Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Enhancement and Assessment of the PlanX Toolbox for Constructing AI Planning Systems

Manuel Stamm

Course of Study:	Informatik
Examiner:	Dr. Ilche Georgievski
Supervisor:	Dr. Ilche Georgievski
Commenced:	April 4, 2023
Completed:	October 4, 2023

Abstract

The PlanX toolbox provides different planning functionalities to compose an advanced planning system for solving planning problems. The research field about planning systems is lacking in flexibility, portability and more to provide users an optimal experience, without the need of software engineering knowledge or AI planning knowledge. The toolbox provides services with good interoperability and the option to change the workflow of the created planning system as desired. One aspect of this work is to demonstrate the usability of the PlanX toolbox. For this reason, we conduct a case study with a simplified real-world example, as well as a user study. The usability is evaluated under different aspects and with different user feedback. The usability is all in all good with room for further improvement like better documentation. Furthermore, the architecture is an important part of this work. The modularity and lose coupling is lacking in the current toolbox. We propose a new service to refine the architecture. The new system monitoring service is a modular service with an encapsulated functionality, which is split up from an existing service. This results in a better overall architecture that can be further optimised in the future.

Kurzfassung

Die PlanX-Toolbox bietet verschiedene Planungsfunktionalitäten, um ein fortschrittliches Planungssystem zusammenzustellen zum Lösen von Planungsproblemen. Im Forschungsbereich der Planungssysteme mangelt es an Flexibilität, Portabilität und mehr, um den Nutzern ein optimales Erlebnis zu bieten, ohne dass Software-Engeneering Kenntnisse oder KI-Planungs Kenntnisse nötig sind. Die Toolbox bietet verschiedene Dienste mit guter Interoperabilität und die Möglichkeit, den Arbeitsablauf des erstellten Planungssystems nach Belieben zu ändern. Ein Aspekt dieser Arbeit ist es, die Benutzerfreundlichkeit der PlanX-Toolbox zu demonstrieren. Aus diesem Grund führen wir ein Fallstudie mit einem vereinfachten realen Beispiel sowie eine Nutzerstudie durch. Die Benutzerfreundlichkeit wird unter verschiedenen Aspekten und mit verschiedenen Nutzer Rückmeldungen evaluiert. Die Benutzerfreundlichkeit ist im Großen und Ganzen gut mit Raum für weitere Verbesserungen wie bessere Dokumentation. Darüber hinaus ist die Architektur ein wichtiger Bestandteil dieser Arbeit. Es mangelt an Modularität und loser Kopplung in der aktuellen Toolbox. Wir schlagen einen neuen Dienst vor um die Architektur zu verfeinern. Der neue Systemüberwachungsdienst ist ein modularer Dienst mit gekapselter Funktionalität, der aus einem bestehenden Dienst abgespalten wird. Dies führt zu einer besseren Gesamtarchitektur, die in Zukunft weiter optimiert werden kann.

Contents

List of Figures

List of Listings

Acronyms

AI Artificial Intelligence. 13

API Application Programming Interface. 18

HTN Hierarchical Task Network. 22

MOM Message-Oriented Middleware. 27

PANDA Planning and Acting in a Network Decomposition Architecture. 13

PDDL Planning Domain Definition Language. 15

PDDL4J Planning Domain Description Library for Java. 23

SOA Service-oriented Architecture. 19

SOA-PE Service-Oriented Architecture for Planning and Execution. 13

SOC Service-oriented Computing. 18

STRIPS Stanford Research Institute Problem Solver. 15

WSL Windows Subsystem for Linux. 41

1 Introduction

Artificial Intelligence (AI) planning solves planning problems that consist of a domain model, which provides actions to change the states of the world, an initial state, which describes the starting point of the world, and a goal state, which describes the objective of the user. The solution to such a planning problem is a course of actions that reach the goal state when starting in the initial state. An AI planning system originates from the need to use its powerful reasoning capabilities for real-world planning problems. A planning system contains not only a solving functionality to solve the planning problem, but also other functionalities such as modelling, visualisation, validation of the solution and much more [GB21]. Real-world examples for the use of AI planning in different domains are: robotic domains [KM20], space mission domains [FPD13], autonomous driving domains [AGPA22] and many more.

There are many standalone functionalities to solve planning problems. But to address real-world problems, there is a need for different functionalities and not just solving. Planning functionalities are usually not compatible with other functionalities which makes it very hard to compose a planning system. There are already different approaches for planning systems. The Planning and Acting in a Network Decomposition Architecture (PANDA) framework [HBBB21] is an advanced AI planning system with similar aspects to our propose. The similarity is that it provides different functionalities like solving, verification, repairing and more. Thus, they implemented a range of functionalities into one system with a common input language. But they do not specify how the PANDA framework can be integrated in other system. There exist other planning systems which uses a service-oriented architecture. The service-oriented architecture provides many benefits and is used for our project. An example from a related work is the Service-Oriented Architecture for Planning and Execution (SOA-PE) [FMJB15]. It is a planning system that uses the service-oriented principles and patterns for the architecture. Moreover, it even provides different services as domain-management-service, planning-service, execution-service and more.

The different existing planning system and planning functionalities all lack in some way as we explain in the following. As stated, the interoperability between the planning functionalities in this research field is lacking. Furthermore, other planning system approaches are not flexible. They usually have a set amount of services that are difficult or impossible to extend. In addition, the workflow for using these systems is already set, so there are no flexible options to meet the user's needs.

Thus, we present the PlanX toolbox, which provides different front-end and back-end services to build flexible, advanced and deployable planning systems. It can be used create standalone planning systems, with the option to add services even in runtime, or it can be used to be integrated into other systems. The modular services possess easily accessible interfaces as well as a common in- and output patterns, thus the communication between the services are no problem at all. The architectural pattern allows a good horizontal scalability and provides the option to extend the

toolbox with more services. The RabbitMQ message broker connects all back-end services and is the reason for the good scalability. Composing such a planning system should be an easy task without any needed knowledge about software engineering or AI planning.

In this work we take a closer look at the the PlanX toolbox. We provide an overview about the services and then talk about the architecture. In focus is the design of the toolbox and the current issues. The front-end services lack in modularity and lose coupling. Currently, the front-end services are all provided as one service. The first research question derived from this is: How can we improve the design of PlanX? Apart from the architecture, the toolbox has no demonstration on a case study and also no usability evaluation as well. This leads to the second research question: How can we demonstrate the usability of PlanX?

The methodology used to work on these research questions are the following. We talk about the design and implementation of the new service that is proposed as a solution. We then use the toolbox in a case study to showcase its usability and conduct a user study to get more opinions on the usability as well as opinions for further improvement.

The work is structured as follows. It starts with Chapter 2 background information to understand the basics. We then move on to Chapter 3 state of the art and show different approaches. We explain what the approaches are about and what they are missing in comparison to our PlanX toolbox. After that, we show in Chapter 4 the PlanX toolbox with every service, we explain the architecture and present the new service. The case study in Chapter 5 and evaluation in Chapter 6 consists of what we already stated above. In the evaluation we consider the new service as well. To finish it off, we conclude our work in Chapter 7 with a conclusion and an outlook.

2 Background Information

2.1 Classical planning

Classical planning usually based on the concept of a state model and is defined over a state space. It consists out of a single initial state, a set of actions, which map each state to another, and a non-empty set of goal states. If a set of actions connects the initial state to the goal state with transitions through different states, we call this sequence of actions a plan [Geo15].

Depending on the size and complexity of planning problem, the state space can grow exponentially. To prevent this from happening, each state is only allowed to have a set of values for variables with finite and discrete domains. As a result, the actions and transitions are defined by the terms of these variables. Most of the times, these states are based on propositional variables, which are for example facts or atoms. Propositional variables have a domain of two values, which are true and false. This is called the Stanford Research Institute Problem Solver (STRIPS) representation and the related problem STRIPS planning problem [Geo15].

STRIPS planning problems consists of a set of propositional variables, a set of operators, initial states and goal states. A state consists of facts, which are propositional variables that have the value true. If a fact is not utilized to describe a state, the default value is false. This is called the close-world assumption. An action is an instance of an operator, which consists of preconditions and effects. The preconditions have to be fulfilled to use an action, and the effect describes the positive and negative effects on the propositional variables of a state [Geo15].

2.2 Planning Domain Definition Language

Planning Domain Definition Language (PDDL) extends STRIPS to first-order logic with a finite sets of constraints, variables and predicates. It intends to describe the „physics“ of a domain, that is, the predicates, the effects of possible actions and the structure of compound actions. At its core PDDL is an action-centred language using the STRIPS formulations of planning problems. This implies the semantics of actions, utilising the pre- and post-conditions to describe the effects of an actions [AHK+98].

A planning problem is created by dividing it into a paring of a domain description and a problem description. The domain description is given in a domain file and the problem description is given in a problem file. One domain description can be used for many different problem descriptions to determine plans for different planning problems in the same domain. The following example shows a domain file and a problem file. We shortly explain the domain description from Listing 2.1. The name of the domain is „vehicle“. The keyword „requirements“ stands for the required features. In this case, the domain requires the usage of STRIPS and typification of entities. There are three

Listing 2.1 Domain description in PDDL[FL03].

```
(define (domain vehicle)
  (:requirements :strips :typing)
  (:types vehicle location fuel-level)
  (:predicates (at ?v - vehicle ?p - location)
               (fuel ?v - vehicle ?f - fuel-level)
               (accessible ?v - vehicle ?p1 ?p2 - location)
               (next ?f1 ?f2 - fuel-level))
(:action drive
 :parameters (?v - vehicle ?from ?to - location
               ?fbefore ?fafter - fuel-level)
 :precondition (and (at ?v ?from)
                    (accessible ?v ?from ?to)
                    (fuel ?v ?fbefore)
                    (next ?fbefore ?fafter))
:effect (and (not (at ?v ?from))
             (at ?v ?to)
             (not (fuel ?v ?fbefore))
             (fuel ?v ?fafter))
)
```

different type of entities: „vehicles“, „location“ and „fuel-level’. The „predicates’ part defines four different properties for the objects in this domain. With the help of predicates, we can describe the state of the world within the problem file. After that, one action is defined, whereby positive and negative effects are distinguished with a „not“ for negative effects. Now onto the problem instance we see in Listing 2.2. The name of this instance is „vehicle-example“. In the next line it refers to the corresponding domain. The next parts defines the specific objects for this problem, sorted by the type. The „init“ section describes the initial condition of the domain world, using the predicates from the domain. The last section defines the wanted the goal state for a possible plan. Now everything is defined and set up to use a solver to compute a plan if one exists.

2.3 AI planning systems

2.3.1 AI planning tools

AI planning tools handle automatically the selection and organisation of actions to achieve the user’s objective. There are several planning tools to not only solve planning problems but also to address other important aspect, for instance modeling problems or validating found plans. The problem lies in the non-existence of a tool which can handle all these aspects that are needed in real-world environments. This results in the need to manually combine different, heterogeneous tools. These tools differ in several aspects such as software, design and configuration requirements. Apart from the fact that this is time-consuming and error-prone, it also demands advanced planning and technical expertise. Another problem lies within the fact that existing planning tools are not

Listing 2.2 Problem instance in PDDL[FL03].

```
(define (problem vehicle-example)
 (:domain vehicle)
 (:objects
    truck car - vehicle
    full half empty - fuel-level
    Paris Berlin Rome Madrid - location)
 (:init
    (at truck Rome)
    (at car Paris)
    (fuel truck half)
    (fuel car full)
    (next full half)
    (next half empty)
    (accessible car Paris Berlin)
    (accessible car Berlin Rome)
    (accessible car Rome Madrid)
    (acessible truck Rome Paris)
    (accessible truck Rome Berlin)
    (accessible truck Berlin Paris)
)
(:goal (and (at truck Paris)
            (at car Rome))
)
```

designed to be compatible with other tools. The composition and integration are realised in a way that results in significant differences in data models and technologies, which thus leads to a considerable limitations for the interaction of planning tools [GB21].

Planning tools are capable of a wide range of various functionalities. Some examples are converting, managing, modelling, monitoring, parsing, solving, validating and visualising. We explain each of these shortly in the following enumeration [GG21].

- converting: performs a transformation of planning data from one format to another
- managing: is used as a router or system handler
- modelling: represents an interface between the planning system and the user
- monitoring: observes the status of the execution of plan actions and look out for contingencies
- parsing: creates programming-level objects from models that are specified in the planning language
- solving: creates a plan to achieve the goal
- validating: examines the plan for errors
- visualising: is a front-end functionality to display charts, tables and other statistics

2.3.2 Engineering a planning system

There are various challenges to face when trying to create a planning system. Expertise knowledge of AI planning is important, for example, choosing the right underlying planning model. But proficiency in software engineering is also needed, for instance, to design a system without established interoperability mechanism for planning components [Geo23a].

There are several challenges to engineer a planning system identified by Georgievski [Geo23c] which are divided into 3 groups.

Architecture

As we have already mentioned in Section 2.3.1, the complexity of existing planning tools prevents them from being easily connected. It also makes the identification of functional boundaries of components difficult. Another aspect is the interoperability. There are planning tools that use a common planning syntax like PDDL. This is helpful for the knowledge-engineering part, but it is different for the software-engineering part. A common syntax does not necessarily lead to a possible communication between planning tools. The last architectural challenge is the reusing of planning tools. Based on the complexity and tight coupling of planning components, it is hard to separate between the generic and specific parts.

Development and deployment

Integrating planning tools into a single planning system is difficult because of two reasons. On the one hand, there are a lot of different underlying data models and algorithms. On the other hand, there is a lack of controlling interactions of planning tools via Application Programming Interface (API). As we see in Section 2.3.1, the process of composing planning tools is not easy. The last problem is the continuous deployment because of evolving planning tools. Taking versioning, performance measurement and raising warnings into account, this is a challenging task due to non-existing mechanism to represent planning software in a machine-readable form.

Process of using planning tools

The heterogeneity aspect in the area of planning tools makes the identification, selection and management of planning tools hard to handle. Apart from the already mentioned diversity of functionality, there are a lot of planning tools. However, most of these are just planners, which means they are tools that focus on solving planning problems. At last, another aspect are the different classes of planning. Classical planning (Section 2.1) is a convenient way to solve planning problems. But it has many restrictions for real-world problems, thus there are other classes of planning such as temporal planning and probabilistic planning.

2.4 Service-Oriented Computing

„Service-oriented Computing (SOC) is the computing paradigm that utilises services as fundamental elements for developing applications.“ [PG03]. Services are low-cost components for distributed applications, because they are modular and can work without a specific environment. Other positive factors besides modularity are interoperability, integrity and reusable software. „Interoperability is the ability of software components to communicate, work and exchange messages with each other

and other systems seamlessly, despite being developed differently [...].“ [Geo23c]. Services hide away their code and data integration needed for execution. It encapsulates a capability to perform a certain task [GG21]. Composed services can perform following functions [PG03]:

- Coordination: Controlling the data-flow between the services as well as the execution of them.
- Monitoring: Watching events or information produced by the composed service
- Conformance: Ensuring the integrity of the composed service by matching the parameter types with the one from the components.

2.4.1 Service-Oriented Architecture

Service-oriented Architecture (SOA) is one basic concept of SOC that defines how to make software components interoperable and reusable. This can be achieved through the use of design patterns and communication standards in a way that components can be integrated into new systems without deep integration. The building blocks of a SOA are services. The idea is to be able to recombine these services in various forms for the implementation of a system. It gives you flexibility in choosing the location for service providers and consumers as well as the implementation of technologies. All these benefits result particularly from a single feature, which is the stability of the interface service. This stability limits the cost of subsequent amendments. Another advantage is the possibility to reuse the services as they are. That prevents additional cost of re-implementation or cost of modifying the encapsulated functionality [VAMD09].

Another positive factor of using these architectural pattern is lose coupling. Loose coupling is the key for accessibility with little to no knowledge about the integration by intentionally sacrificing interface optimisations. The achieves extra interoperability between system with various technology, availability and more. Loose coupled software components can communicate to each other with messaging through the messaging system. The messaging system gets messages via messaging channels and distribute it to the appropriate components.

3 State of the Art

3.1 SOA for planning systems

The approach from Feljan et al. [FMJB15] proposes a SOA named SOA-PE for planning and execution in a decentralised, multi-agent system (large scale cyber-physical systems). Other than previous works, which are built for specific problem in specific domains, they want to provide generic frameworks for developing applications. The work WebMed from Hoang et al. [HPK12] has a similar approach „[...] but it lacks in planning and plan execution components we believe are necessary for autonomous systems.“ [FMJB15]. Nevertheless, SOA-PE architecture can be built on the communication and data aggregation basis of WebMed. There are more related approaches mentioned with similar efforts but they lack in the same aspects as WebMed or in the architecture.

The objective of SOA-PE is to provide different functionalities like domain modeling, planning, execution, monitoring and so forth. The management and life cycle of agents in the system involves all these functionalities in the following way. When there is an objective to reach, it is translated into goals and domain models. These are used to compute plans to achieve that goal. The execution is monitored to look out for any anomalies and failures. The new knowledge is returned for replanning or changes in the domain models.

The following real-world characteristics of a system should be payed attention to.

- The system consists of a large number of agents.
- Centralised control is often not a choice.
- Agents might be autonomous in nature, thus they should be able to use the functionalities to plan etc. on their own.

To address these characteristics, the service-oriented approach is chosen. The functionalities are hosted as services, so the agents can use these services to their needs. Each service can access the API of other services and create automated flows. It can also extend itself with hierarchical domains/problems to solve according planning problems. The advantage of this is using the properties of services (Section 2.4), which matches the tasks of the agents, to „orchestrating the functions according to the needs“ [FMJB15].

The service approach is similar to our proposed architecture. But we see here, that the workflow is predetermined. That is not the goal for our project. We want reusable services that users can utilise as needed. The services should be flexible in their communication with other services as needed to reach the user's goal. Our goal is to provide the users a toolbox with good usability and flexibility. The users should be able to freely decide the workflow according to their needs. Moreover, the number of services should not be predefined. We want the horizontal scalability to make planning systems with all various planning services according to the wishes of the user.

Another related work to our propose in terms of architecture is from Fratini et al. [FPD13] and is about „a service-oriented approach for the interoperability of space mission planning systems.“ [FPD13]. Planning problems in space domains tend to be very diverse, making the definition of a generic, reusable planning system a difficult task. They have decide to use a „black box“ planning system to tackle this problem. It consists of very general services that are detailed enough to be usable. The challenge is to find the right balance between assumption, that are made about the black box, and the standardisation level required to define generic planning services. Without knowing much about the implementation of the black box, it becomes difficult to specify a planning request and interpreting the result. Planning systems are depending on the planning technique (PDDL vs. Hierarchical Task Network (HTN)) and thus requiring different information sets. But without making fixed assumption for all compliant systems, it is hard to define one abstract information model. So one can either standardise the syntax or the semantics. With standardised syntax, the relevant semantics for the planning problem has to be part of the system and be already defined within the black box. This approach would need a set of assumptions that are often too optimistic, so a minimum amount of semantics would be needed. By standardising the semantics a common abstract planning meta model would be needed, so the planning system can solve the requests. The suggested middle point is an agreement on the syntax and semantic of a limited set of services provided by the system.

Here we can see a similar problem as in the architecture proposed above. First of all, the set of services is also limited with no intention to scale it further. The general approach is about optimising a planning system for this specific use case. As we will see in more detail later, that is not the goal of our toolbox. We want to provide the possibility to generate advanced planning systems for any kind of use case and optimally for any kind of user. Thus, we want to reach a better usability than they have.

3.2 PANDA

The PANDA framework focuses on solving HTN planning problems. Hierarchical planning provides an additional hierarchy on top of the task, describing how a task can be broken down into several more concrete ones. Höller et al. [HBBB21] introduce their own input language to model these kind of problem. The Hierarchical Domain Definition Language (HDDL) is an extension of PDDL and serves as the common input language for this planning system. Apart from the input language, PANDA provides other functionalities. The predefined workflow starts with the „Preprocessing“ functionality that grounds the input model (make the model variable-free). It continues with the solving process. The framework consists of several planning approaches and also several heuristics for the solving algorithms. The „Plan Explanation “ then elaborates, for example, why a particular action is included in the plan. This functionality includes different kind of explanations. The task of the „Plan and Goal Recognition“ functionality is inferring the goal(s) and plan of the user. When the execution of a plan fails there are two possibilities to continue. The first one is re-planning, which is not an option for HTN planning. Thus, the „Plan Repair “ modifies the old plan by compiling the original model together with it into a new planning problem. The last functionality is the „Plan Verification “, which checks the solution by using the action sequence to try to reach the goal.

This planning system is not a SOC approach. The authors state that it can be integrated into other systems, but it is not explained how. We can assume that software engineering skills are likely to be required. In contrast to this, the planning systems from our toolbox are easily integrable, because of the modular services with understandable interfaces to help system developers. Moreover, Höller et al. [HBBB21] do not seem to plan to extend the planning system, for example, with other input languages. The focus lies in solving HTN planning problems and not being flexible or have a good usability for different use cases. For instance, there is not functionality to monitor the process or nothing about a front-end interface. The PlanX toolbox provides more than this and is easily expandable to integrate new functionalities in the future.

3.3 PDDL4J

Planning Domain Description Library for Java (PDDL4J) is an open source library for developers meant to provide planning tools for the PDDL language. Another main point of the work from Pellier and Fiorino [PF18] is to facilitate research on new planners. PDDL4J consists of several independent modules. Firstly, it contains a full PDDL parser that supports PDDL 3.1. The parser ensures lexical, syntactical and semantical analysis of the domain and problem files. These files are transformed into a compact internal representation before analysis. Next, PDDL4J has a pre-processing module called the instantiation module. This module has several tasks and is defining the planner performances. Moreover, it also decides the informative structures and their corresponding heuristics. Lastly, there are several classical planners included in the library. There are different types of planners implemented. The first type is state-space planners. These kind of planners, like the Heuristic Search Planner (HSP), Fast Forward planner or Fast Downward planner, rely on search algorithms to search in a subset of the state space. Each node represents a state of the world and the transitions are actions, so finding a way from the initial state to the goal state is the plan. The both other types of planners are planning graph planners and propositional satisfiability planners.

The PDDL4J library focuses only on the planners. On its own, it does not provide a good usability for users. There are no options for users to model a planning problem or visual/verify the plan. We think the modularity is good in PDDL4J and we look forward to reach a better modularity as well. The planning tools itself have a good reusability and are used in the PlanX toolbox as well. This includes the planners as well as the parser.

3.4 Planning.Domains

The objective of Planning.Domains from Muise [Mui16] is to provide a set of resources, repositories and tools to discover and develop planning problems. Planning.Domains consists of three main components. The first is an API to all existing planning problems. Next, we have an open and extendable interface as a cloud service to solve planning problems. Finally, there is a fully featured editor for creating and modifying domain and problem files in PDDL.

The API represents a source for existing planning benchmarks. It provides programming access to a version controlled publicly accessible repository. The solver can be invoked by a RESTful interface. The PDDL file is sent as a raw text and the solver running as cloud service returns a plan. The

use of the planner is restricted to 10 seconds and 500Mb. The editor is available as a website and provides standard features, such as syntax highlighting, bracket matching and code folding. But there are custom features as well. Users can use custom deployed solvers and compute/display solutions during editing. There is also PDDL specific auto-completion and more.

This planning system is the closest to the PlanX toolbox. We notice that there are not many provided functionalities. Another negative usability aspect is the missing explanation on how to set domain and problem files. If we press on solve, we see the option to choose from the added files which the domain file is and which the problem file is. This was not clear in the beginning. We want to show that PlanX provides a better interface with at least the same functionality.

4 PlanX toolbox and its architecture

The core of this work resolves around the PlanX toolbox [Geo23b]. The idea of this approach is to offer a toolbox that enables researchers and developers to quickly build, integrate and deploy planning systems without having to worry about interoperability or other internal workings. The several integrated planning functionalities are usable and portable and can be chosen as needed. There is no predefined workflow and any needed combination of services can be chosen. So, the user of the toolbox is given a set of 8 services that can be used to build the planning system. As the architectural design uses modern SOC principles and patterns, it is expandable at any given moment as needed. We take a look at the architecture and identify the flaws with the current toolbox. After that, we present solution to refine the architecture.

4.1 PlanX overview

The PlanX toolbox (https://github.com/PlanX-Universe) can be divided into front-end and back-end services. The Modelling, Visualisation and System Monitoring Services are part of the front-end while the other are part of the back-end. Apart from the 8 planning services there is one messaging component, the Message-oriented middleware. Docker (https://www.docker.com/) is used to build the services to a planning system. We describe the included functionalities as seen in Figure 4.1 in the following using the information from Graef and Georgievski [GG21] [Geo23b].

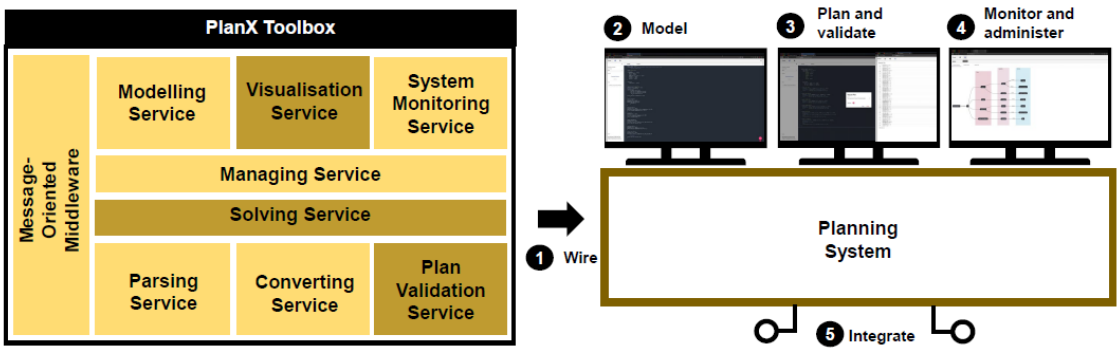

Figure 4.1: Overview of the PlanX toolbox [Geo23b]

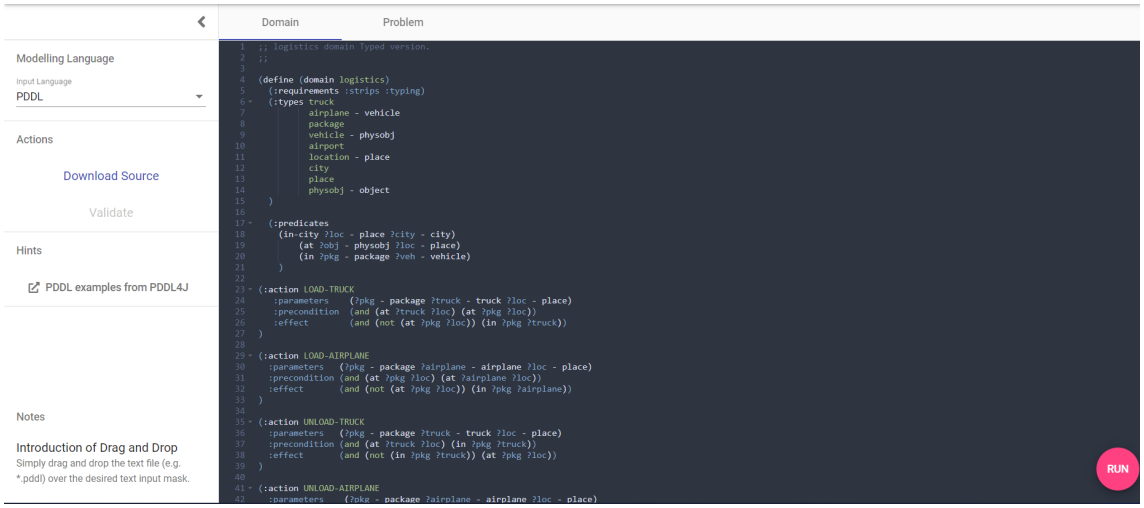

Figure 4.2: Web IDE of the modelling service

4.1.1 Modelling and Visualisation Service

These both services are tightly coupled and appear as a single service in the toolbox. The modelling service consists of a web IDE to work with. As seen in Figure 4.2 there are different adjustment options. First we can change the input language which is at the moment only PDDL. Furthermore there a two windows named 'Domain' and 'Problem' where the user can input the domain model and an appropriate problem instance. The editor [MFMM17] provides syntax highlighting and folding. One last fact worth mentioning is the existence of a sample to guide the user.

The visualisation service has several tasks after the user filled the editor with information. After the user presses the 'RUN' button there are several options to choose a solving algorithm from the solving service (Section 4.1.5). After choosing one, the solving request is sent to the appropriate back-end services. The found plan is shown after that with the option to validate it. If there is an error, a stack trace is also provided.

4.1.2 System Monitoring Service

The system monitoring service is also included with the modelling service and visualisation service as the front-end, but not as tightly coupled as we see later (Section 4.2.2). It enables the user to monitor the planning system while solving a request. There are three different views. The first is the system overview which shows the current composition of the planning system in the graphic as seen in Figure 4.3. The next one gives an overview of the status of each planning service during the runtime. And finally, this service offers an administration panel for RabbitMQ (Section 4.1.4).

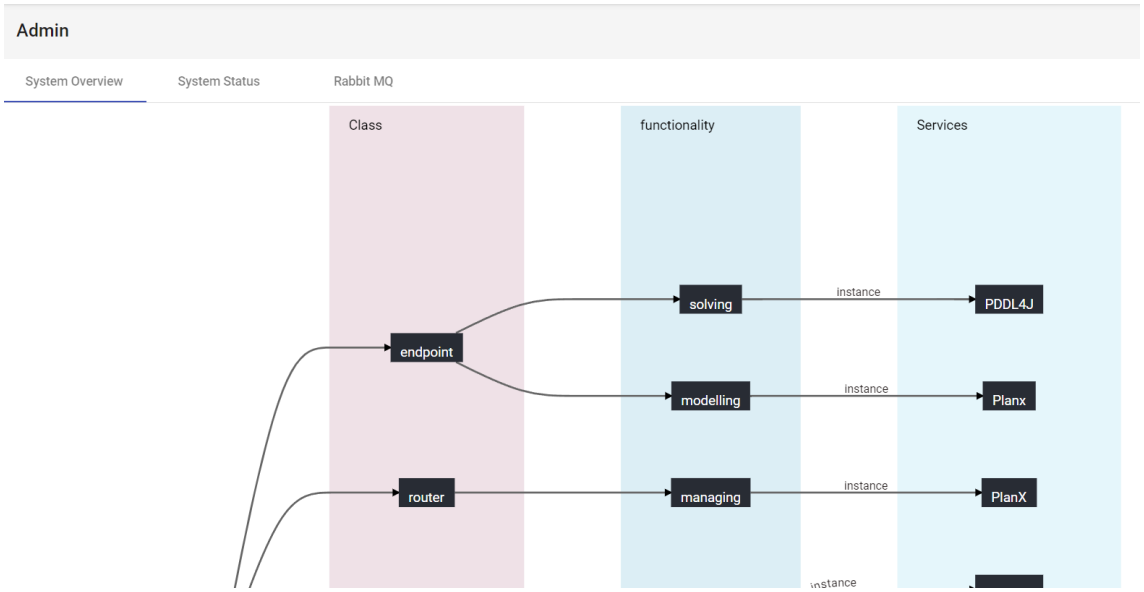

Figure 4.3: The System Monitoring Service interface

4.1.3 Managing Service

The managing service connects the front-end service to the back-end services by handling the user requests and delivering planning results back to the front-end services. There is a set up error queue that delivers the corresponding status of the failing functionality to the user. It is also connected to the message broker, the Message-Oriented Middleware (MOM).

4.1.4 Message Broker

RabbitMQ [VW] is used to implement the MOM. Each planning service is configured to have a message queue for incoming tasks. The RabbitMQ interface (Figure 4.4) provides various information about the flow of messages, all current connections, the messaging queues and more.

4.1.5 Solving Service and Plan Validation Service

After selecting the solving algorithm in the visualisation service (Section 4.1.1), the solving service uses this and decides on its own the planning functionalities. This could be provided by the parsing service for example, which would be the executed before the actual solving process. When choosing a different solving algorithm a different plan may be found.

If the user decided in the front-end interface to validate the plan, the result of the solving service, which is the plan, is sent to the plan validation service. It gives feedback back to the user, therefore back to the front-end.

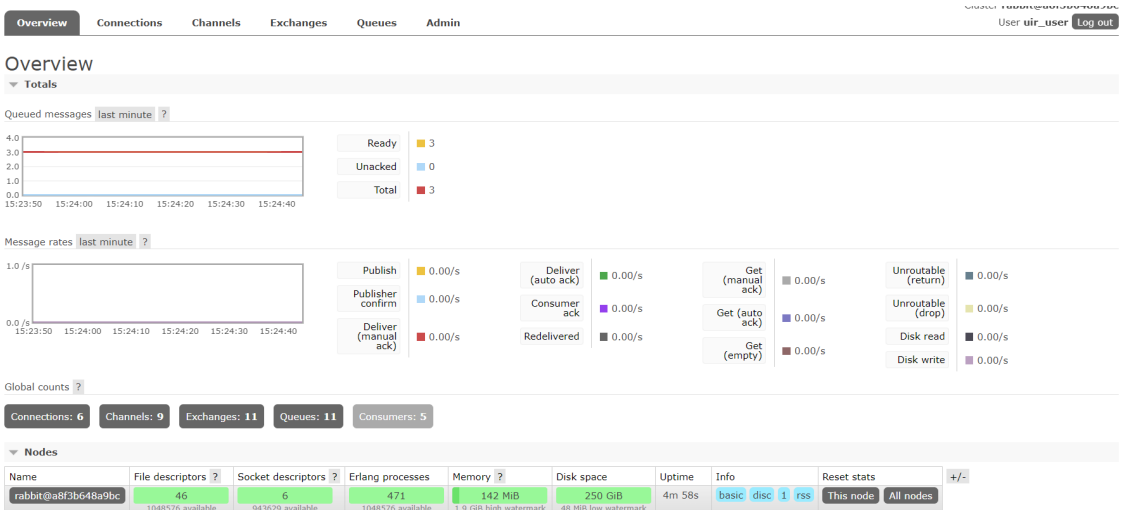

Figure 4.4: RabbitMQ administration interface

4.1.6 Converting and Parsing Service

The converting service takes the PDDL request and performs and encoding step to transform it into its final form. This is typical for most PDDL-based planners. It preserves the representation of the planning problem to maintain the efficient way to determine every reachable proposition from the current state.

The parsing service converts the planning problem into programming level objects. The internal data models are converted into wrapped data models, serialised and made available as JSON messages.

4.2 Architectural Design

The basis of the architecture of the PlanX toolbox is determined in [GG21]. There are different goals that are achieved with the help of an appropriate architecture. We first look into the design choices that are made to accomplish these goals. We then identify the issues with the current toolbox architecture and present a solution.

4.2.1 Design choices

We look at the design choices made as stated in [GG21]. The planning architecture is based on a decentralised approach. This is essential to ensure lose coupling between the services. As already mentioned, the toolbox includes a message broker (Section 4.1.4). Messaging between the planning services via publish/subscribe stack is known to have a good performance. In systems with many clients and high computational load it is especially useful. Another good aspect is the horizontal expandability of the planning system, because every service has its own incoming and outgoing messaging queues. Furthermore, there is the aspect that planning services can request other services without any gateway, which prevents checking for availability and thus boost the performance.

Finally, to preserve the principles of services, they have to be implemented to be stateless. This means that a service capability has to store the state of the process temporarily when it assigns a sub-task to another capability. The result of all these factors is the choice of a decentralised approach.

The following parts will go deeper into details.

The goal of the architecture is to define the composition of the different planning capabilities to make it easier to implement them as services. Therefore, the planning architecture is based on the Hub-and-Spoke pattern. The right choice of the technology for the MOM is essential for the missing scalability of architectures with this pattern, because the MOM is working as the hub. This pattern also makes it easy to connect new planning services to the planning system. Important for an architecture like this is the pattern of the messages and the handling of it by the services.

<*ver*>.<*capability-class*>.<*capability-name*>#<*name*>

The messages are structured in a way that makes it possible to have multiple instances of the same planning service running parallel. The last part of the message, <name>, isn't unique to provide horizontal scalability. The handling of messages is different between front-end and back-end services, because front-end services have to work with direct user input while back-end services do not.

The back-end services subscribes to the hub to exchange messages with it. Incoming messages are placed into a first-in-first-out queue, processed one after the other and the results are pushed into another queue for outgoing messages.

The front-end services receive user input and create a request for the RESTful endpoint of a routing capability, thus the managing service. This service processes the request and sends it to the appropriate service, the solving service. The result goes the same way back through the managing service to the front-end and gets visualised for the user.

In Figure 4.5 we see the overview of the architecture. As already stated, we can see that the architecture is easily horizontal extensible. Right now the toolbox just consists of some of these services connected with RabbitMq as the hub. The front-end is different in the actual toolbox as we see in the next section. It also shows the which services are front-end and which back-end. Furthermore we see the different connection between the front-end and the managing service as well as the connections between the back-end services through the MOM. The web sockets seen between the front-end and the managing service are used for asynchronous messaging while the RESTful connection is for the synchronous messaging.

4.2.2 Identification of an architectural flaws

We can see in Figure 4.5 the optimal architecture for the PlanX toolbox. While the back-end services are implemented in the same way as in the planning architecture, the front-end services are different. The current toolbox has just one front-end services named „frontend-services“. This services contain the modelling service, the visualisation service and the monitoring service as one service. This is not in line with a SOC architecture. The current frontend-services lack in modularity, which also leads to decreased reusability. The severity of this problem is different for the modelling/visualisation service and the monitoring service.

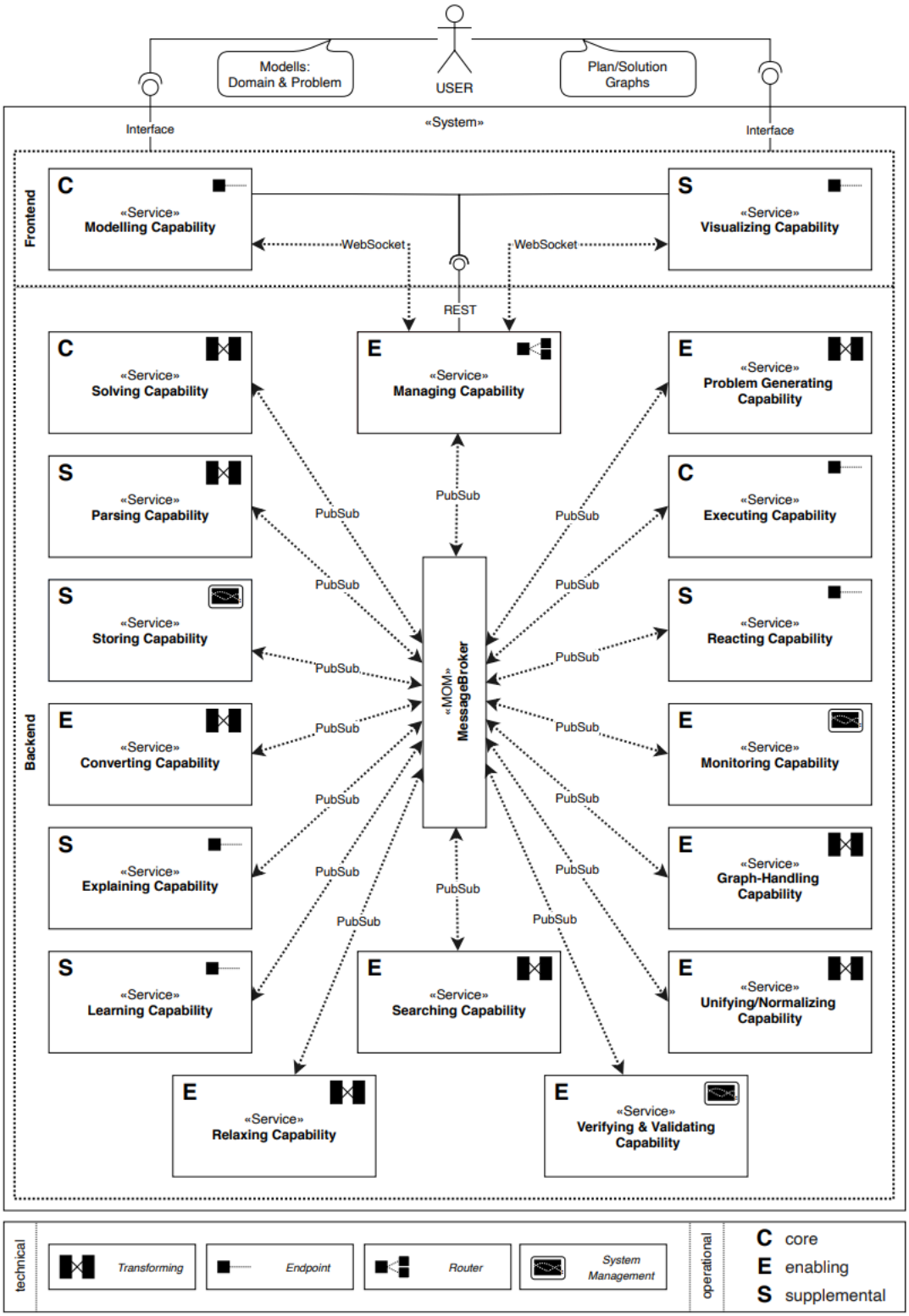

Figure 4.5: Ideal planning architecture for the planning system [GG21]

The problem with the modeling and visualisation service is their tight coupling. We see in Figure 4.2 that the visualisation functionalities are within the editor for modelling. The visualisation functionalities lies within the „RUN“ button, which enables the user to choose a solving algorithm and shows the found plan afterwards. As we see, there is not easy way to just split up these two functions into two different services. A way to communicate between the modelling and visualisation services is needed to make this possible. Furthermore, there should be an option in a possible new visualisation service to show the plan or to just validate it.

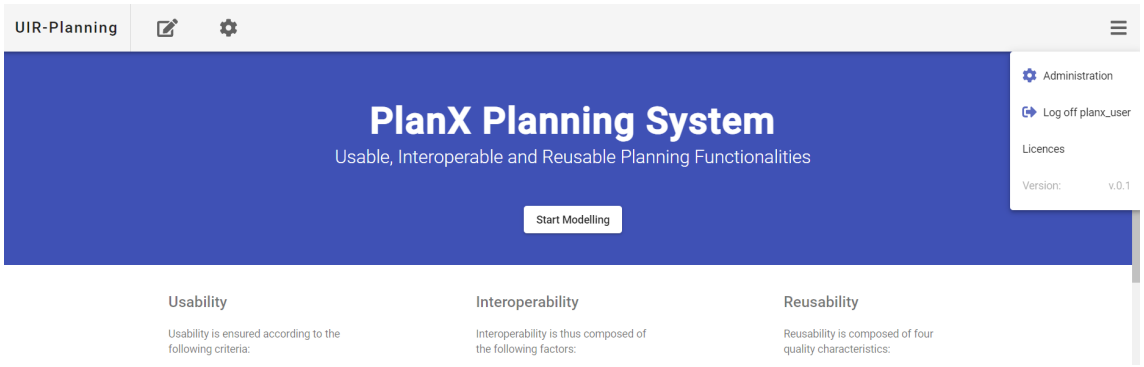

Figure 4.6: Frontend-services homepage

The monitoring service is also part of the frontend-services, but not as tightly coupled. We can see this without reading the code, but just by looking at the structure of the front-end. We see in Figure 4.6 the only two ways to enter the system monitoring service. With a click on the gearwheel or on the „Administration“ tab in the pop-up menu we an enter the monitoring interface as seen in Figure 4.3. When we take a look in the code, we see that the system monitoring even has his own routing-module as well. We can navigate to the system monitoring independent from the rest of the website. There are no dependencies between the the modelling/visualisation and the system monitoring.

This flaw is not just important from a software engineering perspective, but also from a usability perspective. If the services are not modular, the users can not choose if they want a specific service or not. For example, the monitoring service is probably not needed by users who just want to model PDDL planning problems. But in the current toolbox, they have no way of excluding this service. Another negative aspect is the possibility of software errors. For example, if the monitoring service throws a fatal error, the whole front-end is not usable anymore. By providing better modularity, we can assure to reduce the probability of this happening.

4.2.3 Design of the new service

We address these flaws with a new service, the system monitoring service. The system monitoring service fundamentally consists of the same code structure as the frontend-services. We have taken the code of the frontend-services as a basis and modified it to our needs.

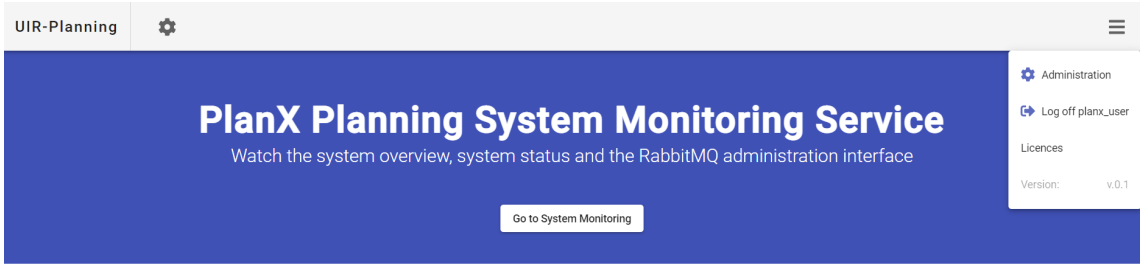

Figure 4.7: System monitoring service homepage

We see in Figure 4.7 that the homepage of the system monitoring service is fundamentally the same as in Figure 4.6. We have kept the structure with a homepage as the first page the users sees and a button in the middle to navigate to the functionalities. The functionalities are exactly the same as it is in frontend-services. The new service has no further interface design changes, except the homepage is empty underneath the blue box.

We have modified the code in the following way to ensure modularity. We have had as a basis a copy of the frontend-services code. After that we have looked into the project structure and have identified all files related to modelling or visualising. We have deleted every one of them and have ended up with the structure of the new service. We have kept all of the existing configuration file or similar files to ensure the building process. There have been many changes necessary as we see in the following section. Next we have kept the homepage as is it, but deleted all the text of it. There is no need to have the same explanation from the homepage twice. Lastly, the core of the service. There are several needed functionalities for the monitoring service that are still there. There has been no need to change anything about that, because it already has had a own routing model and everything to work on its own, without the modelling or visualisation service. The new system monitoring service is completely split up from the other two front-end services without any dependencies. Furthermore, the new service runs on its own localhost on port 5000.

Now we have to do the reverse for the frontend-services. Every file/folder that is important for the monitoring has been deleted in the new version of the frontend-services. This is the same work as in the system monitoring service. After removing every part of the monitoring service, the splitting process is complete. We have only changed two more little things in the interface of frontend-services. We have removed the gear wheel and the „Administration“ option in the pop-up menu now links to the new localhost.

4.2.4 Implementation of the new service

We now talk about the code changes in a more detailed way. For the purposes of the following explanations, we want to clarify that we have started with the code of frontend-services. All the following relative folder paths apply for the frontend-services. Here we can see the difference to the system monitoring service, which has the same folder structure.

First, we go to „apps/planx-frontend/src/app“. We have deleted the „modelling“ folder here. As the name states, we do not need this for the system monitoring service. Next, we have changed the „home.component.html“ (can be found here: „apps/planx-frontend/src/app/home“) for the new homepage. The last change worth mentioning is in „app-routing.module.ts“, where we have removed every option that leads to modelling or visualisation functionalities.

Next we look into „libs“. We see here many folders with functionalities used for modelling or visualisation. We have deleted every folder but „dev-ui“, „login“ and „shared“. These three folder contains all the necessary code for the monitoring. Especially the dev-ui folder, because it contains all the important monitoring functionalities. We have not changed anything here, because the routing as well as the functionality works perfectly fine as it is.

The remaining changes are only in different configuration files or similar ones. We do not mention every little change. There are several references to the deleted functionalities, which we have all removed. We continue with the more significant modifications of files. In the „Dockerfile“ we have only changed the exposed port to 5000. The new service runs on its own localhost, which is on the port 5000. There is one more file we have had to change the port in, which is called „nginx.conf“. Another small but import change is in the „build.sh“. We have changed here the name of the built image, so it does not have the same name as frontend-services. We have decided to call it „planx.toolbox.endpoint.monitoring“. The last big modification is in the „angular.json“. We can see here information and different settings about each functionality that is built. We have looked through the whole file and removed every unnecessary block for functionalities we have deleted.

The process for the new version of the frontend-services has been exactly the same. All the files we have identified as important for the monitoring has been removed. This only includes the „dev-ui“ folder in „libs“. Apart from some small changes in other files where „dev-ui“ has been mentioned, this is all we have had to do. We have removed the gear wheel in the „header.component.html“ found in „apps/planx-frontend/src/app/header“. In the same file we have also changed the „Administration “ button, so it links to the new localhost. We have changed [routerLink]=’admin’ to onclick="window.open(’http://localhost:5000/home’)".

5 Case Study

The remaining work covers the usability aspects of the PlanX toolbox. We want to demonstrate the use of the toolbox as well as the use of the created planning system with a case study. We use the process of executing the case study to evaluate the usability in Chapter 6 to see what the benefits of using the PlanX toolbox are and what changes might be needed.

The case study example is about a „warehouse with robots“ scenario. We see in Figure 5.1 a visualisation of an example in this domain. There is one warehouse with 14 shelves. Each shelf has in this case 2 places, illustrated with white boxes. In theses white boxes can be empty spaces or occupied with products. The robot (in red) defines his position with warehouse and shelf.

There can be various possible warehouses, each with the same amount of shelves. Robots exist in this domain and are able to move between the warehouses and shelves as well as picking up products from the shelves and dropping them in shelves. So, they can organise to warehouse as desired. The complexity of the problem depends on the scale. For instance, if there are two warehouses, each with one shelf, one robot and one product, then there is no complexity in the planning at all. But if we scale it up, we can see the complexity is rising rapidly. If there are a few warehouses, each with several hundred shelves and several robots, the complexity reaches a point where it becomes difficult to solve. These are the cases where planning systems find their application. In the following, we will define this domain as a PDDL file as well as a problem file. After that we look at the problems from different perspective (different possible users) and build a planning system.

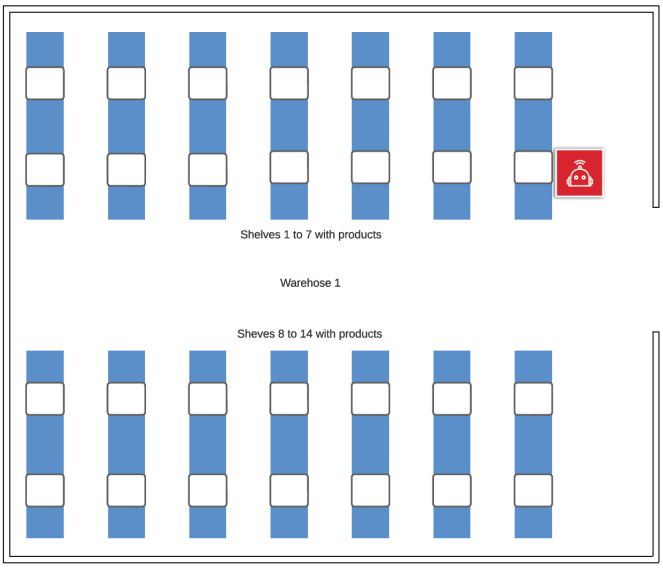

Figure 5.1: Visualisation of the domain (made in https://www.lucidchart.com/)

5.1 Domain and Problem in PDDL

The domain of this case study is a simplified assumption of the real-world. In the real world are more things to consider, for example, how big is the product? How much can one robot carry? How much energy has a robot? How long are the distances traveled and how long does it take? In order to not go beyond the scope of this work and to showcase the toolbox, we decided to model the domain as follows.

Listing 5.1 Types of the domain.

```
(define (domain warehouse-robot)
(:requirements :strips :typing)
(:types robot
        product - entity
        entity
        shelf
        warehouse  - object
)
```

The types „robot“ and „product“ are both defined as entities, because they are the only types in this domain that have a changeable location. The other both types are „warehouse“ and „shelf“ to define the positions of the robot(s) and product(s). The requirements are just the basics, that it has to support the STRIPS level as well as typing.

Listing 5.2 Predicates of the domain.

```
(:predicates
(at ?entity - entity ?warehouse - warehouse ?shelf - shelf)
(robot-holds ?robot - robot ?product - product)
)
```

The predicates of the domain are simple as well. The entities have an initial position in the domain world („at“) which is defined by the warehouse and shelf. The naming for the shelves works like this: shelf12 means shelf 1 place 2. So there is no extra type for the places per shelf, but it is still flexible for the problem instance. We see the use of the type „entity“ here as well. We can use the same predicate to describe the positions of the robot(s) and product(s). The second predicate „robot-holds“ describes the state when a robot is holding a product.

Listing 5.3 Movement actions of the domain.

```
(:action MOVE-TO-WAREHOUSE
  :parameters (?robot - robot ?loc-from - warehouse ?loc-to - warehouse ?shelf - shelf)
  :precondition (at ?robot ?loc-from ?shelf)
  :effect (and (not (at ?robot ?loc-from ?shelf)) (at ?robot ?loc-to ?shelf))
)

(:action MOVE-TO-SHELF
  :parameters (?robot - robot ?warehouse - warehouse ?loc-from - shelf ?loc-to - shelf)
  :precondition (at ?robot ?warehouse ?loc-from)
  :effect (and (not (at ?robot ?warehouse ?loc-from)) (at ?robot ?warehouse ?loc-to))
)
```

The movements of a robot are split up in moving between warehouses and moving to a certain shelf. As already stated, we want to keep this example simple, thus the robot can only move to another warehouse to the exact same shelf. We assume, that every warehouse has to be a copy of each other. The preconditions and effect make sure that the robot leaves his old position and reaches the new one.

Listing 5.4 Pick-up and drop actions of the domain.

```
(:action PICK-UP
  :parameters (?robot - robot ?product - product ?warehouse - warehouse ?shelf - shelf)
  :precondition (and (at ?robot ?warehouse ?shelf) (at ?product ?warehouse ?shelf))
  :effect (and (not (at ?product ?warehouse ?shelf)) (robot-holds ?robot ?product))
)
(:action DROP
  :parameters (?robot - robot ?product - product ?warehouse - warehouse ?shelf - shelf)
  :precondition (and (at ?robot ?warehouse ?shelf) (robot-holds ?robot ?product))
  :effect (and (at ?product ?warehouse ?shelf) (not (robot-holds ?robot ?product)))
)
)
```

Here we see the actions a robot can perform. He can pick up a product if there is one in the shelf, and he can put a product down in the shelf if he holds one. There is no restriction that a robot can only hold one product. Again, there is no upper bound, but we determine that for this domain world and the corresponding problem instances, there are no problems. We should limit the products a robot can carry if we would go for a larger scale with different kind of problems. The same applies to the amount of products per shelf, as there is no limit to the number of products per shelf.

All in all, we make a lot of assumptions about this domain world that would not work in a real-life scenario. Nevertheless, that does not mean it is not possible to model more complex domain models. Thus, we conclude that an easy example is better to show of the toolbox.

At last, we take a look a at an exemplary problem instance file.

Listing 5.5 Problem instance of the domain.

```
(define (problem warehouse-robot-problem)
  (:domain warehouse-robot)
  (:objects
        rob1 - robot
        prod1 prod2 - product
        wareh1 wareh2 - warehouse
        shelf11 shelf12 - shelf
  )

  (:init
        (at rob1 wareh2 shelf11)
        (at prod1 wareh1 shelf12)
        (at prod2 wareh2 shelf12)
   )

  (:goal (and
        (at prod1 wareh2 shelf11)
        (at prod2 wareh1 shelf11)
    )
  )
)
```

This is simple example with one robot, two warehouses with two shelves each and one product each. The goal is to transport the products from one warehouse to another to a different shelf. The robot starts in warehouse1. We will use this example for the quantitative evaluation and scale it up with copy and paste of the same scenario.

5.2 Building and using a planning system

After we model the domain and a problem instance we want to solve, we continue to use the toolbox to build an appropriate planning system. But before we do that, we have to think about what kind of planning system we need. We first consider different use cases for this toolbox, before we look at the kind of planning system we need for this work.

There are probably three different kind of users who will use this toolbox [Geo23b]. We start with system developers. System developers do not need the functionalities of a planning system to solve planning problems. In this case, the architecture of the toolbox is more interesting. The services are delivered as Docker containers, thus system developers can wire the services to a planning system by manually composing them. In more detail, they can change several facts by changing the „docker-compose.yml“. There are the different images listed with their names as an image, their names as a container, the host location where a service can access and store data, other services a service has connections with and the network of containers it belongs to. After changing all the setting, Docker builds the system automatically.

The next user group are system administrators. For example, if the planning system is integrated into another system, information about the performance and the current system status are important. For this or similar cases, they can use the system monitoring service newly introduced in this work. But the functionality already existed in the frontend-services. There are different options to see the state of the system, the composed planning system and different information from the RabbitMQ administration interface.

Most likely the largest group are domain authors. They are the group who actually want the planning systems to solve their domain problems. We assume here that the domain authors work alone and do not get a set up planning system. So, a domain author needs a planning system with a front-end to model the domain and the problem instance(s). The visualisation of the plan is also important as well as the plan validation service, to check the resulting plan. As for the other services, the parsing, converting and solving services are the core. Also worth mentioning is the second solving service, which can be useful to expand the possible solving algorithms to get the best possible result. In the current state of the toolbox, all services but the system monitoring service can be interesting for a domain author.

Now we look at the planning system we want to create to solve the example from Section 5.1. Before we continue, we clarify that we use the new frontend-services with the separate system monitoring service. In the Section 6.1 we discuss the difference to the old PlanX toolbox for this case study.

The first requirement for our planning system is that we need a standalone system. Thus, the use of the frontend-services is absolutely necessary. Without the frontend, we can not model the domain or problem instance for this case. Furthermore, for the sake of the evaluation later (Chapter 6), we also use the system monitoring service. Apart from testing of the new service we also can see the message throughput and more. The irreplaceable back-end core of a working planning system are RabbitMQ as the MOM, the converting service, the parsing service and one solving service. As for the solving service we decide to use the PDDL4J one without any further reasoning. The managing service is absolutely necessary as well to connect the front-end to the back-end.The plan validation service lies not in our interest. The quality of the resulting plan as well as the quality of the different solving algorithms is not in the scope of this work.

In order to compose the planning system we need to change the „docker-compose.yml“ and comment the services out we do not want. By default, all services get composed, thus even without knowledge of Docker it is possible to compose a planning system. Another prerequisite are the finished Docker images. For Unix operation systems, there are bash files to build every service and one „build-all.sh“ to execute every build file with just one click. In this case we use Windows, which leads to a different approach for the building. We can use the contents of the building files and enter then manually in the console to build every service. After successfully doing that we can find the images in Docker to confirm every needed image is built. Finally, we can use „docker-compose up -d“ and the composed planning system is built consisting of the services as containers. It also starts the planning system automatically which concludes the building and starting process.

In the last step we open a browser and navigate to `http://localhost:4200/` to use the frontend-services. After navigating to the web editor we can either start to model in there or drag and drop a prepared PDDL file. Finally, we just need to press „RUN“, choose a solving algorithm and look at the system monitoring and Docker logs to see how the toolbox is doing. More precisely we see in the Docker logs of the solving service the time to find a plan. In Figure 5.2 we see what a plan looks like for the example problem instance of Listing 5.5.

Sequential Plan

000	**move-to-shelf** (robot, warehouse, shelf, shelf) rob1 wareh2 shelf11 shelf12
001	**pick-up** (robot, product, warehouse, shelf) rob1 prod2 wareh2 shelf12
002	**move-to-warehouse** (robot, warehouse, warehouse, shelf) rob1 wareh2 wareh1 shelf12
003	**pick-up** (robot, product, warehouse, shelf) rob1 prod1 wareh1 shelf12
004	**move-to-shelf** (robot, warehouse, shelf, shelf) rob1 wareh1 shelf12 shelf11
005	**drop** (robot, product, warehouse, shelf) rob1 prod2 wareh1 shelf11
006	**move-to-warehouse** (robot, warehouse, warehouse, shelf) rob1 wareh1 wareh2 shelf11
007	**drop** (robot, product, warehouse, shelf) rob1 prod1 wareh2 shelf11

Figure 5.2: Plan for the problem instance of the case study

This concludes the case study problem description and the way to solve it. In the following Chapter 6 we evaluate the case study as well as the system monitoring service. For the case study quantitative evaluation we consider the same problem instance from the case study, but we scale it up with more warehouses (so more shelves and more products as well) and see how the toolbox performs.

6 Evaluation

The evaluation consists of different parts. We start with the evaluation of the case study. We talk about different usability factors and our experiences using the toolbox and a created planning system. We showcase the benefits of using the toolbox and the negative aspects to give a complete picture of the PlanX toolbox. Next, we evaluate the new system monitoring service. Lastly, we have asked five users to use the toolbox and have interviewed them to tell us about their experiences and ideas to further improve the toolbox in the future.

6.1 Case Study

We divide this section into qualitative evaluation, quantitative evaluation and talking about specific problems. We start with the qualitative evaluation. Here we look at the usability factors from Kabir et al. [KRM16] and evaluate each of them from our perspective. Moreover, we use this part to compare the old PlanX toolbox to the new one. The quantitative evaluation is a smaller part, where we look at the message throughput and the solve time for scaling problem instances. In the last part, we consider the problems we have encountered while building and using the toolbox or the planning system we have created.

6.1.1 Qualitative evaluation

We look at the twelve quality factors that Kabir et al. [KRM16] has proposed and see how the PlanX toolbox is doing in this case study from our point of view.

Operability

Operability describes the ease of use of the system. This means how easy it is for the user to operate and control the system. We use the toolbox in Windows, which means the bash files to build the box are not working. This is a negative aspect in the operability of the toolbox. In this context we have to mention the existence of Windows Subsystem for Linux (WSL) that solves this problem. With the help of this Linux interface for Windows we can use the bash files and build the toolbox without any extra effort. This is still a negative point, because the user is forced to use one type of operation system, which is also is not good for portability. Still, it is not impossible to build it in Windows. We can just copy the commands out of the bash files and enter them in the console manually, which is what we have done as well. A positive point is the possibility to choose the services we desire for a planning system. The only option we have to change for that is the docker-compose file which is very understandable and simple, even without much Docker experience. The planning system itself is very clearly structured and shows every capability and option. Another negative aspect is that the editor does not save the domain and problem files when you run the planning system. When we

navigate back to the editor after we have a plan, we see the standard example again. Thus, when we want to run the same domain with different problem instances, we always have to enter the domain file again.

Efficiency

Efficiency indicates how quickly the users can perform tasks after learning the system. With this thought in mind, we look at building and setting up the toolbox and creating a planning system. We conclude that there is no much room for time improvement with more knowledge about the toolbox. The building process is well comprehensible and there is not much to improve on the second time around. This looks different with the use of the toolbox. The option to drag and drop PDDL files into the editor fairly improves the time to get a plan. A user can prepare a domain file and one or more problem instances beforehand and use the toolbox way more efficient this way.

Effectiveness

Effectiveness is about the accuracy and completeness of the user goal achievement. To be more precise, this is about the succession and failure rate while trying to achieve a goal. We have found several problems while using the toolbox in our specific environment, which we discuss in Section 6.1.3.

Learnability

Learnability describes how easy it is for the user to finish the task for the first time. According to our point of view, it is very easy to accomplish the desired goal, even if we are using the toolbox for the first time. Navigate through the website to the editor, entering our domain and problem into the editor and press the red „RUN“ button seems very understandable to us. The difference in using the toolbox after gaining experience is limited to efficiency aspect only.

Training

Training means how the system is teaching the user. There is not much to say, because there is not much to learn about using the toolbox or the created planning systems. But there is one positive aspect: the standard example. For example, if the user is familiar with STRIPS planning problems, but not with PDDL syntax, we can use the example as a template to enter our problem. In our case it helped to solve a parsing problem, as we further discuss in Section 6.1.3.

Satisfaction

Satisfaction is about the subjective feeling when using the software. Apart from the problems it is all in all a very satisfying experience. The building process is easily executable and many different processes are automatic without the need to understand it. The planning systems are easy to use and the visualisation of the plan is clearly arranged as well. The login process seems slightly useless in our opinion, but we do not know what is planned in the future for this software. If it is needed in the future and the feature is already implemented for that use, we can call it a luxury problem, because the username and password are saved after we have entered them for the first time.

Understandability

Understandability is about how easy a user can understand a task in the system. We have already said everything regarding this topic, so we continue with the next one.

Helpfulness

Helpfulness indicates the guidance given to the user to solve the task. This is definitely given in this toolbox. When we open the frontend-services, there is the „Start Modelling“ button which we can not miss. It leads us directly to the different options to deal with the problem. Every option has an explanation on what it is and what the user can use it for. Currently there is only one option, „Create full Model“, to navigate to the web IDE. From here on it is obvious where to enter the domain/problem and how to solve it. We get smoothly navigated trough the front-end without any problems. The same goes for the system monitoring service. After pressing the same button as in the frontend-service, that is called „Go to System Monitoring“ here, we have an overview that is self explanatory.

We skip the last four factors. The reason for three of them, „Attractiveness“, „Usability Compliance“ and „Human Engineering“, is that we do not have anything (new) to say about these. We discuss the „Reliability“ in the following sections.

Finally, we look at the difference between the old and the new toolbox. Most factors are exactly the same between both versions. One difference is that the system monitoring was better accessible through the old frontend-services interface. The gear in the header links directly to the system monitoring. We think that the gear symbol does not indicate clearly where it links to, but after gaining knowledge about the toolbox it is a convenient way. Another difference is the missing option in the old toolbox to choose whether we want the system monitoring service or not. Otherwise the old toolbox does not change anything for the actual use nor the building process. It makes no difference in terms of effort to build one more service, but it does take some extra time. So, in the end we have a new toolbox with a refined architecture and better options for the user, while there are no negative aspects. More about the changes with the new service is discussed in Section 6.2.

6.1.2 Quantitative evaluation

For the quantitative evaluation, we use the domain model from Chapter 5 and fundamentally the same problem instance. We scaled the problem instance the following way: We have the same amount of warehouses and products. For the first execution there are two warehouses and two products with one robot. For the goal the products have to switch the warehouse. We ignore the shelf option, thus every product always starts on the same shelf and needs to be brought to another shelf as the goal, which is always the same as well. There is only one robot for every instance. So, a problem instance with 20 warehouses has 20 products, which all need to be moved one warehouse by only one robot.

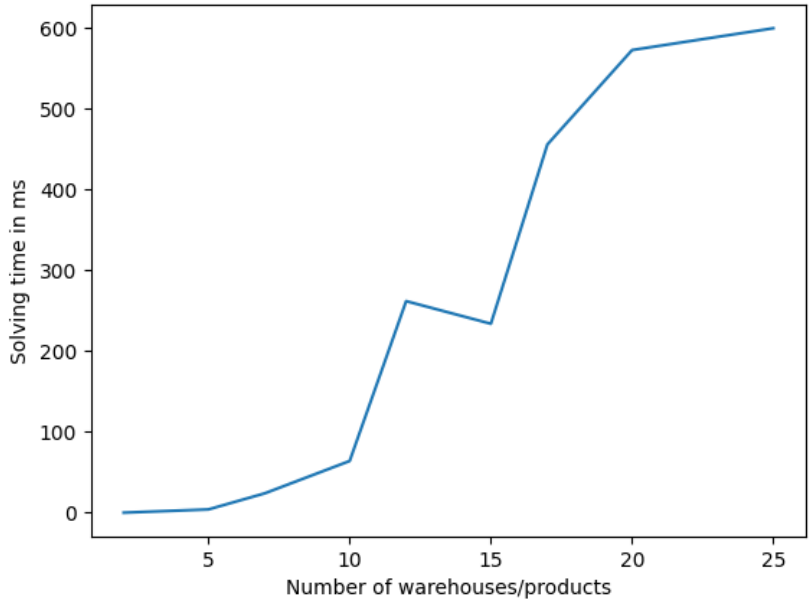

Figure 6.1: Diagram to show the solving time related to the size of the problem (created in python)

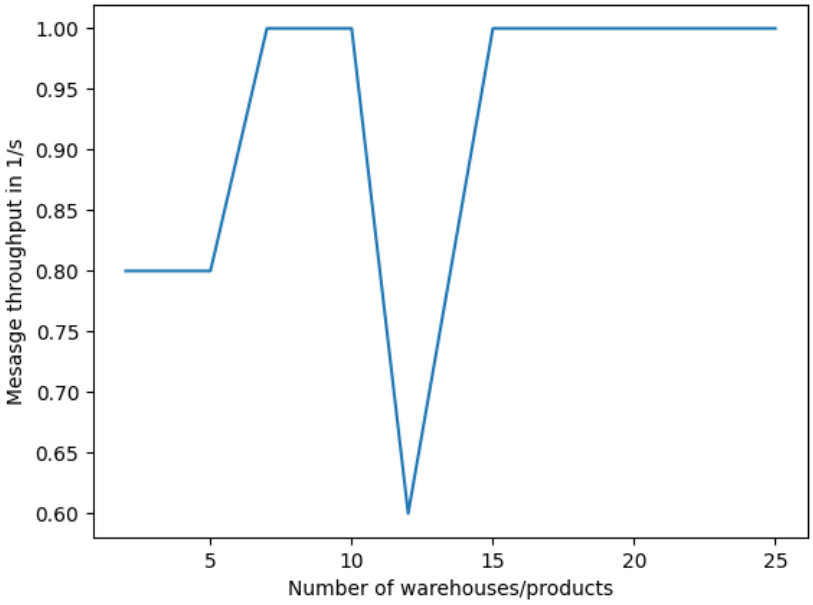

Figure 6.2: Diagram to show the message throughput related to the size of the problem (created in python)

As the solving algorithm we use the „Fast Forward Planner“. We first have used the „Heuristic Search Planner“ which has turned out to be not that good, because there has been no plan after just a few more warehouses. Furthermore, the response time was much longer than with the Fast Forward Planner. We have not tested the remaining to planners, because the Fast Forward Planner is more than enough for our example. There is the possibility, that the other planners may perform better/worse or they find plans for bigger state spaces. We do not care about that for this work and use the Fast Forward Planner as the representative planner.

In Figure 6.1 we see the solving time of the planner depending of the number of warehouses/products. We need to look at the real sizes of the problems to better understand the big differences in the solving time. For example, when we double the warehouses from 10 to 20, the solving time increases many times over. The Fast Forward Planner is used to solve problems in state spaces [Hof01]. As we already mentioned in Section 2.1 the state space for a problem can grow exponentially. We do not know how this solver actually works, but we can expect that the size of the state space increases way more than twice if we double the number of warehouses.

We can see in Figure 6.1 that the solving time is increasing as expected, depending on the warehouses. But there is one irregularity at 12 warehouses. The solving time is higher than the time for 15 warehouses. We also see in 6.2 that the message throughput, which we obtained from the RabbitMQ administration interface, drops for this execution. Here we can continue the qualitative evaluation for the „Reliability“. We see that the message throughput is most of the times constant. The size of the problem does not matter in the communication. In addition, the communication is most of the times very consistent, even though it is sometimes a little bit slower, as we can see in the beginning of the graph in Figure 6.2. Coming back to the actual irregularity, we can only assume what happened there. The fact, that it has solved the problem, means that there is no bigger error. There has probably been a error while sending messages. It is a bit surprising that the actual solving time is higher as well, but we do not know when the service starts and stops the timer. If the timer starts with receiving the messages, it makes more sense. Maybe the message was broken and came in a second time, which lead to the irregularity in the graphs. As a conclusion, we can say that even this irregularity is not even noticeable. Therefore, the reliability factor does not suffer from this.

Another important point to talk about is that it did not find a plan for 20 or 25 warehouses. After it has worked for 17 warehouses, we have tried to find a plan for 20 warehouse, but the planner has not found a plan. The same goes for the Heuristic Search Planner, and again we have not tried the other two. Just to be sure, we have entered a problem instance with 25 warehouses, but it has not found a plan either. Apart from the fact, that there is a limit on how big a problem can be, we want to know how the limit is compared to other planning systems. For this reason we have looked for another planning system and have found the PDDL editor from Planing.Domains (see Section 3.4) which is online available under `http://editor.planning.domains`. Sadly, there is no option to show the solving time, so we can not compare the planning systems. But as a reference we have entered our domain and the problem instances with 20 and 25 warehouses. The solving time is noticeable longer than the smaller examples, but we got a plan. Thus, there is the possibility for the PlanX toolbox to create planning systems that can handle bigger problems. The question is, how big can planning problems get in the real-world? Even without knowing that, we conclude that it is better to provide the opportunity to solve as big planning problems as possible.

6.1.3 Issues with PlanX

We have run into several problems while building and running the toolbox and the created planning system. We look into these problem, evaluate how severe the problems are and assume if it is an environment problem or a problem for every possible user. As a reminder, we have worked on a Windows system without WSL. The only difference in using WSL should be the speed in the building process, because we can use the „build-all.sh“ file to build everything with one click. But the building process obviously remains the same, even if we use the Windows console and build every service separately.

The first issues have occurred while building it for the first time on our desktop PC. We have wanted to build only the frontend-services to look at it and try to split up the system monitoring, which is one of the results in this work. Even after several attempts and several unsuccessful fixes, we still have not been successful to fix one error. The builder is not able to find the python installation on our desktop PC. We have ensured that python is installed, „python –version“ is working and entered as a variable in PATH. As we see in the survey in Section 6.3 we are the only ones with this problem. So, we assume this is a specific error from this certain environment. There is the possibility, that there has been a different kind of error, but the feedback from the building process has been inaccurate.

We have continued to work on our laptop. While running it on the laptop, it has worked except for some small software errors, for example, typos or missing imports in files. These kind of errors are easily fixable. This is no problem for future users, because errors like this are fixed before it will be published. So our conclusion is that the build process is working and should be on most computers.

Creating the planning system with Docker has worked smoothly without any errors. But there are some problem while running the toolbox. First of all, we do not receive a plan every time a plan is found. When we look into the Docker logs, we see in the logs from the solving service if a plan is found. In approximately 50% of the cases the management service does not handle the request correctly. The problematic part is after the solving service sends the found plan to the managing service. The managing service recognises the request every single time and prints in the logs „Handle new websocket Session“ as well. But it stops there and does not pass on the plan to the frontend-services to visualise it. The planning systems uses Web Sockets for asynchronous messaging. We assume there is a problem with the Web Sockets on our system, but not a software issue. The occurrence of the „error“ seems to be random. Maybe there is a pattern, but we have not been able to find it out after much testing. Thus, it seems to be a specific problem depending on the environment, in this case our laptop.

The last problem we have encountered is about the parsing service. We have modelled our domain and problem file directly in the web editor provided by the frontend-services, not in a separate editor/IDE. After we have finished, we have always received a parsing error, which has not been specified. As a reference, we have used the Plannings.Domain editor and it has worked there without a parsing error. After trying several times to fix it, we have been able to parse our problem. We have used the standard example by the toolbox as a template. We have not changed any spacing or anything and entered our types/predicates/actions. This approach has lead us to a working example. We assume that the parser is not able to handle a „Tab“ as a spacing tool or it can not handle too many spaces. For example, this line „:parameters (?robot - robot ?product - product ?warehouse - warehouse ?shelf - shelf)“ originally has looked more like this „:parameters (?robot - robot ?product

- product ?warehouse - warehouse ?shelf - shelf)“. The idea has been to make the files easier to read. We have not tested it further to see exactly where the problem lies, but there should be no other explanation. We conclude that this is a problem for future users as well. There should either be a warning that tabs/spaces (or whatever else) are not allowed or there is a need for change in the parsing service.

6.2 System Monitoring Service

Next we look at the refined architecture with the new system monitoring service. As we already stated in Section 2.4, one of the most important factors of a SOA is lose coupling and therefore modularity as well. We want to achieve that every service of the toolbox has just one encapsulated functionality. Creating the new service is a big step into this direction. The new service has exactly one encapsulated functionality, and the new frontend-services is closer to that goal as well. As we see in the case study (Chapter 5), most users probably do not want to use the system monitoring, thus giving the users more options on which functionality to build is an improvement towards a more user friendly toolbox. But it is also important from the software engineering perspective. If the system monitoring has an issue which causes the entire frontend-services to crash, the planning system as a standalone becomes unusable. With the new architecture that is not a problem anymore. The same idea works with modifying or maintaining the service. With better modularity and lose coupling these operations become easier to execute.

Another good part is that there are no changes in the user experience but the option to include/exclude the system monitoring as desired. In the new frontend-services is a linkage to new system monitoring service in the pop-up menu under the „Administration“. But we removed the gear symbol of the header of the homepage. We believe it is unclear that this symbol links to another localhost page. Since it is just a symbol without the option to add text or anything we think it is better to delete it and to split the frontend-services more from the system monitoring service. There is definitely the need to add the link to the localhost:5000 in the README of the „compose-planning-system“, so that users can find it. Moreover, we believe that using the same layout with the same homepage helps with understanding the system monitoring interface. If users only use the frontend-services first and add the system monitoring later, they should be able to understand the interface with their experience immediately. The starting homepage looks the same in both services with the same button to proceed through the website.

Finally, there is one issue with the new service. The time it takes in the building process is too high. As we stated in Section 4.2.3, the basic structure of the new service is a copy of frontend-services. It was really helpful way to address the problem and create the new service. The only problem is the number of dependencies the frontend-services have. When we look in „package.json“ or „package-lock.json“ in frontend-services, we see a large number of necessary dependencies for it to work. But we do not know the functionality and use of every dependency that has been added to the service. We are sure that there is much room for a performance boost for the building, if we delete the unnecessary dependencies for the system monitoring. Since the knowledge is missing, we decided it is better to stick to the current way, even if it takes more time.

6.3 Survey

For the survey, we have asked five different users to try and build the toolbox and compose a planning service. Four of them are computer science students and one has very little experience in this field. Three of the users have built the system on Windows, one of them has used WSL and one Linux. The survey has been conducted in the form of an interview. Thus, the users have tried to build the system and after that we have asked them for their opinion. Before they have started, we have given them verbal instructions as a help. None of them have any experience with plannings problems or PDDL and just one has used Docker before.

We have decided to send pre-built back-end services. If an error occurs while trying to build the back-end services with gradle, we can still build the Docker images. The idea is to get the most feedback out of every user who tries to build it. But this leads to a problem, because the RabbitMQ service throws an error due to something about „ownership of cookies“. Thus, this survey is fully about the experience of building every service, creating a planning system and taking a look at the user interface.

We start with the first question. *Were there any problems while building, and if so, what were they? And how was the overall experience?*

The prerequisites for building this toolbox are Python, Java and Docker. Most of the users have already Java and Python installed and every one of them have had to install Docker. There has been not many problem with these and the users has been content with the Docker installation as well, because it has been easy and understandable. The only issue has been a missing WSL update which is not complicated to solve. The first user, who uses WSL, has had problems with gradle. He has always got permission denied errors. We are certain that these are environment related problems and have nothing to do with the software. Because the services has been pre-built, he still has been able to build the images without further problems. Two other Windows users have had other problems with gradle as well. For example, one error is „gradle build failure: unable to start daemon process“. We assume that there are problems with the installed java versions. Building the Docker images has been not a problem. The remaining Windows user has had no problems at all. The Linux user, in contrast, has not been able to build any Docker images, because Docker has thrown all kinds of errors and not a single image has been built. We do not know the cause of these and have not investigated the problems further. Again, it has seemed to us that the errors are completely environment dependent as well, so there is nothing to change in the toolbox to improve it.

The overall experience has not been bad. No one has said something negative about too many prerequisites, a too long building time or a too complicated process. The experience for Windows users without WSL has not been the best. The users have complained about building every service on its own without an appropriate building file for Windows. This can also easily lead to self-made errors, when they use the wrong Docker build command for a service. These build commands are all very similar but the name of the image, which can lead to confusion or errors from indulgence. But it has still been an easy, understandable task, especially building the images in Docker. Even the user without much computer science experience has had no problems building the system, which speaks in favour of good usability.

The next question is about composing a planning system. *How understandable is it to integrate/leave out the individual services for composing?*

As we have already stated, the users have little to no experience in Docker. The command to build a planning system with every service can be found in the GitHub, so there has been no problem at all. But due to the missing experience none of them knew the importance of the „docker-compose.yml“ file or how to use it. After a short explanation every one have understood it immediately. Thus, it has not been understandable for the users to do it on their own. We conclude that there is a need for a short explanation in the README.

But every one of them have liked using Docker. They say that Docker is easy to install and to use as well. The interface is understandable and easily usable from the beginning. The images and containers are well presented and it is clear on how to stop the containers, how to start them again and how to see the logs. All in all Docker seems to be a good software to build the planning systems.

We continue with a question about the user interface. *What is your opinion on the user interface?*

The user interface has been a satisfying experience for the users. The homepage is clearly structured and easy to understand. The users have got navigated through the website without any problems. Finding the editor to enter the planning problems has been no problem as well. One user has pointed out that the menu on the left side of the editor has an arrow to minimise it, but the arrow to show it again is on another position. Another point two users have made is that the placement and form of the „RUN“ button is not good. Every option for the editor is on the left side in the menu, but the „RUN“ button is alone on the lower right on the website. We think the red button has a good visibility, but maybe it can be moved to left side in the menu as well. One user has stated that the system monitoring page is not easily accessible through the frontend-services. The one linkage can not be easily found and entering another localhost address is not very user friendly. We agree to some degree and think there could be a visible link on the frontend-services homepage, but we do not think it is necessary. But the new link still needs to be added in the README of the compose-planning-system repository. For this survey, we have simply told the users where to find it. One last opinion on the system monitoring homepage has been that it seems useless, because there is no extra information and just a button. We agree, but the idea is to leave space for the future to perhaps provide some kind of information before the user enters the system monitoring functionalities.

The last question was about changes for the building and the interface. *What would you change about the building/interface? What is missing?*

We have already talked about the „RUN“ button and the user have suggested to either make it bigger or move it to different place to make it more visible. Apart from the accessibility of the system monitoring homepage from the frontend-services, the users have been content with the interface.

There are several ideas to change the building process and make the experience better. First of all, every one of the users have remarked the lack of documentation. Especially because the building process is not complicated, a few lines of documentation in the README would have helped them to understand the structure of the software better. More importantly, some documentation for using Docker is essential. Even without any knowledge, a user does not need to fully get Docker to build the toolbox. Quite the opposite is the case. The users have said that with a bit of simple documentation about the „docker-compose.yml“ they would have had no problem with composing a planning system by themselves. The accessible video in the GitHub is not understandable enough for users with no prior experience about anything regarding the toolbox. Other than that, some users have proposed the idea of a build script, that is executable in Windows without WSL. One user has

had some ideas to change the present build script. First, he would have liked a version check for the required software (Java and Python). The next idea is that the „baseDir“ should be able to handle empty spaces in the name of a folder. Users have all kind of different folder structures and the build script should be able to run on all of them. The last propose was adding tags in the build script, so the user can include or exclude which service should be build with the „build-all.sh“ script.

This concludes the survey, which gives us a better perspective on the toolbox and especially its shortcomings. Apart from individual problems with their computer, we can summarise that there is a need for better documentation and usability while building the PlanX toolbox. But the user interfaces are well designed and give the users a good experience while using their created planning system.

7 Conclusion and Outlook

Conclusion

The PlanX toolbox purpose is to provide the possibility to build advanced planning systems without any advanced expertise. We believe to have shown the benefits of the use of the toolbox. The building process is simple and doable without even knowing how it works. For example, building the Docker images works on its own with the provided Docker files in every service. A user does not need to understand any of this to create the images and finally a planning system. We have used the case study we see in Chapter 5 to demonstrate the usability of the toolbox. Using the toolbox to solve a problem is a understandable and good experience. We have evaluated the toolbox under many different usability factors and shown the good and bad aspect for the user experience. We can say it is most of the time satisfying to use the toolbox. We have also pointed out negative aspects to improve in the future. For example, the web socket issue with the managing service, the parsing errors because of spaces/tabs or that the editor does not save the domain/problem files. Apart from these and some problems while building, we think the toolbox is a good overall solution to provide planning functionalities to build a desired planning system that is able to solve and visualise planning problems. But we have to keep in mind that the usability evaluation from our perspective about one case study might be lacking. Different kind of users with different needs or with different examples might see the toolbox in another light. We believe that our view on the PlanX toolbox is as general as possible. The users from the survey have given us a new perspective on some problems that have occurred, but none of these were severe or the fault of the software. But it has been a good insight on the strengths and weaknesses of the toolbox. The building process is fundamentally simple and meets our expectations. We have also seen that documentation as well as other small points are missing to provide an optimal user experience. But we have to mention here, that the user study would have been better, if the users actually needed this toolbox for research or similar work. In this case, we could have asked experts on how they perceived the toolbox, which would provide an even better insight.

The refining of the architecture was fundamentally successful as well. We have not been able to split up the modelling and visualisation service, but we have been successful with the monitoring service. We have improved the lose coupling and modularity of the toolbox without any negative impact on the functionality. But there is more work to do with the new service, especially the building speed, but also the design of the user interface.

Outlook

There is room for more improvement in the future. One aspect to further improve is the usability. As we can see in the feedback from the users, there are many ideas to make the toolbox more user friendly. All of these possible changes are just small and easily implementable and will most likely happen in the future. We believe the most important task is to completely split the remaining two front-end services. There is probably a need to fully change the visualisation to make it work on its own without the modelling service. It is a bigger challenge but nothing impossible to do. Moreover, the new system monitoring service needs improvement for the building process. It is possible to lower the number of dependencies and thus improve performance while building. Another point to talk about is the possibility to add different input languages and hence different planning types in the future. We can already see the option to choose the input language, even though the only option is PDDL right now. That means, there will come more languages and probably appropriate parsing, converting and solving functionalities as well. Finally, we can also talk about the option to scale to toolbox with more services. We see in Section 4.2.1 possible new services to add into the toolbox to further improve the usability. With more services available, there are more potential users who can solve their problems with their personalised planning system. The chosen architecture also allows to do this easily and leaves room for software engineers to add their own services and use them for their planning systems. The flexibility of the PlanX toolbox is convenient for future improvements of any kind.

Bibliography

[AGPA22] E. Alnazer, I. Georgievski, N. Prakash, M. Aiello. “A role for HTN planning in increasing trust in autonomous driving”. In: *2022 IEEE International Smart Cities Conference (ISC2)*. IEEE. 2022, pp. 1–7 (cit. on p. 13).

[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”. In: *Technical Report, Tech. Rep.* (1998) (cit. on p. 15).

[FL03] M. Fox, D. Long. “PDDL2. 1: An extension to PDDL for expressing temporal planning domains”. In: *Journal of artificial intelligence research* 20 (2003), pp. 61–124 (cit. on pp. 16, 17).

[FMJB15] 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 International Conference on Smart Sensors and Systems (IC-SSS)*. IEEE. 2015, pp. 1–6 (cit. on pp. 13, 21).

[FPD13] S. Fratini, N. Policella, A. Donati. “A service oriented approach for the interoperability of space mission planning systems”. In: *Workshop on Knowledge Engineering for Planning and Scheduling*. 2013, pp. 39–43 (cit. on pp. 13, 22).

[GB21] I. Georgievski, U. Breitenbücher. “A vision for composing, integrating, and deploying AI planning functionalities”. In: *2021 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. IEEE. 2021, pp. 166–171 (cit. on pp. 13, 17).

[Geo15] I. Georgievski. “Coordinating services embedded everywhere via hierarchical planning”. English. PhD thesis. University of Groningen, 2015. ISBN: 978-90-367-8148-0 (cit. on p. 15).

[Geo23a] I. Georgievski. “Conceptualising Software Development Lifecycle for Engineering AI Planning Systems”. In: *2023 IEEE/ACM 2nd International Conference on AI Engineering – Software Engineering for AI (CAIN)*. 2023, pp. 88–89. DOI: 10.1109/CAIN58948.2023.00019 (cit. on p. 18).

[Geo23b] I. Georgievski. “PlanX: A Toolbox for Building and Integrating AI Planning Systems”. In: *IEEE International Conference on Service-Oriented System Engineering*. 2023 (cit. on pp. 25, 38).

[Geo23c] I. Georgievski. “Towards Engineering AI Planning Functionalities as Services”. In: *Service-Oriented Computing – ICSOC 2022 Workshops*. Ed. by J. Troya, R. Mirandola, E. Navarro, A. Delgado, S. Segura, G. Ortiz, C. Pautasso, C. Zirpins, P. Fernández, A. Ruiz-Cortés. Cham: Springer Nature Switzerland, 2023, pp. 225–236. ISBN: 978-3-031-26507-5 (cit. on pp. 18, 19).

[GG21] S. Graef, I. Georgievski. “Software architecture for next-generation ai planning systems”. In: *arXiv preprint arXiv:2102.10985* (2021) (cit. on pp. 17, 19, 25, 28, 30).

[HBBB21] D. Höller, G. Behnke, P. Bercher, S. Biundo. “The PANDA framework for hierarchical planning”. In: *KI-Künstliche Intelligenz* (2021), pp. 1–6 (cit. on pp. 13, 22, 23).

[Hof01] J. Hoffmann. “FF: The fast-forward planning system”. In: *AI magazine* 22.3 (2001), pp. 57–57 (cit. on p. 45).

[HPK12] D. D. Hoang, H.-Y. Paik, C.-K. Kim. “Service-oriented middleware architectures for cyber-physical systems”. In: *International Journal of Computer Science and Network Security* 12.1 (2012), pp. 79–87 (cit. on p. 21).

[KM20] E. Karpas, D. Magazzeni. “Automated planning for robotics”. In: *Annual Review of Control, Robotics, and Autonomous Systems* 3 (2020), pp. 417–439 (cit. on p. 13).

[KRM16] M. A. Kabir, M. U. Rehman, S. I. Majumdar. “An analytical and comparative study of software usability quality factors”. In: *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)*. IEEE. 2016, pp. 800–803 (cit. on p. 41).

[MFMM17] M. C. Magnaguagno, R. FRAGA PEREIRA, M. D. Móre, F. R. Meneguzzi. “Web planner: A tool to develop classical planning domains and visualize heuristic state-space search”. In: *2017 Workshop on User Interfaces and Scheduling and Planning (UISP@ ICAPS), 2017, Estados Unidos.* 2017 (cit. on p. 26).

[Mui16] C. Muise. “Planning. domains”. In: *ICAPS system demonstration* (2016), pp. 242–250 (cit. on p. 23).

[PF18] D. Pellier, H. Fiorino. “PDDL4J: a planning domain description library for java”. In: *Journal of Experimental & Theoretical Artificial Intelligence* 30.1 (2018), pp. 143–176 (cit. on p. 23).

[PG03] M. P. Papazoglou, D. Georgakopoulos. “Service-oriented computing”. In: *Communications of the ACM* 46.10 (2003), pp. 25–28 (cit. on pp. 18, 19).

[VAMD09] M. H. Valipour, B. AmirZafari, K. N. Maleki, N. Daneshpour. “A brief survey of software architecture concepts and service oriented architecture”. In: *2009 2nd IEEE International Conference on Computer Science and Information Technology*. IEEE. 2009, pp. 34–38 (cit. on p. 19).

[VW] A. Videla, J. Williams. *RabbitMQ in action. 2012* (cit. on p. 27).

All links were last followed on September 28, 2023.

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.

---

place, date, signature