Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Designing and implementing usable, interoperable, and reusable services of AI planning capabilities

Sebastian Graef

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. Marco Aiello
Supervisor:	Dr. Ilche Georgievski
Commenced:	June 15, 2020
Completed:	December 15, 2020

Abstract

Artificial Intelligence (AI) has become an essential part of our globalized world over the last years, with applications ranging from laboratory software to autonomous cars and space missions. Since the challenges to be solved by AI Planning are becoming more complex and versatile, decomposition of problems and outsourcing planning steps offers a possible way out. However, there is a lack of interoperability and reusability of AI planning capabilities. Planners and planning systems are often overloaded to meet the requirements, which results in a lack of usability. Thus, developers are forced to dig into the theory and planner details to use these existing systems.

This thesis investigates how planning capabilities need to be designed to be usable, interoperable, and reusable. It presents a novel architectural approach to create abstract and domain-independent planning capabilities. Through literature research, the typical planning capabilities were identified and then classified. Two metrics were developed to classify capabilities, each focusing on different aspects of the capabilities and their composition. Based on the findings, requirements were derived that must be met to optimize usability, interoperability, and reusability. Since one classification metric is based on the *Enterprise Integration Patterns*, the use of a Service-oriented Architecture (SOA) is recommended. This architecture approach offers a platform solution of planning capabilities as a service. Through messaging, the classified capabilities can be integrated according to pipes and filter based engineering patterns.

This thesis also includes a prototype of the approach, representing a minimal subset of the capabilities. Using the prototype, it is possible to model a domain and a problem in a Web application with the Planning Domain Definition Language (PDDL) and create a sequential plan.

The prototype shows that it is possible to integrate AI planning capabilities into SOA to make them usable, interoperable, and reusable. However, the transformation of existing planners to planning capabilities can lead to difficulties in slicing and serializing data structures. The presented approach allows universal use without the need to define specific standard interfaces. The architecture allows a planning capability to have multiple service instances and thus provide different interfaces.

Kurzfassung

Künstliche Intelligenz (KI) ist im Laufe der letzten Jahre zu einem wesentlichen Bestandteil unserer globalisierten Welt geworden, mit Anwendungen, die von Laborsoftware bis hin zu autonomen Autos und Weltraummissionen reichen. Da die von KI Planung zu lösenden Herausforderungen immer komplexer und vielseitiger werden, bietet die Dekomposition von Problemen und die Auslagerung von Planungsschritten einen möglichen Ausweg. Allerdings mangelt es an Interoperabilität und Wiederverwendbarkeit von KI Planungsfähigkeiten. Planer und Planungssysteme sind oft überladen, um den Anforderungen gerecht zu werden, wodurch es zu einem Mangel an Benutzerfreundlichkeit kommt. Daher sind die Entwickler gezwungen, sich tief in die Theorie und die Details der Planer zu vertiefen, um diese bestehenden Systeme zu nutzen.

In dieser Arbeit wird untersucht, wie Planungsfähigkeiten gestaltet werden müssen, damit sie nutzbar, interoperabel und wiederverwendbar sind. Es wird ein neuartiger Architekturansatz zur Schaffung abstrakter und domänenunabhängiger Planungsfähigkeiten vorgestellt. Mittels Literaturrecherche wurden die typischen Planungsfähigkeiten identifiziert und dann klassifiziert. Zur Klassifizierung der Fähigkeiten wurden zwei Metriken entwickelt, die sich jeweils auf verschiedene Aspekte der Fähigkeiten und ihrer Zusammensetzung konzentrieren. Auf der Grundlage der Ergebnisse wurden Anforderungen abgeleitet, welche erfüllt werden müssen, um die Nutzbarkeit, Interoperabilität und Wiederverwendbarkeit zu optimieren. Da eine Klassifizierungsmetrik auf den *Enterprise Integration Patterns* basiert, wird die Verwendung einer SOA empfohlen. Dieser Architekturansatz bietet eine Plattformlösung von Planungsfähigkeiten als Services. Durch Messaging können die klassifizierten Fähigkeiten nach dem technisches Muster von “Pipes and Filters” integriert werden.

Im Rahmen dieser Arbeit wurde auch ein Prototyp des Ansatzes entwickelt, der eine minimale Teilmenge der Fähigkeiten darstellt. Mit dem Prototyp ist es möglich, eine Domäne und ein Problem in einer Web-Anwendung mittels PDDL zu modellieren und einen Ablaufplan zu erstellen.

Der Prototyp zeigt, dass es möglich ist, KI Planungsfähigkeiten in eine SOA zu integrieren, um sie nutzbar, interoperabel und wiederverwendbar zu machen. Die Transformation vorhandener Planer in Planungsfähigkeiten kann jedoch zu Schwierigkeiten beim Aufteilen und Serialisieren von Datenstrukturen führen. Der vorgestellte Ansatz ermöglicht einen universellen Einsatz ohne die Notwendigkeit, spezifische Standardschnittstellen zu definieren. Die Architektur erlaubt es einer Planungsfähigkeit, mehrere Service-Instanzen zu haben und somit unterschiedliche Schnittstellen bereitzustellen.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ADL Action Description Language. 28

AI Artificial Intelligence. 3

ANML Action Notation Modeling Language. 13

BNF Backus-Naur-Form. 78

CNF Conjunctive Normal Form. 24

CPEF Continuous Planning and Execution Framework. 35

CPS Cyber-Physical System. 43

CSP Constraint Satisfaction Problem. 24

ESA European Space Agency. 38

FaaS Function as a Service. 127

Fdr Finite Domain Representation. 96

HDDL Hierarchical Domain Definition Language. 13

HPDL Hierarchical Planning Definition Language. 31

HTN Hierarchical Task Network. 30

ICAPS International Conference on Automated Planning and Scheduling. 78

IDE Integrated Development Environment. 26

IPC International Planning Competition. 28

LAPKT Lightweight Automated Planning ToolKiT. 43

MDP Markov decision process. 22

MOM Message-Oriented Middleware. 10

NDDL New Domain Definition Language. 32

PDDL Planning Domain Definition Language. 3

PELEA a Domain-Independent Architecture for Planning, Execution and Learning. 37

POMDP partially observable MDPs. 22

RDDL Relational Dynamic Influence Diagram Language. 25

REST Representational State Transfer. 72

RQ	Research Question. 18
SAT	Satisfaction Problem. 24
SHOP	Simple Hierarchical Ordered Planner. 30
SOA	Service-oriented Architecture. 3
SOC	Service-oriented Computing. 17
STRIPS	Stanford Research Institute Problem Solver. 24
XML	Extensible Markup Language. 37

1 Introduction

These days, planning problems tend to become more and more complex. AI is supporting developers in dealing with complex topics for several years now. The rise in complexity is due to globalization and an increasing number of dependencies.

While AI has gained popularity in many areas of software engineering, for instance, on the field of image processing, comparatively few approaches exist in the field of AI Planning [FPD13]. The lack of encapsulated, abstract, and standardized capabilities that are needed to solve real planning problems is causing this difference. Consequently, AI planning has a low level of acceptance in the industry. Furthermore, the integration of planning in complex applications is difficult and only partially effective [FPD13]. The composition of multiple planners is necessary to solve complex planning problems, which can be relaxed to a subset of underlying problems. Furthermore, AI planning requires various capabilities(1. parsing 2. modelling 3. planning 4. validation 5. converting 6. execution 7. monitoring) to address the problem [Geo15]. The composition of these various capabilities is the entering-point of Service-oriented Computing (SOC). An SOA forces standardized interfaces to enable interoperability [PG03].

Since most AI planning systems are implemented standalone, the redundancy of modules like parsers is apparent [GKP20]. Through a SOA, a platform can be created which solves this redundancy and allows the integration of different planning capabilities.

1.1 Problems

This section summarizes the identified challenges. Below is a list of the individual shortcomings:

Lack of Classification

Planning services are usually only implemented for a specific type of planning requests. However, there is a large number of different tasks and entities [Geo15]. The selection of a satisfactory service is necessary, which requires a classification of these services to select the correct planning service to solve the problem.

Lack of Interoperability

Most real-world planning problems are too complex to be solved by only a planning capability. A decomposition of the problem to subproblems and tasks is necessary (according to the System-of-systems paradigm [FMJB17]).

Lack of Reusability

Reusability can be applied to different applications, but also among planning capabilities. For instance, many planners require PDDL parser. While some implement their parsers, others integrate full planners to use the parsing functionality [GKP20].

Lack of Usability To integrate the planning application, a developer has to get bogged down in theory, and details [GKP20]. As the systems are often overloaded to meet the requirements, complexity increases. This behavior further reduces the level of usability.

1.2 Idea

The problems mentioned on Section 1.1 are addressed in this thesis and lead to the major Research Questions:

(RQ_1) How should planning capabilities be designed and realize to make them *usable*, *interoperable*, and *reusable*?

The lack of classification (see Section 1.1) is included in the second RQ:

(RQ_2) How can planning capabilities be discerned and classified so that they represent encapsulated and separate concerns?

The lack of usability, interoperability, and reusability is addressed in the third RQ:

(RQ_3) Which requirements need to be satisfied to guarantee usability, interoperability, and reusability?

RQ_{2-3} paves the way for the design of new services, which have to be developed under a set of derived criteria. This leads to the last group of RQs:

($RQ_{4.1}$) How to design a **parsing service** that provides a standardized and generic interface and supports the implementations of the "most popular" existing planners?

($RQ_{4.2}$) How to design a **modelling service** that provides a standardized and generic interface and supports the implementations of the "most popular" existing planners?

($RQ_{4.3}$) How to design a **solving service** that provides a standardized and generic interface and supports the implementations of the "most popular" existing planners?

1.3 Method

This thesis aims to answer the research question RQ_1. To solve the first RQ_1, RQ_2-RQ_4 must processed before, as shown on Figure 1.1.

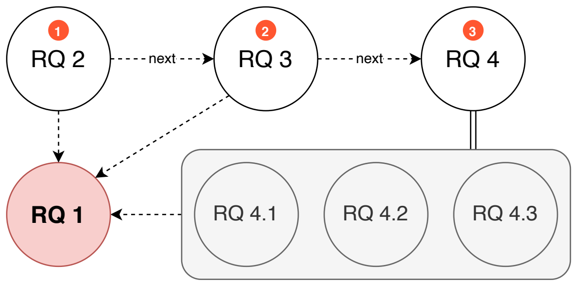


Figure 1.1: Research Question Dependencies and Process

In order to gain a better understanding of the research fields “AI Planning” and “SOC”, the basic principles are initially elaborated in Chapter 2.

Regarding the application of the research question RQ_2, in Chapter 3, related works are examined and differentiated from this thesis’ approach. These AI planners are the result of an extensive literature research, which is performed by means of snowballing, based on the following literature:

- “CPEF: A Continuous Planning and Execution Framework” [Mye99]
- *Automated Planning* [GNT04]
- “PELEA: a Domain-Independent Architecture for Planning, Execution and Learning” [GAP+12]
- “A Service Oriented approach for the Interoperability of Space Mission Planning Systems” [FPD13]
- “HTN planning: Overview, comparison, and beyond” [GA15]
- *Automated Planning and Acting* [GNT16]
- “Planning.Domains” [Mui16]
- “Coordinating services embedded everywhere via hierarchical planning” [Geo15]
- “SOA-PE: A service-oriented architecture for Planning and Execution in cyber-physical systems” [FMJB17]
- “Introduction: Service-oriented computing” [PG03]
- *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)* [Erl07]

All results for related research systems are presented including their architecture and briefly discussed.

However, the research question RQ_2 is answered later in Chapter 4. Apart from the collected AI planner systems in Chapter 3, AI planner solvers gathered during the literature research are also considered.

Subsequently, requirements for the three quality attributes *(usability, interoperability, and reusability)* from RQ_3 are elaborated. All requirements apply to each capability.

Based on these essential requirements, and architecture is designed, which allows the composition of the classified capabilities in an AI planner. The interface design is developed based on the knowledge acquired so far and considers all SOC properties. For this purpose, the interface design is developed that satisfies RQ_4.

Afterward, a minimal system prototype, providing the three capabilities mentioned on RQ_4, is created as a proof of concept.

After all, the prototype is evaluated, and the results are discussed.

2 Background

This Chapter contains the foundation of this thesis. Starting with the groundings about planning on Section 2.1, followed by modeling knowledge on Section 2.2. Finally SOC and SOA are introduced on Section 2.3.

2.1 Planning

This section defines planning and gives an overview of the faceable planning problems. At first, it will be shown where AI Planning is placed in the AI landscape (see Section 2.1.1). Afterward, Section 2.1.2 presents an overview of the problem categories. This introduction to the technical basics serves the understanding of the classification of the various planners. These planner classes are introduced and briefly summarized in Section 2.1.2. As a general definition of planning Definition 2.1.1 is used in this thesis.

Definition 2.1.1 (Planning)
"Planning is the art and practice of thinking before acting [Has]."

2.1.1 AI Planning

The wide field of AI can be separated, to the groups of "Symbolic" and "Sub-symbolic" AI [NM19].

Sub-symbolic AI

On the one hand, the group of *sub-symbolic AI* can be characterized by data usage. Most of its approaches are model-free, which leads to a lack of explainability. The most famous examples are (Deep) Learning, Pattern Recognition, Perception (e.g., vision), and many more.

Symbolic AI

On the other hand, *symbolic AI* can be characterized by knowledge usage. This means that they are often model-based so that they can be explained. Examples are knowledge representation and reasoning, Logic, Search, **AI Planning** [NM19].

Thus AI planning is in principle a symbolic AI. However, there are also sub-symbolic AI approaches in this area [AF17]. In the context of this work, only symbolic AI planners are considered.

2.1.2 Classification of Planning Problems

Planning problems can be classified by the properties *dynamics*, *observability* and *horizon* [NM19].

dynamics $dynamics \in \{deterministic, nondeterministic, probabilistic\}$
Deterministic dynamics means an action and the current state clearly determine the successor state (as known from automata). *Nondeterministic* dynamics means action, and the current state has multiple possible successor states (as known from automata). *Probabilistic* dynamics means there is a probability distribution over possible successor states, for actions and current state.

observability $observability \in \{full, partial, none\}$
Full observability means the current world state can be determined. *Partial observability* means the current world state can not be determined. The current state is one of several possible ones. *No observability* means it is impossible to narrow down possible current states.

horizon $horizon \in \{finite, infinit\}$
Finite horizon means the algorithm terminates after a finite number of steps. *Infinite horizon* means the algorithm never terminates.

Table 2.1 shows the configurations and problem classes
($D \equiv deterministic,\ ND \equiv nondeterministic,\ P \equiv probabilistic,\ infinit \equiv \infty$).

	dynamics	**observability**	**horizon**
classical planning	D	full	finite
conditional planning with full observability	ND	full	finite $\vee$ ∞
conditional planning with partial observability	ND	partial	finite $\vee$ ∞
conformant planning	$D \vee ND$	none	finite $\vee$ ∞
Markov decision processes	P	full	finite $\vee$ ∞
Partially observable MDPs (POMDP)	P	partial	finite $\vee$ ∞

Table 2.1: Classification of Planning Problems [NM19]

Classical Planning This problem class describes planning problems in which a solution has a goal state. An example of the class is the following problem: “Build a three blocks height tower.”

Conditional Planning with full observability This problem class describes planning problems in which the solution stays in goal states indefinitely. An example of the class is the following problem: “Build towers, which are maximal five blocks in height.”

Conditional Planning with partial observability This problem class describes planning problems in which solutions maximize the probability of reaching a goal state. These problems are getting more complicated, as we can see in the example: “Buy a house in 2050 by studying hard and saving money [NM19].”

Conformant planning This problem class describes planning problems in which the solution collects the maximal rewards. An example of the class is the following problem: “Maximize your future income [NM19].”

Markov decision processes (MDPs) This problem class describes planning problems, which deal with fully observable sequential decision problems [NM19]. MDP compute the *optimal policy* in an accessible, stochastic environment with a known transition model [RN10, pp. 645–650]. *Policy* is defined as the complete mapping form states to actions ($policy : States \mapsto Actions$). For this class, the Markov property is given because the transition probabilities depend only on the current state and not on the history of predecessor states.

The most common example of this is a robot, which is moving on a defined grid [Suc15, Chapter 11]. This robot has a transition model (e.g., different costs for moving directions), reward functions, and obstacles can also be placed on the grid. Since there is an uncertainty in the result of each action taken by the robot, the usage of MDP is necessary. The complete example can be found in the book of Sucar [Suc15, Chapter 11].

Partially Observable MDPs This problem class describes planning problems, which deal with partial observable sequential decision problems [NM19].

Since POMDPs are too complex to solve them directly (only partly accessible environments), the physical state space has to be reduced to the corresponding belief-state space [RN10, pp. 658–692]. MDP solve problems with-in belief-state space [RN10, p. 660]. In order to enable this convention in a sufficient way, POMDPs use *dynamic decision networks* to represent the transition models, to update its *belief-state*, and to provide possible action sequences [RN10, p. 685].

For instance, the example of MDPs can be reused [Suc15]. The only difference that has to be applied is that the real world is not perfect, and the physical state space is not fully observable. This means the robot maybe do not run the command correctly.

2.1.3 Classification of Planners

According to Rintanen and Hoffmann, planners can be classified to following categories [GKP20; RH01]. In this way, a distinction is made between different types of planning. To solve the different planning problems (cf. Section 2.1.2), different planners are necessary.

Classical Planning

This group of planners is used to solve classical planning problems. They can be characterized by the ability to solve deterministic problems with full observability in a static environment [RN10]. Typically the modeling standard PDDL is used for this purpose (see Section 2.2).

Solving classical planning problems by finding the shortest path between the initial state and goal state in the transition graph leads to the “real” problem. In most cases, the state spaces are far too huge to construct the full transition graph at once. Most planning algorithms are much more efficient than distinct solution methods like the Dijkstra algorithm [NM19].

Within this category, further subgroups can be identified:

- State-Space Planning (see [GNT04, p. 101])
  e.g. Stanford Research Institute Problem Solver (STRIPS)
- Plan-Space Planning (see [GNT04, p. 101])

The major difference between state-space planning and plan-space planning is that the state-space approach is finite [GNT04, p. 101].

Neoclassical Planning Nevertheless, *Classical Planning* is also further refined under the term “Neoclassical Planning”.

- Planning Graph (see [GNT04, p. 112])
  These planners build a graph using the initial state as the starting point [RN10].
- Constraint Satisfaction and Propositional Satisfiability (see [GNT04, p. 168])
  These planners solve Constraint Satisfaction Problems (CSPs) (in general Satisfaction Problem (SAT)). Therefore a Conjunctive Normal Form (CNF) formatted problem input is required.

Temporal Planning

Temporal planning takes a problem from classical planning and solves it. Since in classical planning, it is possible to define what is to be done and in which order, the additional information about the duration of the action [RN10, p. 401] is missing. This information is contained in the temporal planning, so it can be defined for each action, when it starts and how long it lasts (see [HLMM19, Chapter 5]).

Hierarchical Planning

In hierarchical planning, methods are not assumed to be atomic actions, as other approaches do. Hierarchical planning represents actions in networks, so it is possible to process nested actions [RN10, p. 406]. Decomposition as a sequence of atomic actions is possible.

Probabilistic Planning

Like temporal planning, this approach can be seen as an extension of classical planning. In this case, probabilities for the occurrence of actions are added. For example Relational Dynamic Influence Diagram Language (RDDL) can be used for modeling (see [San10]).

Conformant Planning aka Sensorless Planning

Conformant planning is a planning approach that does not impose conditions on the observability of the environment [RN10, p. 415], hence the second name “Sensorless Planning”. This type of planning is mainly used in non-deterministic environments, including online planning or replanning in unknown environments [RN10, p. 415].

Multiagent Planning

The so-called multi-agent planning is a distributed approach to solve planning problems [RN10, p. 425]. It focuses on shared resources, such as activities and goals. A particular challenge is the merging of plans [RN10, p. 425].

2.2 Modeling

This section introduces the modeling of AI Planning. Modeling focuses on approaches, constructs and languages used to define a planning problem [Geo15]. In the specific field of planning modeling, the model contains definitions of a domain model and the problem.

AI planning can naively be described as a state-transition system (see Definition 2.2.1). However, some assumptions are made which cannot be made in the real world, including the finiteness of states and actions and that all events can be observed.

Definition 2.2.1 (State-transition system [GNT04])
A state-transition system is defined as a tuple $\Sigma = \langle S, A, E, \gamma \rangle$, *where:*

- $S = \{s_1, \dots, s_n\}$ *is a finite set of states.*
- $A = \{a_1, \dots, a_n\}$ *is a finite set of actions.*
- $E = \{e_1, \dots, e_n\}$ *is a finite set of events.*
- $\gamma : S \times A \times E \rightarrow 2^S$ *is a state-transition function.*

Since this model cannot be applied directly, more detailed ones must be used (see Section 2.2.1 and Section 2.2.2).

2.2.1 Domain Model

Domain models are used to define domain knowledge, in order to solve planning problems [Geo15, p. 50]. It denotes a representation that portrays behaviors as found in the real domain and provides semantics for the constructs in the domain [McC02].

Definition 2.2.2 (Domain Model)
"The Domain Model is a Domain Specification which is in an operational form, containing explicit details of domain dynamics, and suitable for processing by a planning engine [McC02]."

It contains the following formal definitions (Definition 2.2.3 and Definition 2.2.4).

Definition 2.2.3 (Domain Model in classical planning [Lot17])
A planning domain is defined as a tuple $d = \langle \mathcal{T}, \mathcal{R}, \mathcal{P}, \mathcal{A} \rangle$, where:

- $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ *is a set of types.*
- $\mathcal{R}$ *is an inheritance relation on types.*
- $\mathcal{P}$ *is a set of predicates.*
- $\mathcal{A}$ *is a set of actions.*

Definition 2.2.4 (Domain Model in hierarchical planning [GNT04])
A planning domain is defined as a tuple $d = \langle O, \mathcal{A} \rangle$, where:

- O *is a set of operators.*
- $\mathcal{A}$ *is a set of methods.*

The Domain Model needs to be specified for each user individually. Therefore several approaches exist. Since the definition of a domain model mentioned in Definition 2.2.2 shows, the model has to be suitable for processing. A purely textual description in natural language is not suitable for this purpose.

Manual input It is not user-friendly to write code manually, e.g. in PDDL (cf. Section 2.2.3). Even with an Integrated Development Environment (IDE) this is extremely time intensive. Moreover, the defined environment can change fast, and manual input cannot face such use-cases. Only workbenches and other test cases fit perfectly for this input method.

Web interface The model creation through a web interface can decouple the input language and the required model input. In the GUI case, the required knowledge on the customer's side can be reduced by far. An example of such an interface can be found in Hoekstra and Georgievski thesis [HG13]. The approach is reused in Georgievski thesis [Geo15, Section 6.3.2]. Figure 2.1 shows a screenshot of the input mask for modeling an action.

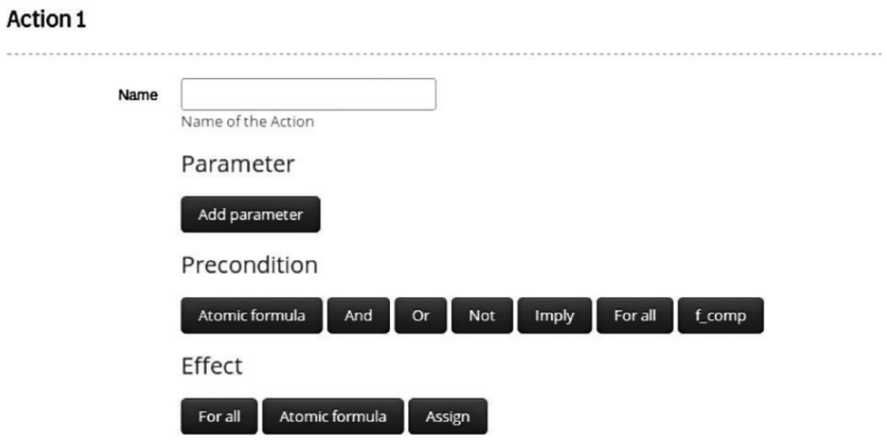


Figure 2.1: Screenshot of an example web modelling tool [Geo15]

Automation The third option is the most user-friendly. In this case, the user does not have to make any inputs. The information is collected directly from sensors and automatically generated. In 2011 Hidalgo et al. presented an approach for planning daily activities [HCM+11]. Fernández et al. introduced another approach for automatic plan generation for data mining in 2013 [FRF+13]. Alternatively, for example, Ortiz et al., who created an approach for automated building planning [OGOB13]. Such approaches can make a system usable and therefore such an integration is highly desirable.

2.2.2 Problem Model

The Problem model is defined in Definition 2.2.5.

Definition 2.2.5 (Problem Model in classical planning [Lot17])
A planning problem is defined as a tuple $p = \langle \Omega, s_0, goal \rangle$, where:

- $\Omega = \Omega_1 \cup \cdots \cup \Omega_n$*, where* $\Omega_i = \{\omega_1, \ldots, \omega_n\}$
  $\forall \omega_i \in \Omega_i$ *that (*ω_i *is off type* τ_i*).*
  So Ω *is a set of object-sets off the same type.*
- s_0 *is the initial state.*
- $goal$ *is the goal condition.*

Definition 2.2.6 (Problem Model in hierarchical planning [GNT04])
A planning problem is defined as a tuple $p = \langle s_0, w, O, \mathcal{A} \rangle$, where:

- s_0 *is the initial state.*
- w *is the initial task network.*
- O *is a set of operators (see domain).*
- $\mathcal{A}$ *is a set of methods (see domain).*

A planning problem contains the **initial state** and **goal**, which can be automatically generated from the environment definition. This information are usually extracted from a domain model [Geo15]. Consequently, the information must first be merged in order to process the planning request.

2.2.3 Languages

Languages are definitions of syntax, which is used to pass physical properties and specific knowledge. This modeling languages are used to define some properties:

- Actions
- Domain-specific knowledge
- Current state of an environment
- Predicates
- Variables or functions
- The goal
- Compound tasks *(in case of hierarchical planning)*

Since there are multiple researchers, there is not *one* standard language. Each research group extends languages and creates new ones. In the following paragraphs, several popular languages are introduced.

Planning Domain Definition Language (PDDL)

PDDL[1] is kind of a community standard for the representation and exchange of planning domain models in context of respective planning. Since International Planning Competition (IPC) domains and problems are written in PDDL, it became a standard input format for classical *(nonprobabilistic)* planning problems. According to Georgievski, the majority of the primary studies use PDDL as a modeling syntax for their domain models [Geo15].

STRIPS (see [Byl94]) and Action Description Language (ADL) inspired Aeronautiques et al. to create PDDL [AHK+98].

In the following sample PDDL is used as the modeling language. Listing 2.1 shows the PDDL domain and Listing 2.2 the problem definition. For this sample we want to move `blocks` from `table x` to `table y`. Additionally multiple blocks are permitted on a `table`. The only available action is `move` [Bec15].

[1]PDDL has currently three major versions (August 11, 2020):
Ver. 1 see [AHK+98], *Ver.* 2 see [FL03], *Ver.* 3 see [GL05]

Listing 2.1 PDDL domain example (cf. [Bec15]).

```
(define (domain blocksworld)
  (:requirements :strips)
  (:action move
     :parameters (?b ?t1 ?t2)
     :precondition (and
        (block ?b)
        (table ?t1)
        (table ?t2)
        (on ?b ?t1)
        (not (on ?b ?t2))
     )
     :effect (and (on ?b ?t2) (not (on ?b ?t1)))
  )
)
```

Listing 2.2 PDDL problem example (cf. [Bec15]).

```
(define (problem move-blocks-from-a-to-b)
    (:domain blocksworld)
    (:objects a b x y )
    (:init
        (block a) (block b)
        (table x) (table y)
        (on a x) (on b x)
    )
    (:goal (and (on a y) (on b y)))
)
```

The second line of the domain definition in Listing 2.1 defines the PDDL requirements. In this example, only STRIPS is assumed. But generally other possibilities are offered, for example `:negative-preconditions` [HLMM19, p. 19], `:typing` [HLMM19, p. 25], or *:action-costs* [HLMM19, p. 32].

Pellier and Fiorino created a Java-based PDDL toolkit (PDDL4J) [PF18], which is used in the prototype of this thesis.

According to Haslum et al., PDDL should also be able to be used in the future to model hierarchical planning problems [HLMM19, pp. 145–147].

Simple Hierarchical Ordered Planner Language (SHOP)

Simple Hierarchical Ordered Planner (SHOP)[2] is a domain-independent automated-planning system. It is based on ordered task decomposition, which is part of Hierarchical Task Network (HTN) planning.

Since SHOP became increasingly popular [NAI+05], researchers have started to use the modeling language. Therefore SHOP can be seen as a kind of standard language for HTN planning. The language has changed within several years and is now available in the third version. However, it must be noted that SHOP3 contains significant changes that change the behavior. Nevertheless, the SHOP language was also changed to make domain engineering easier [GK19a; GK19b].

The main difference to non HTN planning is the decompression of non-primitive tasks. Thus, tasks are created in a tree structure, where the leaves are considered primitive tasks [GNK07, p. 4].

Definition 2.2.7 (Methods in HTN planning [GNT04])
A method is defined as a tuple $m = \langle name(m), task(m), subtasks(m), constraints(m)\rangle$, where:

- $name(m)$ *is a unique string-literal.*
- $task(m)$ *is a not primitiv task.*
- $\mathcal{TN} = (subtasks(m), constraints(m))$ *is a task network.*

java implementation of SHOP so called JSHOP.

Listing 2.3 JSHOP example of a move operator.

```
(:operator (!move ?agent ?from ?to)
  (; preconditions
    ; Downgrade types to predicates
    (agent ?agent) (hallway ?from) (hallway ?to)
    (at ?agent ?from)
    (not (at ?agent ?to))
    (adjacent ?from ?to)
  )
  (; delete effects
    (at ?agent ?from)
  )
  (; add effects
    (at ?agent ?to)
  )
)

```

In the latest version SHOP3 several features have been taken from PDDL, for example, the quantifiers and conditional effects in methods and operators [GK19b]. SHOP3 is now even able to use PDDL natively [GK19a; GK19b].

[2]SHOP has currently three major versions (August 11, 2020):
$Ver.$ 1 see [NCLMA99], $Ver.$ 2 see [GNK07], $Ver.$ 3 see [GK19a; GK19b]

Action Notation Modeling Language (ANML)

The Action Notation Modeling Language (ANML) is created for hierarchical planning [SFC08]. While a large part of the semantic is close to PDDL, in combination with the capability of handling HTN, the syntax changed by far [SFC08]. Listing 2.4 shows, that the syntax is way shorter and easier to read than e.g. PDDL.

Listing 2.4 ANML move operator (cf. [SFC08]).

```
action move (agent agent, hallway from, hallway to) {
  before {
    agent.pos == from;
    position == from ::= to;
  }
}
```

ANML is for example used on the FAPE planner [DBMIG14].

Hierarchical Planning Definition Language (HPDL)

Georgievski introduced 2013 the Hierarchical Planning Definition Language (HPDL). Inspired by Castillo et al., he created a PDDL like, feature oriented, planning language [CFOGPP06]. The most important part is that HPDL is capable to model HTN planning domains. HPDL is for example used on the real world energy-saving system approach of Georgievski et al. [GNA13].

HPDL and HDDL (see Section 2.2.3) are very similar, only the syntax varies minimal, for this compare [GNA13] and [HBB+19].

Hierarchical Domain Definition Language (HDDL)

HDDL actions are no different from PDDL. This can't be observed in Listing 2.5, because it is only a singe operation. Since HDDL extends the language focusing on the hierarchical aspects required by HTN planners [HBB+19].

Listing 2.5 HDDL example of a move action.

```
(:action move
  :parameters (
    ?agent - agent
    ?from ?to - hallway ; Both ?from and ?to are hallways
  )
  :precondition (and
    (at ?agent ?from)
    (not (at ?agent ?to))
    (adjacent ?from ?to)
  )
  :effect (and
    (not (at ?agent ?from)) ; This line is usually forgotten
    (at ?agent ?to)
  )
)
```

New Domain Definition Language (NDDL)

The New Domain Definition Language (NDDL) was found by NASA in 2002 to enable temporal planning for space missions [BS07; FJ03]. It is forked initially from PDDL1 and developed over a couple of years. Since PDDL3 supports most of features, that are necessary for temporal planning, NDDL is only rarely used [BCC12].

Relational Dynamic Influence Diagram Language (RDDL)

RDDL is made of *PPDDL*[3] (see [YL04]), because among other reasons unrestricted parallelism is necessary for many real-world problems, which PPDDL cannot handle due to its PDDL2 origin [San10]. It also adds the capability to handle only partial observable problems [San10].

2.2.4 Plan Model

Definition 2.2.8 (Plan Model in classical planning)
A planning solution π for the problem p is defined as a list $\pi_p = [a_1, \ldots, a_k]$, where:

- *$a \in \mathcal{A}$ is the name of the action.*

Definition 2.2.9 (Plan Model in hierarchical planning [GNT04])
A planning solution π for the problem p is defined as a list $\pi_p = [a_1, \ldots, a_k]$, where:

- *$a_i = name(u_i)$ for $u_i \in U$.*
- *If there is an instance (U', C') of $(U, C) = \mathcal{TN}$*
  see definition of a task network $\mathcal{TN}$ [GNT04, p. 244].

[3]An extension to PDDL for expressing planning domains with probabilistic effects.

- *The plan π is executable in s_0.*

If the task network $\mathcal{TN}$ is not primitive, the tasks have to be decomposed until it is.

2.3 Service-oriented Computing and Architecture

This section provides a breath overview of SOC and SOA. The border between SOC and SOA isn't always clearly defined, therefore Definition 2.3.1 and Definition 2.3.2 are provided.

Definition 2.3.1 (SOC)
"SOC is a paradigm that uses services to support the development of rapid, low-cost, interoperable, evolvable, and massively distributed applications [PTDL07]."

Definition 2.3.2 (SOA)
"SOA is an architectural model that aims to enhance the efficiency, agility, and productivity by promising loose coupling between components within a distributed architecture. SOA is an element of SOC, which enables service discovery and integration [Erl07]."

2.3.1 Loose-Coupling

Loose coupling originality come from the field of enterprise architecture, and integration [HW04]. As already mentioned in Definition 2.3.2 loose coupling is also a key feature of SOA. Kaye shows, loose coupling intentionally sacrifices interface optimization, to achieve flexible **interoperability** among systems that are disparate in *technology*, *location*, *performance*, and *availability* [Kay03]. This properties lead to Leymanns' definition of Autonomy Aspects for loose coupling [Ley18]:

- **Reference Autonomy**:
  Reference Autonomy means, the producers and consumers do not know each other. Only the place where to read/write data is known.
- **Time Autonomy**:
  Time Autonomy means, the communication between producers and consumers is asynchronous. The access channels at their own pace, and the Data exchanged is persistent.
- **Format Autonomy**:
  Format Autonomy means, the exchange data format may differ from producers and consumers. In order to allow such behavior, mediation/transformation on the pipe is required.
- **Platform Autonomy**:
  Platform Autonomy means, producers and consumers may do not share the same environments or languages.

2.3.2 Messaging

Messaging is the most common way to reach loose coupling properties (see Section 2.3.1). Essentially a message is transmitted in five steps [HW04]:

1. *Create*: (Endpoint *A*)
   The sender creates the message and adds data.
2. *Send*: (Endpoint *A*, Channel)
   The sender pushes the message to a channel.
3. *Deliver*: (Channel, Route, Transform)
   The messaging system makes the message available to the receiver.
4. *Receive*: (Channel, Endpoint *B*)
   The receiver consumes the message from the channel.
5. *Process*: (Endpoint *B*)
   The receiver extracts the data from the message.

Figure 2.2 shows an abstract architectural concept of a messaging approach. The pattern behind this approach is the popular principle of *Pipes and Filters* [Ley18].

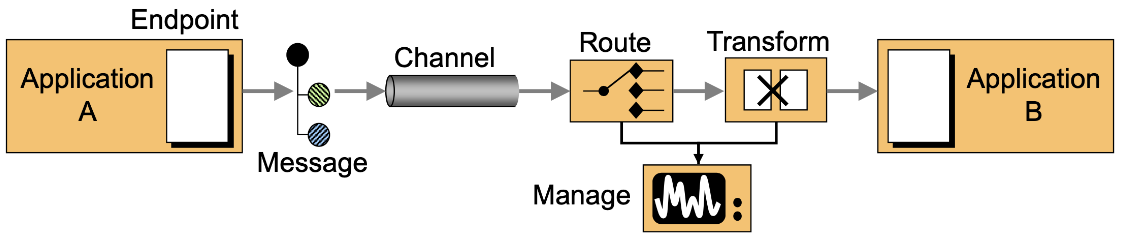


Figure 2.2: Architectural Building Blocks [Ley18, Messaging]

In this case an application *A* provides some data for application *B*. Each Block (Endpoint, Messages, Channels, Routing, Transformation and Managing) of Figure 2.2 symbolizes a category of patterns [HW04]. For instance, a message is sent through a MOM. The Route block handles the delivery of the message, and the Transform block transforms or enriches the message to fit the requirements of an application *B*.

3 Related Work

This chapter surveys related work in the field of AI planning systems. A common theme in much of this work is the desire for reusability and interoperability.

In general, the architecture of most approaches can be represented by the abstract model from Figure 3.1.

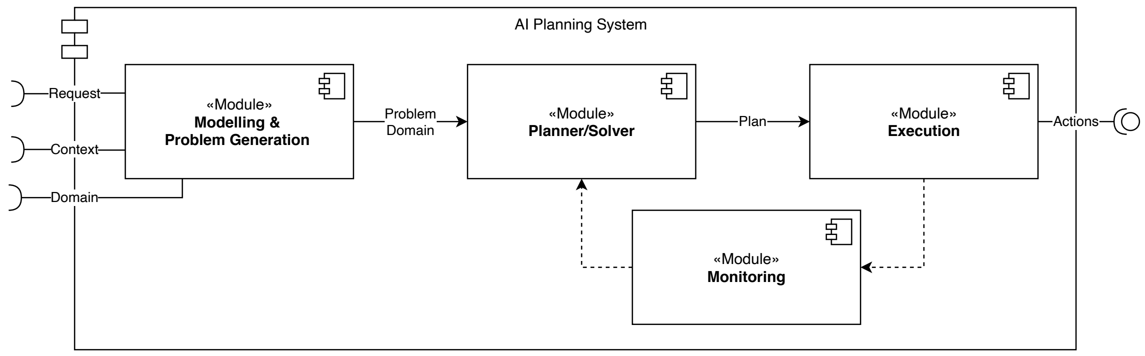


Figure 3.1: Component diagram of typical planning systems.

Figure 3.1 shows an abstraction of most planning systems. The diagram differs only slightly from Georgievski [Geo15, p. 57]. However, the creation of the problem and the domain is combined in one module, while Georgievski requires a store.

All the following systems (cf. Sections 3.1 to 3.7) are selected, as already explained in Section 1.3. This chapter provides a shortlist of all related work. Besides a general description, there is a list of capabilities and a subsequent discussion of each system's advantages and disadvantages.

Since the systems are arranged chronologically, the evolution to a SOA becomes apparent.

3.1 CPEF

Continuous Planning and Execution Framework (CPEF) is a Continuous Planning and Execution Framework, founded by Myers in the year 1999 [Mye99]. CPEF uses several capabilities as execution, monitoring, and repairing, to solve complex tasks in unpredictable and dynamic environments [Mye99].

As already contained in the title CPEF is based on a continuous planning approach, which requires an abstract view of planning as dynamic artifacts. Each change of the environment triggers the system, which initializes plans or starts reparations. Since the interface is the only part of interaction with the user, all input has to be made manually. The significant innovation of CPEF was the supervision of execution of capabilities. The *Plan Manager* can process user changes and request the next steps [Mye99].

In general CPEF can be seen as a composition of multiple existent components. For instance, for the fast high-level planning, SIPE-2 [Wil90] is used. The rest of the components is based on PRS, a procedural reasoning system [GI89].

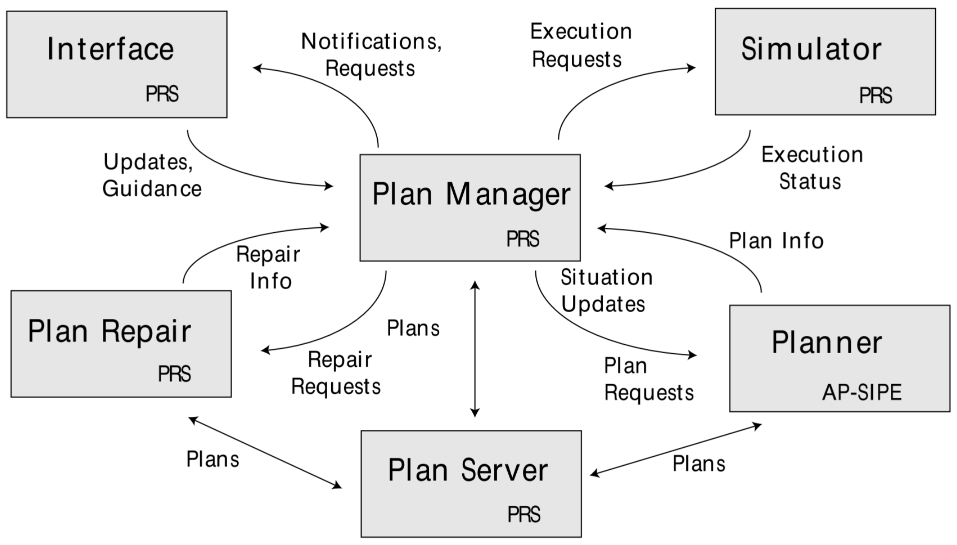


Figure 3.2: Architecture Overview of CPEF [Mye99, Fig. 1].

The following capabilities can be derived from this architecture overview on Figure 3.2:

- *Modelling and Visualizing Capability (Interface - PRS)*
  This capability is intended for communication with the user. Besides defining all necessary elements (problem, domain), it also displays notifications.
- *Managing and Monitoring Capability (Plan Manager & Plan Server - PRS)*
  This capability is responsible for the coordination of requests, states of execution, and, if necessary, the initiation of replanning.
- *Executing Capability (Simulator - PRS)*
  This capability is responsible for simulating the execution of plans. The state of execution is transferred to the Plan Manager.
- *Solving Capability (Planner - AP-SIPE)*
  This capability is necessary to solve the planning problem.

CPEF is one of the first AI planning systems. Therefore no SOA were used. Nevertheless, the capabilities are distributed in single components to increase interoperability and maintainability. From today's point of view, the architecture is no longer desirable (lack of flexibility), but the capabilities can still be used as abstractions.

3.2 PELEA

A Domain-Independent Architecture for Planning, Execution and Learning (PELEA) was found by Guzmán Alvarez et al. in 2012. The developers, in particular, focused on the lack of generality and reusability possibilities. They fixed the lack of flexibility of CPEF (see Section 3.1) by allowing engineers to quickly generate new applications by reusing and modifying the components and even to include their techniques [GAP+12].

Figure 3.3 shows the full architecture of PELEA. One special feature of the architecture the two state/plan models is remarkable. PELEA has a low- and high-level model. For internal communication an own Extensible Markup Language (XML) schema *XPDDL* [AGP+10], which is capable to transport all information of PDDL and HTN [GAP+12], was defined.

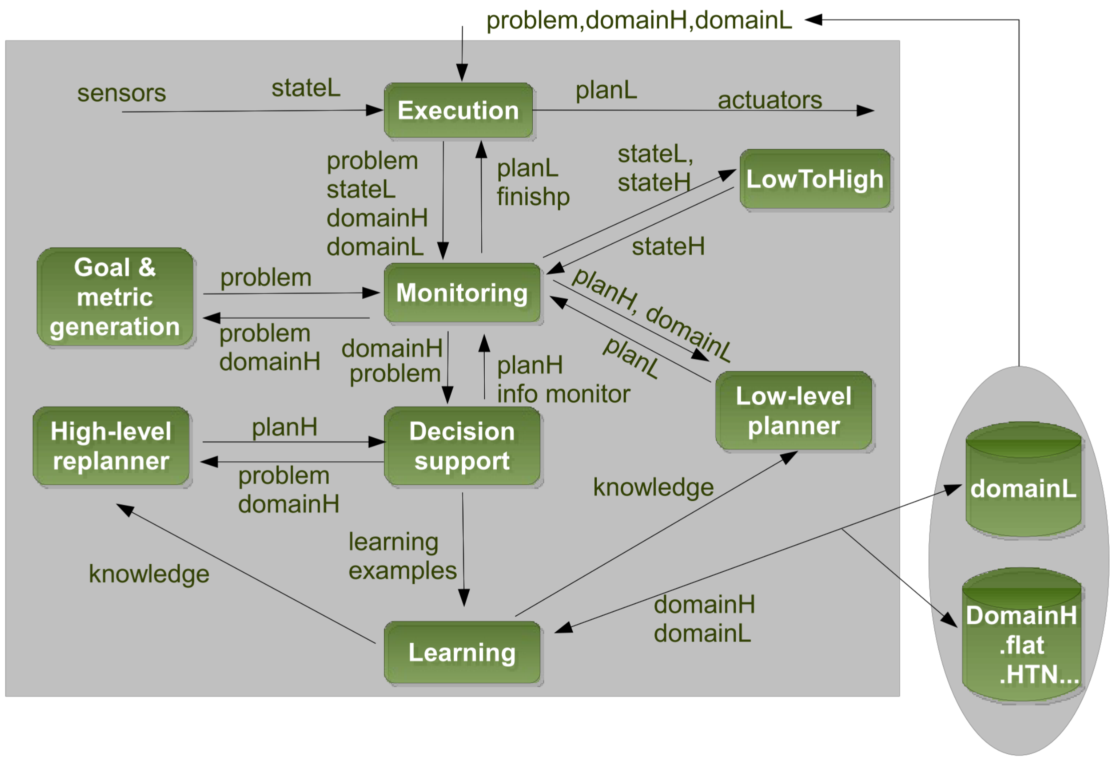


Figure 3.3: Architecture Overview of PELEA [AGP+10, Fig. 1]

The following capabilities can be derived from this architecture overview on Figure 3.3.

- *Modelling Capability (Goal & Metric generation)*
  This capability is used to define the goals by modeling the problem.
- *Executing Capability (Execution)*
  Since the system is designed for the continuous application, the Executing Capability also serves as a bridge for the Monitoring Capability control flow. On the other hand, plans are forwarded to actuators and executed.

- *Monitoring and Managing Capability (Monitoring)*
  This capability is not only necessary to record the state of the system. Due to its central position in architecture, parts of the utilities of a Managing Capability are also required.
- *Reacting Capability (Deciton support)*
  This capability is directly connected to the monitoring capability and decides whether replanning is necessary.
- *Solving Capability (Low-level planner & High-level replanner)*
  This capability is necessary to create solutions from a problem and the domain.
- *Storing Capability (Databases for Domains)*
  This capability is realized in this system by providing two databases.
- *Converting Capability (LowToHigh)*
  This capability is responsible for the conversion from low-level states to high-level states.
- *Learning Capability (Learning)*
  This capability is used to improve solving capabilities continuously. For this purpose, training data is permanently transferred, leading to insights in combination with the domain information.

The system offers the possibility to react quickly to changes in the real world and introduce plan changes. Besides, it scores with the combination of symbolic and sub-symbolic AI.

However, since PELEA only divides the capabilities into separate modules, like CPEF, and does not use SOA, it is not flexible from architecture viewpoints. Changing or adding new capabilities is very complicated and requires a rebuild of the whole system.

3.3 Space Mission Planning System

This system does not have any specific name since it is a prototypical architecture. It was found by Fratini et al. from European Space Agency (ESA) in 2013. To be more precise, this service-oriented approach brings interoperability to *Space Mission Planning Systems* [FPD13].

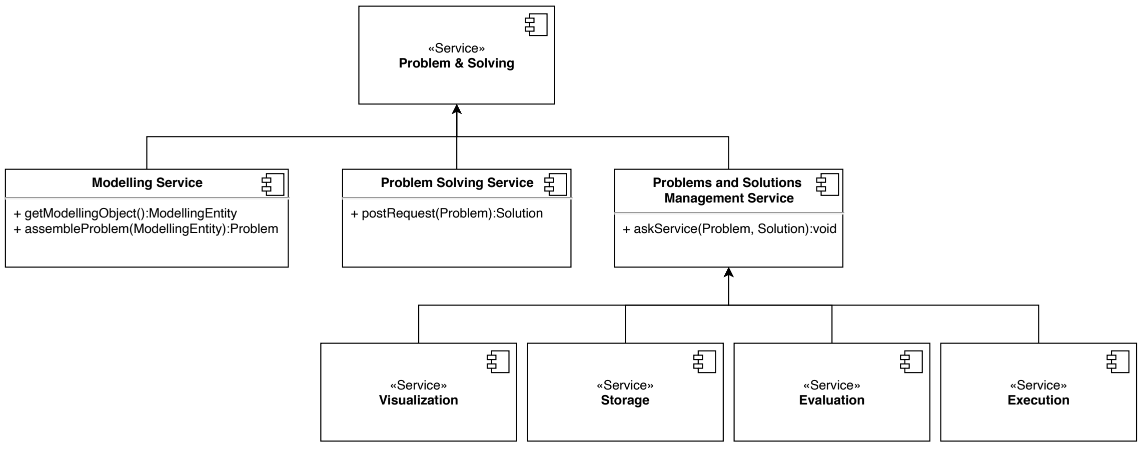

«Service»
Problem & Solving
Modelling Service
+ getModellingObject():ModellingEntity
+ assembleProblem(ModellingEntity):Problem
Problem Solving Service
+ postRequest(Problem):Solution
Problems and Solutions Management Service
+ askService(Problem, Solution):void
«Service»
Visualization
«Service»
Storage
«Service»
Evaluation
«Service»
Execution

Figure 3.4: Architecture Overview of the Space Mission Planning System (cf. [FPD13, Fig. 2]).

Figure 3.4 shows the system's service architecture overview, and it is derived from Fratini et al. [FPD13, Fig. 2]), which represents the conceptual model of the system. Several services are providing different capabilities.

- *Modelling Capability (Modelling Service)*:
  This capability manages the modeling entities like basic entities, problems, domain theories and solutions. Procedures are HTN decompositions of planning objects, while rules are the mandatory specification of conditional activation of necessary restrictions [FPD13]. Additional solutions and problems are compatible, which allows the usage of solutions as problems for another solving process.
- *Modelling Capability (Problem Solving-Service)*: This capability manages the solving and optimization processes. It offers the following options for specifying and propagating restrictions (e.g., conflicts), planning, generating, and optimizing timelines [FPD13].
- *Evaluation Capability (Evaluation Service)*
  This capability manages the evaluation of solutions (e.g., quality, robustness, flexibility).
- *Validation and Verification Capability (V&V Service)*
  This capability manages the validation and verification of properties on problems, solutions, and domain (can be seen as part of the *Evaluation Service*).
- *Executing and Monitoring Capability (Execution Service)*
  This capability manages the execution of timelines and monitoring the process.
- *Visualizing Capability (Visualization Service)*
  This capability manages the visualization of problems and solutions.
- *Storing Capability (Storage Service)*
  This capability manages the storing of problems and solutions.

The approach offers the possibility to create new capabilities without a deep understanding of the planner himself. This possibility is achieved by a high degree of interoperability, which is possible with standardized interfaces. Thus, the implementation of new capabilities can be easily passed on to other development teams.

Currently, the approach is only theoretical because there are too many standards for defining problems and solutions in practice [FPD13]. The system offers meta models but is not able to react quickly to new inventions. Changes to this meta-model force all developers of the encapsulated services to adapt their interfaces.

3.4 RuG Planner

The RuG Planner is a domain-independent AI planning system, which was published in 2013 by Kaldeli et al. [KLA13]. This approach's unique feature is to model the planning problems as CSP [KLA16]. During the development of the architecture, special attention was paid to the application's scalability and flexibility. Thus the RuG Planner can react to dynamic changes from the environment [KLA13]. Figure 3.5 shows an architectural overview of the system.

The following capabilities can be derived from the architecture on Figure 3.5.

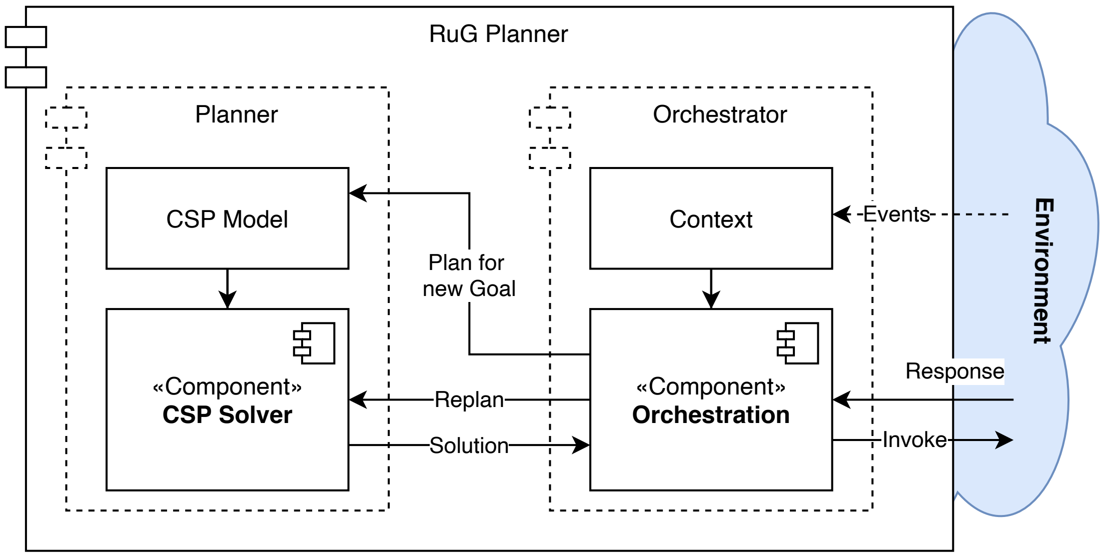


Figure 3.5: Architecture Overview of the RuG Planning System (cf. [KLA16, Fig. 1]).

- *Solving and Converting Capability (CSP Solver)*
  This capability creates plans that are obtained by entering a problem and a domain. This input is given as PDDL and converted to CSP models within the service.
- *Problem Generating, Managing and Reacting Capability (CSP Solver)*
  This capability is a combination of several skills. For example, problems are modeled from live status data and targets and provided as PDDL *(Problem Generating)*. Monitoring or reacting capabilities are used to update the context and trigger replanning in the case of a failure. Since both parts of the system are distributed on different services, the correct allocation of server instances must also be ensured by a managing capability.

The system achieves essential flexibility by the option of using any CSP solver. Another service instance can be started by defining the interfaces, which contains another solver, for example. However, this approach's primary advantage is the scalability, and continuous processing of the environment data [KLA16].

The extension of the planner is a challenge. Hence, the given architecture restricts reusability since large parts of the behavior are implemented in the orchestrator component.

3.5 SH Planner

The scalable hierarchical planning system ("SH-Planner") is the outcome of the PhD thesis of Georgievski. SH-Planner is a service oriented approach for HTN. It is designed to solve ubiquitous computing problems and was used in an experiment to control IoT devices such as lighting [Geo15].

Figure 3.6 shows the UML component diagram of the SH-Planner. It can be observed that, in contrast to most comparable AI planners, the user does not need to enter any environmental information. This information can be consumed directly from the environment. Only the domain model has to be created manually. For modeling the domain HPDL is used.

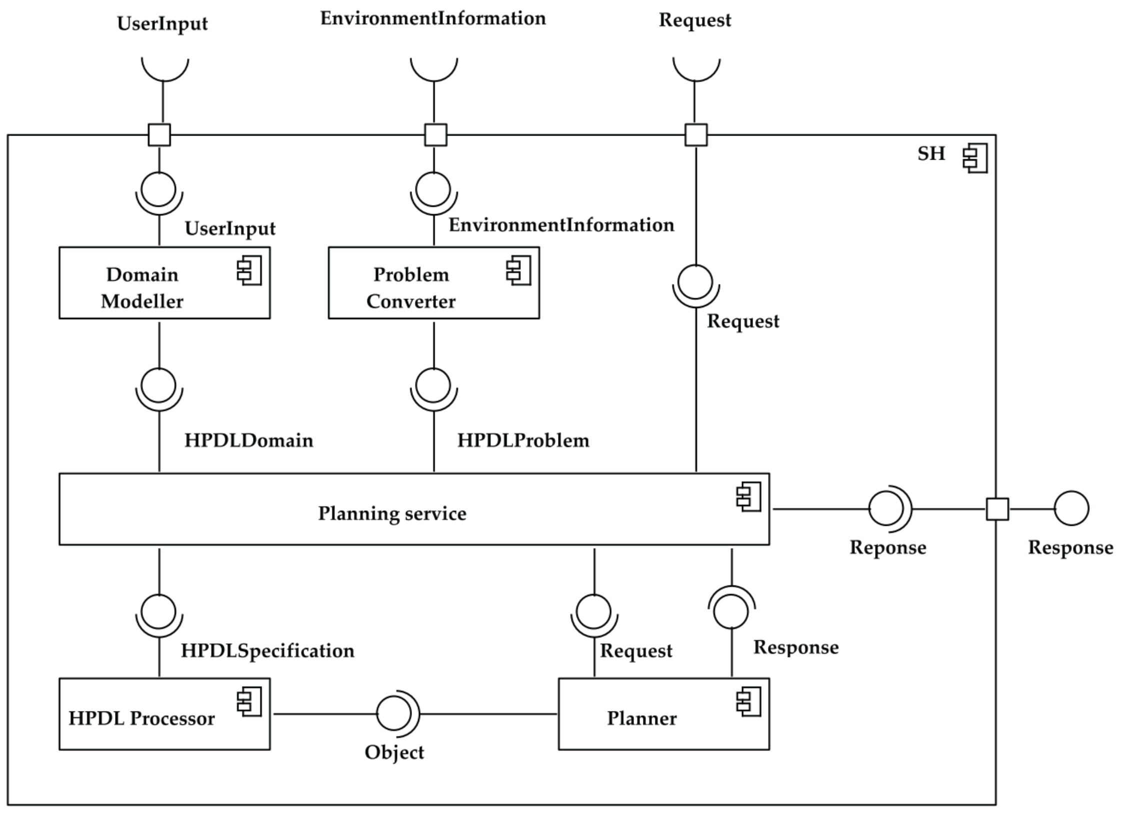


Figure 3.6: Architecture Overview of the SH-Planner [Geo15, p. 148].

The following capabilities can be derived from the architecture on Figure 3.6.

- *Modelling Capability (Domain Modelling)*
  This capability is used to model the domain using HPDL by the user.
- *Problem Generator Capability (Problem converter)*
  This capability receives the information gathered by sensors. Thus, a problem can be constructed from the current status and the goal, encoded in HPDL.
- *Parsing Capability (HPDL Processor)*
  This capability processes the complete HPDL problem and domain specification and converts it into an object which is subsequently used to solve the problem.
- *Solving Capability (Planner)*
  This capability consumes the parser's object and creates plans on request, which are forwarded to the service in the form of action lists.

- *Managing Capability (Planning Service)*
  This capability is the managing core of the system and, additionally, also the executing core. At this point, the plans created are passed on to the environment for execution.

In contrast to JSHOP2 and other planners, the SH-Planner is much more flexible. This flexibility is mainly due to the usage of a SOA. Furthermore, the system can lead the previous planners from the laboratory to the real world. Since other designers only deliver the final plan, which is rarely used afterward, they are not suitable for industrial use. By integrating the executive component, a relation between the real world and the modeled world can be established.

The lack of expandability is one of the most significant disadvantages. So it is not possible to integrate additional capabilities without changing the code. Furthermore, the existing prototype has been created in a no longer executable scale version, which makes it impossible to use the system at the moment. Another point is the missing possibility to include parallelism of actions in the plan. This problem only affects the planner itself and has no impact on the rest of the system architecture.

3.6 Planning.Domains

Planning.Domains is a platform solution to make AI planning easier and serves as a repository for different sample problems and domains. The system has been published 2016 by Muise and is still accessible online (at `http://planning.domains`). It is initially designed for researchers and lecturers [Mui16].

The system consists of a landing page, which refers to the different services [VK20, pp. 91–105]. Figure 3.7 shows the architecture overview.

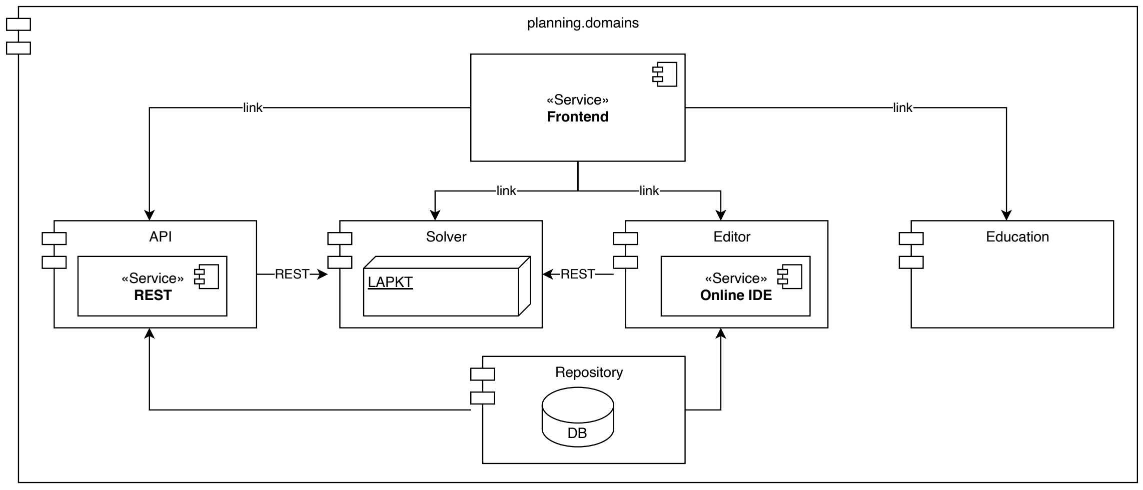


Figure 3.7: Architecture Overview of Planning.Domains

The following capabilities can be derived from the architecture on Figure 3.7.

- *Solving Capability (Solver)*
  This capability is suitable for solving a STRIPS planning problem encoded in PDDL. To solve the problem the user can choose between a range of solvers (e.g. FF [HN01], BFS_f [LRMG14]), which are included in the Lightweight Automated Planning ToolKiT (LAPKT) (see [RLM15]) [Mui16].
- *Storing Capability (Repository)*
  This capability allows quick access to predefined domains and problems of the IPC.
- *Modelling and Visualizing Capability (Editor)*
  This capability is responsible for manual, textual modeling using PDDL. Besides, plans can also be executed directly in this online IDE.
- *Explaining Capability (Editor & Education)*
  This capability is distributed on two sides. On the one hand, plans can be visualized in the editor with a plugin [VK20, pp. 91–105]. On the other hand, many explanations about the algorithms' functionality are provided on the education page.

The system offers a high degree of comfort and enormous time savings for the user by providing a web application. This is probably also the reason for its popularity, as the system is immediately ready to use by all students. When creating the editor, special care was taken to implement a plugin framework, which offers significant extensibility advantages.

However, PLANNING.DOMAINS also has decisive disadvantages, such as too much encapsulation of many capabilities in the Solving Service. Thus, the extension of the solver is not possible without sensitive modifications. This lack of flexibility is also reflected in the interfaces used, which are only designed for PDDL. A further disadvantage is a limitation to research purposes because it does not offer any extension possibilities to connect to real environments. Thus the productive advantage of the application is not guaranteed.

3.7 SOA-PE

SOA-PE is a SOA for planning and execution in context of Cyber-Physical System (CPS) [FMJB17]. This architecture is closely related to PELEA (see Section 3.2). The researchers assume that SOA is the appropriate approach. Since SOA is ideal for large, autonomous systems and offers typical advantages such as reusability, independent development and deployment, platform independence, transparency, and flexibility [FMJB17]. As expected for a SOA, SOA-PE provides a clean separation between all services (domain modeling, planning, execution, monitoring, and actuation) [FMJB17].

Figure 3.8 gives an architectural overview of SOA-PE, made for CPS. As a communication backbone, the service-oriented middleware WebMed[1] is used.

The following capabilities can be derived from the architecture on Figure 3.8.

[1]Service-Oriented Middleware Architecture for CPS [HPK12].

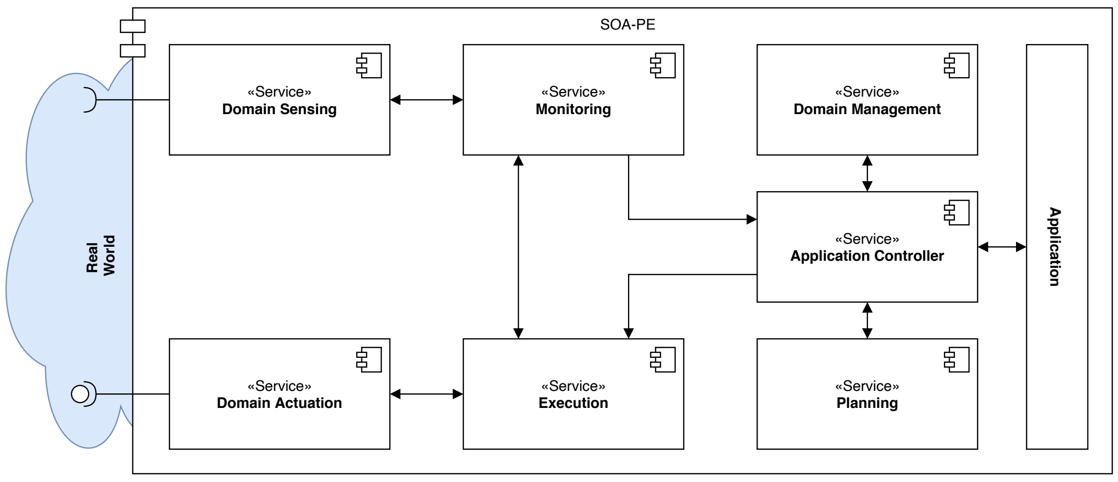


Figure 3.8: Architecture Overview of SOA-PE (cf. [FMJB17, Fig. 2]).

- *Monitoring Capability (Domain Sensing & Monitoring)*
  This capability is responsible for automatically detecting and updating environmental states. By splitting the capability into two services, system logic can be separated from application logic (depending on individual usage). By linking monitoring and execution services, any arising problems can be handled before passing them to the environment.
- *Executing Capability (Domain Actuation & Execution)*
  This capability is responsible for the execution of the created plans. Like the monitoring capability, this capability is split up into individual responsibilities. In order to maintain the flexibility of the system, the domain actuation service is outsourced.
- *Managing and Problem Generating Capability (Application Controller)*
  This capability assumes a managing function. It processes requests from the application, generates problems (from state and goal), and then calls the solver.
- *Modelling and Storing Capability (Domain Management)*
  This capability manages domain information and includes the modeling of this information as well as its storage.
- *Solving Capability (Planning)*
  This capability is responsible for the straightforward creation and replanning of problems.
- *Visualizing Capability (Application)*
  This capability can vary considerably, which makes the system incredibly flexible in terms of its possible applications. In most cases, the application has a user interface, which makes it a visualizing capability.

SOA-PE has excellent connectivity to the environment, mainly because the system is designed for CPS. Besides, basic rules for the creation of SOA have been followed, allowing enormous flexibility on the environment and the application side.

However, the system lacks flexibility regarding the interchangeability of the solver. Therefore the reusability of the whole system as AI Planner is not entirely satisfying.

4 Planning Capabilities

This Chapter deals with AI planning capabilities. It aims to answer the second research question (cf. RQ_2 from Section 1.2). First, existing planning systems are analyzed, and common capabilities are extracted. Subsequently these are classified (see Section 4.2).

In order to find capabilities, all levels of planning systems have to be considered. Some capabilities occur inside planners. Others are already extracted and can be found on system-level, as components or services. Section 4.1 comes up with a list of capabilities, which are evaluated and grouped on Section 4.2.

The Table 4.1 is the result of the extended literature research. The first two columns list the name and source of the publication of the AI Planner. The type column can be used as an indicator for the planner.

A “P” stands for an AI planner designed to solve a plan only (solver). This planner stands in contrast to an AI planner system, which can be identified by a “S”. Such a system does not have its solver but integrates an external planner to be fully functional. To cover all types of planners, compound planners are indicated by “P+S”, which describes a complete AI planning system with a dedicated solver.

Sometimes a capability is offered but not provided by the system itself, or it is not available in all system versions. Then it is indicated by “(X)”.

Name	Source	Type	Capabilities																	
			Parsing	Modelling	Solving	Visualizing	Ver. & Val.	Executing	Monitoring	Managing	Reacting	Converting	Searching	Graph-Handling	Relaxing	Storing	Learning	Explaining	Normalizing	Problem Gen.
CPEF	[Mye99]	P+S	X	X	X	X		X	X											X
FAPE	[DBMIG14]	P	X	X	X	X		X												
Fast Downward	[Hel11]	P	X		X								X	X	X				X	
Fast Forward	[HN01]	P	X		X								X	X	X				X	
GTOHP	[RPFP17]	P	X		X															
Marvin	[RH01]	P	X		X			X					X		X					
O-Plan2	[TDK94]	P+S	X	X	X	X	X	X	X	X	X									X
PANDA	[BKB14]	P+S	X	X	X	X	X	X		X		X		X	X			X	X	X
PELEA	[GAP+12]	P+S	X		X			X	X								X			X
ROSPlanner	[CFL+15]	S	X		(X)		X	X												
RuG Planner	[KLA16]	P+S	X	X	X			X	X	X	X									X
SH Planner	[Geo15]	P+S	X	X	X			X	X	X		X				X			X	X
SHOP	[NCLMA99]	P	X		X															
SHOP2	[NAI+03]	P	X		X	(X)	X						X	X						
SHOP3	[GK19a]	P	X		X		X						X	X	X				X	
SIADEX+BACAREX	[FOCGPP06]	P+S	X	X	X	X			X		X									X
SIPE-2	[Wil00]	P+S	X	X	X	X	X	X	X				X							X
SOA-PE	[FMJB17]	P+S	X		X			X	X	X	X									X
SMP System	[FPD13]	S	X	X	(X)	X	X	X								X				X
UMCP	[EHN94]	P	X		X															

Table 4.1: Capabilities Collection Overview

4.1 Collection

In this section, the dedicated planners are analyzed first and followed by the compositions/systems. The literature and source-code research results are provided on Table 4.1.

According to RQ_2 (“How can planning capabilities be discerned [...]”), the planners and planning systems are analyzed in terms of components, modules, and service levels. Some of the investigated systems are only provided as detailed explanations. The source code is not published. In these cases, only textual analysis is applied.

The following Sections 4.1.1 to 4.1.18 shortly describe each capability.

4.1.1 Parsing Capability

This capability enables the system to accept textually modeling input. Typically, a textual input contains formal modeling of the domain and a problem definition. However, this has a significant impact on reusability since not all users use the same language (e.g., PDDL see modeling languages Section 2.2.3). In order to enable the system to process textual input, an object must be generated. It is essential to make sure that the generated object can be serialized to allow sending using standard formats like JSON or XML. All analyzed planners and systems provide such a capability (cf. Table 4.1).

This step of transforming from a modeling language to a serializable object can be understood as a parsing capability.

4.1.2 Modeling Capability

This capability is tightly coupled with the *Visualizing Capability* (cf. Section 4.1.4). A *Modeling Capability* is usually implemented as a user-interface to ensure easy *Knowledge Engineering*.

Some of the planners and most of the systems provide *Modeling Capabilities* (see Table 4.1). In addition, there are various standalone solutions for modeling using a graphical user-interface (cf. [Geo15; VTC+12; WCM14]).

Thus it can be argued that the modeling step can be considered as a self-contained process step and consequently represents a capability.

4.1.3 Solving Capability

The *Solving Capability* encompasses the core concepts and aspects of AI planning. This capability aims to generate an optimal plan. The exact type of solver is not decisive for the definition, and the specification includes all types (e.g., classical, temporal, hierarchical cf. Section 2.1.3). In the case of a classical planner, a sequence of actions is typically created.

All investigated systems and planners provide structures that enable such behavior and can be classified as *Solving Capability*. Some systems only provide required interfaces to connect a solving component (e.g. SH-Planner with the RuG-Planner [Geo15]).

Since the introduction of hybrid planners (e.g., Panda [BKB14]), the use of such solver modules is also possible, reducing the number of subgroups of solvers. In such a case, problems of different types can be solved with the same solving capability.

However, the internally used supporting capacities can also be subdivided, as shown in the following sections.

4.1.4 Visualizing Capability

Some systems provide embedded user-interfaces for information (e.g. FAPE [DBMIG14], SMP-System [FPD13] or PDDL4J statistics [Pel16]). This capability aims to visualize problems, solutions, or other statistics to entail a mixed-initiative approach [AGH99; FPD13].

All these features can be grouped as *Visualizing Capabilities*.

4.1.5 Verifying & Validating Capabilities

Especially if there is no *Modeling Capability* with great usability[1] available, verification and validation of the user-input are mandatory. Depending on the input supported languages or meta-models, this capability provides syntax and semantics checks. Thus, it can be called *Verifying & Validating Capabilities*.

4.1.6 Plan Execution Capability

The *Plan Execution Capability* provides functionality like simulation or direct execution on the physical world. This capability is not present in most scientific planners (cf. Table 4.1). Only the planning systems provide this capability, which makes them economically active.

Especially in the field of cyber-physical systems the direct execution is mandatory. This can be seen in the case of robotics (e.g. [KLA16]) or IoT (e.g. [Geo15]). Furthermore, some older planners need a plan executions to find a solution (e.g. [DBMIG14; GAP+12; Mye99; RH01]).

4.1.7 Monitoring Capability

Particularly with the use of an *Executing Capability* (see Section 4.1.6), the use of monitoring functionalities is necessary. Especially if the plan execution fails, the monitoring helps to understand the behavior and triggers notifications (e.g., [KLA16]). However, monitoring is also helpful in other cases such as data collection for statistical analyses (e.g., [FMJB17; GAP+12; Mye99]). This capability only occurs on planning systems (cf. Table 4.1) standalone planners do not use encapsulated monitoring units. This behavior can be classified as *Monitoring Capability*.

[1] A good usability should prevent the user from making mistakes. Norman describes various behavior-shaping constraints and forcing functions to prevent errors [Nor88].

4.1.8 Orchestrating & Managing Capabilities

Orchestrating and Managing Capabilities are required if the planning system is already service oriented or at least allows multiple agents (see [FMJB17; Geo15; KLA16]). This capability only occurs on planning systems (cf. Table 4.1).

4.1.9 Reacting Capability

The *Reacting Capabilities* is mandatory if the system or planner delivers plans on runtime. Especially if the plans are executed immediately (see Section 4.1.6). In this case the *Reacting Capability* consumes the *Monitoring Capability* (see Section 4.1.7) and triggers a *re-plan* or *abort* action. This capability is used for instance on RuGPlanner [KLA16] or SOA-PE [FMJB17]. The *Reacting Capability* is defined as a generic event consumer, which calls internal system capabilities to react.

4.1.10 Converting Capability

Converting Capabilities are often required by planners. For instance *HTN2STRIPS* of Panda [BKB14], the problem-converter of SH-Planner [Geo15] or the problem encoding on pddl4j [PF18].

Generally, the *Converting Capability* is optional and depends on the underlying planner. Thus, the definition of the *Converting Capability* is extensive.

4.1.11 Searching Capability

Several of the analyzed planners use search algorithms, to provide a solution [GNK07; Hel11; HN01; NAI+03; RH01; Wil00]. For instance, the *Enforced Hill-Climbing* algorithm is often used and is also part of the Fast Forward Planner. It can be observed that especially older planners rely on this kind of algorithms. Also, many of the planners of that period have a similar structure, as can be seen for example in Marvin, which contains a searching module like the FF-Planner [RH01]. Thus it can be concluded that there is a *Searching Capability*, which can vary in the exact implementation style. In general, this capability is not available in all applications. The requirement is dependent on the *Solving Capability*.

4.1.12 Graph-Handling Capability

Since graphs are used as data structures in planners, *Graph-Handling Capabilities* are required.

For instance the FF-Planner and FD-Planner or Panda-Planner provide such capabilities [BKB14; Hel11; HN01]. This capability is optional and depends on the *Solving Capability* used.

4.1.13 Relaxing Capability

In general, there are two methods to come up with a heuristic to solve a plan. One of them is relaxation. Therefore we consider a less constrained version of the problem. It is easy to find *any* relaxed plan (e.g., greedy approach, can solve in polynomial time). To find the optimal relaxed plans, the *relaxed planning problem* is NP-complete.

This capability is typically also monitoring the *Searching Capability* for heuristic evaluation of states. Generally, *Relaxing Capabilities* are not required by all solvers. The FF-Planner, FD-Planner, Marvin-Planner, require *Relaxing Capabilities*, which are often part of the planners core [Hel11; HN01; RH01].

4.1.14 Storing Capability

In order to reuse domain specifications or other parts *Storing Capabilities* are required. Such capabilities are usually only contained on planning systems since the quick access of domain files is getting an issue. This capability is optional because it only supports the user, but it is unnecessary to get a plan at all. The only exception, which can occur, is the requirement of providing a cache to store temporal data for a planning request. This capability can be found on the Space Mission Planning System or SH-Planner (see [FPD13; Geo15]).

4.1.15 Learning Capability

Since sub-symbolic AI is getting more common, *Learning Capabilities* are also used on AI planning systems. In general, this capability is optimizing performance but is not always necessary for the plan generation. For instance *Pelea* is using learning capabilities [GAP+12].

There are also other hybrid concepts to face new planning problems using a combination of *symbolic* and *subsymbolic* AI. As example Asai and Fukunaga created a system to solve image-puzzles [AF17].

4.1.16 Explaining Capability

The *Explaining Capability* is optional and only rarely used. It is provided by the Panda Planning System [BKB14; SMSB12] and only used for educational purposes.

This capability shows, why the model is correct and what is precisely planned. Therefore the causal and decomposition structure of the plan is shown as a diagram. According to Panda documentation, the *Explaining Capability* aims to answer the following questions [Ins12]:

- “Why do I have to execute this action at all?”
- “Why do I have to execute this action before this one?”

Figure 4.1 shows an example for the explanation of planning steps. Other systems rely on animations to visualize the plan. For example the plugin “Planimation”, which is offered for Planning.Domains [CDE+20].

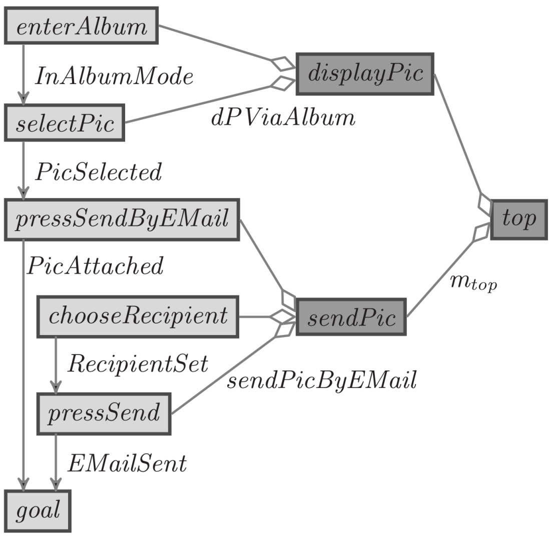


Figure 4.1: Explaining Capability – Plan Steps example [SMSB12, Fig. 1]

4.1.17 Normalizing & Unifying Capability

Normalization can be found in several parts of planning applications. The most common use-case is the normalization of the heuristics. It can be found on the Panda-Planner [BKB14]. This form of normalization can be found on the SH-Planner [Geo15]. Other forms of normalizing can be found on a unifying approach from Kautz and Selman. In this approach, a translation from STRIPS to SAT is used to work on a normalized base [KS99].

4.1.18 Problem-Generating Capability

A classical planning problem consists of the problem-description and the current state. This current environment state has to be checked out before the planner can perform any kind of solving. The *Problem-Generating Capability* is used on some planning systems, e.g. the SH-Planner [Geo15]. In this particular case of use, the current state is automatically gathered using interfaces to communicate with IoT devices.

4.2 Evaluation & Classification

This section aims to answer, how planning capabilities can be classified, so that they represent encapsulated and particular concerns (RQ_2).

All capabilities can be classified in two different ways. The first point of view is derived from business context, described by Leonard-Barton [LB95]. Section 4.2.1 shows the classification from the operational process perspective. The second point of view is derived from *Enterprise Integration Pattens* (see [HW04]). Section 4.2.2 contains the technical viewpoint.

4.2.1 Operational View

Generally all capabilities (see Table 4.1) can be classified into following three categories: *Core Capabilities*, *Enabling Capabilities*, and *Supplemental Capabilities*.

Core Capabilities

Leonard-Barton, Prahalad and Hamel defined core capabilities as capabilities, which satisfy the following three criteria [PH90]:

1. Does the competence provide access to a wide variety of markets?
2. Does the competence make a significant contribution to the perceived customer benefits of the end products?
3. Is the competence challenging to imitate?

Applied to the context of AI planning, the question to be answered is the significant contribution to the perceived customer benefit. According to this definition the minimal set of *Core Capabilities* is a *Modelling*, *Problem Generating*, *Solving*, and *Executing Capability* (see Table 4.2).

Enabling Capabilities

Enabling capabilities are defined as capabilities that do not provide a competitive advantage on their own but are necessary for other capabilities [PH90]. These are *Converting*, *Graph-Handling*, *Managing*, *Monitoring*, *Normalizing*, *Relaxing*, *Searching*, and *Verifying & Validating capabilities* (see Table 4.2).

Supplemental Capabilities

Supplemental capabilities are easily accessible and provide a competitive advantage for the user. This class of capabilities often requires enabling capabilities to run. These are *Explaining*, *Learning*, *Parsing*, *Reacting*, *Storing*, and *Visualizing capabilities* (cf. Table 4.2).

Operational Classification

This section contains a short explanation of the classification for each capability. Table 4.2 provides an overview of the capabilities and operational classification.

Converting

capabilities are **enabling** capabilities, because the benefit for the user is minimal. In most scenarios, this capability is only required to enable other supplemental functionalities.

Executing

capabilities are **core** capabilities; thus, the value of a plan only occurs during execution.

	Capability-Class		
Capability	Core	Enabling	Supplemental
Converting		X	
Executing	X		
Explaining			X
Graph-Handling		X	
Learning			X
Managing		X	
Modelling	X		
Monitoring		X	
Normalizing		X	
Parsing			X
Problem Gen.	X		
Reacting			X
Relaxing		X	
Searching		X	
Solving	X		
Storing			X
Ver. & Val.		X	
Visualizing			X

Table 4.2: Operational view of capability classification

Explaining capabilities are **supplemental** capabilities. Hence the user benefits directly from this feature. Nevertheless, AI Planning is possible without explanation.

Graph-Handling capabilities are **enabling** capabilities since the user does not benefit from this capability.

Learning capabilities are **supplemental** capabilities. Thus the benefits for the user are enormous. In some scenarios, fewer inputs are required, or higher performance can be achieved. Nevertheless, these features are not mandatory to enable essential AI Planning.

Managing capabilities are **enabling** capabilities because the process management is provided, and no direct benefits can be derived.

Normalizing capabilities are **enabling** capabilities since the normalization and unification only affect some capabilities, which is not mandatory for a minimal subset. The user cannot benefit from these normalized objects.

Parsing capabilities are **supplemental** capabilities, hence the most common input-format this days is textual. So the user benefits directly from this opportunity.

Problem Generation capabilities are **core** capabilities since the problem is one of the central elements of AI Planning. In order to provide a solution, the problem is required.

Reacting capabilities are **supplemental** capabilities, because the user benefits from bound activities, e.g., replan triggers or other event-driven reactions. Nevertheless, this capability is not mandatory to ensure basic functionality.

Relaxing capabilities are **enabling** capabilities. Thus they are required for some solvers but do not generate any benefits if they are used standalone.

Searching capabilities are **enabling** capabilities. Since they are required for some solvers but do not generate any benefits if they are used standalone.

Solving capabilities are **core** capabilities. Hence it is the most critical and central capability of AI Planning. Without solving a plan, the system would not generate any benefit at all.

Storing capabilities are **supplemental** capabilities since the user saves time for, e.g., recreating domain models each time.

Verifying and Validating capabilities are **enabling** capabilities because the user can get notifications about failures at an early stage.

Visualizing capabilities are **supplemental** capabilities, thus user-interfaces provide information for the user. So the results or other vital information can be made visible.

4.2.2 Technical View

As already mentioned by Alexander in 1979, patterns are an highly scientific and timeless approach to describe behavior and its context [Ale79]. This section aims to a more technical and engineering perspective than Section 4.2.1. Hohpe and Woolf created a catalogue of *Enterprise Integration Pattens*, which can be applied on the capabilities [HW04].

To reduce complexity, the process of planning can be seen as a black-box pipeline, where each capability is a filter. By analyzing this capabilities it can be observed, that some capabilities are *Source-*, *Filter-* or *Sink-Nodes*. Table 4.3 gives an overview of the node-type of all capabilities.

	Node-Type		
Capability	Source	Filter	Sink
Converting		X	
Executing			X
Explaining			X
Graph-Handling		X	
Learning	X	X	
Managing		X	
Modelling	X		
Monitoring		X	
Normalizing		X	
Parsing	X	X	
Problem Gen.		X	
Reacting		X	
Relaxing		X	
Searching		X	
Solving		X	
Storing			X
Ver. & Val.	X	X	
Visualizing	X		X

Table 4.3: Capability Node-Type Mapping

With this knowledge, the capabilities can be mapped to the *Enterprise Integration Pattens*. Table 4.4 shows all capabilities and the related patterns. Since these patterns are grouped into four classes, each capability is classified by its pattern. These classes are **Endpoint**, **Transforming**, **System-Management** and **Routing**. Each category is shown on the overview on Figure 4.2.

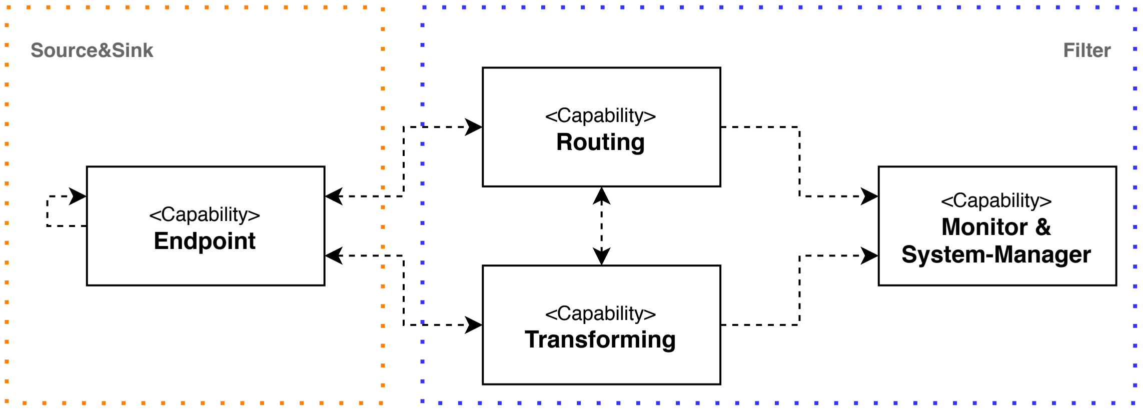


Figure 4.2: Technical Capabilities Classification-Overview

Endpoints represent source- and sink-nodes. These endpoints can call each other or call other filters from the routing or transforming class. System-managers also monitor routing and transforming capabilities.

Capability	**Enterprise Integration Pattern**	**Class**
Converting	“Message Translator”	Transforming
Executing	“Service Activator”	Endpoint
Explaining	“Message Endpoint”	Endpoint
Graph-Handling	“Message Translator”	Transforming
Learning	“Message Endpoint”	Endpoint
Managing	“Process Manager”	Routing
Modelling	“Message Endpoint”	Endpoint
Monitoring	“Control Bus”, “Wire Tap”	System-Management
Normalizing	“Normalizer”	Transforming
Parsing	“Message Translator”	Transforming
Problem Gen.	“Content Enricher”	Transforming
Reacting	“Event-Driven Consumer”	Endpoint
Relaxing	“Message Translator”	Transforming
Searching	“Content Filter”	Transforming
Solving	“Message Translator”	Transforming
Storing	“Message Store”	System-Management
Ver. & Val.	“Content Based Router”, “Detour”	Router
Visualizing	“Polling Consumer”, “Event-Driven Consumer”	Endpoint

Table 4.4: Capability Enterprise Integration Pattern Mapping

Pattern Introduction

This section contains all the necessary patterns to classify capabilities according to the *Enterprise Integration Pattens*:

“Message Translator” – Transforming   The Message Translator pattern is initially a generic solution to remove dependencies between applications based on a standard message format [HW04; Ley18]. In this context, it can be interpreted as a transformation of the content without additional knowledge. Figure 4.3 illustrates this pattern.

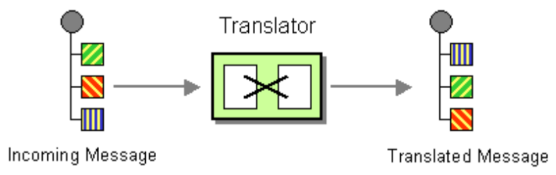


Figure 4.3: Visualization of pattern – “Message Translator” [HW04]

“Normalizer” – Transforming The Normalizer pattern is used to translate messages arriving in different formats into a standard format [HW04; Ley18]. Figure 4.4 illustrates this pattern.

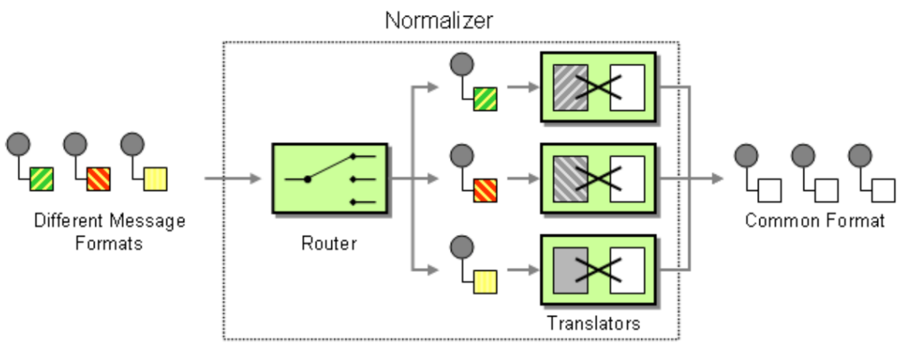


Figure 4.4: Visualization of pattern – “Normalizer” [HW04]

“Content Enricher” – Transforming The Content Enricher pattern provides additional information that is not included in the original message. It looks for missing information and adds it to the outgoing message [HW04; Ley18]. Figure 4.5 illustrates this pattern.

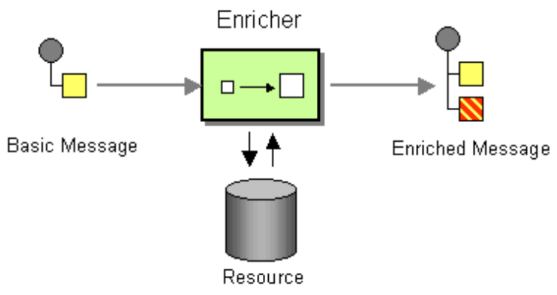


Figure 4.5: Visualization of pattern – “Content Enricher” [HW04]

“Content Filter” – Transforming The Content Filter pattern reduces the number of elements inside a message of the only required ones [HW04; Ley18]. Therfore, it is kind of complementary to the Content Enricher pattern. Figure 4.6 illustrates this pattern.

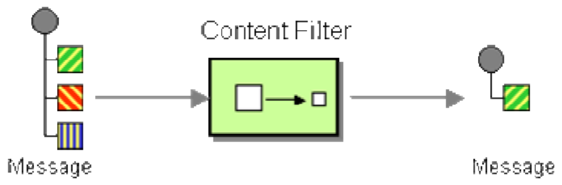


Figure 4.6: Visualization of pattern – “Content Filter” [HW04]

“Service Activator” – Endpoint The Service Activator pattern is used to enable blocking application calls (e.g., non-messaging invocations) without losing the asynchronous behavior of the system [HW04; Ley18]. Figure 4.7 illustrates this pattern.

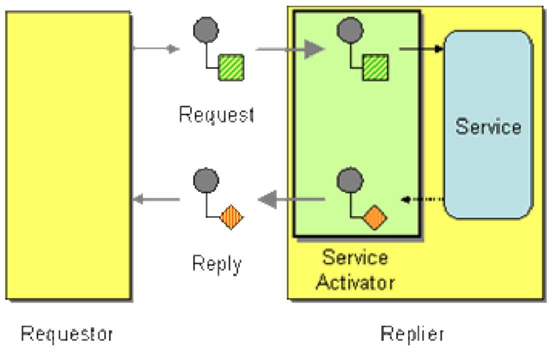


Figure 4.7: Visualization of pattern – “Service Activator” [HW04]

“Event-Driven Consumer” – Endpoint The Event-Driven Consumer pattern enables a service to consume a message as soon as it becomes available [HW04; Ley18]. Figure 4.8 illustrates this pattern.

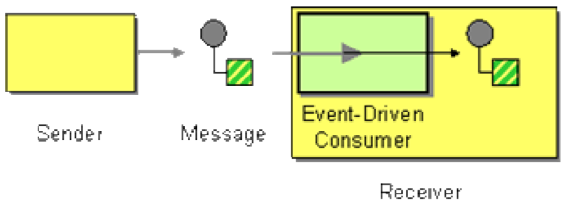


Figure 4.8: Visualization of pattern – “Event-Driven Consumer” [HW04]

“Message Endpoint” – Endpoint The Message Endpoint pattern is a combination of several endpoint patterns to separate messaging, and application logic [HW04; Ley18]. In this context, it can be interpreted as a general pattern of consuming or producing endpoints. Figure 4.9 illustrates this pattern.

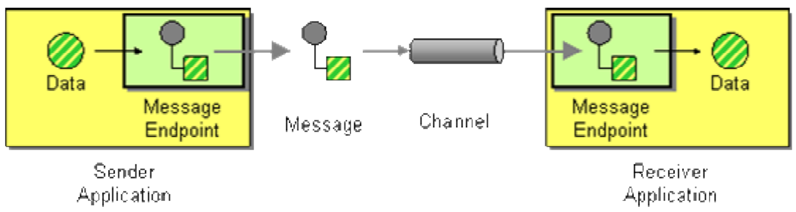


Figure 4.9: Visualization of pattern – “Message Endpoint” [HW04]

“Polling Consumer” – Endpoint The Polling Consumer pattern is also called “synchronous receiver”, and its only behavior is consuming messages (e.g. REST-*GET*) [HW04; Ley18]. Figure 4.10 illustrates this pattern.

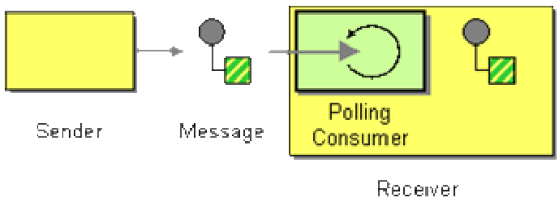


Figure 4.10: Visualization of pattern – “Polling Consumer” [HW04]

“Content-Based Router” – Routing The Content-Based Router pattern enables routing depending on the message content. It is usually used if one function is given by several services for different types [HW04; Ley18]. Figure 4.11 illustrates this pattern.

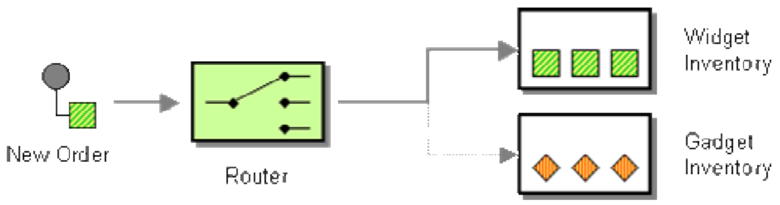


Figure 4.11: Visualization of pattern – “Content-Based Router” [HW04]

“Process Manager” – Routing The Process Manager pattern is used to split one request into multiple sub-tasks and send them to the correct services. Moreover, it is used to maintain the state of the processing steps executed last, caching intermediate results required by future processing steps, and determining the next processing steps based on previous results [HW04; Ley18]. Figure 4.12 illustrates this pattern.

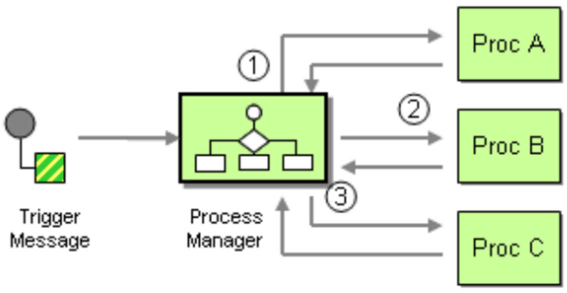


Figure 4.12: Visualization of pattern – “Process Manager” [HW04]

“Control Bus” – System-Management The Control Bus pattern is characteristic for controlling behavior. It is used to provide a single logical point of control [HW04; Ley18]. Figure 4.13 illustrates this pattern.

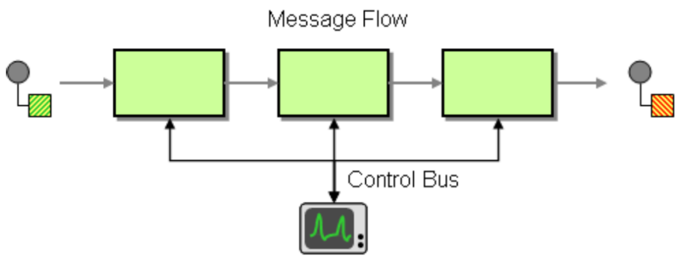


Figure 4.13: Visualization of pattern – “Control Bus” [HW04]

“Detour” – System-Management The Detour pattern enables dynamically routing through additional steps, like, e.g., logging [HW04; Ley18]. These steps are only additional and do not affect the application. Figure 4.14 illustrates this pattern.

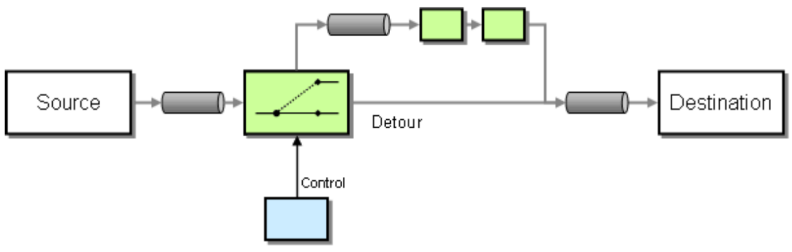


Figure 4.14: Visualization of pattern – “Detour” [HW04]

“Wire Tap” – System-Management The Wire Tap pattern is characteristic for observing and analyzing behavior. It is a pattern to allow the observation of messages at a certain point [HW04; Ley18]. Figure 4.15 illustrates this pattern.

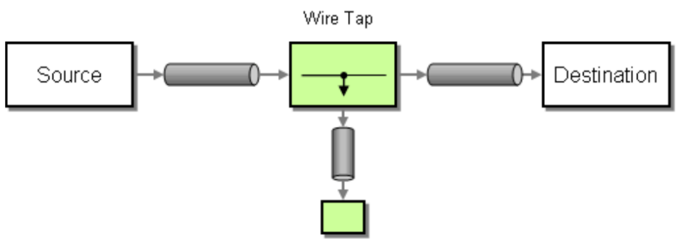


Figure 4.15: Visualization of pattern – “Wire Tap” [HW04]

“Message Store” – System-Management The Message Store pattern saves each message sent to a channel. It can be realized by duplicating each message and publishing it to a particular channel representing the message store [HW04; Ley18]. Figure 4.16 illustrates this pattern.

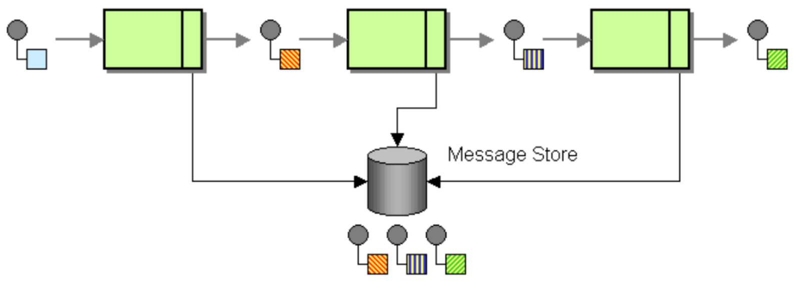


Figure 4.16: Visualization of pattern – “Message Store” [HW04]

“Routing Slip” Pattern This pattern is required in the later course of the work (see Section 5.3.2). According to Hohpe and Woolf the pattern offers the following advantages [HW04]:

- An improvement in the message flow’s efficiency, as only those steps are called, which are necessary to process the request.
- An improvement of efficiency in terms of resources, since far fewer channels are required (no message ping-pong).
- An improvement in Flexibility because each message can be handled individually, and a dynamic process can be mapped. For example, in intermediate steps, messages can be wisely inserted or omitted.
- An improvement in maintainability, since the number of required data models is reduced and thus potential errors can be avoided. This aspect depends strongly on the avoidance of ping-pong messages, so the number of response models can be reduced.

Figure 4.17 illustrates this pattern.

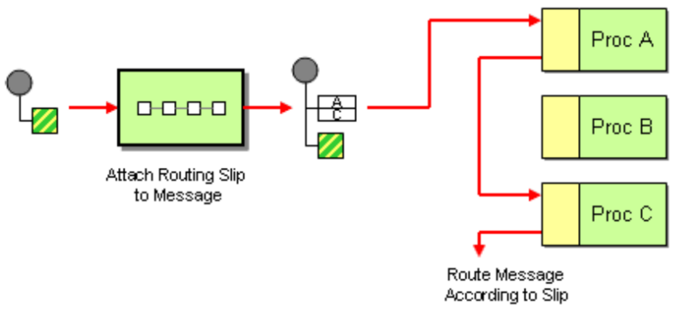

Attach Routing Slip to Message
A
B
C
Proc A
Proc B
Proc C
Route Message According to Slip

Figure 4.17: Visualization of pattern – “Routing Slip” [HW04]

Applying patterns to capabilities

This section contains the explanation of the mapped pattern for each capability. All capabilities and patterns can be found on the overview Table 4.4.

Since *converting capabilities* are a filter-node and only change the syntax of an incoming object the Message Translator patterns fits best (cf. Table 4.4). Thus the capability is classified as *Transforming Capability*.

Executing capabilities are sink-nodes and can be realized by using the Service Actuator pattern. Thus the capability is classified as *Endpoint Capability*.

Explaining capabilities are also sink-nodes. The specific implementation depends on the supported functions; nevertheless, the underlying pattern is Messaging Endpoint. Thus the capability is classified as *Endpoint Capability*.

Graph-Handling capabilities provide several utility functions on graph data-structure. From a system-level view, these functions only modify an incoming object. Hence, the Message Translator pattern is fitting, and the capability is classified as *Transforming Capability*.

Learning capabilities are commonly used as wrapping layers, which makes them source- and sink-nodes. Since the implementation depends on the dedicated functionality, only a general pattern seems to fit. In this case, the generic Messaging Endpoint pattern is considered. Thus the capability is classified as *Endpoint Capability*.

Since AI-Planning is process-oriented, *managing capabilities* are required to keep track of the results, subtasks and route the requests to the correct services. The Process Manager pattern covers precisely this behavior. Thus the capability is classified as *Router Capability*.

Modelling capabilities are commonly used to receive model-related information directly from the user, like problems or domains. This makes this capability a source-node and leads to the usage of the Messaging Endpoint pattern. Thus the capability is classified as *Endpoint Capability*.

Monitoring capabilities are often required if real-world systems are involved. In order to make decisions transparent, logging is mandatory. Two related patterns can realize this task. On the one hand, a Control Bus can be used. The benefits are apparent since all commands can be easily logged. On the other hand, it is possible to add Wire Taps on each point, where logging is required. The second is a more punctual approach. So the usage of the correct pattern depends on the case of use. Nevertheless, both patterns belong to the same category, so the capability is classified as *System-Management Capability*.

Normalizing capabilities are filters that are normalizing and unifying objects. This leads to the transforming pattern category. Since there is an existing pattern for normalizing, the Normalizer pattern is selected. Thus the capability is classified as *Transforming Capability*.

Parsing capabilities are required in multiple scenarios, where the user already modeled the domain and the problem in textual form in a planning language like, e.g., PDDL. In order to use the system, the string needs to get translated to an object data structure. This transformation can be realized by using the Message Translation pattern. Therefore the capability is classified as *Transforming Capability*.

In order to minimize the user-input, some systems provide *problem generating capabilities*. These capabilities add additional information to a given request by requesting, e.g., the current environment state from the other capability. The Content Enricher pattern describes this behavior. Hence the capability is classified as *Transforming Capability*.

Reacting capabilities are sink-nodes and characteristics for their reactive behavior. Since they can react to environment changes or other events, the pattern of Event-Driven Consumer fits best. For this reason, the capability is classified as *Endpoint Capability*.

Relaxing capabilities are filters, modifying data structures, so the message Translator pattern can be applied. Thus the capability is classified as *Transforming Capability*.

Searching capabilities are removing elements from the input to return only fitting element ones. The Content Filter pattern can describe this capability. Hence the capability is classified as *Transforming Capability*.

Solving capabilities are creating a solution (plan). From a system point of view, this capability only modifies inputs. Hence the Message Translator pattern can be applied. Therefore the capability is classified as *Transforming Capability*.

Since *storing capabilities* are mandatory to minimize user-inputs, some data has to be stored. This could include domain models or other large temporary data. The Message Store pattern can describe this sink-node capability. Thus the capability is classified as *System-Management Capability*.

Verifying and Validating capabilities can be used in two different cases. They commonly use the verification and validation of a model before it is passed to the next steps. The application of the Content-Based Router can describe this behavior. It also ensures the user is notified about an error. The second option is only used for monitoring reasons and can be implemented by using the Detour pattern. Since the second scenario is only rarely used and grounded on a router, the capability is classified as *Router Capability*.

Visualizing capabilities are sink-nodes, which are only used to visualize results or various statistics. Depending on the requirements, two patterns of the Endpoint category can be chosen. On the one hand, the Polling Consumer pattern can be used. This pattern has some drawbacks, according to the performance. On the other hand, the Event-Driven Consumer pattern can be used to enable reactive high-performance components. Thus the capability is classified as *Endpoint Capability*.

5 Designing Capabilities

This Chapter provides an analysis of requirements, on Section 5.1, that have to be fulfilled to create usable, interoperable, and reusable capabilities. Afterward, on Section 5.2, the architecture of a capable *AI Planning System* is presented.

5.1 Requirements

This section aims to answer RQ_3 ("Which requirements need to be satisfied to guarantee usability, interoperability, and reusability?" cf. Section 1.2) First, on Section 5.1.1, the usability quality attributes are analyzed and defined. Afterwards, the interoperability attributes on Section 5.1.2 and last but not least, reusability attributes on Section 5.1.3 are elaborated.

5.1.1 Usability

Usability starts on the interaction side. Independent of the user's skills, the user should be able to use the system. Thus a certain level of "ease of use" is mandatory to be usable.

The ISO 9241-11 standard provides a formal definition for usability (see Definition 5.1.1).

Definition 5.1.1 (DIN ISO 9241-11:2018 [Iso])
The usability of software depends on the context in which it is used. The usage context includes the users, the goals, the tasks, the resources [...]. In general, the following properties can be measured:

- ***effectiveness** for the solution of a task*
- ***efficiency** in task processing*
- *user **satisfaction***

Since not all capabilities aim to provide a user-interface, some only provide software interfaces. The requirements need to be defined in a way to be applied to both kinds of capability services. In order to provide best possible usability, the "Ten Usability Heuristics" of Nielsen is refereed here [Nie05]. These heuristics are initially created to describe user-interfaces, but they can also be applied to APIs and complete systems. Some of the roles are tightly coupled to the "Ten Golden Designing Roles" of Shneiderman et al. [SPC+16].

1. *Visibility of the system status*
   The system should always keep users informed of what is going on by providing appropriate feedback within a reasonable time [Nie05].

2. *Match between system and the real world*
   The system should speak the users' language and follow the conventions of the real world [Nie05].
3. *User control and freedom (only for UI)*
   The system should support undo and redo functionality, to give the user full control.
4. *Consistency and standards*
   The system should be consistent and aim for standardized solutions.
5. *Error prevention*
   The system should prevent the user from making mistakes, e.g., early validation and good error messages.
6. *Recognition rather than recall*
   The system should provide all the necessary information to minimize the short-term memory of the user.
7. *Flexibility and efficiency of use (only for UI)*
   The system should allow shortcuts for advanced and frequent-users.
8. *Aesthetic and minimalist design (only for UI)*
   The system should only view important information to do not diminishes their relative visibility.
9. *Help users recognize, diagnose, and recover from errors*
   Error messages should be expressed in clear language, indicate the problem exactly and suggest a constructive solution [Nie05].
10. *Help and documentation*
    The information should be easy to find, focus on the user's task, list concrete steps, and not be too extensive [Nie05].

5.1.2 Interoperability

Interoperability is majorly achieved by using unified models on every endpoint. So concepts for messaging and Loose Coupling (see Section 2.3.1) are mandatory.

Delgado provided a level model of interoperability in 2013 [Del13]. It was initially created in the IoT context but can be generalized and applied to this thesis. Delgado used a simple transaction to extract four levels of interoperability. Figure 5.1 shows a simple transaction on a decentralized system. The red arrow indicates a request, which passes all four levels on the customer and the provider side.

The interoperability levels are described as following [Del13]:

- *Organizational*: Each request follows a purpose. This level contains the strategy, the partner's choice, and the outsourcing of these value chain steps.
- *Semantic*: To ensure both services can interpret the resource in the same way. This means the domain, the rules, and the choreography must act similarly. Thus the meaning of a resource is the same on both sides.

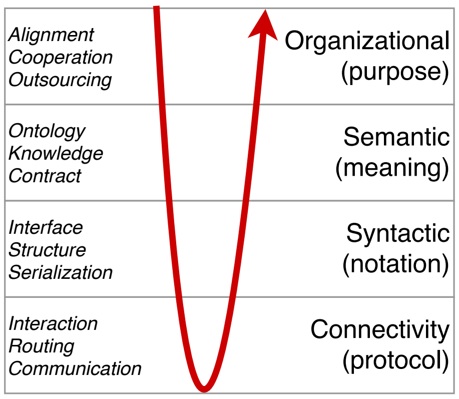


Figure 5.1: Levels of interoperability in a simple transaction (cf. [Del13, Tab. 1])

- *Syntactic*: This level aims to the notation (format) of the request. The resource must follow the same structure according to defined schematics. In most cases, REST APIs or messaging channels use JSON or XML as a message format.
- *Connectivity*: This level aims to the protocol, which is the essential step of transferring a request. It contains the message protocol selection, the routing to the correct endpoint, and the underlying network protocol.

In order to support the best possible interoperability, the capabilities have to aim for the best interoperability on each level (*organizational, semantic, syntactic, connectivity*) [Del13].

The following actions can be derived from these requirements and impact the architecture:

- The *connectivity interoperability* is reached by using a common protocol on all capabilities. This leads to the decisions to use REST and MOM.
- The *syntactic interoperability* is reached by using a standard message format. To lower the complexity, JSON is used.
- The *semantic interoperability* is reached by providing detailed schema definitions at the documentation and the endpoint implementation.
- The *organizational interoperability* is reached by providing only a single point of interaction to the user. The *Modelling Capability-Service* is the only service which process user inputs.

5.1.3 Reusability

In order to allow the reuse of the capabilities, the number of dependencies has to be minimal. This requirement is also related to the Loose Coupling autonomies (see Section 2.3.1). Since reusability is not included in the software quality attributes of ISO 9126, other definitions and evaluation metrics are necessary [KG12].

Choi and Kim created a quality model for evaluating reusability of services in a SOA [CK08]. While approach for object-oriented programming on component level (e.g. [SAKS10]) can not be efficiently applied in SOA, Choi and Kim created a capable quality model. The model is grounded on a different definition of reusability (see Definition 5.1.2), based on the IEEE definition of reusability.

Definition 5.1.2 (Reusability)
"Reusability of a service is the degree to which the service can be used in more than one business process or service application, without having much overhead to discover, configure, and invoke it [CK08]."

Reusability Metrics of Choi and Kim

Choi and Kim came up with five quality attributes of reusable services, which was derived from the key service features. All metrics are normalized ($M_x \in [0, 1]$).

Business Commonality – customer dependent The Business Commonality metric indicates the relationship between functional and non-functional commonness of requirements by domain users [CK08]. To ensure that a wide range of service consumers can use a service, it should offer various applicable functionalities and non-functionalities, besides not being application-specific. The Equation (5.1) shows the formula for this metric M_{BC}. It consists out of Equation (5.2) and Equation (5.3).

$$(5.1)\quad M_{BC} = \frac{\sum_{i=1}^{n}(FunctionalCommonality_i * NonFunctionalCommonality_i)}{n}$$

$$(5.2)\quad FunctionalCommonality_i = \frac{\text{\# customers require operation } i}{\text{total \# customers}}$$

$$(5.3)\quad NonFunctionalCommonality_i = \frac{\text{\# customers require operation } i}{\text{total \# customers}}$$

In order to support the best Business Commonality, most requested cases have to be covered. Additionally, no application-specific properties should be contained.

Modularity The Modularity metric measures the degree of independence of service to other services [CK08]. The Equation (5.4) shows the formula for this metric M_{Mod}.

$$(5.4)\quad M_{Mod} = 1 - \underbrace{\frac{\text{\# dependent}}{\text{total \#}}}_{\text{dependency of operations}}$$

To support the best Modularity, the number of dependent service operations has to be minimal (best case = 0).

Adaptability – customer dependent The Adaptability metric measures the capability of the service to be well-adapted to different consumers by considering customer satisfaction related to consumers' expectations [CK08]. The Equation (5.5) shows the formula for this metric M_{Ad}.

$$(5.5)\quad M_{Ad} = \frac{\text{\# satisfied customers}}{\text{total \# customers}}$$

To support the best adaptability, the requirements of a capability have to be analyzed and fulfilled. Additionally, all settings should be configurable by the environment variables instead of being hard-coded.

Standard Conformance The Standard Conformance metric measures the conformity of mandatory and optional standards. The Equation (5.6) shows the formula for this metric.

$$(5.6)\quad M_{StdCon} = \underbrace{W_{mandatory} * \frac{\text{\# conform}}{\text{total \#}}}_{\text{mandatory standards}} + \underbrace{W_{optional} * \frac{\text{\# conform}}{\text{total \#}}}_{\text{optional standards}}$$

$W_x \in [0, 1] \mid W_{mandatory} + W_{optional} = 1$
W_x is the weight to define the importance of optional standards.

In order to support the best Standard Conformance, the number of satisfied standards has to be maximal.

Discoverability – customer dependent Discoverability is intended to measure the extent to which consumers can quickly and correctly understand the service. To allow the service to be quickly and correctly detected, the service should be specified in detail concerning syntax, and semantics [CK08]. The Equation (5.7) shows the formula for this metric.

$$(5.7)\quad M_{Disc} = \underbrace{W_{syntax} * \frac{\text{\# described}}{\text{total \#}}}_{\text{syntax elements}} + \underbrace{W_{semantics} * \frac{\text{\# described}}{\text{total \#}}}_{\text{semantics elements}}$$

$W_x \in [0, 1] \mid W_{syntax} + W_{semantics} = 1$
W_x is the weight to define the importance of syntax and semantics.

In order to support the best Discoverability, the number of To be able to use the services most effectively, the services should first be specified in an easily understandable form.

Reusability Metrics of Cardino et al.

Since Choi and Kim many metrics are based on user surveys, not all of them are applicable in such an early stage of research and can only be used after release. Additionally, essential aspects of reusability were not reflected. Cardino et al. presented 1997 a metric for the assessment of the reusability of frameworks, which can be transferred to the planned approach [CBV97]. Figure 5.2 shows the elements of which reusability is composed.

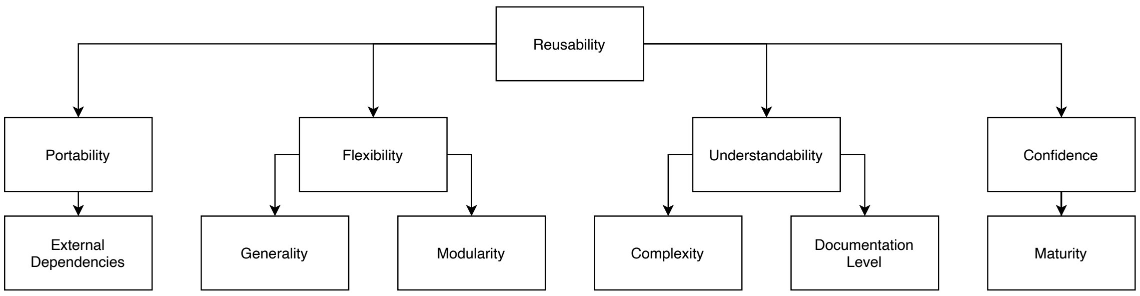


Figure 5.2: Overview of factors-criteria decomposition for assessing framework reusability (cf. [CBV97, Fig. 2]).

According to Cardino et al. reusability is based on four pillars of quality attributes:

Portability The first pillar is *Portability*, which consists of the number of external dependencies (here we can see a great similarity to Equation (5.4)). Portability is about removing the service from its previous environment and deploying it in a new one. For this purpose, it is necessary to minimize the number of external dependencies [CBV97].

Flexibility The quality attribute of *Flexibility* is about adaptability, also within the service. Therefore a certain degree of *Generality* must be maintained. *Flexibility* is composed of two underlying attributes, whereby *Generality* is the first [CBV97]. For instance, are all internal functions limited to the use within one field of application, or can parts be extended to other areas without much additional effort? The second is *Modularity*, which is also called in other papers (e.g. Modularity of Service (MoS) [OLK11]). The *Modularity* is about the internal dependencies a service has on libraries or similar.

Understandability In general the *Understandability* is the crucial point [OLK11]. If developers do not directly understand the operations and usage of a service, it will be rejected for reusability in most cases. The *Complexity* aspect plays a decisive role in this respect [CBV97]. Cardino et al. distinguishes between structural complexity and inheritance complexity [CBV97]. These two factors focus strongly on object orientation, but complexity in this context can also refer to the entire data structure. If it is not possible to get an overview of the service structure within a short period, it is probably too large.

The complexity can also be reduced by adequate documentation. Complex content can be simplified by proper *Documentation*, especially by using standard diagrams (e.g. UML). Besides, particularly in the context of AI, the scientific description (or at least linking to the corresponding literature) of algorithms is necessary.

Confidence The attribute *Confidence* is the most difficult to grasp. Since it can only be measured during an evaluation of the user, some of the metrics of Choi and Kim can. Cardino et al. suggests a metric of *Maturity* for this purpose. This is based on the number of errors during a design review and thus reduces the *Confidence*.

5.1.4 Summery

In this section, the requirements are summarized in the form of quality attributes. The following design decisions are made based on the optimization of these requirements.

Usability requirements (cf. Section 5.1.1): With good usability, it does not matter if a capability is a front-end or back-end service; the appropriate usability criteria are the same. Only the metrics for evaluating them depend on the context. In summary, the quality characteristics to be maximized are the following:

- Effectiveness
- Efficiency
- User satisfaction

Interoperability requirements (cf. Section 5.1.2): Interoperability applies to the entire system and must be implemented at all levels to meet the requirements. As already mentioned, interoperability can be divided into four sub-areas, each of which must achieve maximum interoperability. In summary, the following four requirements result:

- Connectivity interoperability
- Syntactic interoperability
- Semantic interoperability
- Organisational interoperability

Reusability requirements (cf. Section 5.1.3): To reduce the development effort for new AI Planning capabilities, the reusability of existing concepts and services is indispensable. The literature research has resulted in a summary of four reusability quality characteristics, which serve as requirements for designing new capabilities. These are:

- Portability
- Flexibility
- Understandability
- Confidence

5.2 Architecture

In principle, it is possible to create two architects to implement capabilities as services. These mainly differ in the way they communicate and in the degree of decentralization.

The first approach implements a strictly Representational State Transfer (REST) based SOA. The second approach is based on a mainly message oriented communication, using a MOM (see Figure 5.4). In this case, REST is only used for front-end communication.

Since this thesis aims to develop a usable, interoperable, and reusable system, a central capability service used for pure process control does not seem suitable. Since the central capability service must always be adapted when changing a capability, this architecture lacks usability. In theory, this service could be set up generically to be configured via its environment, but this would not be easy to realize because of the time frame. Besides, the scalability of the system would be endangered. Thus, the first approach is not sustainable.

The second approach causes a considerable overhead in the implementation since a message broker must always be started in addition to the capabilities (Apache ActiveMQ, RabbitMQ, or Apache Kafka can be used for this purpose). Nevertheless, this approach offers the best loose-coupling properties. Especially with a high system load and several clients, messaging-based communication in the publish-subscribe stack is very performant. Furthermore, the system scales very well horizontally since each capability has its own incoming and outgoing topic. However, the most crucial point is that Capabilities themselves can request other Capabilities directly without another gateway, checking the availability first. It is essential to increase reliability and not violate the principles of services. Hence, capabilities must be implemented stateless.

5.2.1 System Architecture

This section introduces the basic architecture model. In consideration of the general service requirements, a specific architecture is created.

In general, the hub-and-spoke pattern is used in this architecture (see [HW04]). In most cases, scaling the hub in such a pattern is more complicated, but new services can be easily connected to a hub and scaled just as quickly.

Figure 5.3 shows an abstract view of the system architecture. The system has a front-end service, the MOM, and various back-end capability services.

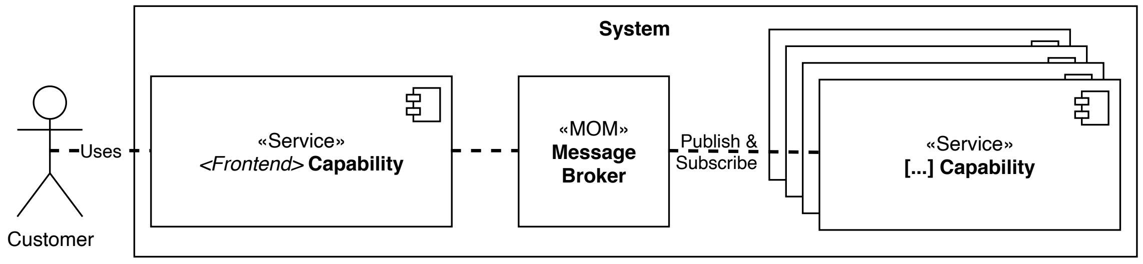


Figure 5.3: Abstract Architecture Overview

The explicit architecture is a more detailed description of the abstract architecture (see Figure 5.3). In this explicit architecture, all the capabilities included in Section 4.1 are provided. This architecture aims to define the composition and thereby facilitate the implementation of the capability services.

The communication is based on a MOM, while the front-end capability services are implemented using REST communication. Additionally, the *managing capability* is used as an entrance gate to the message broker. Thus, system relevant information about routes does not have to be stored in the front-end. All capabilities are colored according to their class on Figure 5.4. As already shown in Figure 5.4 some capabilities require additional interfaces. For example, the *executing capability* is connected to the outside world to execute plans. Also the *Storing capability* needs a database.

5.2.2 Messaging

Within the MOM, a standard structure is essential to integrate new capabilities as efficiently as possible. For this purpose, each capability service is implemented with at least one messaging endpoint.

- Request topic endpoint *(CONSUME)*:
  `<ver>.<capability-class>.<capability-name>`
- (Optional) Monitoring endpoint *(PUBLISH)*:
  `<ver>.<capability-class>.<capability-name>.monitoring`

The incoming topic is used for order acceptance. The exact implementation depends on the MOM used.

To avoid blocking code, outgoing topics are not provided since this would encourage polling endpoints. It is necessary to specify a response topic in the request message to route the message after processing dynamically. The underlying pattern is “Request-Reply” with an extension of the “Return Address” pattern [Ley18].

In case several sub-steps are necessary for the processing of a task, which is divided into other capabilities, the state must be stored temporarily. To keep the IDEAL properties (see [BL19; FLR+14]) of the system, the use of an isolated state is necessary. Therefore capabilities, which

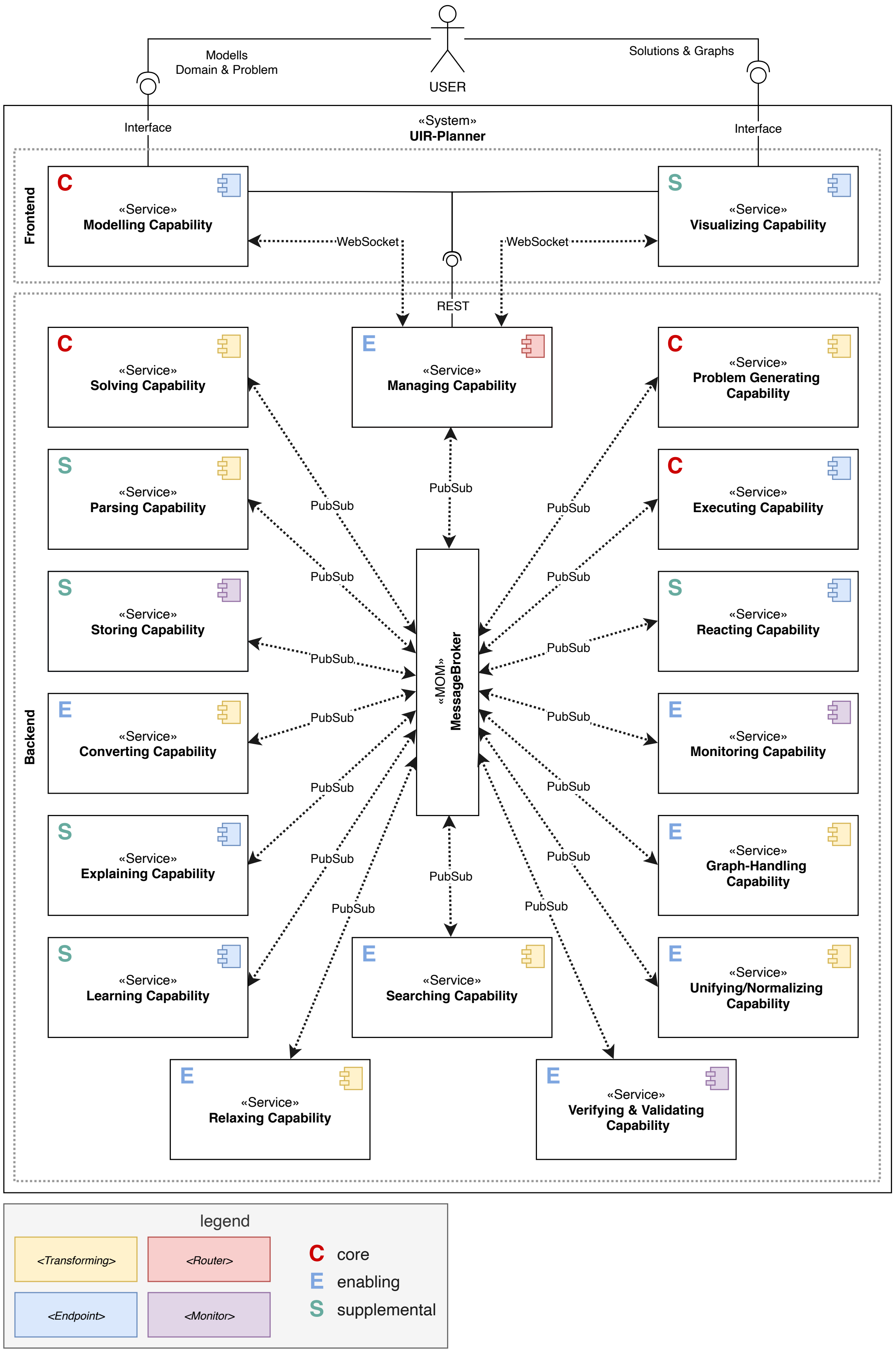

USER
Modells
Domain & Problem
Solutions & Graphs
Interface
Interface
«System»
UIR-Planner
Frontend
C «Service» Modelling Capability
S «Service» Visualizing Capability
WebSocket
WebSocket
REST
Backend
C «Service» Solving Capability
E «Service» Managing Capability
C «Service» Problem Generating Capability
S «Service» Parsing Capability
C «Service» Executing Capability
S «Service» Storing Capability
S «Service» Reacting Capability
E «Service» Converting Capability
E «Service» Monitoring Capability
S «Service» Explaining Capability
E «Service» Graph-Handling Capability
S «Service» Learning Capability
E «Service» Searching Capability
E «Service» Unifying/Normalizing Capability
E «Service» Relaxing Capability
E «Service» Verifying & Validating Capability
«MOM» MessageBroker
PubSub
legend
<Transforming>
<Router>
<Endpoint>
<Monitor>
C core
E enabling
S supplemental

Figure 5.4: Architectural Overview of the UIR-Planner

call other capabilities to place requests, must store them externally. To merge the results in the corresponding capability service, a so-called “Correlation Identifier” is used to guarantee the correctness of the correlation [Ley18].

For general monitoring the common topic (`<ver>.management.monitoring-capability`) is provided. Thus all relevant information can be delivered to a *monitoring capability* (cf. control-bus pattern). Each message is a log-entry, so the sender capability name is mandatory to provide a better overview.

Since some of the capabilities use capability invocation, an additional generic sending endpoint is mandatory.

5.2.3 Interaction-Concepts of Capabilities by Classification

This section shows architecture patterns for each class of capabilities. Each pattern describes a way to connect the capability with others. Therefore a visualization and a short step by step explanation are provided.

Endpoint Capabilities

Since endpoint capabilities can occur in different variants, a general pattern is not possible. Therefore, a distinction between front-end and back-end services must be made.

Back-end All capabilities within the class “Endpoint Capabilities”, which does not have to process direct user input, can be considered back-end capabilities. Figure 5.5 shows an example case for the processing of a request.

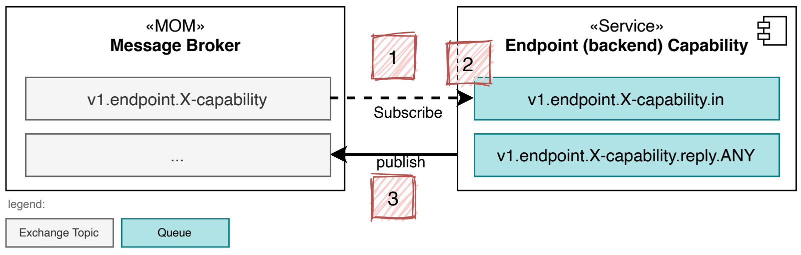


Figure 5.5: Endpoint Capability (back-end)

The following steps are also highlighted on Figure 5.5.

1. The back-end capability service has an established subscription to its incoming exchange topic and starts processing the message.
2. The request is received and placed in a queue. All the requests will be processed after each other (FIFO). The results are pushed to the corresponding reply queue.
3. The response is taken from the reply queue and send to the corresponding topic.

Front-end All capabilities within the class “Endpoint Capabilities”, which have to process direct user input, can be considered front-end capabilities.

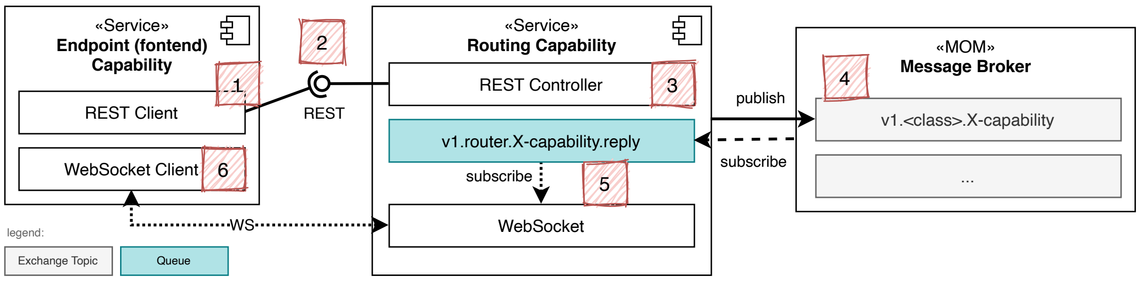


Figure 5.6: Endpoint Capability (front-end)

1. The front-end capability service receives new information from user input. All information is placed at the request body of a REST POST message.
2. The front-end capability service calls the REST endpoint of a routing capability.
3. The routing capability service process the request and sends it to a solving capability using the MOM.
4. After the system processed the request, the response or error is passed back to the routing capability. A WebSocket is used to send the asynchronous response back to the front-end.
5. The Front-end receives the response or a custom-error and processes it (e.g., visualization).

Transforming Capabilities

Since all capabilities in the class “Transforming Capabilities” are identical from an architectural point of view. One pattern can be applied to all capabilities.

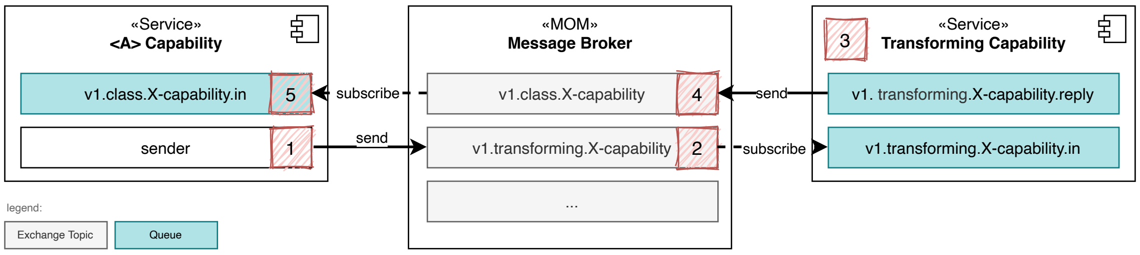


Figure 5.7: Transforming Capability

Figure 5.7 shows a sample request processing of a transforming capability.

1. Service A creates a new request and pushes the message to the corresponding transforming topic on the MOM.
2. The transforming service has an established subscription to its incoming topic and the message is pushed to the queue of the transforming capability.

3. The transforming service process the request and sends it to the given response topic.
4. The response is placed on service A's capability topic in the MOM.
5. Service A has an established subscription to its incoming topic and starts processing the message.

Since the architecture allows dynamic routing, the transforming capability is free to decide if any other transformation is necessary to reach the requested response format.

System-Management Capabilities

System management and monitoring capabilities differ slightly at the architecture level. Only the origin of the information varies within the MOM.

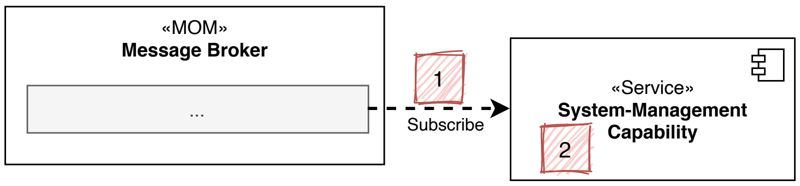


Figure 5.8: System-Management Capability

Figure 5.8 shows an example of the system-management capability behavior.

1. The Service has an established subscription to at least one topic.
2. The Service process the message.

Router Capabilities

Since there is only one capability classified as "Router Capability" yet, one architecture can be applied. The basic functionality was already introduced on Figure 5.6. So this explanation will focus on the global error handling, which is done by the routing capability.

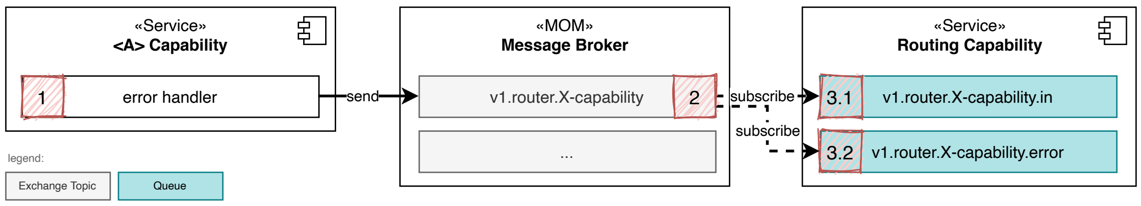


Figure 5.9: Router Capability

Figure 5.9 shows the architecture pattern and behavior for a "Router Capability".

1. Service A throws a custom error on processing a request. The request is handled and sent to the routing capability topic.

2. The routing service consumes the message from the topic and pushes it to the correct queue.
3. Normally the request is pushed to the in-queue (see 3.1 on Figure 5.9). In this Example the message is an error, so the error-queue is responsible (see 3.2 on Figure 5.9). Depending on the specific implementation, the response or error is processed.

5.3 Interfaces and Standards

This section describes the interfaces that the system requires. Section 5.3.1 describes the state of the art interfaces. Afterward, the messaging interfaces are described in Section 5.3.2.

5.3.1 Standard-Interfaces

The AI Planning community has few general standards. In general, the community meeting at IPC is the only major group of researchers that can adopt AI Planning standards. These competitions are organized by the International Conference on Automated Planning and Scheduling (ICAPS) and are an essential part of the exchange.

To compare the planners, they use benchmarks that can be submitted in advance. Also, the languages used for modeling are announced in advance. In the case of PDDL, a validator (e.g., VAL) is also provided to pre-validate the domains and problem files. The output formats also vary among the years and the corresponding tracks.

Over time, the focus of the competitions and the modeling languages of the tasks have changed. In the following, the IPCs of the years 2014 to 2020 are given as an example:

Year	IPC Name (languages)
…	…
2014	Deterministic & Learning & Probabilistic Planning Track [VCG+14]: Modelling languages: PDDL, RDDL
2016	Unsolvability: This IPC aims to the ability to detect early on classical automated planning, if a plan has no solution [Mui16]. Modelling language: PDDL
2018	Classical & Probabilistic & Temporal Track [PTB+18]: Modelling language: PDDL
2020	Hierarchical Planning: This IPC is the first hierarchical planning comptition [Neb20]. Modelling languages: SHOP2, HDDL

There are no global standards besides the conventional input by defined languages (typically defined using Backus-Naur-Form (BNF)). The data models used inside the planners are dedicated and optimized for performance in the majority of projects.

To handle all possible required fields of the planners, a gigantic interface would have to be created, which does not improve the system's usability. Inheritance can simplify the structure, but not all programming languages allow multiple inheritances, which weakens interoperability.

To create a sustainable meta-model, it must first be validated whether the corresponding AI planners and planning systems can be broken down to provide the classified capabilities. Subsequently, all occurring classes must be listed and compared with each other (semantic and syntactic). When using hybrid planners, creating a meta-model is difficult because wrappers or many abstract classes are already used for implementation. Since splitting the planners into single capabilities is very extensive, this research step is not included in this thesis's scope.

Because of this interface diversity, the new design does not require any unique attributes and naming. The restriction of the interfaces reduces the interoperability and forces to use of additional converters.

5.3.2 Standardized Messages

Every message sent via the MOM is based on the `BaseMessageInterface` (see Listing 5.1).

Listing 5.1 Kotlin example of the base-message interface.

```
interface BaseMessageInterface {
    var requestId: String
    var callStack: MutableStack<CallStack>
    val content: Any?
}
```

It consists of three attributes. The first one is the `requestId`, which is used as correlation identifier (see [HW04]). So all messages can be assigned to an initial user request. The second field `callStack` is a stack implementation of `CallStack` objects (see Listing 5.2). The `CallStack` is used to implement the *Routing Slip* pattern (see [HW04]).

To realize the routing slip pattern, the introduction of an address-stack is essential. The first element of the stack represents the next service, which is called. All elements below it represent the following steps. After the arrival of the message the top element of the call stack is extracted (`peek()`) and removed from the stack (`pop()`).

A call stack element consists of several fields (cf. Listing 5.2). The field `topic` contains the service's address (also called the main subject). The `routingKey` determines which queue will later process the request. Since some steps require additional information which may not be transferred (contained on the `content` field of the request) by the intermediate steps, a state map is optionally available (see `state`). The architecture provides a storing capability, which can store single objects via key-value. In the case of large objects (too large to be always transferred), the `state` can contain unique keys and load the objects from the store on usage. The field `replyClass` specifies the class name of the expected return type.

Listing 5.2 Kotlin implementation of the CallStack.

```kotlin
data class CallStack(
        val topic: String = "",
        val routingKey: String = "",
        val state: Map<String, String>? = null,
        val replyClass: String = Map::class.java.name
)
```

Finally, the last field of the `BaseMessageInterface` is the `content`, which varies depending on the capability.

6 Prototyping Capabilities

This chapter describes the implementation of the prototype. According to its intended quality characteristics, the prototype is called UIR-Planning. Where "U" stands for usable, "I" for interoperable, and "R" for reusable.

This prototype serves to illustrate the processes in UIR-Planning. Currently, only PDDL is supported in this prototype. Whereby the interfaces and the whole architecture are designed to be extended. Special attention is paid to dynamic routing to avoid the classical ping-pong process and to increase reusability.

First of all, Section 6.1 explains implementation details that span the full capability. This includes especially the technology selection.

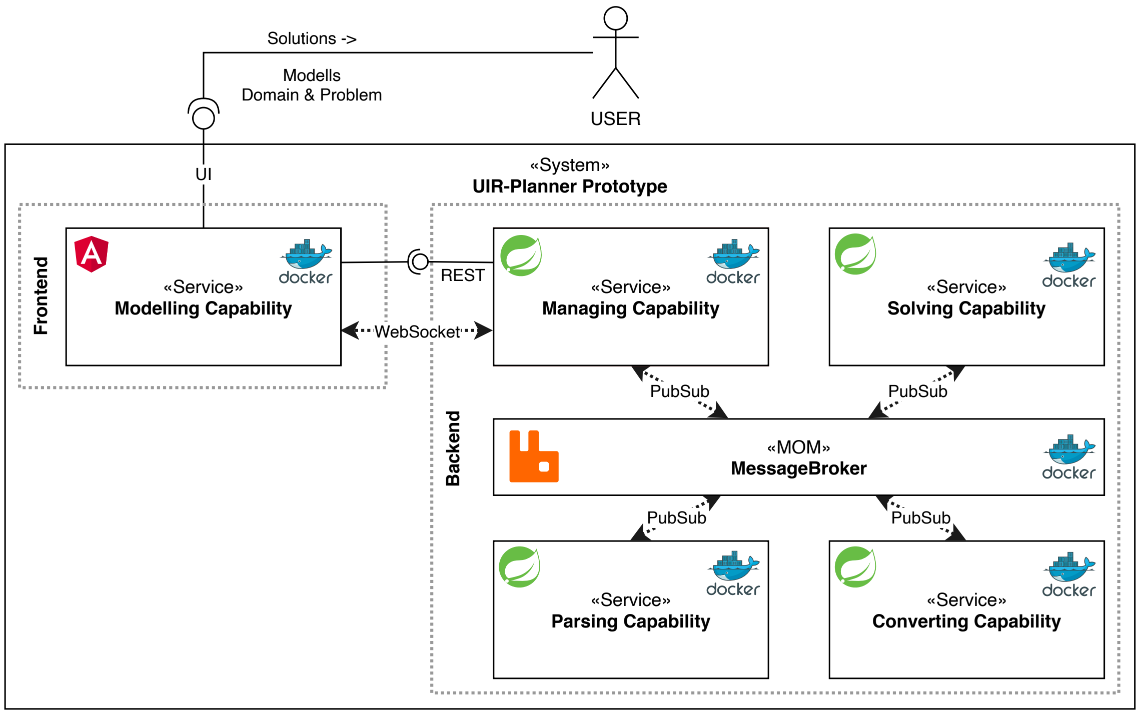


Figure 6.1: Overview of the UIR-Planning Prototype

Figure 6.1 shows an overview of the prototype. Besides the general division into front-end and back-end, the technologies used are also shown. The exact description of the technologies is in Section 6.1.

Furthermore, all capabilities are explained individually. First of all, the managing capability in Section 6.2 is discussed. Afterward, the solving capability in Section 6.3 is explained. Subsequently, several implementations of the parsing capability are listed in Section 6.4. Next, a converting capability in Section 6.5 is presented. Finally, the front-end is introduced in Section 6.6, provided in the form of a modeling capability.

6.1 Common

To implement the architecture described in Section 5.2, the prototype is implemented using *Spring Boot* and *Kotlin*.

Kotlin[1] is a Java-based programming language, which differs only in a small number of improvements [SB17]. So two of these advantages, the zero pointer security and the concise, are due to a significant reduction of the boilerplate code. Another interoperability advantage is the native support of Java libraries. Spring Boot[2] is a Java-based open source framework used to create a micro service [Red17]. In combination with RabbitMQ[3] as MOM the presented architecture can be easily implemented. Docker[4] is used to implement the containerization of the micro-services. All services can be started together using docker-images. For this purpose *docker-compose*[5] is used. The docker images must first be created using `build.sh` (see Section 6.7). This script calls the appropriate package managers. For all back-end capabilities, Gradle[6] is used. The front-end capabilities, use *NodeJS*[7]. The images are named according to the following scheme:
`uir.prototype.<class>.<capability-name>` The detailed procedure is described in Section 6.7.

Each micro-service that is to be connected to the MOM has a SpringBoot AMQP[8] configuration. Thus, a central topic is created by each capability service according to its capability class. For this purpose the `QueueConfigInterface` is implemented, which requires a main queue and a dead letter queue (see Listing 6.1).

[1]https://kotlinlang.org
[2]https://spring.io/projects/spring-boot
[3]https://www.rabbitmq.com
[4]http://docs.docker.com
[5]https://docs.docker.com/compose/
[6]https://gradle.org
[7]https://nodejs.org/en/
[8]https://spring.io/projects/spring-amqp and https://www.amqp.org

Listing 6.1 Kotlin queue configuration interface.

```kotlin
interface QueueConfigInterface {

    /**
     * Method to get the main topic request queue.
     *
     * It requires the environment property "uir.queues.request.name"!
     */
    fun mainQueue(@Value("\${uir.queues.request.name}") mainQueueName: String): Queue

    /**
     * Method to get the dead letter queue.
     *
     * It requires the environment property "uir.queues.dlq.name"!
     */
    fun deadLetterQueue(@Value("\${uir.queues.dlq.name}") deadLetterQueueName: String):
Queue
}
```

Furthermore, corresponding bindings for the routing keys are set up on the main topic (so-called Exchange Topic) (see Listing 6.2).

Listing 6.2 Kotlin binding configuration.

```kotlin
@Configuration
class BindingConfig {
  // ...

    @Bean
    fun exchange(): TopicExchange {
        return TopicExchange(topicExchangeName)
    }

    @Bean
    fun mainBinding(): Binding {
        return BindingBuilder.bind(mainQueue).to(exchange()).with(mainRoutingKey)
    }
}
```

A listener is set up for each queue. Listing 6.3 shows an example of the `MainReceiver` of the solving capability.

Listing 6.3 Kotlin RabbitMQ example listener.

```kotlin
@Component
@RabbitListener(queues = ["\${uir.queues.request.name}"])
class MainReceiver(private var solvingService: SolvingService) {
    var logger = getLoggerFor<MainReceiver>()

    @RabbitHandler
    fun handleMessage(message: SolvingBody, row: Message) {
        logger.info("Request received! \n (RequestID = ${message.requestId})")
        solvingService.solveProblem(
                content = message.content,
                requestId = message.requestId,
                callStack = message.callStack
        )
    }
}
```

The routing keys and queue names can be defined in the spring boot environment properties (see `src/main/kotlin/.../resources/*.properties`).

6.2 Managing Capability – Back-end

The Managing Capability is classified as an enabling capability (see Section 4.2.1). As already mentioned, there is no direct benefit from using this capability, but it is necessary to connect the system with the user. Therefore, a different communication interface is necessary for the connection with the front-end. Described by the architecture (cf. Section 5.2 and Figure 5.6), REST is used for the communication with the modeling capability. The corresponding controller offers an API under `http://<localhost>:8090/v1/managing/...`. To provide asynchronous responses of the system efficiently, a WebSocket is provided. This can be reached under the address `ws://<localhost>:8090/v1/websocket/`. For the connection to RabbitMQ the corresponding exchange topic `v1.router.managing-capability` is created.

Due to the decentralization of the message flow, it is necessary to set up an error queue, which transmits the corresponding status to the user if a capability step fails (also executing capability conceivable). The error queue is filled with `CustomErrorMessage` objects (see Listing 6.4) using the routing key (`error.message`).

Listing 6.4 Kotlin implementation of the custom error message.

```kotlin
@JsonInclude(JsonInclude.Include.NON_NULL)
data class CustomErrorMessage(
        val sender: String = "",
        val requestId: String = "",
        val errorMessage: String? = "",
        val stackTrace: List<StackTraceElement>? = null
)
```

The Managing capability uses Reactor[9] to process non-blocking asynchronous event streams.

Listing 6.5 shows the implementation of both streams. The first one is the planning results stream (see `getPlanResponseWebsocketMessageByRequestId()`), the second one is for the incoming error messages (see `getErrorResponseWebsocketMessageByRequestId()`). Afterward, the streams are merged, and the WebSocket sends the response to the front-end.

Listing 6.5 Kotlin implementation reactive message handling.

```kotlin
fun registerSocket(requestId: String): Flux<UirWebSocketMessage> {
    logger.info("RegisterSocket, waiting for results of {}", requestId)
    return Flux.merge(
            getPlanResponseWebsocketMessageByRequestId<Plan>(requestId),
            getErrorResponseWebsocketMessageByRequestId(requestId)
    )
}
```

To give all capabilities an overview of which capability interfaces are currently available, it is necessary to introduce a capability repository. Besides a list of capabilities, the respective input and output interface is of interest. The pure listing of all available capabilities is already realized in the prototype by using the RabbitMQ API. Therefore the prototype has a minimal solution for storing capability interfaces. Currently, only an in-memory solution is integrated, which is not used by other capabilities. The extension of the messaging controller to implement this functionality is planned for future versions of the prototype.

6.3 Solving Capability – Back-end

The Solving Capability is a core capability (see Section 4.2.1). Like the other planning capabilities, PDDL4J is used to implement the prototype [PF18]. Therefore the prototype is currently limited to solving PDDL modeled problems.

In its current version the solving instance accepts requests containing a `SolvingRequest` in the content field (see Listing 6.6).

Listing 6.6 Kotlin interface for solving requests.

```kotlin
interface SolvingRequest {
    var planner: Planner?
    var language: Language?
    var domain: String? // encoded base64 string
    var problem: String? // encoded base64 string
}
```

[9]Reactor is a reactive library based on the Reactive Streams specification for building non-blocking applications on the JVM. See https://projectreactor.io

The service subsequently decides on its authority which capabilities are to be used. In this example, the method `solveProblem()` is called, which sends a request to the parsing capability. This step is currently preconfigured, but the goal is to search a repository for the required interface and select the appropriate capability instance. The cornerstone with a *H2 in-memory db*[10] is already present in the prototype's managing capability. However, currently, only a list of interfaces is provided via REST, so the solving capability cannot access it.

Like all other capabilities, request messages are processed according to the "Fire-and-Forget" principle. This allows all services to operate stateless and non-blocking.

After receiving the reply, e.g., from a parser or in the prototype of the conversion capability, the problem's real solution begins. The user-defined planner is used for this. Since the data model may not contain a planner field for the response, the `CallStack` state is used at this point.

Depending on the solver, different results can be handled here. So it is possible to get sequential plans (cf. Listing 6.7) and parallel plans (cf. Listing 6.8). Both variants are derived from the PDDL4J `fr.uga.pddl4j.util.AbstractPlan`.

Listing 6.7 Kotlin implementation of a sequential plan.

```kotlin
data class SequentialPlan<A>(
        override var cost: Double? = null,
        override var actions: List<A>? = emptyList()
) : Plan<List<A>> where A : Action
```

Listing 6.8 Kotlin implementation of a parallel plan.

```kotlin
data class ParallelPlan<A>(
        override var cost: Double? = null,
        override var actions: List<Set<A>>? = emptyList()
) : Plan<List<Set<A>>> where A : Action
```

An exception is thrown in the event of an error, which is sent as an error message to the error handling topic of the managing capability (see Listing 6.9).

Listing 6.9 Kotlin implementation of a solving capability error.

```kotlin
class SolvingError(
        override val requestId: String = "",
        override var internalErrors: String? = ""
) : BaseException()
```

If no plan is found, the class `NoPlan` is used. For this purpose a Kotlin `when()` is implemented in the prototype. In the absence of a plan previously created, `plan` equals `null`. If the plan is sequential, Listing 6.7 is selected.

[10] `http://www.h2database.com/html/features.html` using Embedded Mode

Listing 6.10 Part of the implementation for the selection of response body using Kotlin.

```kotlin
// ...
val responseBody = when (plan) {
   null -> NoPlanBody(
      requestId = requestId,
      callStack = callStack
   )
   is SequentialPlan<*> -> SequentialPlanBody(
      content = plan,
      requestId = requestId,
      callStack = callStack
   )
   else -> TODO("Unsupported plan type!")
}
// ...
```

Figure 6.2 shows the process within the solving capability. The workflow is split into two parts. Since the managing capability has no information about the solving capability's behavior, the solving request is sent directly to the solving capability. Therefore, the first part of the request is first looked at and decides which capability will perform the next step. In this case, the parsing capability is addressed. The remaining information is written to the call stack state in the prototype to keep the service stateless. The design indicates that there will be a storing capability in the future, which stores or caches large elements. So the message size can be kept minimal by using references. Furthermore, a self-reference is created in the CallStack, which ensures the Routing Slip pattern's receipt of the message. Finally, the message is placed in the corresponding topic with the appropriate Correlation id and the CallStack.

The second part is executed when a message arrives in the reply object queue. Assuming that the object has the correct format, real solving is started. The prototype provides different algorithms (with different search strategies and heuristics) to solve planning problems [PF18]. Afterward, it is checked if the plan exists and if it is sequential. This behavior has already been explained. Finally, the result is sent back to the managing capability.

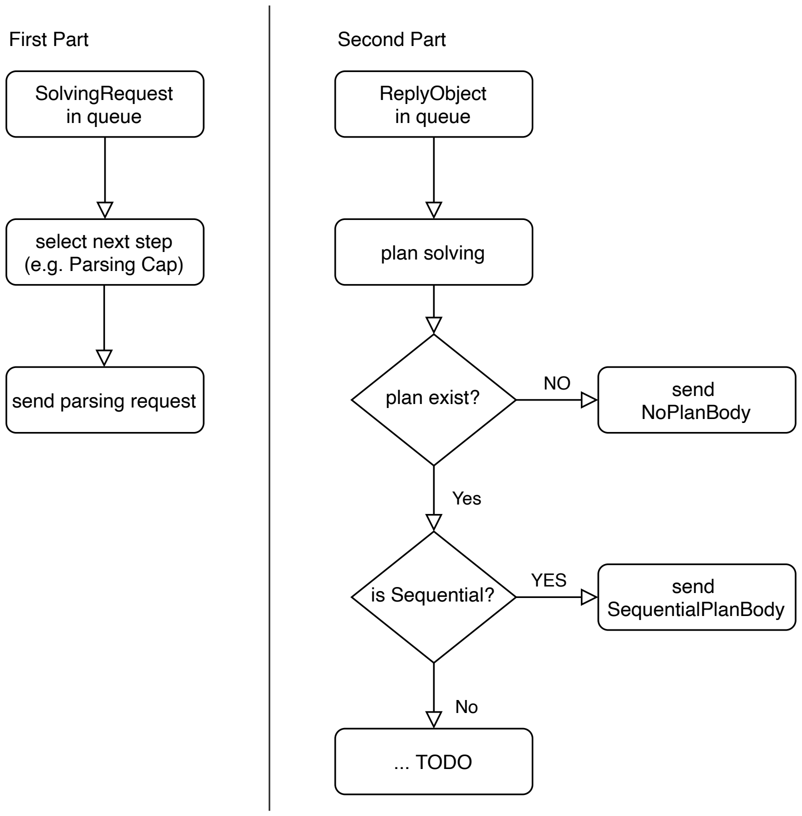


Figure 6.2: Activity diagram for the prototype solving capability.

To extend the functionality of the prototype of the UIR planning system, additional existing planners were integrated, including Panda 3 and KPlanning .

6.3.1 Panda3 Instance

Since the Panda 3 core can solve HDDL and SHOP, it is interesting for integration. Because Panda 3 is based on the JVM, the integration is possible with relatively little effort. However, Panda 3 is written in Scala and Java, making integration in a Kotlin microservice skeleton more difficult.

The first step in the integration was to integrate the parsing capability. Because of problems encountered during the application's slicing, it was not possible to integrate Panda 3 as a solving capability. More detailed information on the causes is given in Section 6.4.2 and Section 6.4.3.

Integration would have been easy without splitting the capabilities, but this would not have strengthened the system's design and questioned parts of the reusability.

6.3.2 KPlanning Instance

KPlanning is a starter kit and wrapper for the JavaFF Planner. As in the previous case, the integration as parsing capability has been started. However, serialization problems with the data structure occurred during the integration. Therefore the exact causes and problems are described in Section 6.4.4.

6.4 Parsing Capability – Back-end

A parsing capability service always implements the `ParsingRequest` interface (see Listing 6.11). This interface includes textual modeling of the domain and the problem. For technical reasons, the input is expected as base64 strings to reduce the message size.

Listing 6.11 Kotlin interface for parsing requests.

```kotlin
interface ParsingRequest {
    // base64 encoded string
    var domain: String?
    // base64 encoded string
    var problem: String?
}

interface ParsingRequestBody : BaseMessageInterface {
    override val content: ParsingRequest?
}
```

Depending on the implementation, automatic recognition of the input savings is possible. In the prototype, such a function is not provided. The input is interpreted according to the stored information in the call-stack.

Besides the possibility of multiple services implementing the parsing capability, the strategy pattern can be used to manage multiple inputs within a service. The Strategy pattern enables the developer to encapsulate a set of interchangeable algorithms individually [Lav19, p. 97]. Figure 6.3 shows how the strategy pattern could be applied to the parsing capability service.

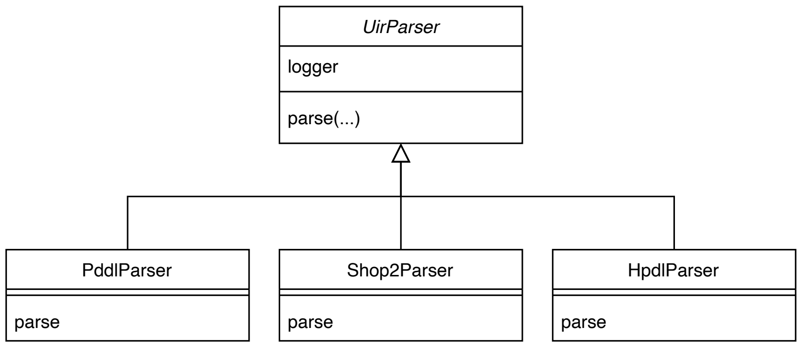


Figure 6.3: Strategy pattern applied on parsing capability.

To create capabilities for the prototype, several approaches were followed. Since it is necessary to wrap all existing planners, the basic concept is similar in all approaches. Each micro-service, like the other capability services, complies with the specifications and supports messaging. According to the interfaces, the libraries or created executables are executed internally, and the results are provided as JSON messages.

In the following six attempts were made:

- Usage of PDDL4J (see Section 6.4.1)
- Usage of Panda 3 scala core (see Section 6.4.2)
- Usage of PandaPIparser (see Section 6.4.2)
- Usage of PandaPIgrounder (see Section 6.4.2)
- Usage of JSHOP2 (see Section 6.4.3)
- Usage of KPlanning (see Section 6.4.4)

This section briefly explains each approach and outlines the encountered problems.

6.4.1 PDDL4J Parser – Java

Since mainly functionalities of the PDDL4J library are embedded in the prototype, the strategy-pattern is implemented but currently offers no further advantages because only PDDL is supported [PF18]. The PDDL4J developers are currently working to provide HDDL support for the IPC 2020. But this work isn't available on the frameworks release branch (only feature branch[11]) and the PDDL4J candidate on the IPC2020 was disqualified [BDB20]. According to Dr. Gregor Behnke, the PDDL4J Planner was disqualified for wrong solutions. This only happened once for total-ordered Problems, but multiple for partial-ordered problems (cf. Appendix B.2.2). Therefore, only the release candidate was used in the prototype and not the IPC2020 branch.

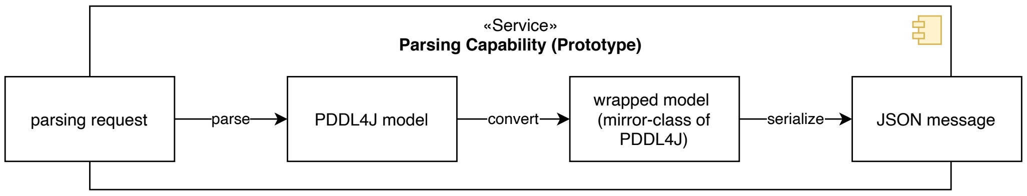


Figure 6.4: Data-flow of requests on the prototype of a parsing capability using PDDL4J.

Figure 6.4 shows the data flow within the parsing service. It can be seen that the PDDL files are parsed using PDDL4J and therefore, are provided in the frameworks, internal data models. Since these models are not made for direct data access, conversion into an intermediate data structure is necessary serializable. Therefore, data classes of the PDDL4J classes are created (see `PddlProblem` Listing 6.12 and `PddlDomain` Listing 6.13). This way, a kind of snapshot of the current state can be captured and sent to another capability.

[11] https://github.com/pellierd/pddl4j/tree/ipc20 (November 2, 2020)

Listing 6.12 Kotlin implementation of the parsed problem, based on PDDL4J models.

```kotlin
data class PddlProblem(
        override val name: String = "",
        override val domain: String = "",
        override val initialState: List<ExpData>? = null,
        override val objects: Set<TypedSymbolData>? = null,
        override val requirements: Set<RequireKeyData>? = null,
        override val goal: ExpData? = null,
        val constraints: ExpData? = null,
        val metric: ExpData? = null
) : Problem
```

Listing 6.13 Kotlin implementation of the parsed domain, based on PDDL4J models.

```kotlin
data class PddlDomain(
        override val name: String = "",
        override val requirements: Set<RequireKeyData>? = null,
        override val types: Set<TypedSymbolData>? = null,
        override val predicates: Set<NamedTypedListData>? = null,
        override val functions: Set<NamedTypedListData>? = null,
        val constants: Set<TypedSymbolData>? = null,
        val constraints: ExpData? = null,
        val methods: Set<MethodData>? = null,
        val tasks: Set<OpData>? = null,
        val derivedPredicates: Set<DerivedPredicateData>? = null
) : Domain
```

All class names ending with "`*Data`" are internal data models used for serialization (e.g. `RequireKeyData`, `TypedSymbolData`, ...).

6.4.2 Panda

To further diversify, the integration of additional parsers is planned. The Panda planner has recently started to play an essential role in the community.

Some functionalities were outsourced and made available to the participants of the IPC. Among them are parsing and converting capabilities. The Panda core also includes solver components, which makes it a framework (like PDDL4J).

Panda3Core – Scala

Panda3 is the latest version of the Panda planner and is written in Scala and Java. It is exciting for integration as a parsing capability since it supports multiple modeling languages. Among these are PDDL, HDDL and SHOP as well as serialization formats such as XML[12].

During the integration of the `Panda3Core.jar`, there were difficulties between Java, Scala, and Kotlin. These JVM problems, which appeared as `java.lang.ClassNotFoundException` and *java reflaction exceptions*, could be solved by adjusting the package manager (Gradle).

During the subsequent attempts to implement a complete roundtrip, difficulties arose again. After a new generation of the Antlr[13] parsing files, Panda3Core can be parsed within a micro-service as HDDL file. However, a complete roundtrip requires both directions, which means that the results must be equivalent even after transformation. In the case of HDDL files, this roundtrip was possible. Trying to add an export as XML in between will result in error messages (e.g. `java.lang.AssertionError` in `en.uniulm.ki.panda3.symbolic.writer.*`). Also, the following error message is displayed in the console:

> “The current version of PANDA3 does not support arbitrary formulas (in preconditions and effects).”

In the following, the Panda System developers were contacted to verify the reproduction of the bugs and help in solving them if necessary. Including Dr. Pascal Bercher, who referred to the newer and more up-to-date PandaPIparser and PandaPIgrounder systems (see [BHS+20]).

> “[...] Note that meanwhile, we do not only have newer *planners*, but also a newer parser and grounder. Some of the software if available on GitHub already, e.g. the parser and grounder:
> pandaPIparser and pandaPIgrounder [...](see Appendix B.1.1) ”

Dr. Gregor Behnke answered the question about the problems with XML generation in another email (see Appendix B.2.1).

> “[...] the XML format is highly deprecated. [...] We have replaced this format with HDDL around 2015/2016. [...] But this writer has not been maintained or used since about 2016 – actually it should be removed [translated from full email Appendix B.2.1].”

Since the XML format cannot be used, the objects of the library must be serialized again. Because providing converters is a very time consuming task, Panda could not integrate to the extent of this work.

[12]See `en.uniulm.ki.panda3.symbolic.parser.xml`
[13]Antlr 4.8: `https://github.com/antlr/antlr4`

PandaPIparser – C++

The PandaPIparser offers the possibility to parse HDDL files, and after a grounding to the PANDA base, to export them to SHOP [BHS+20].

While creating the binaries, the PandaPIparser[14] encountered a problem (`make: *** [src/hddl.cpp ] Error 1`). The problem has been reported in a GitHub issue (cf. https://github.com/panda-planner-dev/pandaPIparser/issues/2). Through close cooperation with Dr. Gregor Behnke, the problems could be solved, and the convertor is compilable.

However, it turned out that the internal models are not suitable for output as objects. This problem was addressed in an email from Dr. Gregor Behnke:

> “[...] This means that sometimes the reading in can not only create an object in the environment, but can have arbitrary side effects. Especially in C/C++ this is not that unusual, even if it is not such good software engineering. For example, the pandaPIparser does not create a single object, but fills a set of global data structures [translated from full email Appendix B.2.2].”

Since only the final output is suitable for distribution, the PandaPIparser can only be used to convert capability. Thus it is possible to create SHOP files from HDDL or vice versa.

PandaPIgrounder – C++

Like the PandaPIparser, the PandaPIgrounder[15] is part of the renewed PANDA grounding [BHS+20]. However, integration is not possible, since the system, or a submodule (`boruvka`[16]), cannot be built. Besides the building difficulties, this PandaPI application also has the problem described in Section 6.4.2.

6.4.3 JSHOP2

To support further input forms, a JSHOP parsing service is created. This service is based on a combination of JSHOP2 (see [GNK07]) and the Panda SHOP reader (`de.uniulm.ki.panda3.util.shopReader`[17]). The processing sequence is illustrated in Figure 6.5.

[14] https://github.com/panda-planner-dev/pandaPIparser
[15] https://github.com/panda-planner-dev/pandaPIgrounder
[16] https://gitlab.com/danfis/boruvka/-/commit/50571195b3c6dbe98680e5b650f90bf9c631ca32
[17] https://github.com/galvusdamor/panda3core/tree/master/src/main/java/de/uniulm/ki/panda3/util/shopReader

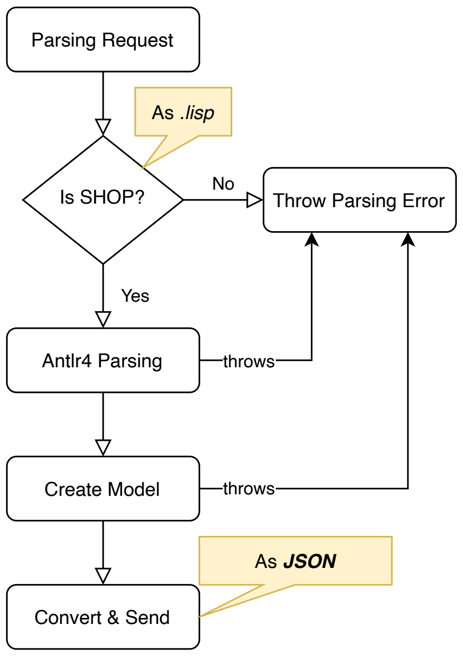


Figure 6.5: Request flowchart of the JSHOP parsing capability.

First, the inputs are converted to objects using Antlr4. Since this conversion should be done without manual building, a framework is used. However, not the internal JSHOP2 Antlr is used, but instead a grammar sheet (`shop.g4`) from Panda3. The results are then presented using internal models (see Listing 6.14). These models are also taken from Panda3Core and have only been translated into Kotlin. Finally, the data models are serialized and sent as JSON.

Listing 6.14 Kotlin model for a SHOP planning instance.

```
data class ParsedShopProblem(
        var consts: MutableSet<String> = emptySet<String>().toMutableSet(),
        var predicates: Map<String, Int> = emptyMap(),
        var tPrimitive: Map<String, Int> = emptyMap(),
        var tAbstract: Map<String, Int> = emptyMap(),
        var methods: List<ShopMethod>? = null,
        var actions: List<ShopOperator>? = null,
        var initialState: List<Array<String>>? = null,
        var initialTN: List<Array<String>>? = null
)
```

- `consts` is a list of all constant names.
- `predicates` defines the predicates and the number of parameters.
- `tPrimitive` defines the primitive tasks.
- `tAbstract` defines the abstract tasks.
- `methods` is a list of abstract methods objects (see Definition 6.4.1).

- `actions` is a list of operations (see Definition 6.4.2).
- `initialState` is the initial state.
- `initialTN` is the initial task network.

The methods are defined as shown in Definition 6.4.1.

Definition 6.4.1 (cf. Panda3 [BKB14])
A ShopMethod is a 3-Tuple $\langle decompTask, ifThen, taskVars \rangle$

- $decompTask$ *is a pair* $\langle abstract, method \rangle$ *which represents the decomposition information off the method.*
- $ifThen$ *describes the effects (positive and negative). It consists of* $[prec, tn]$ *where* $prec$ *is a precondition and* tn *the according to task network.*
- $taskVars$ *is an optional attribute.*

The Operators are defined as shown in Definition 6.4.2.

Definition 6.4.2 (cf. Panda3 [BKB14])
A ShopOperator is a 4-Tuple $\langle name, pre, add, del \rangle$

- $name$ *is an array, where the first element defines the name of the operator, and the rest defines all properties.*
- pre *defines the precondition.*
- add *defines the effect.*
- del *defines the effect.*

The reason for the absence of JSHOP is the time-consuming creation of double-sided converters. Since the Panda3 data structure is not serializable, the capability cannot send the parsing results (the parsed structure was adapted to fit panda). The prototype would currently receive no added benefit by integrating this capability, so it was deliberately withheld.

6.4.4 KPlanning

KPlanning[18] KPlanning is a wrapper for the integration of JavaFF (java implementation based on fast-forward planner, see [HN01]), PDDL4J parsers, furthermore three additional planners (JavaGP, JavaFF Planner, JavaFF Graphplan) are delivered.

[18]https://github.com/guilhermekrz/KPlanning

In this approach, it is attempted to extract the JavaFF parser and create it as a standalone service. It is planned to use a serialized object of the class `GroundProblem` as output format. The creation of the parsing capability did not cause any difficulties, but it is not possible to serialize the class via standard libraries like Gson[19] or Jackson[20]. All classes must be rebuilt, and an extra converter must be provided to map both directions (serialize, deserialize).

In this case, a particular problem is the several interfaces allow a generic implementation (polymorphic objects). To be able to map them with Jackson, annotations on the models are necessary. Listing 6.15 shows in lines 5 to 8 how such subtypes can be defined. This also includes the annotation of the class as property (see line 2 on Listing 6.15). To reduce the JSON size null-values can be removed from the JSON (see line 1 on Listing 6.15).

Listing 6.15 Jackson JSON model annotations.

```
1 @JsonInclude(JsonInclude.Include.NON_NULL)
2 @JsonTypeInfo(use = JsonTypeInfo.Id.CLASS, include = JsonTypeInfo.As.PROPERTY)
3 data class ExampleClass { ... }
4
5 @JsonSubTypes(value = [
6     JsonSubTypes.Type(value = SubClassA::class),
7     JsonSubTypes.Type(value = SubClassB::class)
8 ])
9 abstract class SuperClass { ... }
```

However, since only the self-created models can be annotated, it is necessary to clone the library's entire model structure, in this case of JavaFF. Thus, an integration of the service into the overall system is not possible without disproportionate effort.

6.5 Converting Capability – Back-end

The Converting Capability is classified as an enabling capability (see Section 4.2.1). This capability is included in this prototype to demonstrate dynamic routing and independent insertion of intermediate steps. A converting capability is not mandatory to get a minimal planning system.

In this prototype, the converting capability service is used to encode the problem and the domain. By using the PDDL4J framework, this step is necessary [PF18]. So the planning problem is encoded and simplified in this step. The encoded variant is a compact representation of the planning problem.

This encoding step is typical for most PDDL planners [PF18] to transform the planning problem into its final form. By embedding PDDL4J, this capability is able to encode both the *logical representation* and the Finite Domain Representation (Fdr) [PF18].

[19]`com.google.gson.Gson` from `https://github.com/google/gson`
[20]`com.fasterxml.jackson.module` from `https://github.com/FasterXML/jackson`

Listing 6.16 Kotlin interface for encoding requests.

```kotlin
interface EncodingRequest<P, D> : ConvertingRequest
        where P : Problem, D : Domain {
    var problem: P?
    var domain: D?
}

data class PddlEncodingRequest(
        override var problem: PddlProblem? = null,
        override var domain: PddlDomain? = null
) : EncodingRequest<PddlProblem, PddlDomain>
```

Listing 6.16 shows the interface and the corresponding class provided in this prototype. The return type is given by the class `PddlEncodedProblem`, which is shown in Listing 6.17.

Listing 6.17 Kotlin implementation of a PDDL encoded problem.

```kotlin
data class PddlEncodedProblem(
        var types: List<String>? = null,
        var inferredDomains: List<Set<Int>>? = null,
        var domains: List<Set<Int>>? = null,
        var constants: List<String>? = null,
        var predicates: List<String>? = null,
        var predicatesSignatures: List<List<Int>>? = null,
        var inertia: List<InertiaData>? = null,
        var relevantFacts: List<IntExpData>? = null,
        var goal: BitExpData? = null,
        var init: BitExpData? = null,
        var operators: List<BitOpData>? = null,
        var functions: List<String>? = null,
        var functionsSignatures: List<List<Int>>? = null
) : EncodedProblem
```

The objects' structure is taken from the PDDL4J framework to keep the conversion effort as low as possible. To avoid serialization problems, the whole data structure is wrapped by custom classes.

- `types` is a list of all types defined on the domain. This list's order is critical because other fields use `Int` values to reference these objects by index.
- `constants` contains all objects defined for the given domain. The order of elements also matters for this list.
- `domains` is the assignment of elements in `constants` to a element of `types`. In the case of a logistics domain, this could be, for instance, the assignment of the type package to several objects `{"package" to ["obj1", "obj2",...]}`.

- `predicates` are defined on two different fields. The `predicates` list only contains the string representation. The `predicatesSignatures` add the types of the parameters, by referencing the index of the `types`. For instance in logistics this could be `["at", "in", ...]`(defined in `predicates`) and the type informations of `predicatesSignatures` result in `"PHYOBJ" at "PLACE"` or `"PACKAGE" in "VEHICLE"`.
- `relevantFacts` uses the invariants discovery, which is expressed by mutually exclusive sets of sentences. PDDL4J makes use of mutex propositions, which allow a minimal number of possibilities [PF18].
- `goal` is defined as the target state. It is given by a list of indexed world-state facts from the `relevantFacts` list.
- `init` is defined as the initial state. It is also given by a list of world-state facts from the `relevantFacts` list.
- `operators` is a list of all actions for each precondition. It can be defined as a tuple:
  $o = (name, preconditions, effects, instantiations, costs, duration)$
  While the name represents an action, which can be applied if the preconditions are true. The action is to process the effects. The instantiations define the objects, which are passed to the action as an input parameter. The costs and duration are also given to calculate the costs of a plan.
- `functions` is a list of functions contained on the problem.

The `PddlEncodedProblem` provides a representation as a causal domain transition graph. This is an efficient way to determine which propositions are accessible from the current state [PF18, p. 22].

6.6 Modelling Capability – Front-end

The modeling capability is classified as an endpoint core capability (see Section 4.2.1) and, according to the analysis, always a front-end service. This section describes the implementation of this capability in the prototype. First of all, an overview of the technologies used in the prototype is provided in Section 6.6.1. Subsequently, all features contained in the prototype are listed in Section 6.6.2.

6.6.1 Technologies

In this prototype, an Angular web application is created to implement front-end capability. It serves as an exemplary input mask for the domain and the problem. As a template, the angular material library[21] is used to avoid the manual design of the input fields. The whole micro-service is based on a node.JS application running on an NGINX server. To make the service as extensible as possible, a *nrwl MonoRepo* is used. This offers the possibility to use other front-end frameworks like React for additional components. Furthermore, it offers the possibility to create libraries quickly and provides a preconfigured Test-Bed for unit and end2end tests. Thereby the Jest Framework is used for the

[21] https://material.angular.io

unit tests. Cypress is used to implement the e2e tests. Both cypress and jest offer the advantage of parallelization of the testing, which is not supported by angular standards like jasmine and protractor.

For the connection to the REST endpoints, the angular HttpClient is used. A Web socket handles the asynchronous responses. All large application parts are divided into individual modules to minimize loading times, provided via lazy-loading. For example the `PlanRoutingModule` (see Figure 6.6), it consumes two services `WebsocketService` and `SolvingService`, which are provided by the `SolvingModule` . Besides, it includes a `VisualizeErrorModule` and a `VisualizePlanModule`, which encapsulates the business logic.

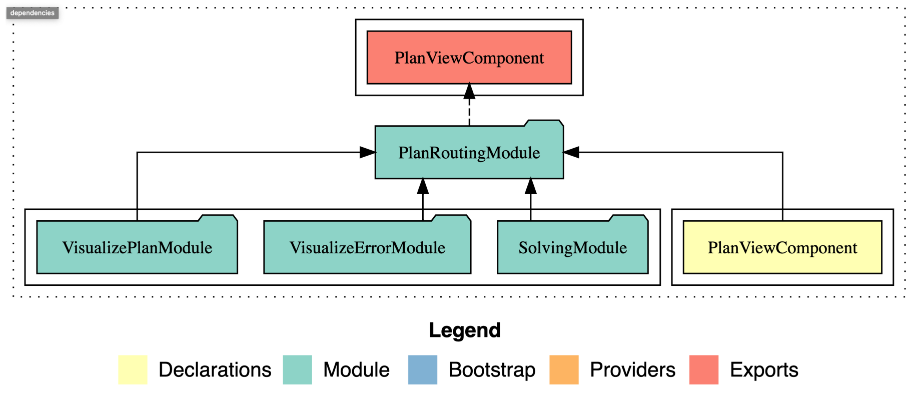


Figure 6.6: Plan module overview of the prototype modelling capability (generated using CompoDoc).

6.6.2 Scope of use

This section contains a summary of the functions provided in the prototype. Figure 6.7 shows a routing diagram of the prototype. The bottom route is a fallback regex to redirect all unhandled requests to a fallback page. The default route follows this `.../`, which points to `/home`.

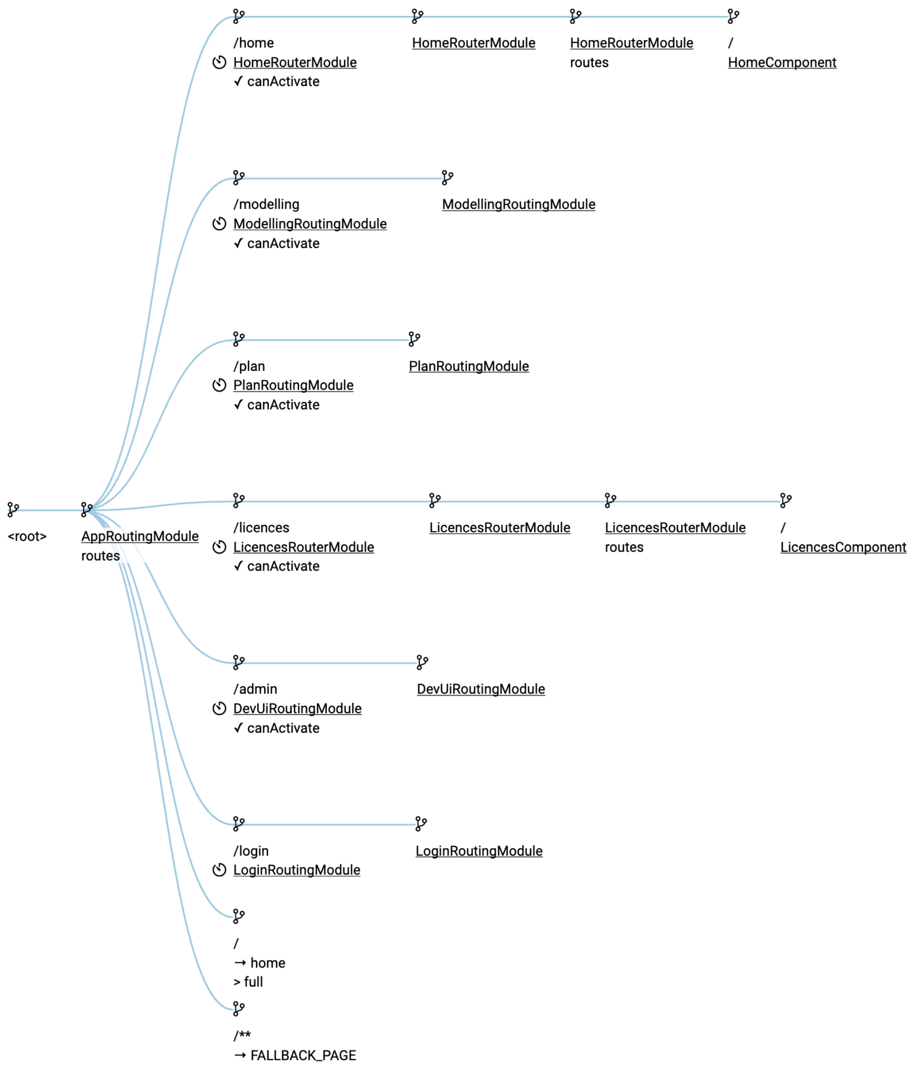


Figure 6.7: Router overview of the prototype modelling capability (using CompoDoc).

Login

Since the system can theoretically be started in a cloud, a login page is provided. The guards are disabled in the prototype, which means that the login page can be unlocked using any username and password. The validation is currently mocked. It only checks for the local storage property `'username'`. It is essential to activate the guards in a release candidate to prevent unauthorized access to the interfaces. Figure 6.8 shows a screenshot of the login page of the prototype.

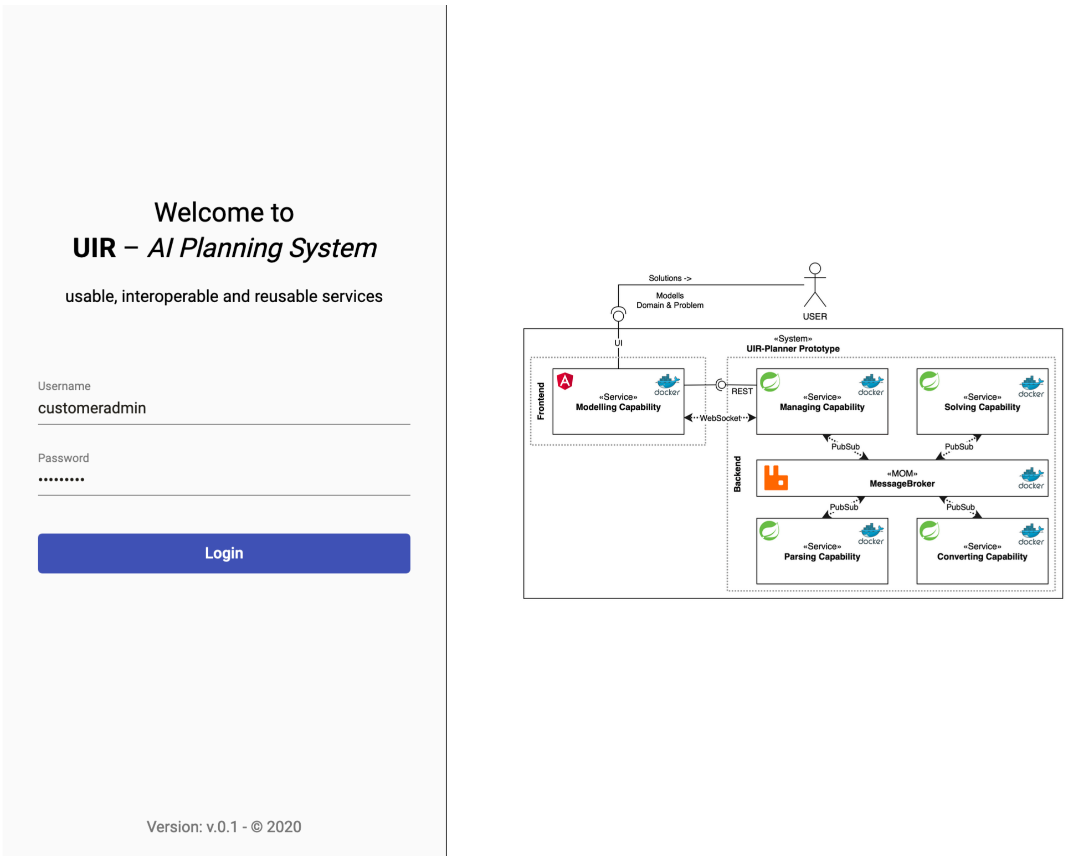


Figure 6.8: Screenshot of the login-page.

Administration

An administration tool is located under the route `/admin`, which provides an overview of the system and the status of capabilities. Figure 6.9 shows all micro-services and routes of MOM. However, the graph in the current prototype is still a mockup and is not updated dynamically.

The system overview is a scalable representation that can be dynamically extended. To generate it, the unique name of the services can be used because it is identical to the graph's structure. For example `v1.endpoint.solving-capability#prototype.pddl4j`. This service would create the nodes "v1", "endpoint", "solving", "prototype.pddl4j" of the graph. Thus, on Figure 6.9 it can be seen that the first cluster corresponds to the capability type. The second cluster represents the classification of the capability. On the last cluster (on the very right side), the implementations are shown. In the example shown, three parsing capabilities are implemented, which have the unique tags "#PDDL4J", "#JSHOP2" and "#Panda3". However, nodes that have no leaves are excluded (see node "monitoring"). Such a node will not be visible if the mock is turned off.

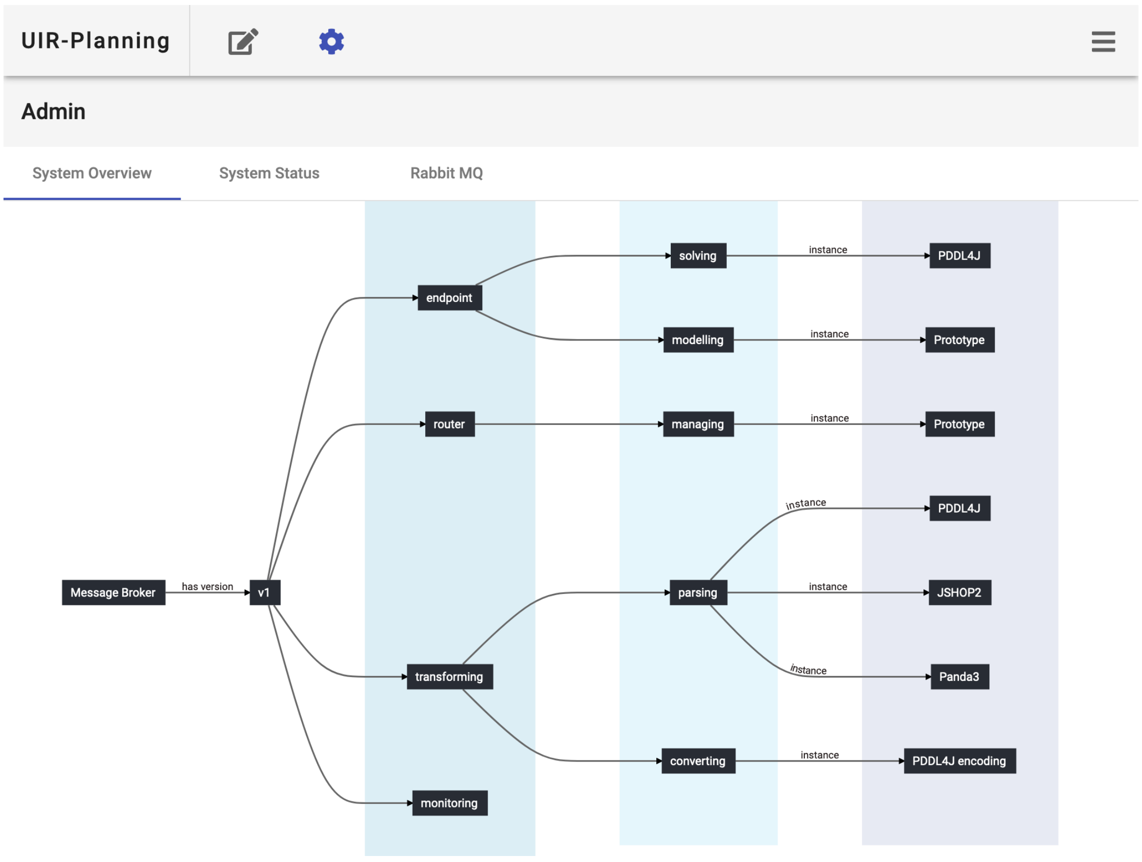
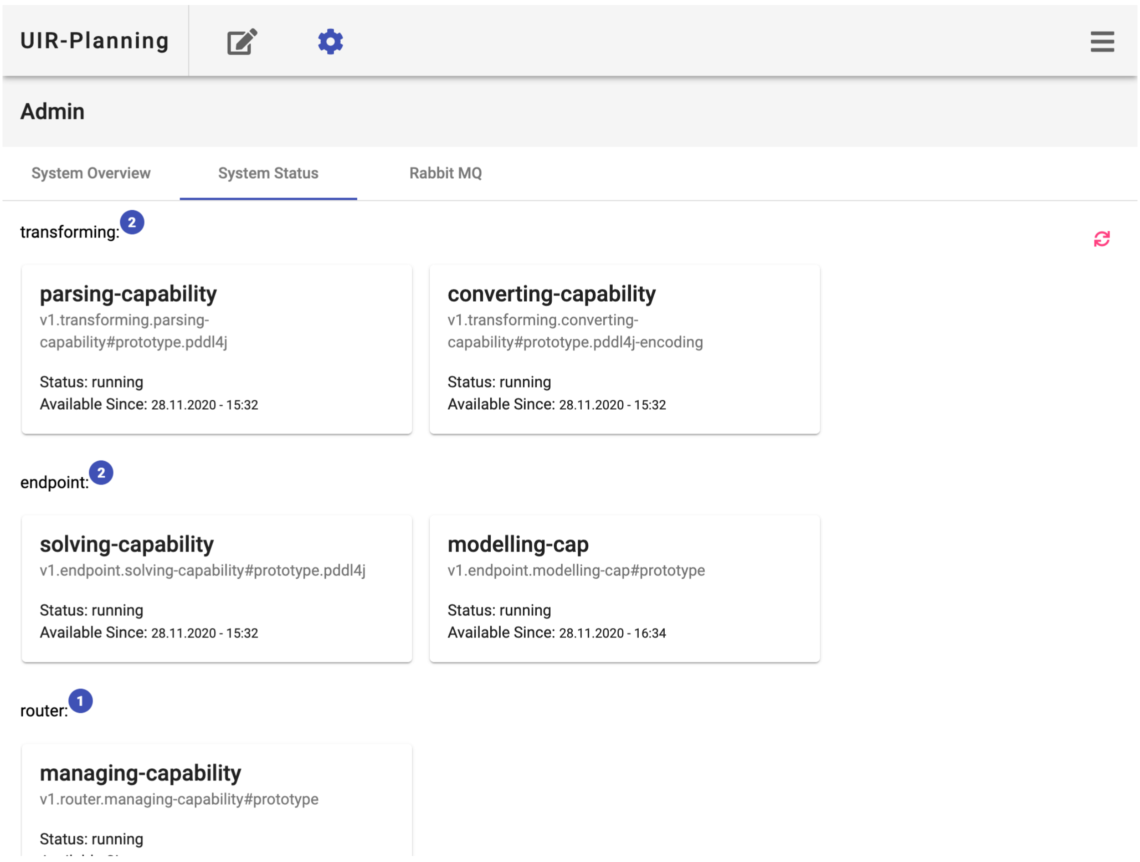


Figure 6.9: Screenshot of the admin view shows system graph.

The second tab ("System Status") displays the live system status based on the information of the MOM (see Figure 6.10). In contrast to Figure 6.9, Figure 6.10 is already connected to the real interfaces in the prototype.

The system status is divided into four categories of capability services. Each of these capability categories contains cards. Each of them represents a planning capability instance. The title indicates the capability class, and the identifier is shown in the line below. Also, information about the status (e.g., "running") and the last connection is provided.



transforming: 2

parsing-capability
v1.transforming.parsing-capability#prototype.pddl4j

Status: running
Available Since: 28.11.2020 - 15:32

converting-capability
v1.transforming.converting-capability#prototype.pddl4j-encoding

Status: running
Available Since: 28.11.2020 - 15:32

endpoint: 2

solving-capability
v1.endpoint.solving-capability#prototype.pddl4j

Status: running
Available Since: 28.11.2020 - 15:32

modelling-cap
v1.endpoint.modelling-cap#prototype

Status: running
Available Since: 28.11.2020 - 16:34

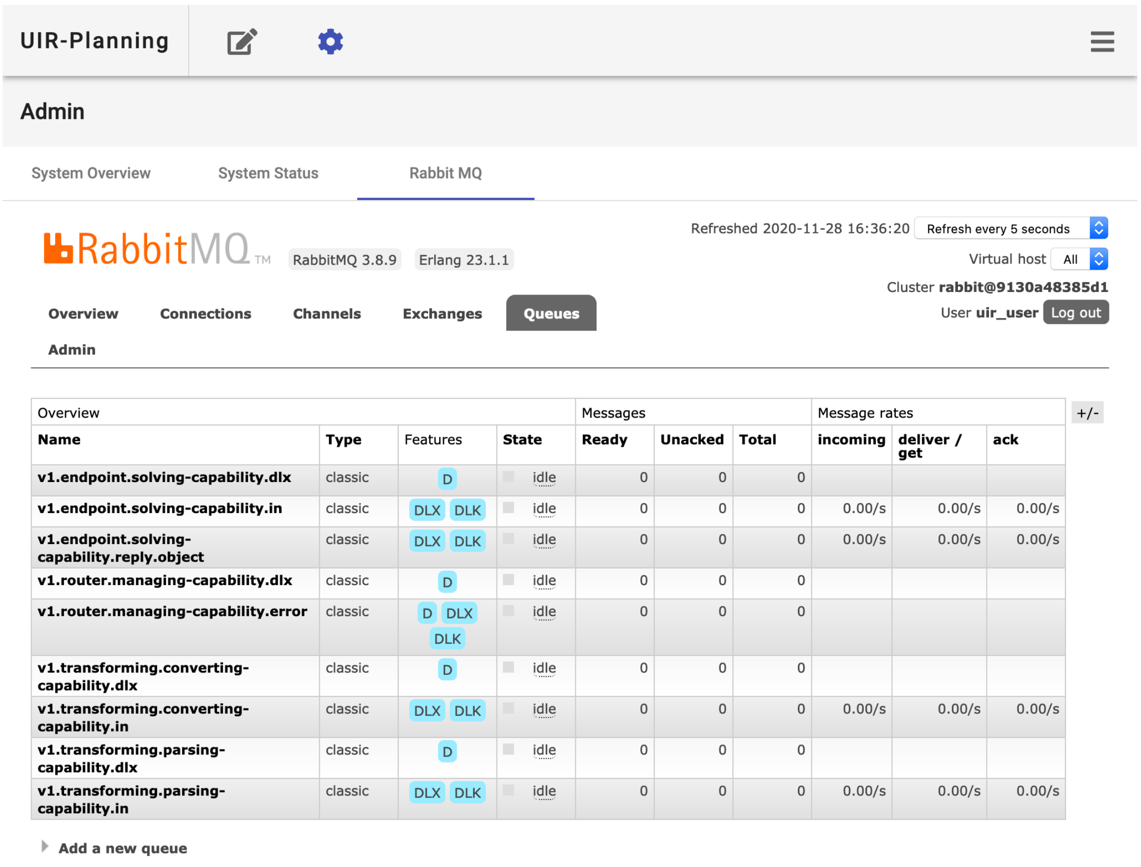
router: 1

managing-capability
v1.router.managing-capability#prototype

Status: running

Figure 6.10: Screenshot of the admin view shows system status.

The third tab (“Rabbit MQ”) contains an embedded administration interface of the MOM. This is where all statistics, clients, queues, …can be accessed and configured.



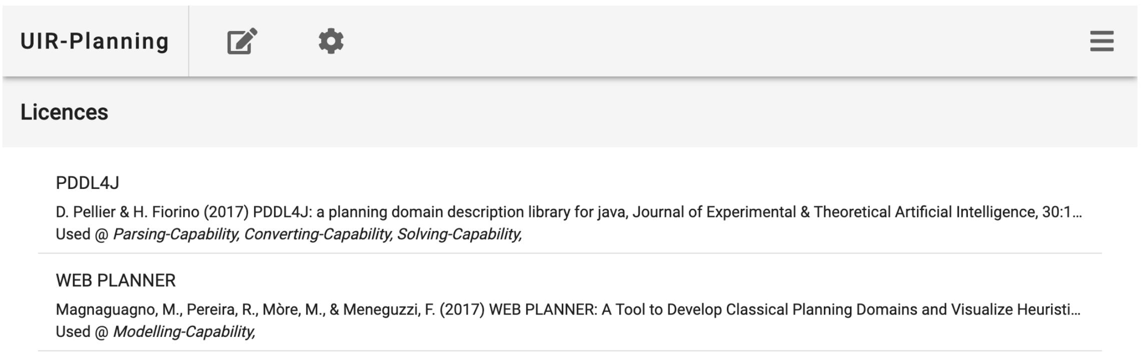
Refreshed 2020-11-28 16:36:20 Refresh every 5 seconds
Virtual host All
Cluster **rabbit@9130a48385d1**
User **uir_user**

Overview				Messages			Message rates		
Name	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**
v1.endpoint.solving-capability.dlx	classic	D	idle	0	0	0			
v1.endpoint.solving-capability.in	classic	DLX DLK	idle	0	0	0	0.00/s	0.00/s	0.00/s
v1.endpoint.solving-capability.reply.object	classic	DLX DLK	idle	0	0	0	0.00/s	0.00/s	0.00/s
v1.router.managing-capability.dlx	classic	D	idle	0	0	0			
v1.router.managing-capability.error	classic	D DLX DLK	idle	0	0	0			
v1.transforming.converting-capability.dlx	classic	D	idle	0	0	0			
v1.transforming.converting-capability.in	classic	DLX DLK	idle	0	0	0	0.00/s	0.00/s	0.00/s
v1.transforming.parsing-capability.dlx	classic	D	idle	0	0	0			
v1.transforming.parsing-capability.in	classic	DLX DLK	idle	0	0	0	0.00/s	0.00/s	0.00/s

Figure 6.11: Screenshot of the embedded admin view of the MOM.

License

The route `/licenses` is provided to offer a list of the used licensed frameworks. Since the current prototype only uses PDDL4J in the back-end, this entry is preset. Figure 6.12 shows a screenshot of the prototype. This information is still hardcoded in the current version, but outsourcing to a managing capability is desirable.



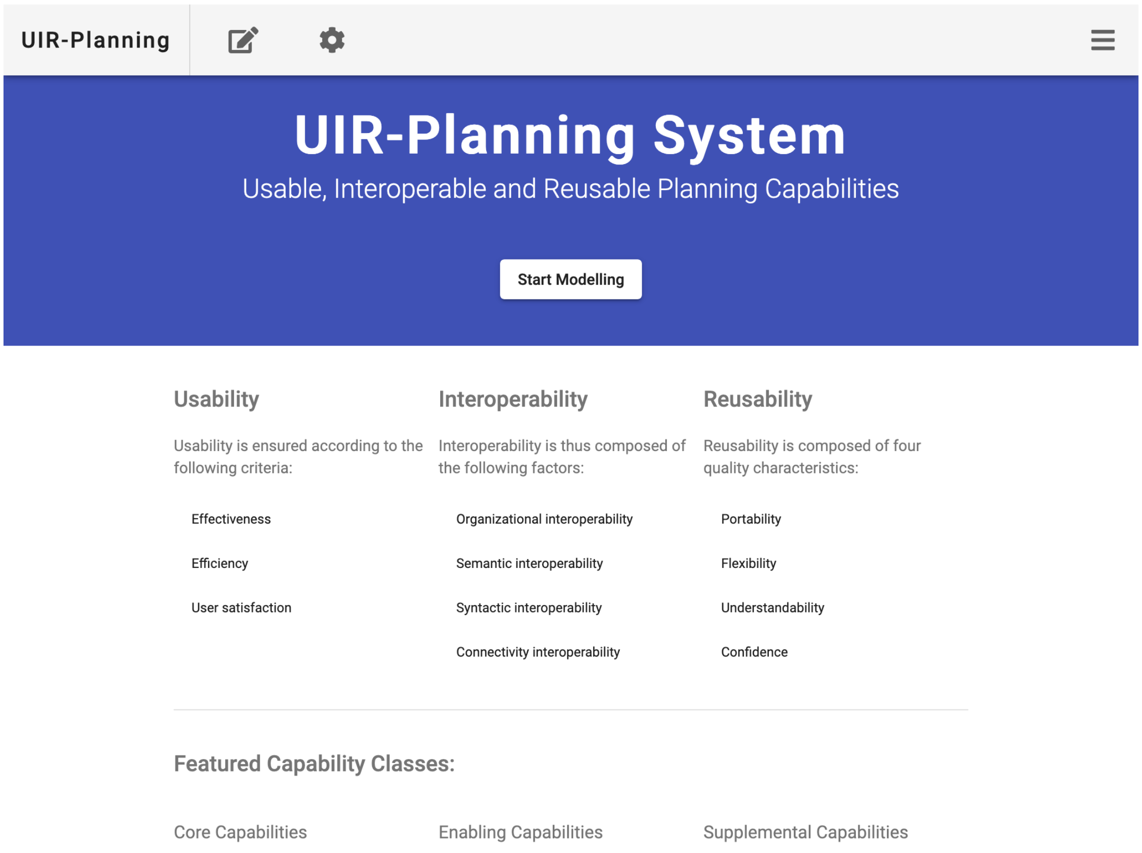
PDDL4J
D. Pellier & H. Fiorino (2017) PDDL4J: a planning domain description library for java, Journal of Experimental & Theoretical Artificial Intelligence, 30:1...
Used @ *Parsing-Capability, Converting-Capability, Solving-Capability,*

WEB PLANNER
Magnaguagno, M., Pereira, R., Mòre, M., & Meneguzzi, F. (2017) WEB PLANNER: A Tool to Develop Classical Planning Domains and Visualize Heuristi...
Used @ *Modelling-Capability,*

Figure 6.12: Screenshot of the license overview.

Home

Beneath the route `/home` the home-screen is placed. Figure 6.13 shows a screenshot of the home page. It lists some facts about the planning system and delivers an entry point for further modeling steps.



UIR-Planning System

Usable, Interoperable and Reusable Planning Capabilities

Usability

Usability is ensured according to the following criteria:

- Effectiveness
- Efficiency
- User satisfaction

Interoperability

Interoperability is thus composed of the following factors:

- Organizational interoperability
- Semantic interoperability
- Syntactic interoperability
- Connectivity interoperability

Reusability

Reusability is composed of four quality characteristics:

- Portability
- Flexibility
- Understandability
- Confidence

Featured Capability Classes:

Core Capabilities

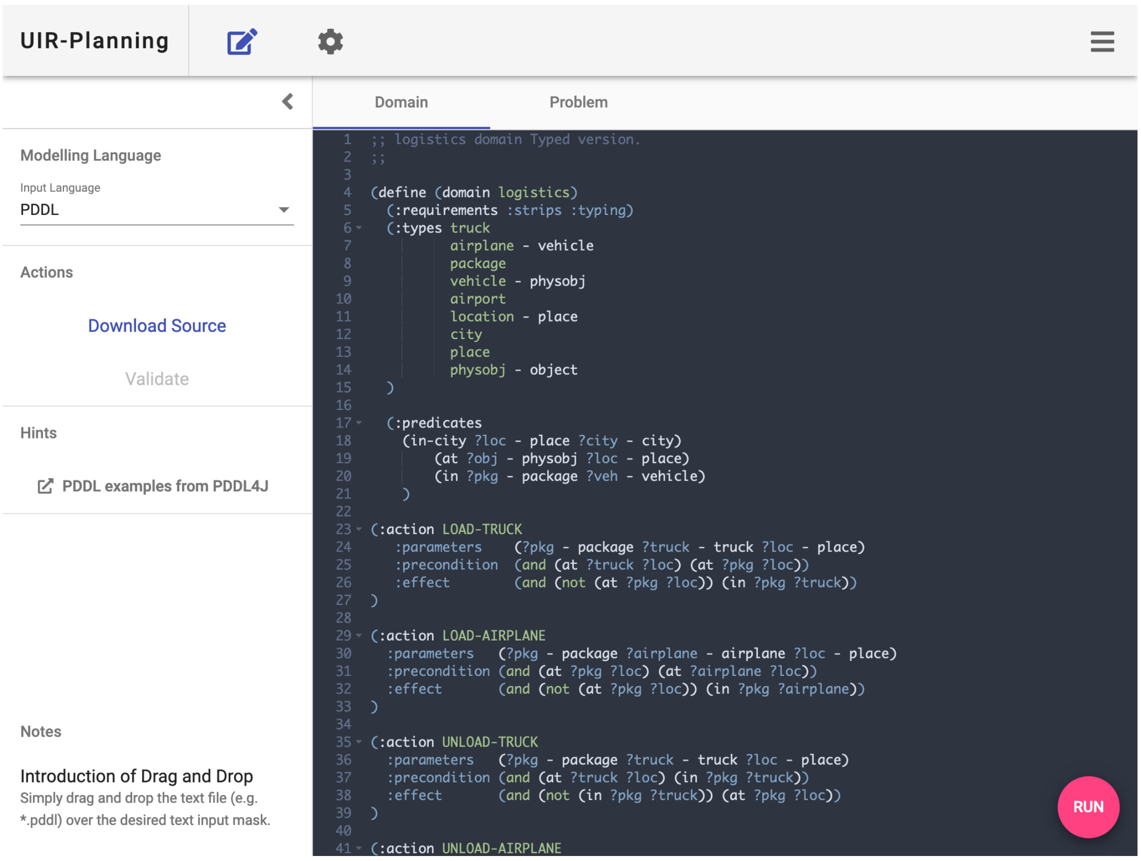
Enabling Capabilities

Supplemental Capabilities

Figure 6.13: Screenshot of the home landing-page.

Modelling

`/modelling` shows all three options to model the domain and the problem. Currently, only manual modeling is possible. `/modelling/editor` offers a Web IDE implemented by using ACE. The syntax highlighting and folding is taken from Magnaguagno et al. WEB-PLANNER [MPMM17]. Figure 6.14 shows a screenshot of the manual modelling area.



Modelling Language

Input Language
PDDL

Notes

Introduction of Drag and Drop
Simply drag and drop the text file (e.g. *.pddl) over the desired text input mask.

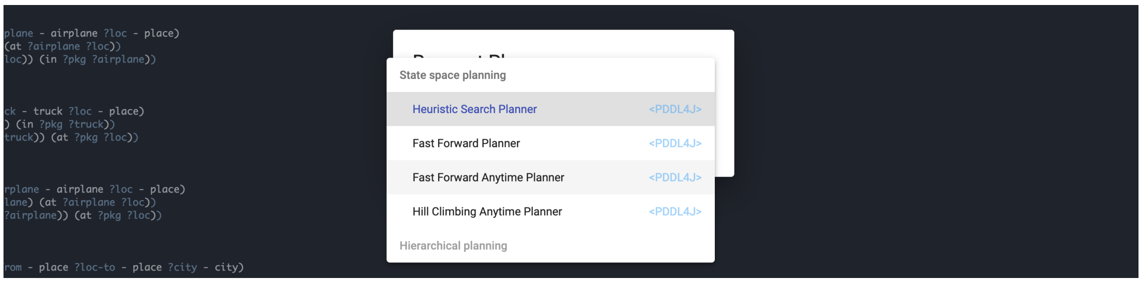
```
;; logistics domain Typed version.
;;

(define (domain logistics)
  (:requirements :strips :typing)
  (:types truck
          airplane - vehicle
          package
          vehicle - physobj
          airport
          location - place
          city
          place
          physobj - object
  )

  (:predicates
    (in-city ?loc - place ?city - city)
        (at ?obj - physobj ?loc - place)
        (in ?pkg - package ?veh - vehicle)
    )

(:action LOAD-TRUCK
   :parameters    (?pkg - package ?truck - truck ?loc - place)
   :precondition  (and (at ?truck ?loc) (at ?pkg ?loc))
   :effect        (and (not (at ?pkg ?loc)) (in ?pkg ?truck))
)

(:action LOAD-AIRPLANE
  :parameters   (?pkg - package ?airplane - airplane ?loc - place)
  :precondition (and (at ?pkg ?loc) (at ?airplane ?loc))
  :effect       (and (not (at ?pkg ?loc)) (in ?pkg ?airplane))
)

(:action UNLOAD-TRUCK
  :parameters   (?pkg - package ?truck - truck ?loc - place)
  :precondition (and (at ?truck ?loc) (in ?pkg ?truck))
  :effect       (and (not (in ?pkg ?truck)) (at ?pkg ?loc))
)

(:action UNLOAD-AIRPLANE
```

Figure 6.14: Screenshot of the modelling web IDE.

The IDE has a page navigation, which can be used to define the input language and to perform additional actions. In this prototype, only a simple download functionality is offered, but at this point, further options are feasible, as indicated by the disabled feature “validate”. In the section below, a link to domain examples is provided. Currently, this is only a GitHub link, but it could also provide access to the storing capability. For example, IPC domains and problems could be provided here.

The main input window provides drag and drop functionality. This allows easy editing of text files in the IDE.

After entering the domain and problem, a planning request can be submitted by clicking the “RUN” button on the right bottom corner (see Figure 6.14). Afterward, a dialog opens, which provides the selection of the solver. This information is in the prototype not dynamically available yet. Nevertheless, four different solvers can be selected. Figure 6.15 shows the dialog and the selection options for the solver.



```
plane - airplane ?loc - place)
(at ?airplane ?loc))
loc)) (in ?pkg ?airplane))


ck - truck ?loc - place)
) (in ?pkg ?truck))
truck)) (at ?pkg ?loc))


rplane - airplane ?loc - place)
lane) (at ?airplane ?loc))
?airplane)) (at ?pkg ?loc))


rom - place ?loc-to - place ?city - city)
```

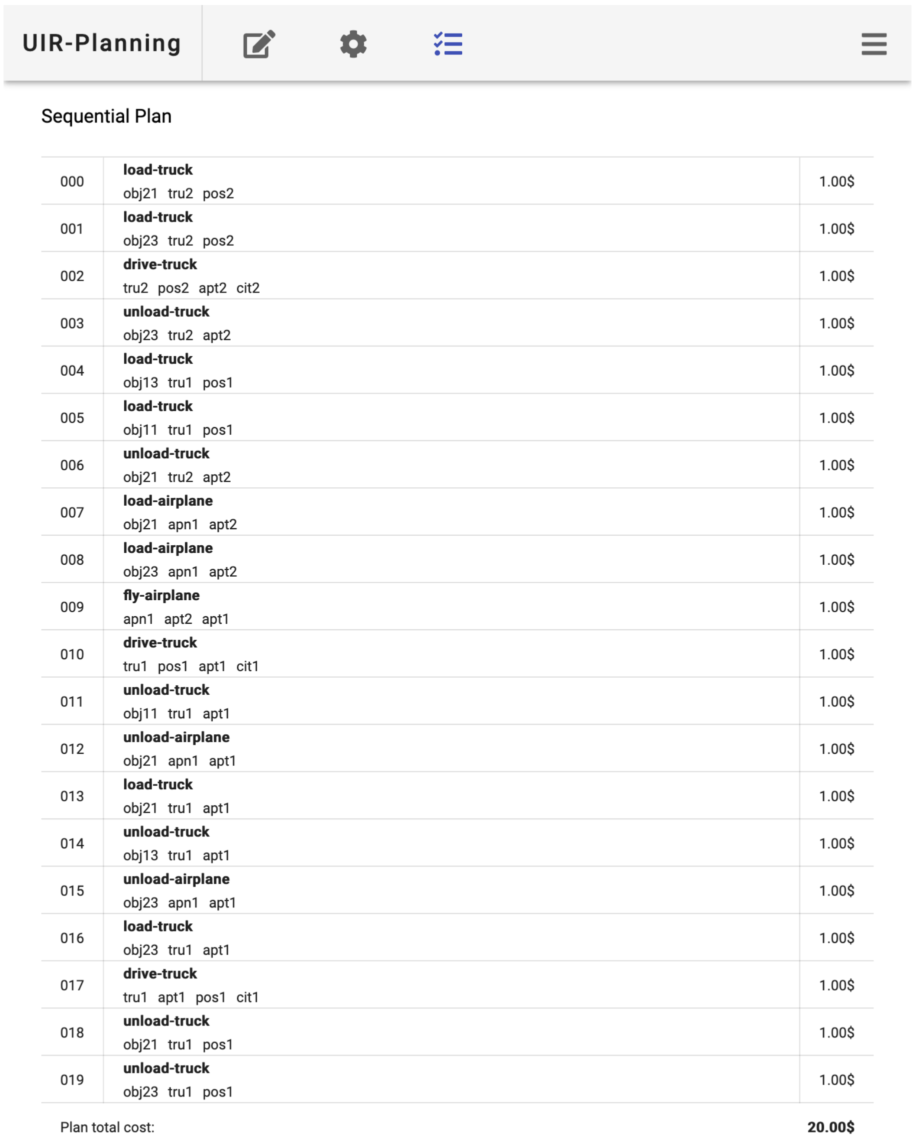
State space planning
Heuristic Search Planner <PDDL4J>
Fast Forward Planner <PDDL4J>
Fast Forward Anytime Planner <PDDL4J>
Hill Climbing Anytime Planner <PDDL4J>
Hierarchical planning

Figure 6.15: Screenshot of a solver selector.

With the confirmation of choice, a planning request is sent to the managing capability.

Plan

In this section, an example result is presented. Figure 6.16 shows a screenshot of the result area. It contains a list of all actions and the corresponding input parameters. This example is a scenario in the logistics domain. It should be noted that only sequential locations can be displayed in the current version. The modular structure of the view allows a quick exchange and extension of other views. Figure 6.16 shows the sequential plan of the problem previously modeled in Figure 6.14.

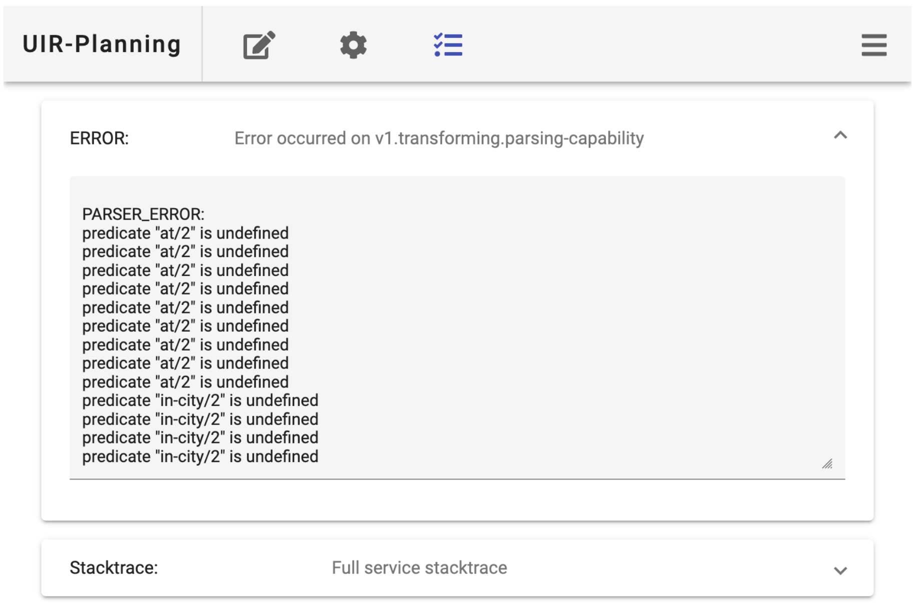


Sequential Plan

000	**load-truck** obj21 tru2 pos2	1.00$
001	**load-truck** obj23 tru2 pos2	1.00$
002	**drive-truck** tru2 pos2 apt2 cit2	1.00$
003	**unload-truck** obj23 tru2 apt2	1.00$
004	**load-truck** obj13 tru1 pos1	1.00$
005	**load-truck** obj11 tru1 pos1	1.00$
006	**unload-truck** obj21 tru2 apt2	1.00$
007	**load-airplane** obj21 apn1 apt2	1.00$
008	**load-airplane** obj23 apn1 apt2	1.00$
009	**fly-airplane** apn1 apt2 apt1	1.00$
010	**drive-truck** tru1 pos1 apt1 cit1	1.00$
011	**unload-truck** obj11 tru1 apt1	1.00$
012	**unload-airplane** obj21 apn1 apt1	1.00$
013	**load-truck** obj21 tru1 apt1	1.00$
014	**unload-truck** obj13 tru1 apt1	1.00$
015	**unload-airplane** obj23 apn1 apt1	1.00$
016	**load-truck** obj23 tru1 apt1	1.00$
017	**drive-truck** tru1 apt1 pos1 cit1	1.00$
018	**unload-truck** obj21 tru1 pos1	1.00$
019	**unload-truck** obj23 tru1 pos1	1.00$
Plan total cost:		**20.00$**

Figure 6.16: Screenshot of a sequential plan.

In case of an error, detailed information about the cause is provided. Please note that the level of detail depends on the capabilities used. Figure 6.17 shows an example. In this case, a part of an action definition was removed to invalidate the PDDL. The error occurred in the parsing capability, and the corresponding stack trace is optionally available. A more user-friendly alternative should be used in future versions because only a few people can work with such detailed information. Recently, notifications are already shown, but no constructive alternative is offered yet. Thus retry actions in case of insufficient availability or timeout are an option.



ERROR: Error occurred on v1.transforming.parsing-capability

```
PARSER_ERROR:
predicate "at/2" is undefined
predicate "at/2" is undefined
predicate "at/2" is undefined
predicate "at/2" is undefined
predicate "at/2" is undefined
predicate "at/2" is undefined
predicate "at/2" is undefined
predicate "at/2" is undefined
predicate "at/2" is undefined
predicate "in-city/2" is undefined
predicate "in-city/2" is undefined
predicate "in-city/2" is undefined
predicate "in-city/2" is undefined
```

Stacktrace: Full service stacktrace

Figure 6.17: Screenshot of an error on planning.

6.7 Building and Startup

This section describes the procedure to deploy the entire system.

First of all, Section 6.7.1 briefly describes the structure and the repositories. Afterwards the building scripts are explained in Section 6.7.2. Afterwards the startup is described in Section 6.7.3.

6.7.1 Repositories

Each capability has its repository. This measure improves reusability and interoperability enormously. It is also easy to add individual capabilities as third party services.

In order to use the complete prototype, all capability repositories must be stored in the same directory. One of the repositories contains the “start all script” and the `docker-compose.yml` (see `./compose`). The repositories listed below must all be present to start the system.

- `compose` – Required for system startup, not for operation.
- `solving-cap` – Implementation of an AI planner.
- `managing-cap` – Required for correct routing.
- `parsing-cap` – Implementation of an encoding capability.
- `modelling-cap` – Implementation of a web user interface.
- `converting-cap` – Implementation of an encoding capability.

6.7.2 Building

During the building process, a distinction between front-end and back-end must be made. In either case, a docker image is created as a result. The prototype assumes that docker uses the locale image, but this can be easily extended by a docker registry for versioning and remote deployment in a cloud environment.

Back-end

Since all back-end services follow the same structure, a uniform building strategy is in place. If additional capabilities in other languages or using technologies are added, an individual `build.sh` must be created.

The `build.sh` is responsible for executing the `Dockerfile` and generating the docker image. Listing A.3 shows an example `Dockerfile` for the solving capability service.

In the first line, a Base-Image is selected. It is necessary to consider the corresponding java version. All back-end services of this prototype are implemented using java 14. Some capabilities may require other versions. For example, the JSHOP parsing capability or capabilities using the Panda3Core libraries require java 8. Also, the use of a java base image is not recommended if the capability is written in another language like Python or C++. On lines 2 and 3, environment variables are initialized (TZ = timezone). A new user is created to prevent root command injections (cf. line 5 – 9), which subsequently takes over the spring-boot application's execution. Afterward, the corresponding jar is placed in the docker container (cf. line 12), and the entry point is defined with the corresponding profile (cf. line 14).

However, in order to include an executable jar, it must be created first. For simplification of the process a `build.sh` is provided in each capability. The building script is shown in Listing A.1. Since the build process takes a long time, especially with active tests, these are disabled in the prototype. To activate them, only `-x test` must be removed (cf. line 3). Finally, the `Dockerfile` described in the previous section is used for the build (cf. line 6). The result of a successful build is the finished docker-image, which is named according to the following scheme:
`uir.prototype.<capability-type>.<capability-class>(.<addintional-name>)`

Front-end

The front-end `Dockerfile` consists of two parts (see Listing A.4). In the first part, the project is built (see lines 1 – 16). To access it in the second part, the stage is tagged with `builder` (see line 1). After building the application, the `app/dist/` folder is created. The second step is to use an NGINX image as a base image configured in line 29. Before this and in the back-end, the time zone is defined, and a new user is created. Line 28 shows how the artifacts are moved from the builder to the NGINX container. Finally, the corresponding port is exposed. In this prototype, this is the angular default port `4200 TCP`.

The build file (`build.sh`) contains in the front-end only the build and tag command (see Listing A.2).

To optimize the developers' understanding of the application, automatic code documentation can be created. The documentation is generated using the `compodoc` framework. Since the front-end application is based on NodeJS, the following command can be used:

```
npm run generate-docs
```

It will act as a wrapper for the angular CLI (`ng run ...`).

6.7.3 Startup

In order to start all docker containers, docker-compose is used. Therefore all capability services and the MOM are stored in a `docker-compose.yml` (see Listing A.5).

Since the services have to communicate with each other, a new network is created. Additionally, all necessary ports are exposed so that the front-end can be reached under port `4200 TCP`. Because RabbitMQ is configured to provide an admin dashboard, the web port `15672 TCP` has to be enabled parallel to the messaging port `5672 TCP`. The managing capability has exposed ports as a service because it offers an REST interface. All logs of the back-end services are mounted to the host system, e.g., an ELK-stack[22] can be quickly upgraded to support a comprehensive system monitoring.

The system can be started up using the command `docker-compose up`. In case of a hidden startup `docker-compose up -d` can be used. To stop the system `docker-compose stop` should be used. To delete the containers and the network, `docker-compose down` is recommended.

[22] Elasticsearch, Logstash, and Kibana (ELK) see https://www.elastic.co/de/what-is/elk-stack

7 Evaluation and Discussion

This chapter contains a crucial analysis of the prototype and the entire system architecture. First of all, an evaluation of the system is performed using the requirements already introduced in Section 5.1 (see Section 7.1). Afterward, the results of the evaluation are discussed in Section 7.2.

7.1 Evaluation

This section applies the metrics presented in Section 5.1 to the created prototype. Furthermore, considerations of the architecture are evaluated, under consideration of the respective point of view.

First, the architecture and prototype usability is evaluated (see Section 7.1.1). Subsequently, Section 7.1.2 examines the quality attributes to fulfill interoperability. Moreover, the reusability properties are evaluated in Section 7.1.3.

In the following, points are awarded for each quality attribute. This score $S \in \{-, O, +\}$ describes the general suitability. In the diagrams (cf.Figure 7.1, Figure 7.2 and Figure 7.3) the following correlation is used: "$-$"$\equiv$ 0.0; "O"$\equiv$ 0.5; "+"$\equiv$ 1.0

7.1.1 Usability

In this section, usability is examined using the following quality attributes:

- Effectiveness
- Efficiency
- User satisfaction

Usually, usability only refers to user interfaces. However, this evaluation also takes the developer's view into account. In order to evaluate usability, studies are conducted, and several tasks are set. Finally, measurements can be derived [AAQ10, p. 212]. However, these only provide comparable results since no absolute values are available. The system design has to be evaluated in the presented system because the prototype alone does not provide much information.

Effectiveness Since effectiveness consists of the three components *task effectiveness*, *task completion* and *error frequency*, these need to be considered more in-depth [AAQ10]. Because the presented approach does not affect *task effectiveness*, it can be neglected. The standalone planners are only distributed over their capability services, but their behavior is not affected. Therefore, it can be assumed that the *task completion* is also identical if the capabilities match. If more capabilities were integrated into the system, more tasks would be possible. Since this assumption cannot be made without the generality restrictions, an equal degree of power can be assumed. In the case of *error frequency*, a standalone system is always preferred. This is mainly due to the architecture. If the system components are distributed in a network, the chance to increase the *error frequency* grows due to unavailability.

However, this problem can be overcome with a SOA, since an additional instance can mitigate a service's failure. The system design enables this kind of resilience. Reliability and fault tolerance is proven by important Web platforms such as Amazon, Google, and many more. Through messaging oriented systems, it is easier to use monitoring. Especially in the case of a search failure, monitoring can be a significant advantage.

In summary, it can be concluded that only a statement about the general effectiveness of the architecture can be made at this early stage. So the effectiveness is as high as in a comparable standalone system without network dependencies. The design offers many possibilities to increase effectiveness. Therefore the thesis system approach scores "+" and the standalone planners "+". This result can be viewed in Figure 7.1.

Efficiency This aspect can be divided into two factors. The first is derived from the initial usage (first startup). Since most systems have a large number of system dependencies, the first startup is often time-consuming. In contrast to this, a platform solution, to which the presented approach also belongs, is a platform solution. Such a platform only requires a browser, which is supported by every standard computer. Thus, in case of one-time or spontaneous use, an online platform can be recommended.

The second factor is the runtime from the submission of the planning request. In this case, a platform solution is slower under the assumption of identical computing power because the serialization and the message transfer mean a considerable additional expenditure. Therefore, a standalone planner is expected to be more efficient. However, since efficiency is not only a matter of performance, it must also be taken into account that the new design offers the possibility of parallelization. Furthermore, collaborative scenarios can also be realized.

In summary, the approach is valued at "+" and the standalone planning solution at "+". A cloud solution can bring significant advantages, especially for large planning problems and when using CPS. However, since this is also a possible instance of the design, this scenario cannot be actively included in the general evaluation. In the following discussion, such cases will be discussed in more detail. The efficiency results can be viewed in Figure 7.1.

User Satisfaction During the creation of the prototype, attention was paid to the compliance with the usability criteria presented in Section 5.1.1. By providing the control elements in a web application, a high degree of comfort is achieved. By using correlation IDs, the new approach can handle multiple requests in parallel. Each request from the front-end is processed individually and sent back to the corresponding web socket. However, since no direct comparison is possible at this early stage, these criteria cannot be included in detail in the evaluation.

In general, it can be assumed that a high degree of user satisfaction can be achieved by providing front-end capabilities. However, due to the ability to extend capabilities, the presented approach offers a significant advantage. The probability of being able to deliver the desired function is correspondingly higher than with a typical planner.

Therefore, the potential of this work's approach, for user satisfaction, is rated "+", while the use of the independent planner reaches only "*O*". The satisfaction results can be viewed in Figure 7.1.

Conclusion

To visualize the result of the usability evaluation, a spider web diagram is used. The axes are inserted according to the three quality attributes, and the value range is shown normalized. Figure 7.1 shows the resulting diagram.

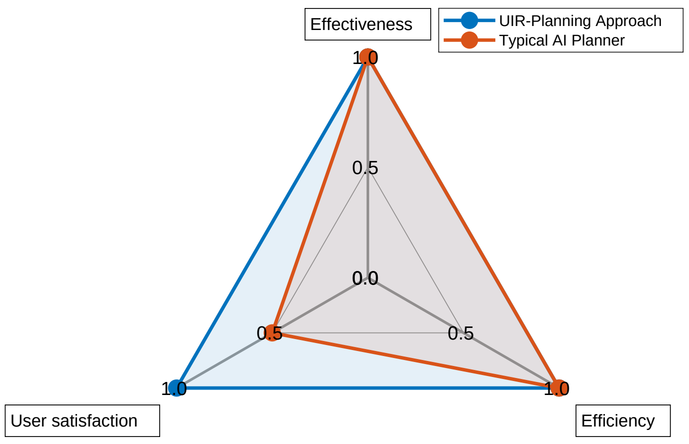


Figure 7.1: Usability comparison between the planning platform and a typical planner

Since this evaluation only examines the approach and the design, the following result is obtained. The design does not succeed in trumping the typical standalone planner in all criteria. This fact is mostly due to the SOA, which can causes disadvantages, especially in planning requests. Although the SOA also offers various possibilities to improve usability. In the following two areas, interoperability and reusability, more advantages can be expected. Nevertheless, a usable design has been achieved since all of the examined criteria are fulfilled.

If several capabilities are implemented in the future, a user study can be created, evaluating the system instance, i.e., the extended prototype. The metrics from Abran and Al-Qutaish could be used for this [AAQ10].

7.1.2 Interoperability

In interoperability, a comparison with other systems is even more difficult since concepts for SOC were assumed as evaluation criteria. Thus interoperability is composed of the following factors:

- Organizational interoperability
- Semantic interoperability
- Syntactic interoperability
- Connectivity interoperability

In the following, the quality characteristics of interoperability are analyzed.

Organizational Organizational interoperability is always given if a system is designed. Because of the earmarking, there is a purpose for every planner. So, in general, a "+" can be assigned for each planner. The Results of organizational interoperability are visualized in Figure 7.2.

Semantic Since semantic interoperability requires a SOA, this attribute cannot be set for all planners and is therefore evaluated as "−". On the other hand, the presented approach can be evaluated with "O" because the corresponding processing is guaranteed by specifying the expected return type, but no mandatory validation is performed. Thus, it is theoretically possible to misinterpret objects if they are misused. This can be further reduced if the interfaces are sufficiently documented. However, this case is unlikely, and therefore a partial fulfillment of the criterion is justified. The semantic interoperability results can be viewed in Figure 7.2.

Syntactic Since syntactic interoperability requires a SOA, this attribute cannot be set for all planners and is therefore evaluated as "−". However, "+" can be assigned for the presented design since it is generally based on a uniform communication format (JSON). Although for technical reasons, front-end capabilities are connected via REST and all other capabilities communicate via MOM, the format is identical. Thus, this system-wide uniform communication is given. The syntactic interoperability result is showen in Figure 7.2.

Connectivity Since connectivity requires a SOA, this attribute cannot be set for all planners and is therefore evaluated as "−". As already mentioned in the previous section, the approach is best supported by commonly used standards. For the front-end REST is used, which is based on the HTTP protocol. The choice of MOM is RabbitMQ, which is also widely supported by a large community. Predefined controllers in most languages support both protocols. Thus the award of the score "+" is justified. The connectivity result can be seen on Figure 7.2.

Conclusion

Since only a few planners have a SOA, the result is not surprising (cf. Chapter 3). To use foreign capabilities in the architecture, they must first be provided with the necessary interfaces using wrappers. The protocols used are suitable for interoperability due to their wide use, as many existing platforms from other areas show. Only in the area of semantic interoperability exists the potential for improvement through the more precise documentation specification. Further possibilities are discussed in Section 7.2.

Figure 7.2 shows again that the interoperability of typical standalone planners is not given. It can be observed that only the organizational interoperability is given.

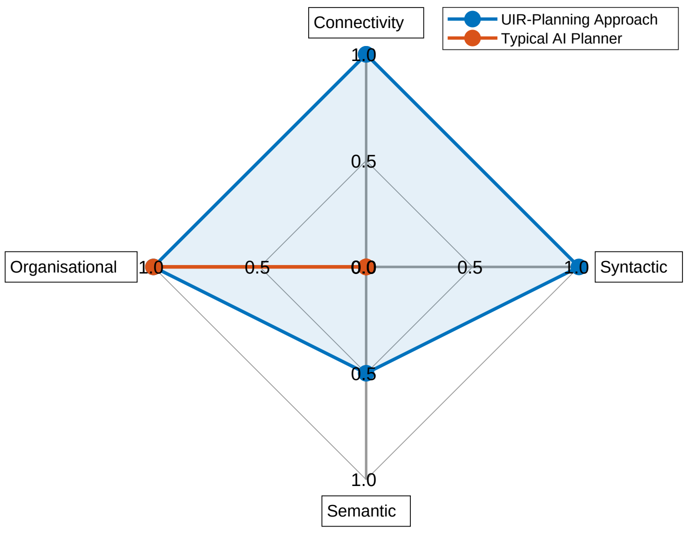


Figure 7.2: Interoperability comparison between the planning platform and a typical planner

7.1.3 Reusability

User studies also measure reusability. Requirements lists can be used to compare the required scope with the current one. Since the prototype only implements a minimal subset of the design, such a comparison is currently not applicable.

Thus, the approach with four quality features presented in Section 5.1.3 is evaluated on the system level. These attributes are [CBV97]:

- Portability
- Flexibility
- Understandability
- Confidence

Portability Since the required dependencies can determine portability, it is necessary to have as few dependencies as possible. The architecture uses micro-services to encapsulate dependencies. Although the services contain many dependencies to external libraries, these are already required during the build. As already shown in the prototype, it is possible to use dependency handlers (e.g., Gradle, Maven, etc.) to minimize repository dependency. During runtime, the services are entirely decoupled apart from the communication interface. Therefore "+" is given for the portability of the system.

Most of the planners are based on system libraries (e.g., PandaPIParser), which may be due to the programming language used (e.g., C++) and the architecture. Only a few planners offer encapsulated solutions, such as the delivery as an image. Even runnable applications are often challenging to find, so it is often necessary to handle all build dependencies. Therefore, "−" is given for the typical standalone planner's portability.

Flexibility The Flexibility attribute consists of two components, generality and modularity. The proposed system offers a high degree of flexibility for both generality and modularity. This is mainly due to the encapsulation of capabilities, which can also be interpreted as individual modules. By using standard communication, generality is also given, and a composition of different capabilities is possible. Therefore the design is rated with "+".

When examining other planners, it was noticed that most of the systems do not use any other modular structures besides the division into packages. Also, the generality of most of the planners is low. However, a set of utility functions is often provided, which can be used without the application context. This allows a minimal degree of flexibility, and the typical planner can be classified as "O".

Understandability Understandability is also divided into two factors. The first factor is the documentation. The new approach has the possibility of detailed code documentation. However, there is no functionality in the prototype that forces this description. Therefore, no full score can be given for this subitem. Complexity is the second factor. Since the system's total complexity is distributed over several services, it can be expected that the functional complexity of the individual services will decrease. However, this must be considered concerning the communication effort that a distributed system incurs. Since the functional complexity outweighs the technical complexity, "+" can be assigned as evaluations combined with both factors.

In the other systems, the documentation is usually very sporadic. However, the behavior is presented as a scientific paper, which can also be seen as a kind of documentation. Nevertheless, there is no guarantee for consistency. Since standalone planner bundles all functionalities, the behavior is much more complicated. Therefore the comprehensibility of the individual components is worse than in a system divided into capabilities. Therefore, in this case, only "O" can be given as a rating.

Confidence According to Cardino et al., confidence is only determined by maturity. An analysis of this criterion is not generally applicable. Thus, in the new approach and all other planners, it can be assumed that confidence is given with "+". Deductions can only be made during operation or during the examination of the implementation.

Conclusion

In summary, the new approach enables improved reusability. Figure 7.3 shows the approach's superiority in terms of reusability.

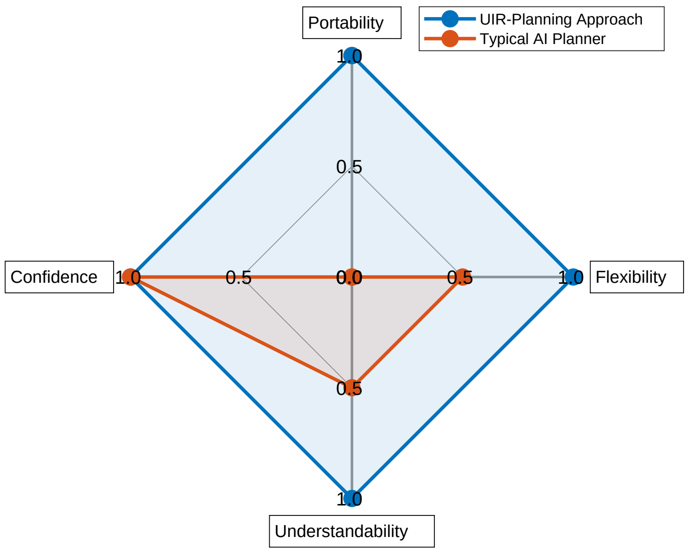


Figure 7.3: Reusability comparison between the planning platform and a typical planner

However, the comparison to an average planner is not very detailed. Moreover, future investigations utilizing benchmarks are necessary to increase significantly. For this purpose, the prototype needs a broader range of capabilities to be comparable with multiple planners.

7.2 Discussion

In the course of this master thesis, the classification of AI planning capabilities and the conditions they have to fulfill to be *usable*, *interoperable*, and *reusable* were analyzed. Furthermore, a system architecture was presented that provides a framework for embedding such capabilities (cf. RQ_1). This thesis's findings show that a SOA is suitable to fulfill the requirements.

In this section, the results of the thesis are discussed. First, the results are summarized and then interpreted.

7.2.1 Findings

During this thesis, the aim is to answer the research question RQ_1.

Literature research has shown that there is a large number of different AI planners. According to the current state of the art, these planners can be classified as a whole in the planner categories (see Section 2.1.3). Within this work, a list of capabilities has been identified and classified using two metrics (see Section 4.2). The first metric is focused on compositional behavior (operational view) and divides capabilities into *Core*, *Supporting* and *Supplemental*. This metric illustrates that specific capabilities require individual other capabilities (see Section 4.2.1).

The second metric is used for positioning within a composition (technical view) and divides capabilities into four classes (*Endpoint*, *Transforming*, *Router*, *System-Management*). The exact division is stated in Section 4.2.2).

Before creating a system architecture design, requirements were derived from the quality attributes *usable*, *interoperable*, and *reusable*. Afterward, the requirements were directly considered in the system design. To verify the fulfillment of these requirements, a comparison to a generic prototypical standard planner was made in the evaluation. This serves the better arrangement of the architecture in the current system landscape.

While working on the research questions, $RQ_{4.1-4.3}$, it was noticed that there are no standards besides the modeling languages. Therefore, it is not possible to design interfaces that support all available planners. These findings are reflected in the option of multiple instances of a capability. These multiple capability instances allow the interfaces to be supported by popular planners. It is therefore not necessary to operate all planners via one interface. The creation of new potential standards is not possible within the scope of this thesis. However, it is a reasonable approach to create JSON interfaces based on the modeling languages. This opportunity is only possible for the interfaces for parsing capabilities.

During the implementation of the prototype, the missing, loose coupling in the existing planners was noticed. Therefore, it is not straightforward to decompose existing standalone planners into their capabilities. This task was significantly complicated by the lack of serializable classes and objects. Therefore it was necessary to integrate every capability into services using a wrapper that uses a serializable data structure. In close exchange with other developers, the problem of interpretation during parsing was addressed. As a result that many planners already optimize their data structure during parsing. Others do not even create objects but set system variables.

7.2.2 Interpretation of the findings

Within this section, the results are interpreted. Initially, the differentiation to the related works is presented. Subsequently, the advantages and disadvantages of the design and the prototype are discussed.

Distinction of related works

This section compares each of the identified related works to facilitate a better understanding of the approach and the developed prototype.

CPEF The CPEF (see Section 3.1) offers an extensive repertoire of planning capabilites. However, the planning system's various tightly coupled components result in a lack of flexibility addressed in the proposed approach. The range of functions exceeds the prototype by far. Nevertheless, the full functional range of the UIR-Approach is far from exhausted.

PELEA The planning system PELEA (see Section 3.2) is not a SOA as well as CPEF. Therefore, a lack of flexibility is evident in this case too. The missing encapsulation of the capabilities makes it difficult to replace components quickly.

Space Mission Planning System The Space Mission Planning System (see Section 3.3) promises, in addition to a SOA, a primary basis for the division of capabilities into different services. However, this system is only theoretical and has not been implemented. The reason for this is not described in the introduction paper (see [FPD13]). It can be assumed that the problems with the implementation correspond to issues found while implementing the prototype of the UIR approach. According to this, the number of available planners, and their combination in a meta-model was impossible.

RuG Planner Since the RuG Planner (see Section 3.4) is already severely restricted by the requirement of CSP solvers, this approach is not as powerful as the UIR approach. Nevertheless, a good performance can be expected from the architecture, and the SOA allows reactive methods. However, like many others, the system has a problem with the modules' extensibility and flexibility. In this example, the SOA is mainly used for scalability of the system and not for encapsulation of capabilities. This thesis's approach allows this kind of performance scaling, but the current prototype does not provide for its use. An enhancement of this functionality is easy to implement since each capability receives a *In-Queue* in addition to the *Routing-Key*. This behavior allows new capability instances with the same *Routing-Key* to be used as workers to process the requests.

SH Planner One of the most significant differences is that the SH-Planner is primarily based on REST (cf. Section 3.5). Only the communication with the IoT sensors is realized with RabbitMQ. Therefore, problems can occur when adding services if no asynchronous communication is available. Since the SH-Planner has few services (Planner, Manager) REST communication is sufficient. However, the UIR approach can provide many services, so a different communication type had to be chosen. The basic idea of providing planning as a service can be found in both approaches, but the UIR approach offers a possibility of dedicated use. If a capability is required, it is sufficient to include a service that provides it. In this way, an application with low complexity can be designed. Comparing the prototypes, a similar range of functions is shown apart from the IoT control and configuration.

planning.domains The planner introduced in Section 3.6 provides various planning aspects. The most significant difference lies in the composition of the services. Although the Planning.Domains only consists of front-end and back-end, while the back-end is entirely based on LAPKT; the presented system architecture can integrate different services from different developers. In addition, planning.Domains is actually only available for PDDL, which can be compared to the prototype's latest implementation.

Besides Planning.Domains there are further online planners for PDDL. For example, the Web-Planner[1] or Stripsfiddle[2]. However, these tools are tools for educational purposes only. The created plans cannot be executed directly or used in other extensions. This is how the UIR approach stands out from these systems, even though the current version prototype does not yet add significant value.

SOA-PE Planner The SOA-PE Planner, which was introduced in Section 3.7, lacks the solver's interchangeability. All capabilities have to be included in a single planner service, which is challenging to realize with increasing functionality. Compared to the new architecture, the communication within the system is dedicated to the services. The services have to identify themselves uniquely to communicate with each other. The UIR-Planning approach allows communication via messaging, using a MOM. This way, for example, scaling can be introduced, and offline services can be better handled.

Advantages and disadvantages

Summary from previous findings, both pros and cons can be derived. This section provides a brief discussion of these topics. First of all, the advantages are considered.

Benefits of the UIR-Planning approach The proposed approach offers further advantages besides the UIR properties (Usability, Interoperability, and Reusability). One of these is the enormous flexibility, which is reflected in the expandability of service capabilities. Furthermore, any data model can be created and made available for further processing in services. The only thing that has to be considered is the serialization. Also, a bidirectional converter should be provided to ensure the interoperability of the capabilities. Since no standards have been decided upon yet, only best practice solutions can be provided. If a suitable interface is already provided in the system, all other researchers are encouraged to use it or create their versions. Thus, the proposed design serves as an ecosystem and incubator for new AI planners since the system's complexity can be reduced by providing pre-configured services, thus saving many problems and time. Especially when using rapid prototyping approaches, the system offers excellent potential. This performance is primarily due to the high degree of reusability of service capabilities, which is further supported by the services' interoperability.

The system can also facilitate beginners' entry into the research field of AI Planning. This concept is already applied in numerous related planners (e.g., Planning.Domains, Web-Planner, ...).

One of the biggest strengths only becomes apparent when the system is connected to the real world (cf. CPS). An executing endpoint capability realizes this. Many existing connector endpoints can be integrated with minimal effort through messaging, and several monitoring approaches can also be implemented.

[1] `https://web-planner.herokuapp.com` (November 24, 2020)
[2] `https://stripsfiddle.herokuapp.com` (November 24, 2020)

With the SOA a horizontal scaling of the capabilities is supported. Thereby the messaging supports the scaling utilizing in-queues. Since all capabilities are implemented stateless (except persisting ones), it is also possible to reduce the number of instances and provide better reliability (e.g., deadline queues).

Drawbacks of the UIR-Planning approach The introduced architectural design, and particularly the implemented prototype, offer room for improvement.

The most significant shortcomings can currently be found in the front-end. Due to this work's tight time frame, it is impossible to implement all types of modeling (e.g., the configuration of automation), so the potential of a web-based application for modeling planning domains was not fully exploited. The modeling capability was also burdened with unspecific underlying functions at some modules. A different approach towards micro-frontend is desirable to better separate the front-end capabilities from the system relevant code. In the case of Planning.Domains this problem is realized with plugins (see Section 3.6).

To achieve acceptance of the system, in the community, it is necessary to implement additional functionality. Beyond that, the available range is also an essential part of the system's usability, which cannot be accomplished in this work.

Another problem still exists. Since there is a lack of standards in AI planning, it is also not possible to provide corresponding standardized interfaces.

As already mentioned, the extraction of capabilities from AI planners is a significant challenge. This is partly due to the neglect of software engineering principles, such as encapsulating logic from data models, and partly due to the focus on behavior. These insights were gained during the implementation of the parsing capability services for the prototype.

In most cases, if there is an encapsulated class for data storage, it is implemented with logic. Some cases have no data structures, and the results are stored in global variables (see Appendix B.2.1). This case occurs mainly with C++ based planners.

Objects are used to pass information, sometimes preventing manual constructor calls in the class (information hiding) to avoid incorrect data structure handling. For the provision of frameworks, such behavior is useful, but the UIR approach requires the forwarding of objects. Therefore the serialization of classes is essential. By decomposing the planner into single capabilities, conversions to serializable classes are necessary.

The only standards the community can agree on are the modeling languages, such as PDDL, HDDL and SHOP. When parsing the domain or problem, fundamental decisions are already taken, which can influence further processing. Many planners also use grounding algorithms directly linked to the parsing process, and only the resulting objects can be used as a basis for planning.

The introduced approach offers no solution for this lack of standards. It only provides a workflow to animate developers to create reusable services in JSON format and reuse others' services. Monitoring can help to find the most popular capabilities and declare the interfaces as a standard.

Another problem is the documentation of various capabilities. To keep track of a large number of active capability services, detailed documentation of each capability and its implemented interfaces is necessary. Since the UIR-approach does not offer a specific way to do this, it is necessary to find a uniform and language-independent approach. A dynamic wiki is conceivable in the planning context, which provides the capability documentation for each topic and the routing key. The currently used JavaDocs are not sufficient for such a platform architecture documentation.

In the prototype, the modeling capability is implemented as a single front-end service. Therefore, the capability contains further functionality needed to provide the results and similar (e.g., login). These functionalities have been outsourced to other modules to minimize this overload, but a refactoring of the system architecture is still reasonable for this area. A micro-front-end could be used to create a platform for front-end services similar to the back-end. In this way, the capability's complexity can be reduced, and system relevant communication services can be outsourced. Furthermore, the use of front-end technologies would be made freely selectable.

8 Conclusions and Future Work

This chapter concludes this thesis. First, Section 8.1 summarizes the content and presents the results of this thesis. In the following Section 8.2, lessons learned are mentioned. Particular attention is paid to the challenges and possible solutions. Finally, an outlook is given in Section 8.4. It addresses possible improvements and extensions to the architecture and implementation.

8.1 Summary

This thesis presented a usable, interoperable, and reusable approach for an AI planning system and implemented it as a prototype.

Initially, typical planning capabilities were identified and subsequently classified (cf. RQ_2). The classification results in two metrics, while one describes the operational view and the other the technical view. The Operational View subdivides the Capabilities based on their applicability. Thus, the following three classes are distinguished: *Core Capability*, *Supporting Capability*, and *Supplemental Capability*. The technical view is composed of four categories, which are modeled based on the *Enterprise Integration Patterns*. These include: *Endpoint Capabilities*, *Transforming Capabilities*, *Router Capabilities*, and *System-Management Capabilities*.

Using these classifications, the appropriate *Enterprise Integration Patterns* were subsequently applied to create a messaging oriented SOA. This work presents the UIR planning system. To guarantee usability, interoperability, and reusability, several requirements have been elaborated to meet (cf. RQ_3). Following this, the capabilities were designed to provide a standardized and generic interface to support a wide a range of planners (cf. RQ_4). Therefore, standard technologies were used to implement the prototype (see Chapter 6). Furthermore, JSON is used as the communication standard format of MOM. The presented architectural design and the implemented prototype provide an answer to the final research question, how planning capabilities should be designed and realized to make them *usable*, *interoperable*, and *reusable* (cf. RQ_1).

The prototype contains a fundamental subset of planning capabilities. This subset includes an *Modelling Capability* for modeling a domain and a problem. Besides manual modeling by utilizing a Web IDE and visualizing features (e.g., a sequential plan), this front-end capability offers essential administrative control. Due to this work's time constraints, only PDDL was integrated as a modeling language. This decision is based on the reason that PDDL is the most common modeling language and since PDDL4J is an easily integrable toolkit for JVM based microservices.

To realize the architecture as described, a *Managing Capability* is implemented in the prototype. This capability is used for asynchronous process management and acts as REST gateway for the front-end services. Furthermore, three additional PDDL4J based capability services are provided.

These include a *Solving Capability*, a *Parsing Capability* and a *Converting Capability* for encoding PDDL problems. Accordingly, the system provides the option to create a plan for a problem specified as PDDL in a distributed AI planning system.

The number of capabilities, in the prototype, is currently still too small for any “real-world” application. However, the potential of the capability services and dynamic routing is illustrated. In the evaluation, the UIR properties are fulfilled by the novel design. Nevertheless, it can be seen in the discussion, that a significant amount of effort must be invested in splitting up the existing planners. This required investment is basically due to the disregard of software engineering principles during the planners’ development. The presented architecture provides a basis/platform, which must first be enhanced with capability services to provide a high level of benefits.

To keep the installation effort on the user side as low as possible, the system was developed with the cloud in mind. The prototype is already designed in compliance with cloud standards. Moreover, it can thus be deployed in the cloud without significant modifications. The current version features build scripts that launch the containerized deployment environment locally. However, user authorization concepts were not initially considered, so online deployment of the prototype is not recommended.

8.2 Lessons Learned

One of the significant challenges has been encountered while trying to create a meta-model. In addition to the different naming of the various attributes, it is also not always guaranteed, that no preconditions are imposed on the data structure. Thus, it is not easy to recognize whether the assumed behavior is always identified as external. Furthermore, there are many modeling languages, which are currently the only foundation for interoperability of the different planners. To design serializable and universal data models, extensive research on existing systems and modeling languages is necessary. The research conditions are aggravated because not all presented planners, including source code, are available online. Mostly the internal interfaces are not described at all.

To overcome these challenges, only a generic base message is possible within the scope of this work. Thus a JSON base message format was created, which uses every capability. Only system-relevant fields are defined, and the content field is freely customizable. By providing different capabilities of the same category, all critical systems can be integrated. The system allows easy reuse of individual capabilities that match the consumer’s requirements.

Communication in a distributed system requires a communication standard. Since the system is based on a MOM, a uniform system internal communication is guaranteed. However, to increase interoperability, it is necessary to choose a standard, that imposes as few conditions on the endpoints as possible. Therein is another challenge. Since almost all endpoints support a JSON-based message, it is necessary to use this form of communication from an architectural perspective. Nevertheless, manual serialization requires a lot of time and effort on the part of the capability developers.

Most messaging-based systems use marshaling instead of serialization, because the developer’s comfort and development speed outweigh the cost. Since interoperability is a star quality attribute of the system, it is necessary to use serialization and deserialization. Through the decision to use Kotlin in the back-end, it is possible to use the Jackson framework as a message converter. However, some structures cannot be converted by automatic serialization. In these cases a manual serialization is necessary. To solve problems, during deserialization, it is usually necessary to set annotations in the data model. For example, polymorphism problems can be solved.

8.3 Limitations

The system depends on the implementation of planning capabilities. This step can be made easier if new standards are developed in the future. So far, it is necessary to manually separate the planners according to their capabilities, where problems can occur considering the programming language and the underlying architecture.

To get to new standard interfaces, further scientific work is required. The development of new interfaces requires further scientific work. The acceptance of the AI planning community is necessary to move from interfaces to new standards.

In the developed prototype, the presented design has difficulties separating the front-end functionalities on the service level. At this point, it is currently not possible to separate the modeling from the visualization. This problem should be considered in future implementations of the system.

8.4 Future Work

Initially, the foundation for the dynamic registration of new capabilities should be integrated into the prototype. Once this feature is integrated into the prototype, feedback from experts and developers can be obtained from other planners. Furthermore, the verification of the approach in the form of a case study is conceivable. By integrating several capabilities, which are compatible in a useful context, it is possible to increase the prototype's value quickly. As already proven by Georgievski [Geo15], IoT and Industrial-IoT are areas of expertise that can make a major contribution to improving efficiency through the use of AI Planning. A CPS could also serve as a showcase.

As already mentioned in the discussion (see Section 7.2), the front-end needs to be rethought, because the current architecture is designed to overload distributed capability services. Thus, visualizing components are displayed in a modeling capability service. To counteract this problem, a refinement of the architecture is necessary. According to the current state of knowledge, a micro-front-end seems to be an alternative solution (see [Gee20], and [PAMM20]). A corresponding approach has to be evaluated separately.

Another improvement is the use of Function as a Service (FaaS) and Serverless [FIMS17]. A lot of system relevant code is managed by a framework in the current version, but many configurations and controllers are necessary to connect the technologies. To further focus on the core competence of capability services, the use of FaaS is worth investigation.

Since the well-known planners started at IPC, they are focused on performance optimization (speed and memory). Thereby, other quality attributes suffer, which are mainly non-functional requirements. These requirements are necessary from an engineering perspective to achieve high software quality (examples include maintainability). Due to the lack of these quality attributes, it is difficult to split the capabilities. A possible impulse could come from the IPC. Besides the classic benchmark competitions, additional evaluation methods could be integrated. For example, reviews could evaluate the quality of the planners. This change creates incentives to improve the planners and is intended to take better account of the engineering aspect.

Bibliography

[AAQ10] A. Abran, R. Al-Qutaish. “ISO 9126: Analysis of Quality Models and Measures”. In: 2010, pp. 205–228. ISBN: 9780470597200. DOI: 10.1002/9780470606834.ch10 (cit. on pp. 113–115).

[AF17] M. Asai, A. Fukunaga. “Classical Planning in Deep Latent Space: Bridging the Subsymbolic-Symbolic Boundary”. In: *CoRR* abs/1705.0 (2017). arXiv: 1705.00154. URL: http://arxiv.org/abs/1705.00154 (cit. on pp. 21, 50).

[AGH99] J. Allen, C. Guinn, E. Horvtz. “Mixed-initiative interaction”. In: *IEEE Intelligent Systems* 14.5 (1999), pp. 14–23. ISSN: 1094-7167. DOI: 10.1109/5254.796083. URL: http://ieeexplore.ieee.org/document/796083/ (cit. on p. 48).

[AGP+10] V. Alcázar, C. Guzmán, D. Prior, D. Borrajo, L. Castillo, E Onaindia. “PELEA: Planning, learning and execution architecture”. In: *PlanSIG’10* (2010), pp. 17–24 (cit. on p. 37).

[AHK+98] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. SRI, A. Barrett, D. Christianson, et al. “PDDL—The Planning Domain Definition Language Version 1.2”. In: (1998) (cit. on p. 28).

[Ale79] C. Alexander. *The timeless way of building*. Vol. 1. New York: Oxford University Press, 1979 (cit. on p. 54).

[BCC12] J Benton, A. Coles, A. Coles. “Temporal Planning with Preferences and Time-Dependent Continuous Costs”. In: *ICAPS 2012 - Proceedings of the 22nd International Conference on Automated Planning and Scheduling* (2012) (cit. on p. 32).

[BDB20] G. Behnke, H. Daniel, P. Bercher. *International Planning Competition 2020 Results*. 2020. URL: http://gki.informatik.uni-freiburg.de/competition/results.pdf (cit. on p. 90).

[Bec15] K. Becker. *blocksworld1 example*. 2015. URL: https://github.com/primaryobjects/strips/tree/master/examples/blocksworld1 (visited on 08/10/2020) (cit. on pp. 28, 29).

[BHS+20] G. Behnke, D. Höller, A. Schmid, P. Bercher, S. Biundo. “On Succinct Groundings of HTN Planning Problems”. In: *Proceedings of the 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*. AAAI Press, 2020 (cit. on pp. 92, 93).

[BKB14] P. Bercher, S. Keen, S. Biundo. “Hybrid planning heuristics based on task decomposition graphs”. In: *Seventh Annual Symposium on Combinatorial Search*. 2014 (cit. on pp. 46, 48–51, 95).

[BL19] U. Breitenbücher, F. Leymann. *Cloud Computing*. Stuttgart, 2019 (cit. on p. 73).

[BS07] S. Bernardini, D. Smith. “Developing domain-Independent search control for EUROPA2”. In: *ICAPS Workshop on Heuristics for Domain-independent Planing* 1 (2007). URL: https://ti.arc.nasa.gov/m/pub-archive/1364h/1364%20%28Smith,%20D%29.pdf (cit. on p. 32).

[Byl94] T. Bylander. “The computational complexity of propositional STRIPS planning”. In: *Artif. Intell.* 69.1-2 (1994), pp. 165–204. ISSN: 00043702. DOI: 10.1016/0004-3702(94)90081-7 (cit. on p. 28).

[CBV97] G. Cardino, F. Baruchelli, A. Valerio. “The evaluation of framework reusability”. In: *ACM SIGAPP Applied Computing Review* 5.2 (1997), pp. 21–27. ISSN: 1559-6915. DOI: 10.1145/297075.297085 (cit. on pp. 70, 71, 117, 118).

[CDE+20] G. Chen, Y. Ding, H. Edwards, C. H. Chau, S. Hou, G. Johnson, M. Sharukh Syed, H. Tang, Y. Wu, Y. Yan, T. Gil, L. Nir. *Planimation*. 2020. DOI: 10.5281/zenodo.3773027. URL: https://doi.org/10.5281/zenodo.3773027 (cit. on p. 50).

[CFL+15] M. Cashmore, M. Fox, D. Long, D. Magazzeni, B. Ridder, A. Carrera, N. Palomeras, N. Hurtós, M. Carreras. “Rosplan: Planning in the robot operating system”. In: *Proceedings International Conference on Automated Planning and Scheduling, ICAPS* 2015-January (2015), pp. 333–341. ISSN: 23340843 (cit. on p. 46).

[CFOGPP06] L. A. Castillo, J. Fernández-Olivares, Ó. Garc\'\ia-Pérez, F. Palao. “Efficiently Handling Temporal Knowledge in an HTN Planner.” In: *ICAPS*. 2006, pp. 63–72 (cit. on p. 31).

[CK08] S. W. Choi, S. D. Kim. “A Quality Model for Evaluating Reusability of Services in SOA”. In: *2008 10th IEEE Conference on E-Commerce Technology and the Fifth IEEE Conference on Enterprise Computing, E-Commerce and E-Services*. 2008, pp. 293–298. DOI: 10.1109/CECandEEE.2008.134 (cit. on pp. 68–71).

[DBMIG14] F. Dvorak, A. Bit-Monnot, F. Ingrand, M. Ghallab. “Plan-Space Hierarchical Planning with the Action Notation Modeling Language”. In: *IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*. Limassol, Cyprus, 2014. URL: https://hal.archives-ouvertes.fr/hal-01138105 (cit. on pp. 31, 46, 48).

[Del13] J. Delgado. “Service Interoperability in the Internet of Things”. In: 2013, pp. 51–87. DOI: 10.1007/978-3-642-34952-2_3. URL: http://link.springer.com/10.1007/978-3-642-34952-2_3 (cit. on pp. 66, 67).

[EHN94] K. Erol, J. A. Hendler, D. S. Nau. “UMCP: A Sound and Complete Procedure for Hierarchical Task-network Planning.” In: *Aips*. Vol. 94. 1994, pp. 249–254 (cit. on p. 46).

[Erl07] T. Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl)*. USA: Prentice Hall PTR, 2007. ISBN: 0132344823 (cit. on pp. 19, 33).

[FIMS17] G. C. Fox, V. Ishakian, V. Muthusamy, A. Slominski. “Status of Serverless Computing and Function-as-a-Service(FaaS) in Industry and Research”. In: *CoRR* abs/1708.0 (2017). arXiv: 1708.08028. URL: http://arxiv.org/abs/1708.08028 (cit. on p. 127).

[FJ03] J. Frank, A. Jónsson. "Constraint-Based Attribute and Interval Planning". In: *Constraints* 8.4 (2003), pp. 339–364. ISSN: 13837133. DOI: 10.1023/A:1025842019552. URL: https://ti.arc.nasa.gov/m/pub-archive/313h/0313(Frank).pdf (cit. on p. 32).

[FL03] M. Fox, D. Long. "PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains". In: *J. Artif. Intell. Res.* 20 (2003), pp. 61–124. ISSN: 1076-9757. DOI: 10.1613/jair.1129. URL: https://jair.org/index.php/jair/article/view/10352 (cit. on p. 28).

[FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns*. Vienna: Springer Vienna, 2014. ISBN: 978-3-7091-1567-1. DOI: 10.1007/978-3-7091-1568-8. URL: http://link.springer.com/10.1007/978-3-7091-1568-8 (cit. on p. 73).

[FMJB17] A. V. Feljan, S. K. Mohalik, M. B. Jayaraman, R. Badrinath. "SOA-PE: A service-oriented architecture for Planning and Execution in cyber-physical systems". In: *2015 Int. Conf. Smart Sensors Syst. IC-SSS 2015* (2017). DOI: 10.1109/SMARTSENS.2015.7873602 (cit. on pp. 17, 19, 43, 44, 46, 48, 49).

[FOCGPP06] J. Fdez-Olivares, L. Castillo, O. Garc\ia-Pérez, F. Palao. "Bringing users and planning technology together. Experiences in SIADEX". In: *Proc. ICAPS*. 2006, pp. 11–20 (cit. on p. 46).

[FPD13] S. Fratini, N. Policella, A. Donati. "A Service Oriented approach for the Interoperability of Space Mission Planning Systems". In: *4th Work. Knowl. Eng. Plan. Sched. (KEPS 2013)* (2013) (cit. on pp. 17, 19, 38, 39, 46, 48, 50, 121).

[FRF+13] S. Fernández, T. de la Rosa, F. Fernández, R. Suárez, J. Ortiz, D. Borrajo, D. Manzano. "Using automated planning for improving data mining processes". In: *The Knowledge Engineering Review* 28.2 (2013), pp. 157–173. ISSN: 0269-8889. DOI: 10.1017/S0269888912000409. URL: https://www.cambridge.org/core/product/identifier/S0269888912000409/type/journal_article (cit. on p. 27).

[GA15] I. Georgievski, M. Aiello. "HTN planning: Overview, comparison, and beyond". In: *Artif. Intell.* 222 (2015), pp. 124–156. ISSN: 00043702. DOI: 10.1016/j.artint.2015.02.002. URL: http://dx.doi.org/10.1016/j.artint.2015.02.002 (cit. on p. 19).

[GAP+12] C. Guzmán Alvarez, V. Alcázar, D. Prior, E. Onaindia, D. Borrajo, J. Fdez-Olivares, E. Quintero. "PELEA: a Domain-Independent Architecture for Planning, Execution and Learning". In: 2012 (cit. on pp. 19, 37, 46, 48, 50).

[Gee20] M. Geers. *Micro Frontends in Action*. Manning Publications, 2020 (cit. on p. 127).

[Geo13] I. Georgievski. *Hierarchical planning definition language*. Tech. rep. Groningen: University of Groningen, JBI 2013-12-3, 2013. URL: https://ilche.io/wp-content/uploads/papers/georgievski2013-hpdl.pdf (cit. on p. 31).

[Geo15] I. Georgievski. "Coordinating services embedded everywhere via hierarchical planning". PhD thesis. University of Groningen, 2015. ISBN: 978-90-367-8148-0 (cit. on pp. 17, 19, 25–28, 35, 40, 41, 46–51, 127).

[GI89] M. P. Georgeff, F. Ingrand. “Decision-Making in an Embedded Reasoning System”. In: *Int. Jt. Conf. Artif. Intell.* Detroit, United States, 1989. URL: https://hal.laas.fr/hal-01980071 (cit. on p. 36).

[GK19a] R. P. Goldman, U. Kuter. “Hierarchical Task Network Planning in Common Lisp: the case of SHOP3”. In: *Proc. ELS '19 Eur. Lisp Symp. (ELS '19)* (2019), pp. 73–80. DOI: 10.5281/zenodo.2633324. URL: https://rpgoldman.goldman-tribe.org/papers/2019-els-SHOP3.pdf (cit. on pp. 30, 46).

[GK19b] R. P. Goldman, U. Kuter. *SHOP3 Manual*. 2019. URL: https://shop-planner.github.io (visited on 08/06/2020) (cit. on p. 30).

[GKP20] K. M. Guido, M. Kanbur, A. Pavlovski. “Analysis of AI Planning Systems”. Fachstudie. University of Stuttgart, 2020 (cit. on pp. 17, 18, 23).

[GL05] A. Gerevini, D. Long. “Plan constraints and preferences in PDDL3”. In: *ICAPS Work. Soft Constraints Prefer. Plan.* (2005) (cit. on p. 28).

[GNA13] I. Georgievski, T. A. Nguyen, M. Aiello. “Combining Activity Recognition and AI Planning for Energy-Saving Offices”. In: *2013 IEEE 10th Int. Conf. Ubiquitous Intell. Comput. 2013 IEEE 10th Int. Conf. Auton. Trust. Comput.* IEEE. IEEE, 2013, pp. 238–245. ISBN: 978-1-4799-2482-0. DOI: 10.1109/UIC-ATC.2013.106. URL: http://ieeexplore.ieee.org/document/6726215/ (cit. on p. 31).

[GNK07] R. P. Goldman, D Nau, U Kuter. “Documentation for SHOP2”. In: *University of Maryland* (2007), pp. 1–42 (cit. on pp. 30, 49, 93).

[GNT04] M. Ghallab, D. Nau, P. Traverso. *Automated Planning*. Elsevier, 2004, pp. 1–635. ISBN: 9781558608566. DOI: 10.1016/B978-1-55860-856-6.X5000-5. URL: https://linkinghub.elsevier.com/retrieve/pii/B9781558608566X50005 (cit. on pp. 19, 24–27, 30, 32).

[GNT16] M. Ghallab, D. Nau, P. Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016, pp. 1–354. ISBN: 9781139583923. DOI: 10.1017/CBO9781139583923 (cit. on p. 19).

[Has] P. Haslum. *Patrik Haslum - Planning*. URL: http://users.cecs.anu.edu.au/~patrik/ (cit. on p. 21).

[HBB+19] D. Höller, G. Behnke, P. Bercher, S. Biundo, H. Fiorino, D. Pellier, R. Alford. “HDDL – A Language to Describe Hierarchical Planning Problems”. In: 2.2 (2019). arXiv: 1911.05499. URL: http://arxiv.org/abs/1911.05499 (cit. on p. 31).

[HCM+11] E. Hidalgo, L. Castillo, R. I. Madrid, Ó. García-Pérez, M. R. Cabello, J Fdez-Olivares. “ATHENA: Smart Process Management for Daily Activity Planning for Cognitive Impairment”. In: *Ambient Assisted Living*. Ed. by J. Bravo, R. Hervás, V. Villarreal. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 65–72. ISBN: 978-3-642-21303-8. DOI: 10.1007/978-3-642-21303-8_9. URL: http://link.springer.com/10.1007/978-3-642-21303-8_9 (cit. on p. 27).

[Hel11] M. Helmert. “The Fast Downward Planning System”. In: *Journal of Artificial Intelligence Research* 26 (2011), pp. 191–246. DOI: 10.1613/jair.1705. arXiv: 1109.6051. URL: http://arxiv.org/abs/1109.6051http://dx.doi.org/10.1613/jair.1705 (cit. on pp. 46, 49, 50).

[HG13] M. Hoekstra, I. Georgievski. “Web-based interface for domain manipulation in Smart Offices”. Bachelor. University of Groningen, 2013. URL: https://ilche.io/wp-content/uploads/theses/hoekstra.pdf (cit. on p. 26).

[HLMM19] P. Haslum, N. Lipovetzky, D. Magazzeni, C. Muise. “An Introduction to the Planning Domain Definition Language”. In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 13.2 (2019), pp. 1–187. ISSN: 1939-4608. DOI: 10.2200/S00900ED2V01Y201902AIM042. URL: https://www.morganclaypool.com/doi/10.2200/S00900ED2V01Y201902AIM042 (cit. on pp. 24, 29).

[HN01] J. Hoffmann, B. Nebel. “The FF Planning System: Fast Plan Generation Through Heuristic Search”. In: *J. Artif. Intell. Res.* 14.27 (2001), pp. 253–302. ISSN: 1076-9757. DOI: 10.1613/jair.855. URL: https://www.jair.org/index.php/jair/article/view/10276 (cit. on pp. 43, 46, 49, 50, 95).

[HPK12] D. Hoang, H.-y. Paik, C. Kim. “Service-oriented middleware architectures for Cyber-Physical Systems”. In: *Int. J. Comput. Sci. Netw. Secur* 12 (Jan. 2012). URL: https://books.nu.edu.sa/uploads/Service-Oriented%20Middleware%20Architectures%20for%20Cyber-Physical%20Systems.pdf (cit. on p. 43).

[HW04] G. Hohpe, B. Woolf. *Enterprise integration patterns: Designing, building, and deploying messaging solutions*. Addison-Wesley Professional, 2004 (cit. on pp. 33, 34, 51, 54, 56–61, 72, 79).

[Ins12] Institute of Artificial Intelligence – University of Ulm. *PANDA Explanation*. 2012. URL: https://www.uni-ulm.de/en/in/ki/research/software/panda/panda-explanation/ (visited on 09/09/2020) (cit. on p. 50).

[Iso] *Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts*. Standard. Geneva, CH: International Organization for Standardization, Mar. 2018 (cit. on p. 65).

[Kay03] D. Kaye. *Loosely Coupled: The Missing Pieces of Web Services*. RDS Press, 2003. ISBN: 1881378241 (cit. on p. 33).

[KG12] T Karthikeyan, J Geetha. “A Study and Critical Survey on Service Reusability Metrics”. In: *International Journal of Information Technology and Computer Science (IJITCS)* 4.5 (2012), pp. 25–31 (cit. on p. 67).

[KLA13] E. Kaldeli, A. Lazovik, M. Aiello. “Domain-independent planning for services in uncertain and dynamic environments”. Proefschrift. University of Groningen, 2013. URL: https://www.rug.nl/research/portal/files/14430948/thesis.pdf (cit. on p. 39).

[KLA16] E. Kaldeli, A. Lazovik, M. Aiello. “Domain-independent planning for services in uncertain and dynamic environments”. In: *Artificial Intelligence* 236 (2016), pp. 30–64. ISSN: 00043702. DOI: 10.1016/j.artint.2016.03.002. URL: https://linkinghub.elsevier.com/retrieve/pii/S000437021630025X (cit. on pp. 39, 40, 46, 48, 49).

[KS99] H. Kautz, B. Selman. “Unifying SAT-based and graph-based planning”. In: *IJCAI*. Vol. 99. 1999, pp. 318–325 (cit. on p. 51).

[Lav19] E. Lavieri. *Hands-On Design Patterns with Java*. Birmingham: Packt Publishing, 2019. ISBN: 9781789809770 (cit. on p. 89).

[LB95] D. Leonard-Barton. “Wellspring of knowledge”. In: *Harvard Business School Press, Boston, MA* (1995) (cit. on pp. 51, 52).

[Ley18] F. Leymann. *Lecture - Loose Coupling*. Stuttgart, 2018 (cit. on pp. 33, 34, 56–60, 73, 75).

[Lot17] D. Lotinac. “Novel approaches for generalized planning”. PHD. Universitata Pompeu Fabra Barcelona, 2017 (cit. on pp. 26, 27).

[LRMG14] N. Lipovetzky, M. Ramirez, C. Muise, H. Geffner. “Width and inference based planners: Siw, bfs (f), and probe”. In: *Proceedings of the 8th International Planning Competition (IPC-2014)* (2014) (cit. on p. 43).

[McC02] L. McCluskey. “Knowledge engineering: Issues for the AI planning community”. In: *Artif. Intell. Plan. Sched.* (2002) (cit. on p. 26).

[MPMM17] M. C. Magnaguagno, R. F. Pereira, M. D. Móre, F. Meneguzzi. “WEB PLANNER: A Tool to Develop Classical Planning Domains and Visualize Heuristic State-Space Search”. In: *Proceedings of the Workshop on User Interfaces and Scheduling and Planning, UISP*. 2017, pp. 32–38 (cit. on p. 105).

[Mui16] C. Muise. “Planning.Domains”. In: *The 26th International Conference on Automated Planning and Scheduling - Demonstrations* (2016), pp. 1–3 (cit. on pp. 19, 42, 43, 78).

[Mye99] K. L. Myers. “CPEF: A Continuous Planning and Execution Framework”. In: *AI Mag.* 20.4 (1999), p. 63. DOI: 10.1609/aimag.v20i4.1480. URL: https://www.aaai.org/ojs/index.php/aimagazine/article/view/1480 (cit. on pp. 19, 35, 36, 46, 48).

[NAI+03] D. S. Nau, T. C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, F. Yaman. “SHOP2: An HTN Planning System”. In: *J. Artif. Intell. Res.* 20 (2003), pp. 379–404. ISSN: 1076-9757. DOI: 10.1613/jair.1141. URL: https://jair.org/index.php/jair/article/view/10362 (cit. on pp. 46, 49).

[NAI+05] D. Nau, T.-c. Au, O. Ilghami, U. Kuter, D. Wu, F. Yaman, J. W. Murdock. “Applications of SHOP and SHOP2”. In: *IEEE Intell. Syst.* 5.1541-1672 (2005), pp. 34–41 (cit. on p. 30).

[NCLMA99] D. Nau, Y. Cao, A. Lotem, H. Munoz-Avila. “SHOP: Simple Hierarchical Ordered Planner”. In: *Proc. 16th Int. Jt. Conf. Artif. Intell. - Vol. 2*. IJCAI’99. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1999, pp. 968–973 (cit. on pp. 30, 46).

[Neb20] B. Nebel. *The 2020 IPC for Hierarchical Planning*. 2020. URL: http://gki.informatik.uni-freiburg.de/competition/ (visited on 11/01/2020) (cit. on p. 78).

[Nie05] J. Nielsen. *Ten usability heuristics*. 2005 (cit. on pp. 65, 66).

[NM19] B. Nebel, R. Mattmüller. *Principles of AI Planning*. Freiburg, 2019. URL: http://gki.informatik.uni-freiburg.de/teaching/ws1920/aip/lecture.html (cit. on pp. 21–24).

[Nor88] D. A. Norman. *The psychology of everyday things*. New York, NY, US, 1988 (cit. on p. 48).

[OGOB13] J. Ortiz, A. García-Olaya, D. Borrajo. "Using Activity Recognition for Building Planning Action Models". In: *International Journal of Distributed Sensor Networks* 9.6 (2013), p. 942347. ISSN: 1550-1477. DOI: 10.1155/2013/942347. URL: http://journals.sagepub.com/doi/10.1155/2013/942347 (cit. on p. 27).

[OLK11] S. H. Oh, H. J. La, S. D. Kim. "A Reusability Evaluation Suite for Cloud Services". In: *2011 IEEE 8th International Conference on e-Business Engineering*. IEEE. IEEE, 2011, pp. 111–118. ISBN: 978-1-4577-1404-7. DOI: 10.1109/ICEBE.2011.27. URL: http://ieeexplore.ieee.org/document/6104606/ (cit. on p. 70).

[PAMM20] A. Pavlenko, N. Askarbekuly, S. Megha, M. Mazzara. "Micro-frontends: Application of microservices to web front-ends". In: *Journal of Internet Services and Information Security* 10.2 (2020), pp. 49–66. ISSN: 21822077. DOI: 10.22667/JISIS.2020.05.31.049 (cit. on p. 127).

[Pel16] D. Pellier. *pddl4j: PDDL4J V3.0.0*. Version v3.0.0. Feb. 2016. DOI: 10.5281/zenodo.45971. URL: https://doi.org/10.5281/zenodo.45971 (cit. on p. 48).

[PF18] D. Pellier, H. Fiorino. "PDDL4J: a planning domain description library for java". In: *Journal of Experimental and Theoretical Artificial Intelligence* 30.1 (2018), pp. 143–176. ISSN: 13623079. DOI: 10.1080/0952813X.2017.1409278 (cit. on pp. 29, 49, 85, 87, 90, 96, 98).

[PG03] M. P. Papazoglou, D. Georgakopoulos. "Introduction: Service-oriented computing". In: *Commun. ACM* 46.10 (2003), p. 24. ISSN: 00010782. DOI: 10.1145/944217.944233. URL: http://portal.acm.org/citation.cfm?doid=944217.944233 (cit. on pp. 17, 19).

[PH90] C. H. Prahalad, G Hamel. "G.(1990).-"The Core Competence of the Corporation"". In: *Harvard Business Review* 68.3 (1990), pp. 295–336 (cit. on p. 52).

[PTB+18] F. Pommerening, Á. Torralba, T. Balyo, B. Say, A. Coles, A. Coles, M. Martinez, P. Sidiropoulos, T. Keller, S. Sanner. *The International Planning Competition 2018*. 2018. URL: https://ipc2018.bitbucket.io (cit. on p. 78).

[PTDL07] M. P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann. "Service-Oriented Computing: State of the Art and Research Challenges". In: *Computer (Long. Beach. Calif).* 40.11 (2007), pp. 38–45. ISSN: 0018-9162. DOI: 10.1109/MC.2007.400. URL: http://ieeexplore.ieee.org/document/4385255/ (cit. on p. 33).

[Red17] K. S. P. Reddy. *Beginning Spring Boot 2: Applications and Microservices with the Spring Framework*. Apress, 2017 (cit. on p. 82).

[RH01] J. Rintanen, J. Hoffmann. "An overview of recent algorithms for AI planning". In: *KI* 15.2 (2001), pp. 5–11 (cit. on pp. 23, 46, 48–50).

[RLM15] M. Ramirez, N. Lipovetzky, C. Muise. *Lightweight Automated Planning ToolKiT*. 2015. URL: http://lapkt.orghttps://hub.docker.com/r/lapkt/lapkt-public/ (visited on 04/24/2020) (cit. on p. 43).

[RN10] S. Russell, P. Norvig. *Artificial Intelligence - A Modern Approach*. Third Edit. New Jersey: Pearson Education, 2010, pp. 645–692. ISBN: 0-13-604259-7 (cit. on pp. 23–25).

[RPFP17] A. Ramoul, D. Pellier, H. Fiorino, S. Pesty. “Grounding of HTN Planning Domain”. In: *International Journal on Artificial Intelligence Tools* 26 (2017), p. 1760021. DOI: 10.1142/S0218213017600211 (cit. on p. 46).

[SAKS10] P. S. Sandhu, Aashima, P Kakkar, S Sharma. “A survey on Software Reusability”. In: *2010 International Conference on Mechanical and Electrical Technology*. 2010, pp. 769–773 (cit. on p. 68).

[San10] S. Sanner. “Relational Dynamic Influence Diagram Language (RDDL): Language Description”. 2010. URL: http://users.cecs.anu.edu.au/~ssanner/IPPC_2011/RDDL.pdf (cit. on pp. 25, 32).

[SB17] S. Samuel, S. Bocutiu. *Programming kotlin*. Packt Publishing Ltd, 2017 (cit. on p. 82).

[SFC08] D. E. Smith, J. Frank, W. Cushing. “The ANML language”. In: *The ICAPS-08 Workshop on Knowledge Engineering for Planning and Scheduling (KEPS)*. 2008 (cit. on p. 31).

[SMSB12] B. Seegebarth, F. Müller, B. Schattenberg, S. Biundo. “Making hybrid plans more clear to human users - A formal approach for generating sound explanations”. In: *ICAPS 2012 - Proceedings of the 22nd International Conference on Automated Planning and Scheduling* (2012), pp. 225–233 (cit. on pp. 50, 51).

[SPC+16] B. Shneiderman, C. Plaisant, M. Cohen, S. Jacobs, N. Elmqvist, N. Diakopoulos. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 6th. Pearson, 2016. ISBN: 013438038X (cit. on p. 65).

[Suc15] L. E. Sucar. “Probabilistic graphical models”. In: *Advances in Computer Vision and Pattern Recognition. London: Springer London. doi* 10 (2015), pp. 971–978 (cit. on p. 23).

[TDK94] A. Tate, B. Drabble, R. Kirby. “O-Plan2: an Open Architecture for Command, Planning and Control”. In: *Intelligent Scheduling*. Morgan Kaufmann, 1994, pp. 213–239 (cit. on p. 46).

[VCG+14] M. Vallati, L. Chrpa, M. Grzes, T. McCluskey, M. Roberts, S. Sanner. *International Planning Competition: Progress and Trends*. 2014. URL: https://helios.hud.ac.uk/scommv/IPC-14/index.html (visited on 11/01/2020) (cit. on p. 78).

[VK20] M. Vallati, D. Kitchin. *Knowledge Engineering Tools and Techniques for AI Planning*. Ed. by M. Vallati, D. Kitchin. Cham: Springer International Publishing, 2020. ISBN: 978-3-030-38560-6. DOI: 10.1007/978-3-030-38561-3. URL: http://link.springer.com/10.1007/978-3-030-38561-3 (cit. on pp. 42, 43).

[VTC+12] T. Vaquero, R. Tonaco, G. Costa, F. Tonidandel, J. Silva, C. Beck. “itSIMPLE4.0: Enhancing the Modeling Experience of Planning Problems”. In: Jan. 2012, pp. 11–14 (cit. on p. 47).

[WCM14] G. Wickler, L. Chrpa, T. L. McCluskey. “KEWI: A knowledge engineering tool for modelling AI planning tasks”. In: *KEOD 2014 - Proceedings of the International Conference on Knowledge Engineering and Ontology Development* (2014), pp. 36–47. DOI: 10.5220/0005034400360047 (cit. on p. 47).

[Wil00] D. E. Wilkins. “USING THE SIPE-2 PLANNING SYSTEM by”. In: *SRI International* 6.1 (2000) (cit. on pp. 46, 49).

[Wil90] D. E. Wilkins. “Can AI planners solve practical problems?” In: *Comput. Intell.* 6 (1990), pp. 232–246 (cit. on p. 36).

[YL04] H. L. S. Younes, M. L. Littman. “PPDDL1. 0: An extension to PDDL for expressing planning domains with probabilistic effects”. In: (2004) (cit. on p. 32).

All links were last followed on December 9, 2020.

A Code Snippets

A.1 Build related files

A.1.1 Building scripts

Listing A.1 Back-end building script (*build.sh*).

```
#!/bin/bash
# build without tests "-x test"
./gradlew -Pbuildprofile=COMPOSE -x test clean build --info

# build docker image from resulting jar
docker build -t uir.prototype.endpoint.solving .
```

Listing A.2 Front-end building script (*build.sh*).

```
#!/bin/bash
# build docker image from resulting /dist
docker build -t uir.prototype.endpoint.modelling .
```

A.1.2 Dockerfiles

Listing A.3 Back-end example Dockerfile for the solving service.

```
FROM openjdk:14
ENV TZ=Europe/Berlin
ENV STAGE=COMPOSE

RUN groupadd -g 9999 uiruser && \
    useradd -r -u 9999 -g uiruser uiruser
RUN mkdir /data
RUN chown uiruser:uiruser /data
USER uiruser

# paste the jar to the container
COPY ./build/libs/solving-cap-0.0.1.jar /

ENTRYPOINT ["java", "-jar", "-Dspring.profiles.active=${STAGE}", "solving-cap-0.0.1.jar"]
```

Listing A.4 Front-end example Dockerfile for the modelling service.

```
FROM node:lts-alpine AS builder
WORKDIR /app
COPY . .

# --no-cache: download package index on-the-fly, no need to cleanup afterwards
# --virtual: bundle packages, remove whole bundle at once, when done
RUN apk --no-cache --virtual build-dependencies add \
    python \
    make \
    g++ \
    && npm install \
    && apk del build-dependencies

RUN npm install && \
    npm rebuild node-sass && \
    node --max_old_space_size=4096 node_modules/@angular/cli/bin/ng build --prod

FROM nginx:alpine
ENV TZ=Europe/Berlin
RUN addgroup -g 9999 -S angularuser && \
    adduser -u 9999 -S angularuser -G angularuser

RUN touch /var/run/nginx.pid && \
    chown -R angularuser /usr/share/nginx /var/cache/nginx /var/run/nginx.pid

USER angularuser

COPY --from=builder /app/dist/apps/uir-frontend/ /usr/share/nginx/html/
COPY --from=builder /app/nginx.conf /etc/nginx/conf.d/default.conf

EXPOSE 4200/tcp
```

A.1.3 Docker-compose

Listing A.5 *docker-compose.yml* system definition.

```yaml
version: "3.8"
services:
  rabbitmq:
    image: rabbitmq:management-alpine
    container_name: rabbitmq
    volumes:
      - ./container_data/rabbitmq/etc/:/etc/rabbitmq/
      - ./container_data/rabbitmq/data/:/var/lib/rabbitmq/
      - ./container_data/rabbitmq/logs/:/var/log/rabbitmq/
    environment:
      RABBITMQ_ERLANG_COOKIE: ${RABBITMQ_ERLANG_COOKIE}
      RABBITMQ_DEFAULT_USER: ${RABBITMQ_DEFAULT_USER}
      RABBITMQ_DEFAULT_PASS: ${RABBITMQ_DEFAULT_PASS}
    ports:
      - 5672:5672 # messaging port
      - 15672:15672 # for the admin ui of rabbitMQ
    networks:
      - uir-planner

  # generic capability example, replace <...>
  backend-cap:
    image: uir.prototype.<class>.<cap-name>
    container_name: managing-service
    volumes:
      - ./container_data/<cap-name>-service/logs/:/logs/
    # expose ports if a REST controller is neccessary
    ports:
      - "8090:8090"
    depends_on:
      - rabbitmq
    networks:
      - uir-planner

  modelling-ui:
    image: uir.prototype.endpoint.modelling
    container_name: modelling-service
    depends_on:
      - managing-cap
    ports:
      - "4200:4200"
    networks:
      - uir-planner

networks:
  uir-planner:
```

B Conversations

B.1 Dr. Pascal Bercher <pascal.bercher@anu.edu.au>

B.1.1 Response from 15. November 2020

Dear Mr. Graef,

Please note that I've added (Dr.) Gregor Behnke and Daniel Hoeller, as they are the main developers of the current version of the system. By the way: We are all German, so if you are too, please feel to write in German. Furthermore, we are all happy with a first-name basis, so don't feel bad approaching any of us on a first-name basis if you prefer.

Regarding your email:
Based on what you wrote I am assuming that you refer to the PANDA system that's available for download on the webpages from the AI Institute of Ulm University. Note, however, that meanwhile we don't only have newer *planners*, but also a newer parser and grounder.

Some of the software if available on githib already, e.g. the parser and grounder:
`https://github.com/panda-planner-dev/pandaPIparser`
`https://github.com/panda-planner-dev/pandaPIgrounder`

I think it would make sense to check out this software and then write us again if you have further questsions?

You also find an up-to-date description of all our planner components on the following webpage:
`https://panda-planner-dev.github.io/`

Also note that for the current IPC on HTN planning (ipc2020.hierarchical-task.net) we have provided various translaters from HDDL to other languages such as the one by SHOP (in case that would be interesting for you).

If your questions are still relevant (based on whether you consider switching to the newer parser), please let us know so that Gregor and Daniel can respond to you on a more technical level.

Best regards, Pascal

B.2 Dr. Gregor Behnke <behnkeg@informatik.uni-freiburg.de>

B.2.1 Response from 16. November 2020

The original message in german language:

> Hallo Sebastian,
>
> das XML-Format ist hochgradig depricated. Dazu etwas Geschichte: Die PANDA Planer wurde an der Uni Ulm entwickelt. Alle Domänen, die es damals gab, lagen in einem XML-Format vor. Dieses Format war allerdings etwas schwer zu interpretieren und (sagen wir mal) kryptisch. Wir haben dieses Format ca. 2015/2016 gegen HDDL ersetzt. Es gab aus Legacy-Gründen noch einen XML Parser in PANDA3 und auch einen Writer, damit wir im Falle von Bugs testen konnte was der älte Planerßu bestimmten Instanzen sagt. Dieser Writer ist aber seit etwa 2016 nicht mehr gewartet oder benutzt wurden – eigentlich sollte man ihn entfernen. Da wir sowieso einen neuen Planer geschrieben haben, ist das aber nie passiert. Kurz gesagt: Ich wundere mich nicht, dass das geschriebene XML sich nicht wieder einlesen lässt. Ich weiß ehrlich gesagt nicht einmal, ob das XML-Format ausdrucksmächtig genug ist um die IPC Domänen darzustellen. Wenn ich mich recht erinnere erlaubt es z.B. keine Methodenvorbedingungen, die quasi jede Domäne in der IPC hat.
>
> Für mein Verständnis: Du willst die geparste Domäne serialisieren, damit du bei einem erneuten Aufruf mit einem anderen Problem die Domäne nicht erneut parsen musst?
>
> Gregor

B.2.2 Response from 17. November 2020

The original message in german language:

> Ah ok. Die Frage, die ich mir dabei stellen würde (und was ich ehrlich gesagt nicht genau weiß) ist wie stark das Einlesen der Domäne in die ïnternen Datenstrukturenïntegriert ist. D.h. das Einlesen kann manchmal nicht nur ein Objekt in der Umgebung erzeugen, sondern quasi beliebige Seiteneffekte haben. Das ist insbesondere in C/C++ gar nicht so ungewöhnlich, auch wenn es kein so gutes Software-Engineering ist. Z.B. erzeugt der pandaPIparser kein einzelnes Objekt, sondern füllt eine Reihe von globalen Datenstrukturen.
>
> Der panda3core Parser wurde aber explizit so entwickelt, dass er keine Seiteneffekte hat, d.h. die geparsten Datenstrukturen sollten zu 100% die Domäne widerspiegeln. Diese Datenstrukturen sind alle in Scala geschrieben. Dafür gibt es einige sehr komfortable json-Bibliotheken, die ohne viel zusätzlichen Aufwand die Konvertierung in JSON und das Einlesen von JSON erlauben. Ich habe für eine Anwendung einmal Argonaut (http://argonaut.io/) verwendet und fand es sehr gut. Den Code kann ich die leider nicht weitergeben, da es ein Industrieprojekt war (wir haben nicht die Domäne, sondern die Pläne serialisiert).

Zu PDDL4J: Der Planer hat in seiner partial-order Konfiguration in einigen Fällen falsche Pläne gefunden. Im Total-Order Benchmark ist das nur einmal vorgekommen (Barman, daher ist diese Domäne auch nur dort DSQ). Für den Partial-Order Track ist das in einigen Domänen vorgekommen, daher wurde er vollständig disqualifiziert. Das Problem scheint zu sein, dass sich der Planer ab und an nicht an die in den Dekompositionsmethoden vorgegebenen Ordnungen hält.

Beste Grüße, Gregor

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

Leonberg, 11.12.2020, S. Graef

place, date, signature