hasOperator(navigation-domain, unvisit)
```

B.2. Pick-and-Place Domain

The following lists show the assertions of the pick-and-place domain. The pick-and-place domain is detailed in Section 3.4 on page 55. This domain consists of two implementations, the partial pick-and-place domain and the complete pick-and-place domain.

B.2.1. Partial Pick-and-Place Domain

The partial pick-and-place domain assertions are listed in five listings. Listing B.4 to B.6 show the methods' assertions, Listing B.7 shows the operators' assertion, and Listing B.8 shows the planning domains' assertion.

Listing B.4: Assertion of the methods "moveobject_p" and "moveobject2_p" for the "partial pick-and-place domain".

```
Method(moveobject_p),
2  hasMethod(moveobject_p, getobject_p),
   hasMethod(moveobject_p, putobject_p),
4  useState(moveobject_p,
     '(at ?val1 ?val2);?val1=I:Fruit,?val2=P:at')
6 Method(moveobject2_p),
   hasMethod(moveobject2_p, moveobject_p),
8  useState(moveobject2_p,
     '(at ?val1 ?val2);?val1=I:Fruit,?val2=P:at')
```

Listing B.5: Assertion of methods "getobject_p" and "getobject2_p" for the "partial pick-and-place domain".

```
1 Method(getobject_p),
   hasOperator(getobject_p, drive-away-f-dex-workspace),
3  hasOperator(getobject_p, drive-to-dexterous-workspace),
   hasOperator(getobject_p, pickup-object),
5  hasOperator(getobject_p, navigate-op),
   useState(getobject_p,
7    '(at ?val1 ?val2);?val1=I:Robot,?val2=P:at'),
   useState(getobject_p,
```

```
9       '(at ?val1 ?val2);?val1=I:Container,?val2=P:at'),
     useState(getobject_p,
11      '(at ?val1 ?val2);?val1=I:Fruit,?val2=P:at')
   Method(getobject2_p),
13   hasMethod(getobject2_p, getobject_p),
     useState(getobject2_p,
15      '(at ?val1 ?val2);?val1=I:Fruit,?val2=P:at')
```

Listing B.6: Assertion of the method "`putobject_p`" for the "partial pick-and-place domain".

```
1  Method(putobject_p),
     hasOperator(putobject_p, put-object),
3    hasOperator(putobject_p, drive-to-dexterous-workspace),
     hasOperator(putobject_p, navigate-op),
5    useState(putobject_p,
        '(at ?val1 ?val2);?val1=I:Container,?val2=P:at'),
7    useState(putobject_p,
        '(at ?val1 ?val2);?val1=I:Robot,?val2=P:at')
```

Listing B.7: Operator assertions for the "partial pick-and-place domain".

```
   Operator(navigate-op),
2  Operator(drive-to-dexterous-workspace),
     useState(drive-to-dexterous-workspace,
4       '(at ?val1 ?val2);?val1=I:Container,?val2=P:at'),
     useState(drive-to-dexterous-workspace,
6       '(at ?val1 ?val2);?val1=I:Robot,?val2=P:at'),
   Operator(drive-away-f-dex-workspace),
8    useState(drive-away-f-dex-workspace,
        '(at ?val1 ?val2);?val1=I:Container,?val2=P:at'),
10   useState(drive-away-f-dex-workspace,
        '(at ?val1 ?val2);?val1=I:Robot,?val2=P:at'),
12 Operator(pickup-object),
     useState(pickup-object,
14      '(at ?val1 ?val2);?val1=I:Fruit,?val2=P:at'),
   Operator(put-object),
16   useState(put-object,
        '(at ?val1 ?val2);?val1=I:Fruit,?val2=P:at')
```

Listing B.8: Planning domain assertions for "partial pick-and-place domain".

```
1  Planning-Domain(pick-and-place_domain_partial),
     hasMethod(pick-and-place_domain_partial, moveobject_p),
3    hasMethod(pick-and-place_domain_partial, moveobject2_p),
     hasMethod(pick-and-place_domain_partial, getobject_p),
5    hasMethod(pick-and-place_domain_partial, getobject2_p),
     hasMethod(pick-and-place_domain_partial, putobject_p),
```

```
7  hasOperator(pick-and-place_domain_partial, navigate-op),
   hasOperator(pick-and-place_domain_partial,
9    drive-to-dexterous-workspace),
   hasOperator(pick-and-place_domain_partial,
11   drive-away-f-dex-workspace),
   hasOperator(pick-and-place_domain_partial, pickup-object),
13 hasOperator(pick-and-place_domain_partial, put-object)
```

B.2.2. Complete Pick-and-Place Domain

Two listings for the complete pick-and-place domain are shown in this section. Listing B.9 shows the methods' assertions and Listing B.10 shows the planning domain's assertion.

Listing B.9: Methods assertions for "complete-pick-and-place-domain".

```
1  Method(moveobject_c),
     hasMethod(moveobject_c,getobject_c),
3    hasMethod(moveobject_c, putobject_c),
     useState(moveobject_c,
5      '(at ?val1 ?val2);?val1=I:Fruit,?val2=P:at')
   Method(moveobject2_c),
7    hasMethod(moveobject2_c, moveobject_c),
     useState(moveobject2_c,
9      '(at ?val1 ?val2);?val1=I:Fruit,?val2=P:at')
   Method(getobject_c),
11   hasMethod(getobject_c, navigate2),
     hasOperator(getobject_c, drive-away-f-dex-workspace),
13   hasOperator(getobject_c, drive-to-dexterous-workspace),
     hasOperator(getobject_c, pickup-object),
15   useState(getobject_c,
       '(at ?val1 ?val2);?val1=I:Robot,?val2=P:at'),
17   useState(getobject_c,
       '(at ?val1 ?val2);?val1=I:Container,?val2=P:at'),
19   useState(getobject_c,
       '(at ?val1 ?val2);?val1=I:Fruit,?val2=P:at')
21 Method(getobject2_c),
     hasMethod(getobject2_c, getobject_c),
23   useState(getobject2_c,
       '(at ?val1 ?val2);?val1=I:Fruit,?val2=P:at')
25 Method(putobject_c),
     hasOperator(putobject_c, navigate2),
27   hasOperator(putobject_c, put-object),
     hasOperator(putobject_c, drive-to-dexterous-workspace),
29   useState(putobject_c,
       '(at ?val1 ?val2);?val1=I:Container,?val2=P:at'),
31   useState(putobject_c,
       '(at ?val1 ?val2);?val1=I:Robot,?val2=P:at')
```

Listing B.10: Planning domain assertion for "complete-pick-and-place-domain".

```
Planning-Domain(pick-and-place_domain_complete),
  hasMethod(pick-and-place_domain_complete, moveobject_c),
  hasMethod(pick-and-place_domain_complete, moveobject2_c),
  hasMethod(pick-and-place_domain_complete, getobject_c),
  hasMethod(pick-and-place_domain_complete, getobject2_c),
  hasMethod(pick-and-place_domain_complete, putobject_c),
  hasMethod(pick-and-place_domain_complete, navigate),
  hasMethod(pick-and-place_domain_complete, navigate2),
  hasOperator(pick-and-place_domain_complete, drive-robot),
  hasOperator(pick-and-place_domain_complete, visit),
  hasOperator(pick-and-place_domain_complete, unvisit),
  hasOperator(pick-and-place_domain_complete,
    drive-to-dexterous-workspace),
  hasOperator(pick-and-place_domain_complete,
    drive-away-f-dex-workspace),
  hasOperator(pick-and-place_domain_complete, pickup-object),
  hasOperator(pick-and-place_domain_complete, put-object)
```

B.3. Blocks World Domain

Below is a description of the helper methods. The `move-block1` method is for improving efficiency by avoiding multiple calculations. This is an example of a general technique in SHOP. In the case of several possible decompositions for a task with common preconditions, one might add another hierarchy in the task level to factorise these preconditions. This method has two cases defined as follows:

`(move-block1 ?x ?z)` *;; method for moving x from on top of y to on top of z*

task: `move-block1(x,z)`

subtasks: $u_1 =$ `!unstack(x,y)`
$u_2 =$ `!stack(x,z)`
$u_3 =$ `!assert((dont-move ?x))`
$u_4 =$ `!remove((stack-on-block ?x ?z))`
$u_5 =$ `check(x)`
$u_6 =$ `check2(y)`
$u_7 =$ `check3(y)`

constr.: $u_1 \prec u_2$, $u_2 \prec u_3$,$u_3 \prec u_4$, $u_4 \prec u_5$, $u_5 \prec u_6$, $u_6 \prec u_7$, `on(x,y)`

`(move-block1 ?x ?z)` *;; method for moving x from table to on top of z*

task: `move-block1(x,z)`

subtasks: $u_1 =$ `!pickup(x)`
$u_2 =$ `!stack(x,z)`

$u_3 =$ `!assert((dont-move ?x))`

$u_4 =$ `!remove((stack-on-block ?x ?z))`

$u_5 =$ `check(x)`

constr.: $u_1 \prec u_2, u_2 \prec u_3, u_3 \prec u_4, u_4 \prec u_5$

Three helper methods are used to test the facts after performing the `move` methods. The first one is `check`. It tests if another block y is ready to be put on block x. It is called whenever block x is moved to its final position. This method is defined as follows:

`(check ?x)` *;; method to check if any block can be put on x*

task: `check(x)`

subtasks: $u_1 =$ `!assert((stack-on-block ?y ?x))`

constr.: `goal(on(y,x)),clear(y)`

The second test method is `check2`. It performs a similar test as the `check` method. However, it is called after a block is removed from on top of block x. It is defined as follows:

`(check2 ?x)` *;; method to check if any block to put on x*

task: `check2(x)`

subtasks: $u_1 =$ `!assert((stack-on-block ?y ?x))`

constr.: `dont-move(x),goal(on(y,x)),clear(y)`

The third test method is `check3`. It tests whether block x can go to its final position. It is called whenever something had been removed from on top of block x. This method is implemented through three cases, which are defined as follows:

`(check3 ?x)` *;; method to test whether x is movable*

task: `check3(x)`

subtasks: $\emptyset$

constr.: `dont-move(x)`

`(check3 ?x)` *;; method to test whether x can be stacked on y*

task: `check3(x)`

subtasks: $u_1 =$ `!assert((stack-on-block ?x ?y))`

constr.: `goal(on(x,y)),clear(y),dont-move(y)`

`(check3 ?x)` *;; method to test whether x can be put on table*

task: `check3(x)`

subtasks: $u_1 =$ `!assert((put-on-table ?x))`

constr.: `goal(on-table(x))`

Listing B.11 shows the planning domain's assertion for the blocks world domain. Details of this assertion are presented in Section 5.2 on page 103.

Listing B.11: Planning domain assertion for the "blocks-world" domain.

```
Planning-Domain(blocks_domain),
  hasMethod(blocks_domain, achieve-goal),
  hasMethod(blocks_domain, assert-goals),
  hasMethod(blocks_domain, assert-goals-nil),
  hasMethod(blocks_domain, add-new-goals),
  hasMethod(blocks_domain, find-nomove),
  hasMethod(blocks_domain, find-moveable),
  hasMethod(blocks_domain, move-block),
  hasMethod(blocks_domain, move-block1),
  hasMethod(blocks_domain, check),
  hasMethod(blocks_domain, check2),
  hasMethod(blocks_domain, check3),
  hasMethod(blocks_domain, axiom-need-to-move),
  hasMethod(blocks_domain, axiom-same),
  hasOperator(blocks_domain, pickup),
  hasOperator(blocks_domain, putdown)
  hasOperator(blocks_domain, stack),
  hasOperator(blocks_domain, unstack),
  hasOperator(blocks_domain, assert)
  hasOperator(blocks_domain, remove)
```

C. Software Engineering

The HDL system was implemented in Java using the Netbean IDE [Mya08]. It is implemented as a Netbean's rich client application. The system is called the HDL Suite. This gives a brief overview of the Suite through snapshots and a series of UML diagrams. The HDL Suite contains several modules, the two most important of which are the OWL Module and the Planner Module. However, there are also several supporting modules that make the suite a user friendly system. These modules are detailed in Appendix C.2. This Suite is implemented completely in Java™. Thus, it can run on any system with a Java™ 5 or above installation.

C.1. HDL Plan Suite

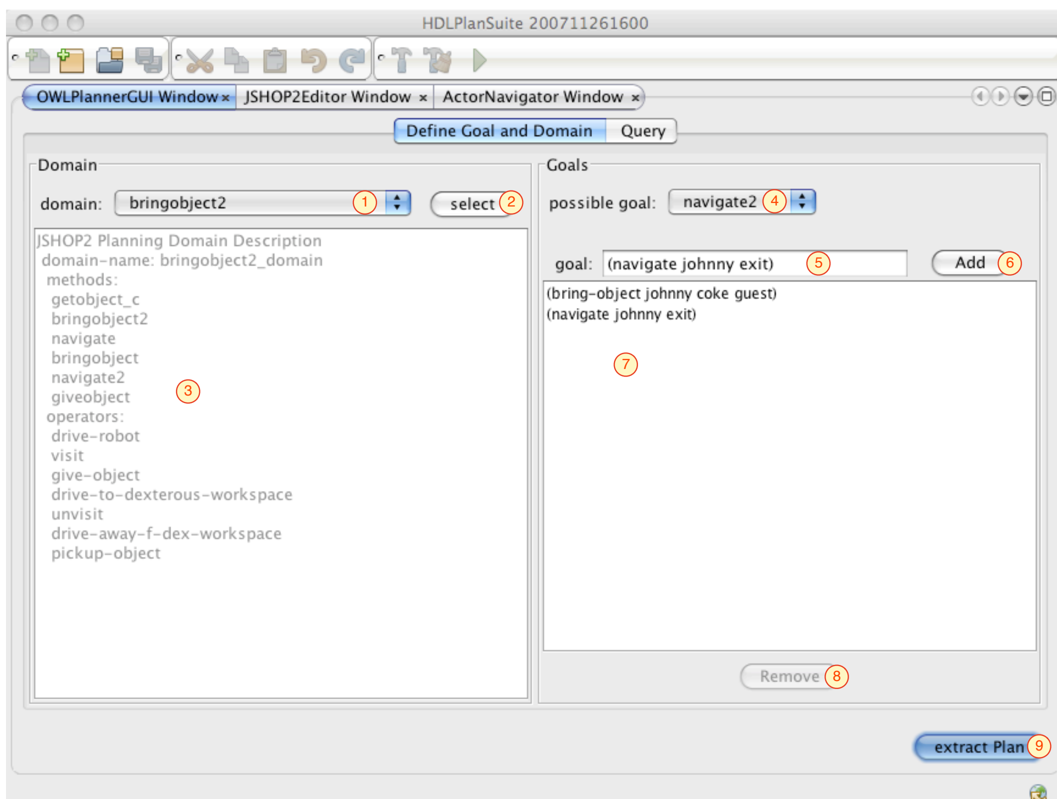


Figure C.1.: Snapshot of the OWLPlannerGUI.

Figure C.1 shows the snapshot of the OWLPlannerGUI of the HDL-Suite. This window consists of two frames; the left side is taken up by the domain chooser and the right side, for the

goal editor. An explanation of the numbered markers seen in Figure C.1 follows.

1. *Domain chooser:*

   Using this combo bar, the user can choose from the available domains in the DL model. The domains are either instances of a class $Method$ or $Planning\text{-}Domain$.

2. *Domain select button:*

   This button is used to commit the selection, after which the relevant methods and operators for the selected domain are queried from the DL model.

3. *Domain viewer:*

   This viewer shows the methods and operators for the selected domain. The content is filled after the select button is pressed.

4. *Goal chooser:*

   After the domain is selected, not only is the domain viewer filled out with the data, but the goal chooser also. The content of the goal chooser depends on the selected domain and will be refreshed when a new domain has been selected.

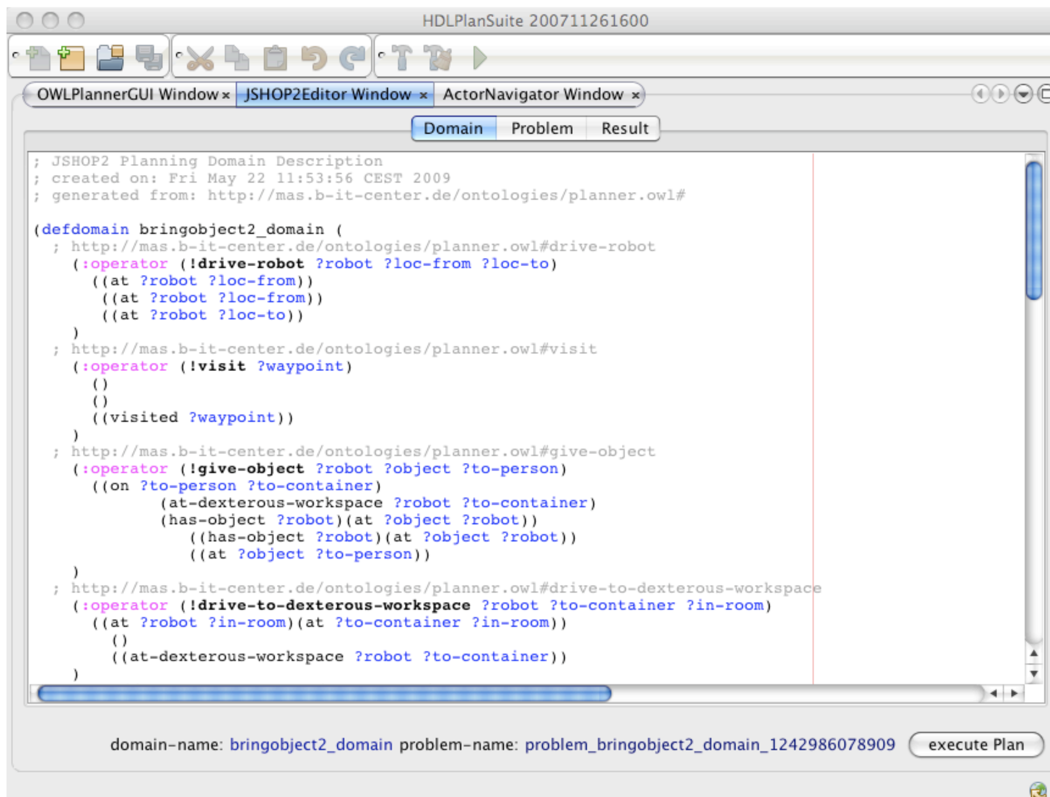


Figure C.2.: Snapshot of the JSHOP2Editor for editing the domain.

5. *Goal editor:*

   After choosing one of the goals, the goal editor will be filled up with the selected goal.

The user has to edit the variable with the real value, e.g. `(navigate johnny exit)`.

6. *Goal add button:*
   Once the desired goal is filled in the goal editor, the user can press the goal add button. Once this button is pressed, the goal will be added into the goal lists.

7. *Goal list viewer:*
   The goal list viewer shows all the objectives for the given planning domain.

8. *Goal remove button:*
   This button is used for deleting entries from the goal list. Thus, the user can make corrections.

9. *Extract plan button:*
   Once the domain is selected and the plan objectives have been defined, this button can be pressed to let the HDL system extract the plans.

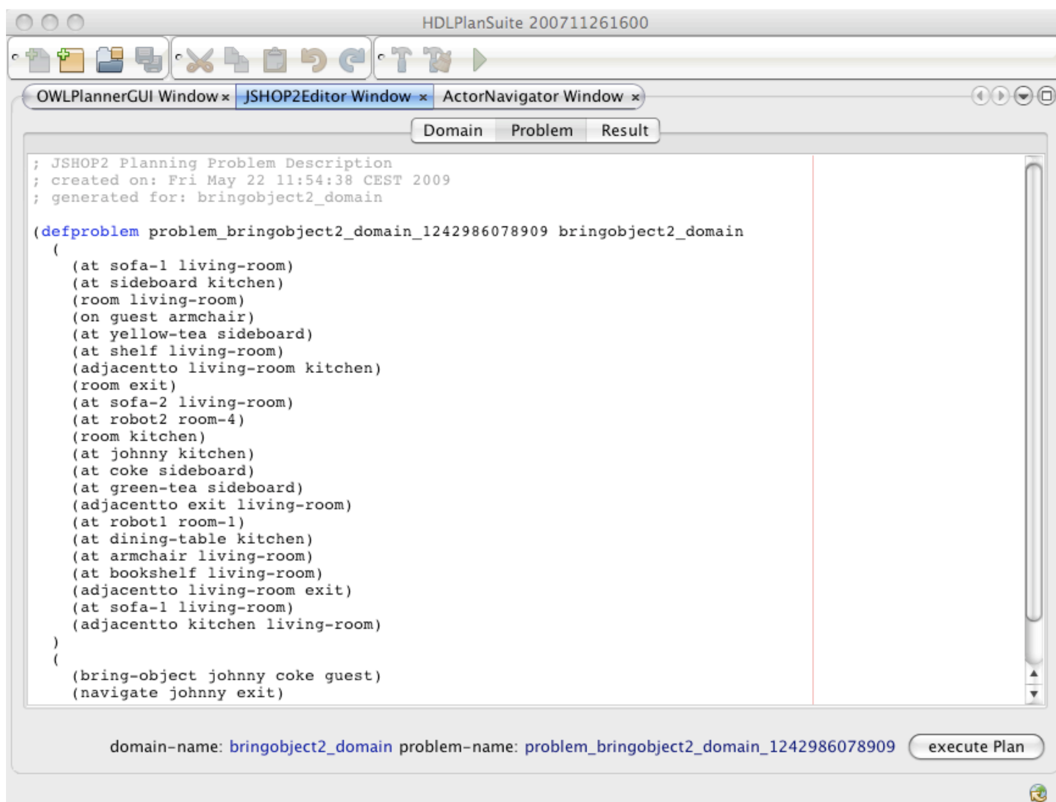


Figure C.3.: Snapshot of the JSHOP2Editor for editing the domain.

Figure C.2 depicts the JSHOP2 editor window. This window is used for viewing and editing the generated domain. Once the extract plan button is pressed, the HDL system generates the planning domain and planning problem. This editor supports the standard editor for programming, such as syntax highlighting, bracket counting, syntax colouring, etc. The user can

also edit the planning domain, to test a different behaviour or new methods for example. If this window is visible, then the planning process has been completed. However, one might trigger the planning process manually after editing the planning domain by pressing the “execute Plan” button.

Figure C.3 depicts the JSHOP2 editor for the planning problem. The content is filled by the HDL system after reasoning about its model for the given planning domain. As with the planning domain editor, it supports standard programming editing styles. One might also edit the planning problem. Thus, it is very useful for developing and testing new planning problems or domains. The JSHOP2Editor can also be used for HTN planning programming. One can write the planning domain in the domain editor and the problem in its editor. The planning process is then triggered by pressing the execute plan button.

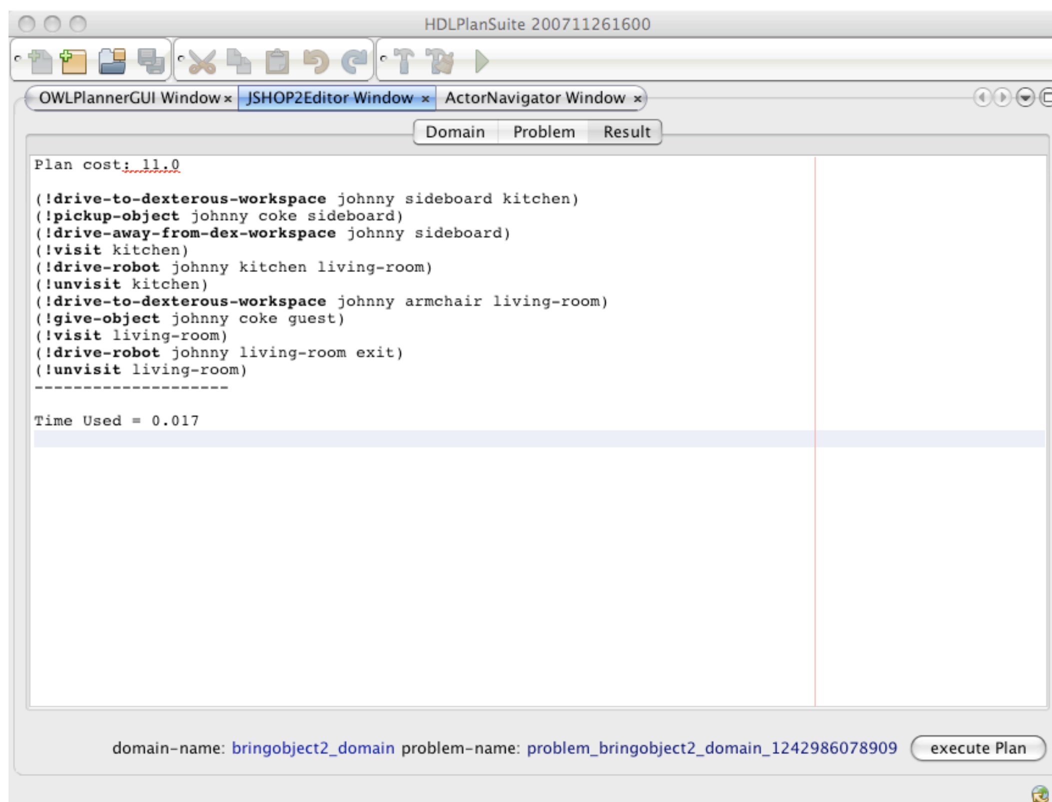


Figure C.4.: Snapshot of the JSHOP2Editor for displaying the planning results.

Figure C.4 shows the result window of the JSHOP2Editor. This window displays the result of the HTN planning for the given planning problem in the problem editor window. The same behaviours as the domain and problem editors are applied here. However, the content is not editable.

Figure C.5 depicts the DL model viewer. The DL model is shown as a tree. Where the classes are shown with a circle icon with a ’C’ inside. An individual is shown as a diamond icon with an ’I’ inside. Using the viewer, the user can browse the content and structure of the

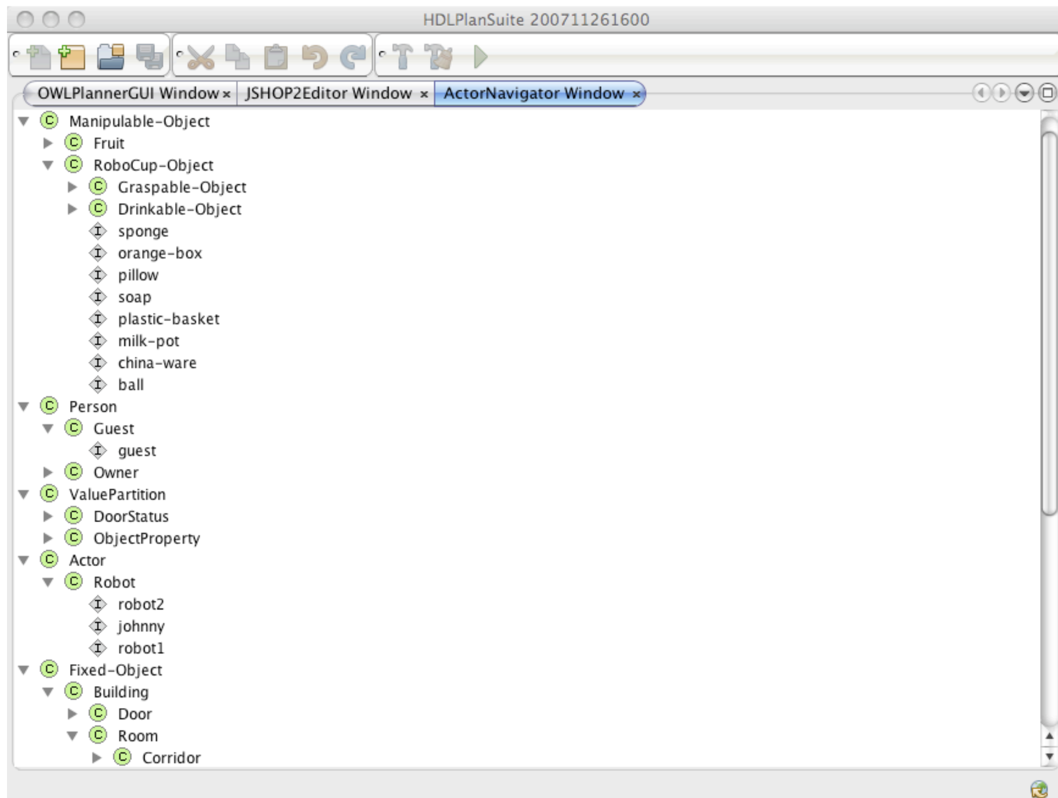


Figure C.5.: Snapshot of the ActorNavigator window.

model. In this version, the user can also edit the properties of the *Actor* class.

C.2. UML Diagrams

The HDL Suite has 18 modules in all. These modules consist of 12 external libraries and 6 HDL implementation modules. The HDL modules are explained in detail in the following sections. The external libraries are required in order to work with the DL reasoning or JSHOP2 planning systems. These libraries are enumerated as follows:

1. ATerm Library
2. Apache Common-IO
3. JAF
4. Java Mail API
5. Jena Toolkit
6. Jetty Library
7. JMS API
8. JSHOP2 API
9. OWL-API Library

10. Pellet Library
11. Sun-IO
12. Sun Multi Schema Validator

C.2.1. HDL Plan Option Module

The HDL plan option module is a module that is responsible for showing the options windows. There are two options windows in the HDL suite; the OWL options window and the JSHOP2 options window. The OWL options window is shown in Figure C.6. With this window, the user can choose the OWL source file either by typing it into the text editor pane or choosing from the browser window. In addition, the OWL URI can also be edited in this window. For the HTN planning domain, the user can edit the namespace of this domain. The JSHOP2 option window is shown in Figure C.7. The option window offers only one property to edit, namely the planner's working directory. The JSHOP2 planner generates some java files for its planning purpose. This directory is where the generated files should be placed.

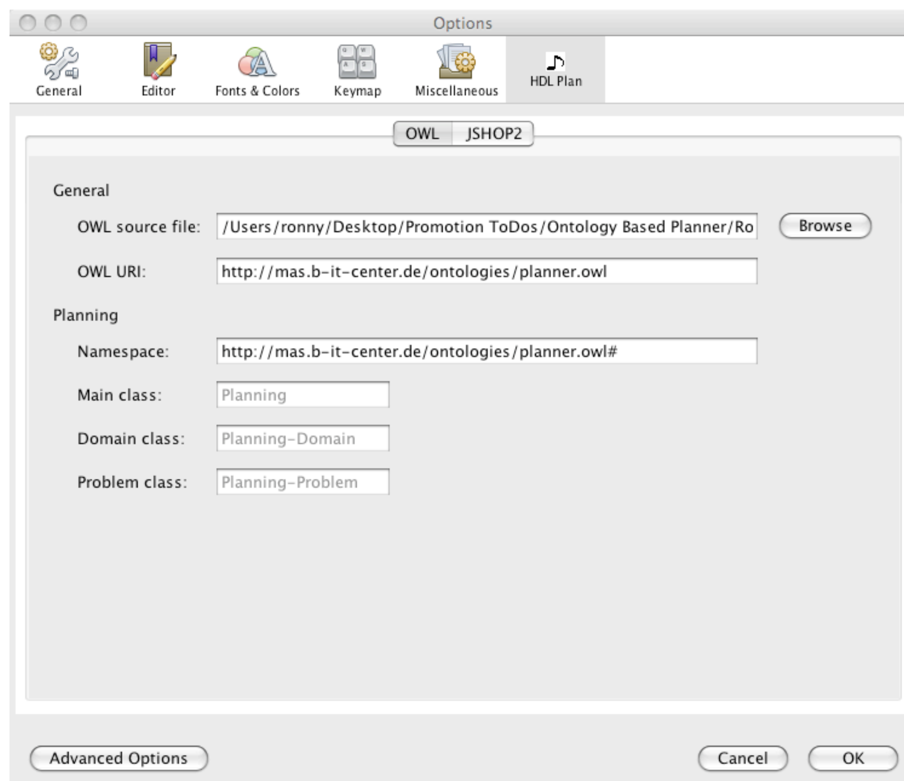


Figure C.6.: Snapshot of the OWL options window.

Figure C.8 shows the UML diagram of the HDLPlan-Option. It shows that the main window is inherited from the `JPanel` class. There are two panels, that implements the `ActionListener` from the java.awt.event. The `GenericFileFilter` inherits from the `FileFilter`. Its function in the options panel is to browse the file system. There are two more classes, namely

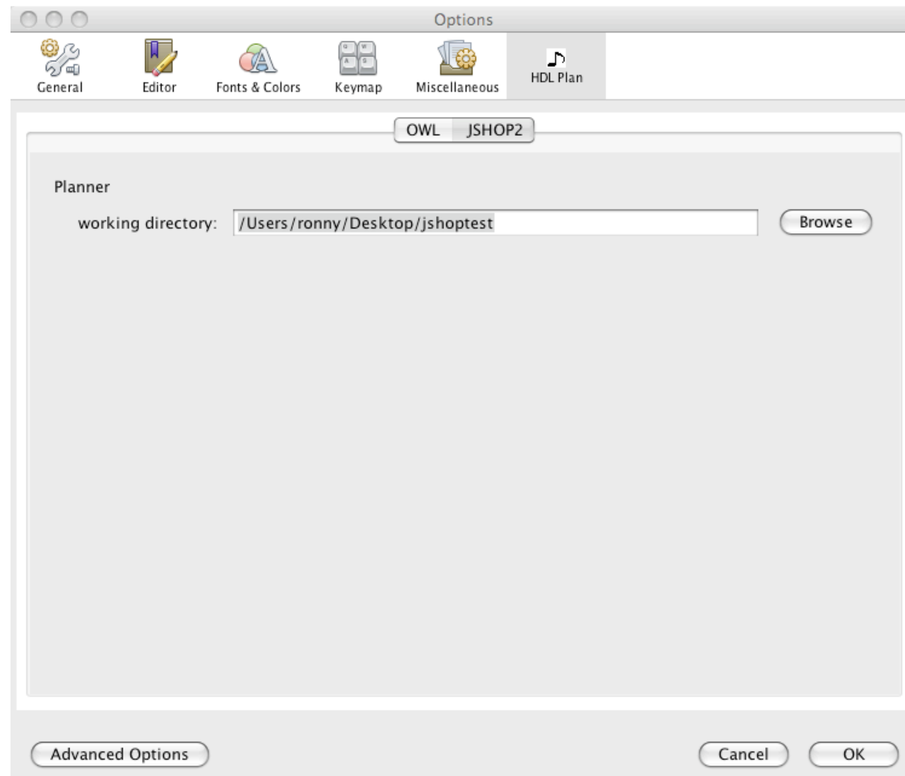


Figure C.7.: Snapshot of the JSHOP2 options window.

`OptionCategory` and `OptionPanelController` which support the functionality of these panels.

C.2.2. OWL Actor Editor Module

Figure C.9 shows the OWL-Actor-Editor UML. The editor window is shown in Figure C.5. The main window in the Netbean rich client platform is implemented as a sub-class from `TopComponent` class. In this window, it is `ActorNavigatorTopComponent`. This editor is implemented using the model-view-controller (MVC) paradigm. The model is represented by the classes `Actor`, `Instance`, and `Concept`. The view is represented with the classes `InstanceNode` and `ConceptNode`. The controller is implemented by the class `ActorNavigatorAction` and with the help of `ConceptChildren` class.

C.2.3. OWL Module

OWL module is the interface to the DL reasoning system. Its purposes are to read the model, reason about it and apply HDL's algorithms to the inferred model. The outputs of this module are the planning domain and planning problem.

Figure C.10 shows the UML diagram for this module. The DL model is defined in the class `OWLModel` that inherits the `OWLModelI` interface. The `OWLModelI` interface defines

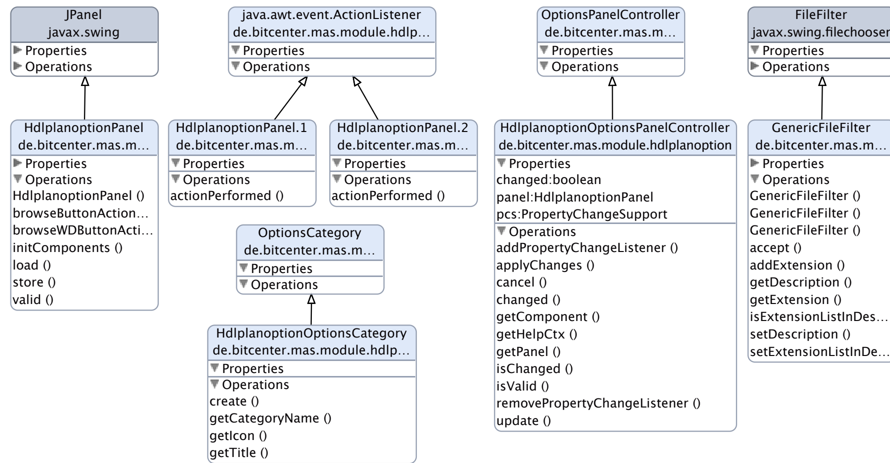


Figure C.8.: HDLPlan-Option UML.

the necessary methods to describe the OWL model, such as `getNS_Prefix`, `getOntModel`, `setNS_Prefix`, and `setOntModel`. The class `OWLPlanningReasoner` provides the connection to the Pellet engine. The queried results are represented within the class `OWLQueryResults`. The `OWLListener` provides the methods for manipulating the model.

In addition to the OWL classes that have been mentioned previously, some classes for JSHOP2 planning are also defined. The `JSHOP2OWLPlanning` class contains the algorithms for reasoning about the model. The `JSHOP2Goal` is the class that is responsible for querying and storing the planning objectives. The `JSHOP2States` queries the model and stores the relevant states for the defined planning problem in the `JSHOP2ProblemDescription`. Two classes are defined for the planning domain and planning problem, namely `JSHOP2DomainDesc` and `JSHOP2ProblemDesc`. The `JSHOP2DomainDesc` class receives its contents from `JSHOP2OWLPlanning` by applying the reasoners' and the HTN's algorithms. The `JSHOP2ProblemDesc` class uses the information in the `JSHOP2DomainDesc`, `JSHOP2Goal`, and `JSHOP2States` for its planning problem. Both classes, `JSHOP2DomainDesc` and `JSHOP2ProblemDesc`, can generates the planning domain and planning problem in SHOP syntax. The `JSHOP2StringFormatter` is the helper class for this code generation process.

C.2.4. OWLPlanner GUI Module

Figure C.11 shows the main graphical user interface (GUI) of the HDL suite. The main window is represented by the `OWLPlannerGUITopComponent`. The main window is the placeholders for the other windows, that have previously been shown as snapshots (Figure C.1 to Figure C.5). These are represented by the `OWLPlannerGUITopComponent.x` (x is 1 to 6). In addition, two helper classes are used by these GUIs, namely `GoalListModel` and `OWLPlannerGUIAction`.

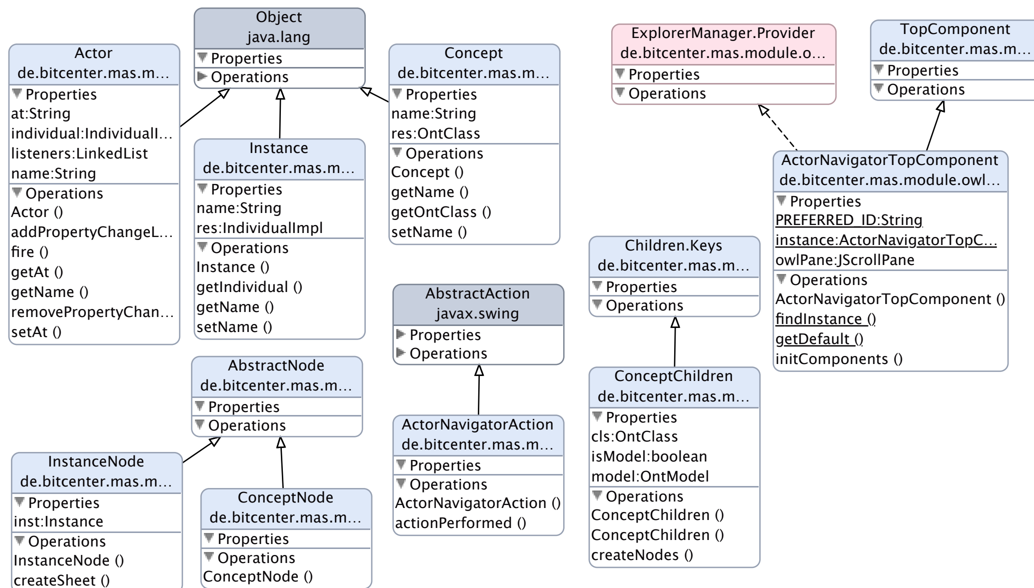


Figure C.9.: OWL-Actor-Editor UML.

C.2.5. Planner Module

The planner module is the interface component to the JSHOP2 planner. It encapsulates JSHOP2 in such a way that the HTN planner can get direct access to its API. Usually, one should execute the planner from the console with the planning domain and planning problem as its parameters. However, in HTN, the planner has to be integrated such that the overall system is coherent.

Figure C.12 shows the UML diagram of the planner module. In this diagram, there are two different parts. The first part is the connection to the JSHOP2 editor window. Although in the main GUI window its already defined, the real contents are fetched directly through this module. The classes that are responsible for the JSHOP2Editor are `JSHOP2EditorTopComponent`, `JSHOP2EditorAction`, and their key listener and action listener.

The interface that connects to the planner is the `JSHOP2Planner`. It implements the interface `PlannerI`. This interface is the generic implementation of a planner, where the necessary methods for executing the planner are defined. These methods are `executePlanner`, `setDomain`, and `setProblem`. These three methods are needed by any planner. Thus, the system is expandable, in case a new planner should be integrated within the HDL system. One has to write an interface to the new planner by implementing the `PlannerI` interface.

As explained previously in Section 6.2 on page 121, the java class loader will not load any class which has previously been loaded. Therefore, in our implementation, the `ClassLoader` class has to be customised. This customisation is done with the following classes: `PlannerClassLoader`, `CompilingClassLoader`, `CustomClassLoader`, and `ByteClassLoader`. These classes are responsible for compiling java codes and loading it, even if the classes have previously been loaded.

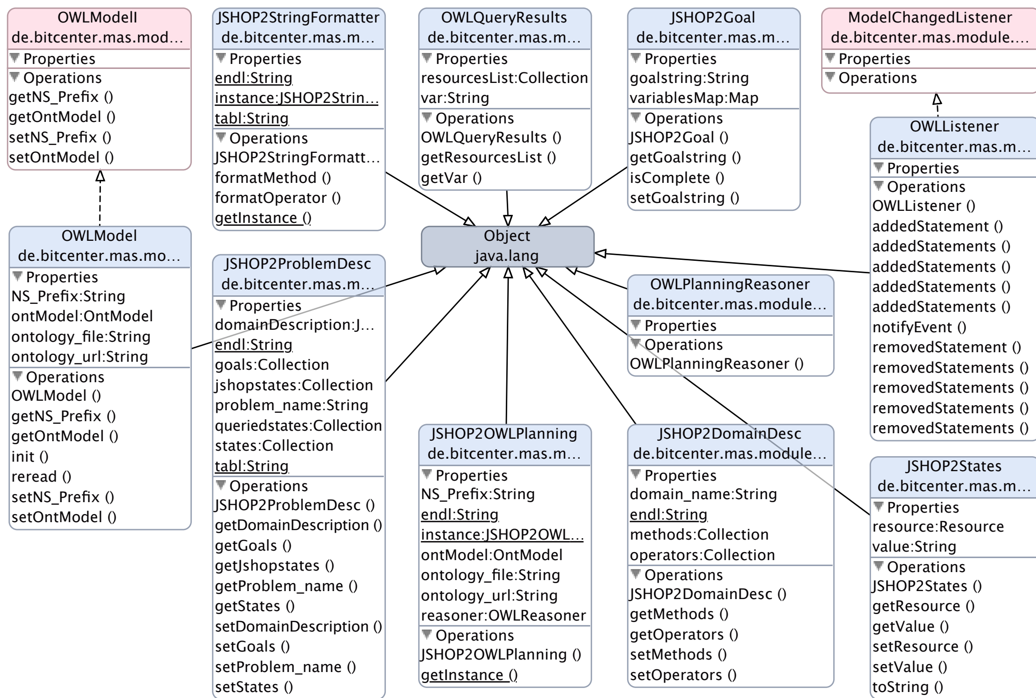


Figure C.10.: OWL module's UML diagram.

C.2.6. SHOP Support Module

The "SHOP support" module serves as the SHOP lexer and language parser. It enables the user to use syntax colouring or highlighting and code completion for writing SHOP files. Basically, it uses the ".nbs" SchliemannNBSLanguageDescription. It is based on the Extended Backus-Naur Form (BNF). The SHOP syntax is extracted from the JSHOP2 documentation [Ilg06]. Therefore, the JSHOP2 editors use this definition to parse and display the SHOP code.

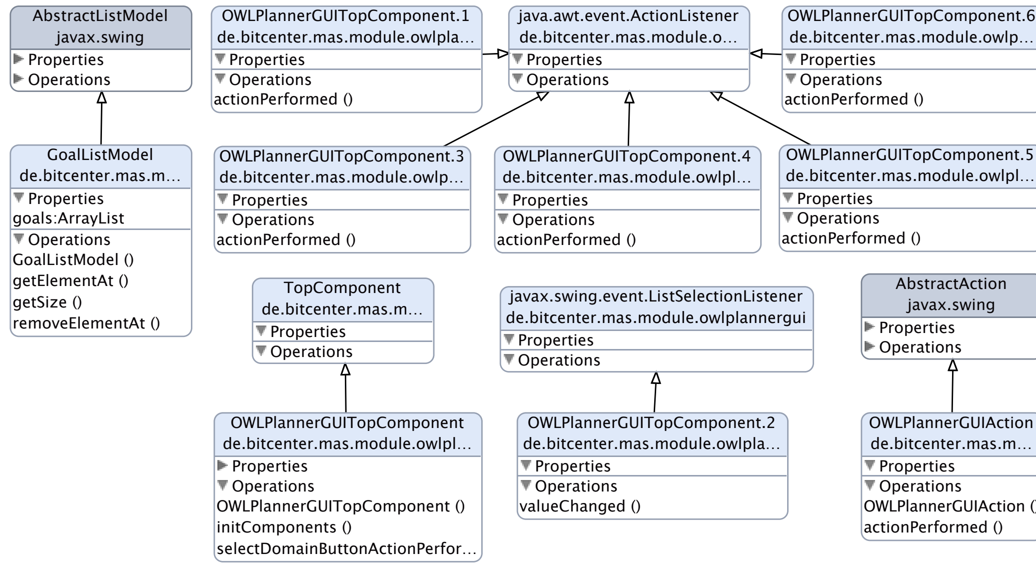


Figure C.11.: OWL Planner GUI UML diagram.

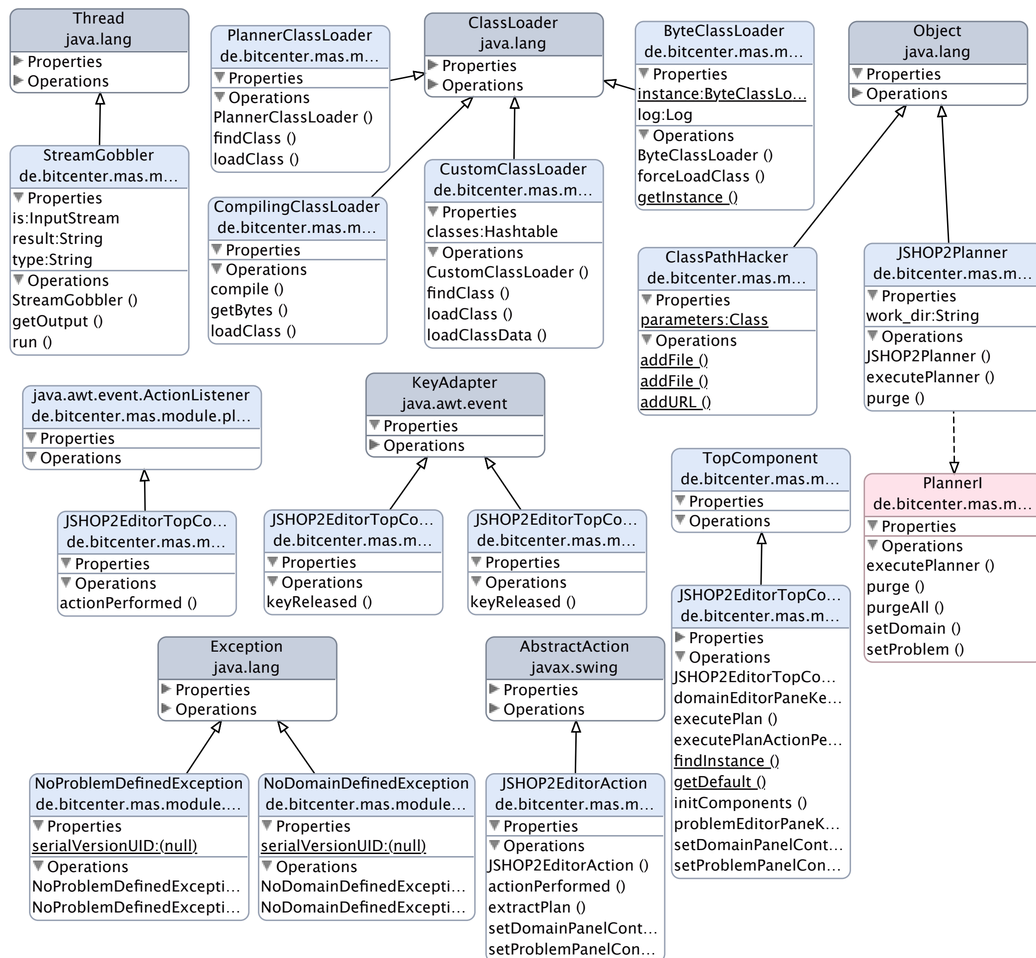


Figure C.12.: Planner module UML diagram.