UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Kelian Kaio

Generating Process-based Mobile Applications for the Internet of Things using Automated Planning

Master's Thesis (30 ECTS)

Supervisor: Jakob Mass, MSc

Tartu 2020

Generating Process-based Mobile Applications for the Internet of Things using Automated Planning

Abstract:

Smartphone devices are being used by more than half of the world population, and this means more opportunities to create mobile applications that help people with their daily lives. This thesis is looking into mobile apps for the Internet of Things, which are used in areas like smart homes, transportation, and healthcare. However, because of the massive scale of smart devices, supporting all of them is not feasible. Automated planning can help the application adapt to user's and device's context and support only those IoT devices which are needed by creating user-specific plans. These plans can be mapped into a business process model so the mobile application could execute them by using a business process engine. The goal of this thesis is to investigate and develop a framework that enables creating dynamic IoT mobile applications, using automated planning and business software management while taking into account user's preferences and mobile device capabilities. Furthermore, it is analyzed which type of planning algorithm fits best for the motivating scenario. A framework prototype consisting of mobile application and backend is created for the motivating scenario is created as a proof of concept. The performance and scalability of the chosen planning algorithm and the developed prototype are evaluated.

Keywords:
Automated Planning, Internet of Things, Business Process Management, Activiti, Android

CERCS: P170 Computer science, numerical analysis, systems, control

Protssesipõhiste mobiilirakenduste loomine asjade interneti jaoks kasutades automatiseeritud planeerimist

Lühikokkuvõte:

Rohkem kui pool maailma elanikkonnast kasutab nutiseadmeid ja see annab üha rohkem võimalusi luua mobiilirakendusi, mis aitavad inimesi nende igapäevaeluga. Käesolevas lõputöös uuritakse asjade Interneti mobiilirakendusi, mida kasutatakse sellistes valdkondades nagu nutikad kodud, transport ja tervishoid. Nutiseadmete tohutu rohkuse tõttu pole mobiilirakenduses kõigi seadmete toetamine siiski teostatav. Automatiseeritud planeerimine aitab rakendustel kohanduda kasutaja ja seadme kontekstiga ning toetada ainult neid nutiseadmeid, mida kasutajal antud kontekstis vaja läheb, luues kasutajapõhiseid plaane. Need plaanid kaardistatakse äriprotsessi mudeliks, nii et mobiilirakendus saab neid oma äriprotsessimootori abil käivitada. Käesoleva töö eesmärk on uurida ja välja töötada raamistik, mis võimaldab luua dünaamilisi nutistu mobiilirakendusi, kasutades automatiseeritud planeerimist ja äritarkvara haldamist. Rakenduse genereerimisel võetakse arvesse kasutaja eelistusi ja mobiiliseadme võimalusi. Analüüsitakse, milline planeerimisalgorithm sobib motiveeriva stsenaariumi rakendamiseks. Kontseptsiooni tõestuseks luuakse motiveeriva stsenaariumi jaoks mobiilsest rakendusest ja taustaprogrammist koosnev raamprototüüp. Väljatöötatud prototüüpi ning valitud planeerimisalgoritmi hinnatakse jõudluse ja skaleeruvuse põhjal.

Võtmesõnad:

Automatiseeritud planeerimine, Asjade internet, Äriprotsesside juhtimine, Activiti, Android

CERCS: P170 Arvutiteadus, arvutusmeetodid, süsteemid, juhtimine (automaatjuhtimisteooria)

Contents

1 Introduction

Mobile devices are being used by more than half of the world population nowadays; in 2019, it was estimated that they were used by 6.8 billion people [32]. Having more mobile devices also means more opportunities for creating applications to help people with their daily lives and work. One of the examples is the Internet of Things (IoT) which with the use of mobile applications significantly benefits areas like smart home, transportation and healthcare. The IoT applications can interact with different smart devices, like a heart rate sensor [3].

On the other hand, creating mobile applications takes much time, effort and outsourcing app's development costs a lot. For example, taking into account multiple surveys, developing a mobile app costs most often from $100,000 to $500,000 [25]. When the app's requirements change, making changes to the code is costly and takes time.

More challenges arise when these applications should also interact with IoT. One of the challenges is the massive scaling of smart devices. It is estimated that eventually there will be trillions of devices on the Internet, but this proposes a question on how to support, access and maintain such a large scale of devices [31]. When creating a mobile app, it is not feasible to add support for all devices but rather the application should adapt to the user's and device's context and support only those devices which are needed.

This thesis is looking into creating a smart application that can be easily changed when the user's context and IoT environment change. The application is realized by a motivating scenario that is introduced next.

1.1 Motivating Scenario

Anna is driving her car, and suddenly one of the tires break. She knows she has an extra tire in the car, but she has never changed one. Anna takes her smartphone and downloads an application that will assist her with the tire change. After opening the app, it asks her what kind of car she has and if she allows the app to use Bluetooth. The app shows her video clips on how to change a tire step by step. After the tire is changed, the tire's pressure needs to be checked. Since the car has Bluetooth capability, the app connects with the car and receives the tire pressure information. Next time Anna is driving a different car and the tire breaks again. She opens the app, fills in the information about her car and checks the "battery-saving" option because her phone does not have much battery left. The application shows her images instead of videos on how to change the tire. Since the car does not have Bluetooth capability, the app gives her instructions on how to check the tire pressure manually.

1.2 Proposed Solution

The presented motivating scenario raises multiple challenges. Firstly, the process of replacing the tire can vary for different types of cars. Instead of handling all of the cases inside the application, the app must be able to create different views based on the given context dynamically.

Second, the motivating scenario highlights the problem of supporting the massive scale of smart devices. The application connects to the smart car via Bluetooth, but various cars can have IoT devices by different vendors. Different devices may require separate implementations of accessing the device. This proposes a question on how the mobile application can support the massive scale of IoT devices?

The thesis addresses these challenges by creating a framework that can generate an IoT Android application based on the app's context and preferences given by the user. The framework would have a public database that contains a list of application templates, which can be thought as different app scenarios and which use a list of actions that can be executed on the mobile app.

The actions can be either interacting with the smart devices (for example checking the car's tire pressure via Bluetooth) or interacting with the users (for example showing a video on how to remove a tire). The database is shared by various IoT device vendors. The smart device actions define the sub-goal (for example, turning on a light), while also including the device-specific information on how to carry out that goal with the given device (for example, the implementation of connecting to the smart light via Bluetooth).

As mentioned before, one of the biggest challenges in IoT is massive scaling of smart devices, but at the same time, the IoT world is very dynamic. The smart devices may suddenly disappear from the network and new ones might appear. Therefore it is hard to know which devices should be supported before developing an IoT system. One of the solutions is to use automated planning, which can suggest which smart devices to use based on the user's context and list of discovered available devices [1].

Automated planning is a branch of artificial intelligence where the main problem is autonomously selecting what action to do next. It uses a model-based approach where the model consists of an initial state, a list of actions and a goal. A plan in this model takes the initial state and finds a set of actions that turn the initial state into a goal state [12]. The planning models are general; they can solve any planning problem without knowing the context. Furthermore, they are human-comprehensible, which means that they are readily accessible and understandable by IT professionals [21].

The automated planner in the framework will take a list of actions from the database and generates a user-specific plan while taking into account the app's context and user's preferences. When the context changes, the planner can quickly generate a new plan with new requirements. There are a lot of automated planning algorithms and implementations (for example, [6], [16] and [4]), and therefore in this thesis, it will be analyzed, which is the best choice for this project. To visualize the generated plan to the user as a

mobile application, the framework will use a business process engine that executes plans modelled using the Business Process Modelling and Notation standard.

Business Process Management (BPM) is about managing the workflow in the organization, to ensure consistent outcomes and trying to improve the performance of business processes. The processes are chains of events, decisions, and activities that add value to the organization and its customers [8]. Business Process Modelling and Notation (BPMN) helps businesses to understand their internal business processes through a graphical notation easily. In addition to graphical notation, the BPMN standard is also executable, when interpreted by a business process engine [13].

As mentioned before, the planning models are general, and therefore any BPM problem that can be converted into PDDL problem can be solved by the planner [21]. The framework maps the received PDDL plan into an executable business process, and the mobile-embedded process engine interprets it. There are already exists business process engines specifically for IoT mobile applications, for example, WiseWare which enables smart goods monitoring [22].

There are many benefits of using business processes for IoT. The processes can reduce the need for completing manual tasks [18]. The engine allows for offline execution and, for example, any data that is collected from the IoT device can be stored in the mobile device [30]. This means that the mobile application can work offline when it already has the required BPMN plan and can be used in scenarios where the network is not always available.

The framework will be developed while taking into consideration that it could easily be extended to use any application context.

1.3 Objectives

This thesis aims to reach three goals.

1. To investigate and develop a framework that can take a template for an IoT application described as planning problem and use it to generate a mobile application using automated planning and business software management while taking into account user's preferences and mobile device capabilities.

2. To analyze which type of planning algorithm fits best for the motivating scenario.

3. Create and evaluate a framework prototype consisting of mobile application and backend for the motivating scenario as a proof of concept.

1.4 Thesis Outline

The rest of this document is organized as follows. In Section 2 background information about IoT, automated planning, different planning algorithms, business process manage-

ment and notation, and Android is given. Section 3 has an overview of the related works. In Section 4, the implementation of the framework that was developed during this thesis is discussed in detail. In Section 5 the framework is evaluated. The last section concludes this thesis and suggests future work.

2 Background and Related Works

This chapter of the thesis gives an overview of the Internet of Things, automated planning and different planners, business process management and notation, and Android.

2.1 Automated Planning

Automated planning is one of the oldest areas in artificial intelligence which is a model-based approach to select what action to do next autonomously. The first automated planner and one of the first AI programs is General Problem Solver in 1959 [26]. The planning has changed a great deal during the recent years and is now seen as an automated solver for mathematical models. The central challenge in planning is scalability in terms of the planning model size [12].

There is a wide range of models used in planning and they all are variations of a *basic state model*. A Basic state model consists of a known initial state, a list of actions, and a goal. A plan in this model takes the initial state and finds a set of actions that turn the initial state into a goal state. An optimal plan has the minimal sum of action costs compared to other plans. This type of planning is also called classical planning. The models are general, and they can be used for any problem and domain [12]. De-facto standard to represent the planning problem and domain is to use Planning Domain Definition Language (PDDL) [23].

2.1.1 PDDL

Planning Domain Definition Language (PDDL) is used to represent a planning task which consists of a planning problem and domain. PDDL was developed by Drew McDermott and his colleagues in 1998 for AIPS-98 planning competition [23]. In 2003, PDDL 2.1 came out that extended the previous PDDL with metrics, durative actions and functions [10]. In 2008, the latest version PDDL 3.1 allowed defining action costs [19]. A short version of the motivating scenario's (Section 1.1) planning task is used to explain further the contents of the planning problem and domain, and what features of the PDDL the scenario uses.

The short version of the motivating scenario is the following. The user has a mobile app that assists with checking the tire pressure. They have two ways to do it - either doing it manually or getting the information from the car via Bluetooth. If the user did it via Bluetooth, it would be quick, but it would use more phone's battery. If the user did it manually, they would follow the instructions on the phone and check the tire pressure themselves. The instructions would consist of an image and some text which would not drain the phone's battery. At the same time checking, the tire pressure manually takes longer time than doing it via Bluetooth.

Planning Domain. A domain file consists of requirements, types, predicates, functions and actions as can be seen in Listing 1. In the first line, the domain's name is defined. In the example code, the domain name is "checkTirePressureDomain".

```
(define (domain checkTirePressureDomain)
    (:requirements :strips :fluents :durative-actions :action-costs)

    (:types car)

    (:predicates
        (has_car ?c - car)
        (car_is_parked)
        (has_bluetooth)
        (tire_pressure_checked)
    )

    (:functions (total-cost) - number)

    (:action park_car
        :parameters(?c - car)
        :precondition(has_car ?c)
        :effect(car_is_parked)
    )

    (:durative-action check_tire_pressure_manually
        :parameters()
        :duration(= ?duration 4)
        :condition(at start(car_is_parked))
        :effect(and (at end(tire_pressure_checked))
                    (at end(increase (total-cost) 1)))
    )

    (:durative-action check_tire_pressure_bluetooth
        :parameters()
        :duration(= ?duration 1)
        :condition(and (at start(has_bluetooth))
                       (at start(car_is_parked)))
        :effect(and (at end(tire_pressure_checked))
                    (at end(increase (total-cost) 2)))
    )
)
```

Listing 1. PDDL code for a domain file.

PDDL supports many requirements, but not all planners support them all. When the planner reads in the domain file, it can quickly tell from the list of **requirements** if it can handle this domain. In Listing 1, the first requirement is *strips*, which tells the planner that actions will only use positive preconditions and deterministic effects. Next requirement is *fluents*, which allows using functions and objects that are mutable numeric

variables. *Durative-actions* allows defining duration for each action. The duration tells the planner how long the action's execution is. *Action-costs* allow specifying a cost for each action [23, 10, 19].

Types are a list of object types which are used in the planning task. In Listing 1, the planning task is using objects with type *car*. **Predicates** show statements about the objects that can be either true or false. They are used, for example, when defining action conditions and effects. All predicates used in the domain file must be declared here [23]. In Listing 1, the domain has four predicates - *has_car*, *car_is_parked*, *has_bluetooth* and *tire_pressure_checked*. The meaning of these predicates will be explained together with conditions and effects. **Functions** can be used to associate a numeric value to a fluent object [10]. In Listing 1, the domain has a function called *total_cost*, which is a requirement of a metric that is explained in the next paragraph.

Actions are used to either complete or get closer to completing a goal [23]. In the mentioned scenario, the user wants to check the tire pressure, and they can do it in two ways. Therefore the two actions to achieve this goal are *check_tire_pressure_manually* and *check_tire_pressure_bluetooth* as can be seen in Listing 1. The action's **parameters** contain a list of all variables that are used in action definition [23]. In Listing 1, the action *park_car* has a parameter *c* with type *car*.

Action's **duration** shows how long the action's execution is, and the unit is up for the developer. This field is specific for durative actions only [10]. In Listing 1, the duration unit is in minutes. Manually checking the tire pressure takes a longer time, and therefore, it is set as four minutes. Checking the tire pressure via Bluetooth takes less time, and therefore the action has a duration of one minute.

The action has **preconditions** which are a list of predicates that have to be satisfied before the action is executable. If the action has no preconditions, then it is always executable. The preconditions can have predicates with parameters [23]. For example, in Listing 1, the action *park_car* has a predicate *has_car* which requires a parameter with a type *car*. For durative actions, the user can define conditions that apply at the start of the execution and the end of execution. Therefore the field is named **conditions** instead of preconditions [10]. In Listing 1, the checking tire pressure actions have a precondition *car_is_parked* which can be satisfied by executing an action called *park_car* first. For checking the tire pressure via Bluetooth, the action has a condition *has_bluetooth*. If the initial state does not declare that Bluetooth is available, this action cannot be executed.

The action's **effect** shows in what state the plan is after this action is executed [23]. Durative actions can define effects that happen at the start and the end of the execution [10]. In Listing 1, checking the tire pressure actions have an effect *tire_pressure_checked* which means that after executing one of these tasks, the user's goal of checking the tire pressure is completed.

Planning Problem. Problem is what planner tries to solve with a given domain [23]. The problem file consists of a list of objects, the initial state, goal specification and metric to be used when finding a suitable plan as can be seen in Listing 2. In the first line, the problem's name is defined. In the example code, the problem name is "checkTirePressureProblem". The domain used with this planning is defined in the second row. In Listing 1, the previously introduced domain is used.

```
(define (problem checkTirePressureProblem)
    (:domain checkTirePressureDomain)

    (:objects audi - car)

    (:init
        (= (total-cost) 0)
        (has_bluetooth)
        (has_car audi)
    )

    (:goal
        (tire_pressure_checked)
    )

    (:metric minimize (total-time))
)
```

Listing 2. PDDL code for a problem file.

Problem's **objects** define a list of objects used in the planning task [23]. The planning domain (Listing 1) requires an object with a type *car*, and therefore in the planning problem (Listing 2), an object *audi* is defined.

Problem's **init** defines the initial state of the planner. All predicates, which are not explicitly defined in the initial state, are assumed to be false by the PDDL [23]. In Listing 2, the initial state has three predicates. The first initializes the plan's total cost to zero. The second predicate *has_bluetooth* is used to state that the user has allowed the use of Bluetooth, and the planner can use actions that utilize this capability. If this predicate was not defined in the initial state, then the planner would not use the Bluetooth task. The third predicate *has_car audi* matches the predicate's object with an object audi.

The **goal** defines what the planner tries to complete [23]. For the scenario, the user's goal is to check the tire pressure. Therefore in Listing 2, the goal is to change the predicate *tire_pressure_checked* to from false to true.

The **metric** defines on what basis the plan will be evaluated. This specification is optional, and it is supported only by planners that support PDDL 2.1 and up. The two standard metrics are minimizing duration or plan's total cost [10, 19]. When the goal is to minimize the duration, the planning problem must define the metric as *minimize total-time* like in Listing 2. For minimizing the plan's total cost, the planning domain must

include a requirement *action-costs* and a function that increases the mutable numeric variable *total-cost* value. The planning problem must define the metric as *minimize total-cost*. It is also possible to define a metric that minimizes a sum of duration and total plan's action cost [19].

Planning Solution. In conclusion, the planner takes in a planning problem and domain file and tries to complete the goal defined in the problem file. The output is a sequence of actions that turn the initial state to goal state [23].

```
0.002: (park_car) [0.002]
0.010: (check_tire_pressure_bluetooth) [1.000]
;Makespan: 1
;Actions:  2
;Planning time: 0.047
;Total time: 0.047
;3 expanded nodes
```

Listing 3. A solution for PDDL planning task.

Example planning solution as output of TFLAP planner (explained further in Section 2.1.2) is shown in Listing 3. The planner gave a solution where the user would park the car first and then check the tire pressure via Bluetooth.

2.1.2 Planning Algorithms And Implementations

There is a wide range of models used in planning - starting from where the current situation is fully known to planners (like **classical planning**) to where the situation is partially known [12]. Furthermore, the planning algorithms can differ from how the produced plans are ordered. Classical planners produce sequential plans, while **partial-order** planners do not. The plans are ordered partially, which means some actions can be done at the same time or in either order [11]. **Temporal** planners support durative actions [10].

For this framework, the temporal partial-order planning algorithm was chosen. The benefit of having actions partially ordered is that some actions could be done at the same time, and this would save the user's time. From IoT perspective, this would allow, for example, turning on a smart light and showing the user a video on their phone at the same time. The benefit of using durative actions is allowing the creation of more complex plans. From IoT perspective, the plan could have two actions that do the same thing - one with IoT device and other a manual task. The planner would choose the action that would fit the user's preferred metric, just like in the example scenario in the previous section.

There are a lot of readily available implementations of partial-order planning algorithms, and therefore it was reasonable to use one of these instead of writing the

implementation itself. The planners are generally either libraries (for example, PDDL4J library[1]) or command-line programs. The program takes in two files or two strings which are a planning domain and problem. Some programs output plans in the command line, others require a third input, a solution file name, and save the output there. Some programs can also give multiple solutions for the planning problem. The program stops when the best or all possible plans are found or when specific time limit (usually 30 minutes) is reached.

To choose the best planner for the developed framework, a selection of planners are tested out from the International Planning Competition[2]. The three command-line planners chosen from the competition are POPF2, OPTIC and TFLAP.

POPF is a forward-chaining partial order planner which competed in 2010 competition. Forward-chaining partial order planning means when an action is added to the plan, it is ordered after the actions in the plan so far to reach the goal and not after all actions in the plan [6].

POPF2 is a planner built on top of POPF and competed a year later in 2011. Improvements were made to cost-optimisation and therefore, the plans found after the first plan are improved. Furthermore, the planner finds more makespan-efficient plans which mean they have better total execution times. POPF2 supports PDDL 2.1 [6].

OPTIC (Optimizing Preferences and Time-dependent Costs) was used in the 2018 competition as baseline [16]. The planner is built on top of POPF and supports PDDL 3. It was extended by supporting preferences (introduced in PDDL 3) which are soft goals or soft preconditions on actions. Furthermore, the planner supports time-dependent costs [4].

TFLAP is a temporal forward-chaining partial order planner which competed in the 2018 competition [28]. It is based on FLAP2 (partial order forward-chaining planner) which follows the principles of POCL (partial order casual-link planning) paradigm [29]. Partial order causal-link is a planning algorithm where, in order to keep track of temporal constraints, structures known as causal links are used [12]. TFLAP is similar to OPTIC, but it does not add permanent constraints between actions to order them if it is not required. Furthermore, it is a more flexible planner in the sense that it can add new actions at any point of the current plan. TFLAP supports PDDL 3.1 [28]

2.1.3 Choosing Suitable Planner

The three planners - POPF2, OPTIC and TFLAP - are compared by scalability and PDDL feature support to choose the most suitable planner for this framework. The scaling of these planners is evaluated by using Blocks World planning problem.

Blocks World [14] represents a scenario where n number of blocks are on the table (either next to each other or some on top of each other). The goal of the task is to create

[1]PDDL4J - https://github.com/pellierd/pddl4j

[2]ICAPS Competitions - http://icaps-conference.org/index.php/Main/Competitions

a tower with them. It is a well known complex planning problem which is simple for people but not so easy for planners which must accept any planning problem or domain and solve it without any additional knowledge. Furthermore, the Blocks World problem with n blocks is exponential in n because to find the solution the planner would have to look through n! possible towers of blocks with additional combinations of lower towers.

For the experiment, the same Blocks World problem was run on the three planners with blocks varying from 3 to 10. The Blocks World planning problem and domain files are taken from the VHPOP repository in Github[3]. The time, when the first solution was received, was recorded. Both TFLAP and OPTIC planners give out multiple solutions for the planning task. Even though the first solution might not be the best, comparing the times when receiving the first solution still gives a good idea of how well the planners scale.

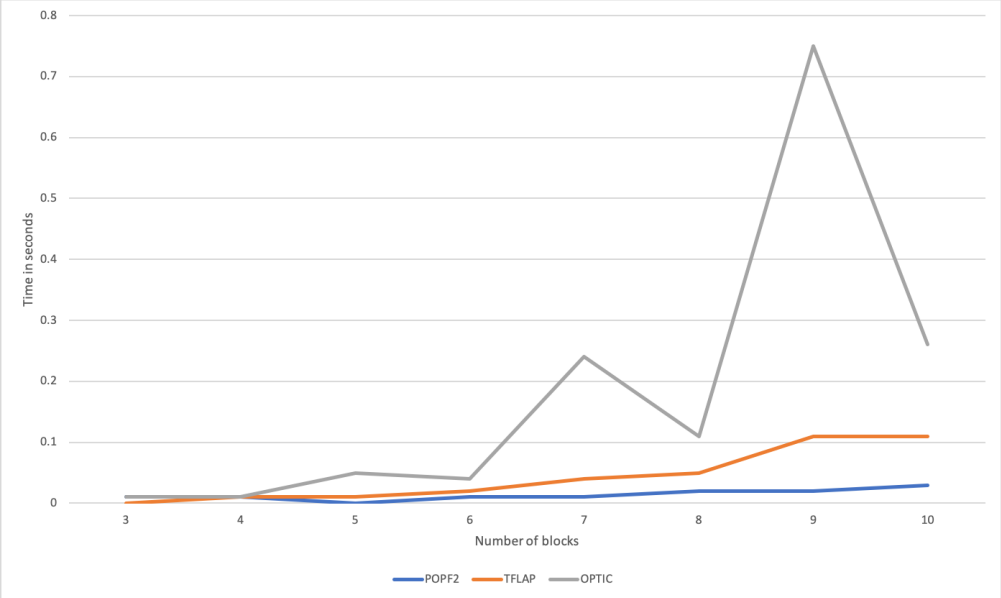

Figure 1. Comparison of the planners with Blocks World.

As can be seen in Figure 1, the best performing planner was POPF2. While POPF2 and TFLAP planner output times were steady, OPTIC output times were fluctuating quite a bit, and it did not perform that well.

For the motivating scenario and to showcase how complicated plans the automated planner could handle, it should support both durative actions and action costs. The planners are run with three different planning tasks. First planning task contains only durative actions, the second supports action costs, and the third one supports them both.

[3]VHPOP Github repository - https://github.com/hlsyounes/vhpop/

It is tested out if the planners give the best possible solution for these tasks.

PDDL features	POPF2	TFLAP	OPTIC
Durative actions	Yes	Yes	Yes
Action costs	No	Yes	No
Durative actions & costs together	No	Yes	No

Table 1. PDDL feature support.

As can be seen in Table 1, only TFLAP supports both durative actions and action costs. The other two, POPF2 and OPTIC, only supported durative actions. POPF2 did not give an error when using action costs and gave a solution, but the solution still did not use an action with a smaller cost even though the planning problem metric was to minimize the cost.

In conclusion, TFLAP deems to be the most suitable planner for this framework. It is faster than OPTIC, and even though it is not as fast as POPF2, it supports durative actions which POPF2 does not.

2.2 Internet of Things

There is a fuzziness around the term "Internet of Things" because syntactically it is composed of two terms. Differences in IoT vision come when the term is looked from either "Internet-oriented" or "Things oriented" perspective. The second perspective means focusing more on generic "objects" to be integrated into a common framework. When these two words are put together, then "Internet of Things" semantically means "a world-wide network of interconnected objects uniquely addressable, based on standard communication protocols" [3].

In IoT, the everyday objects can be sensors, which gather information about their surrounding environment (for example, a thermostat) or actuators which manipulate the environment (for example, turn on the light). Internet connectivity allows them to communicate with digital devices and create complex IoT systems [31]. The smart devices also use technologies like Bluetooth, NFC (Near Field Communication) and RFID (Radio Frequency Identification) [20].

The Internet of Things can be beneficial for different areas, and a lot of research has been done to explore different implementations of IoT in healthcare, environmental (such as smart agriculture and domestic waste treatment monitoring), smart city (such as smart homes and buildings, traffic monitoring), retail and industrial [2].

2.2.1 Automated Planning and IoT

One of the biggest challenges in IoT is massive scaling of smart devices. It is estimated that eventually there will be trillions of devices on the Internet. This proposes a question on how to support, access and maintain such large scale of devices [31].

Furthermore, the IoT world is very dynamic. The smart devices may suddenly disappear from the network and new ones might appear. Therefore it is hard to know which devices should be supported before developing an IoT system. The system should dynamically discover the devices and adapt the system based on the context and user's needs [1].

One of the solutions is to use automated planning. During runtime, the planner can automatically suggest the system which smart devices to use based on the user's context and list of discovered available devices [1].

2.3 Business Process Management

Business Process Management (BPM) is about managing the workflow in the organization, to ensure consistent outcomes and trying to improve the performance of business processes. The processes are chains of events, decisions, and activities that add value to the organization and its customers [8].

2.3.1 Business Process Management Notation

Business Process Modelling and Notation (BPMN) helps businesses to understand their internal business procedures through a graphical notation easily. It is meant for all users, from the business analysts who create the initial draft of the process to the technical developers who will develop the technology to execute the given process. The standard notation includes XML representation, which means these processes are also machine-readable [13].

There are many versions of BPMN, but in this thesis, we will focus on BPMN 2.0 since it is widely accepted standard in academia. BPMN has a small set of graphical notation element categories, so the basic types of elements can be easily recognized in a BPMN diagram [13].

An example process model can be seen in Figure 2. The displayed model is a scenario which is similar to the one discussed in Section 2.1.1. The process starts with the user parking their car. After that, the mobile app connects to the car via Bluetooth and checks the tire pressure. At the same time, it shows information to the user about what is going on. After the tire pressure is checked, the process is over.

The notation has five primary categories of elements, and in this thesis, two of those categories will be used: Flow and Connecting objects [13].

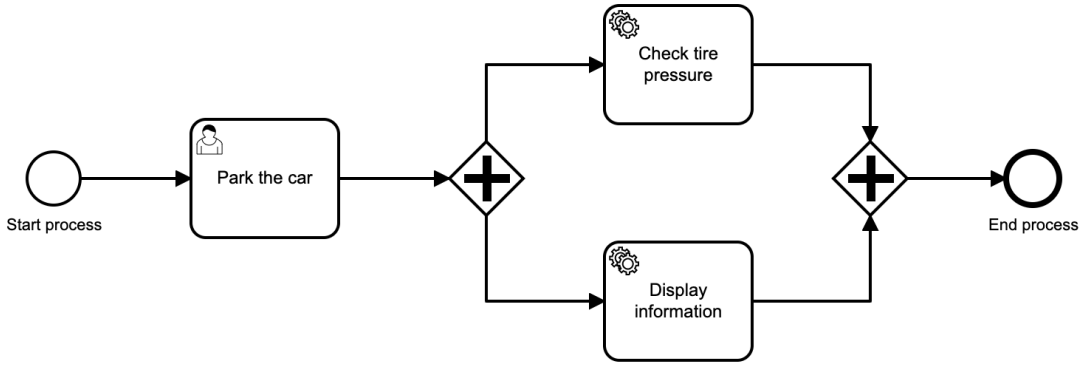

Figure 2. BPMN 2.0 process model example.

Flow Objects. There are three types of flow objects which are the main graphical elements to define process behaviour - events, activities and gateways. An **Event** is something that happens instantaneously. It affects the flow of the model and usually has a trigger and a result. In the thesis, two types of events are used: start and end event [13].

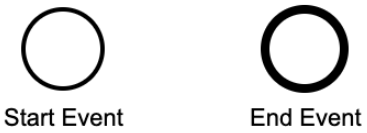

Figure 3. BPMN Event types.

A **Start Event** is an event that starts the business process, such as receiving an order or starting a tour. An **End Event** indicates that the business process will end there (for example, the tire pressure is checked). Events are usually represented as circles (Figure 3). BPMN process model has to have a start and end event [13].

Activities represent units of work that have a duration (for example, a user has to change a tire) and can be seen in Figure 4. A **Task** is an atomic activity and is used when it cannot be broken down into smaller details. It can have different subtypes - user, service, and script task [13].

A **User Task** is used to model work that needs to be done by a human. In Figure 4, the task is indicated by a person icon on the top left corner. A **Script Task** is used to run the snippet of code that is provided for the task. The code can be written either in JavaScript of Groovy. In Figure 4, it is visualized with a script icon. A **Service Task**

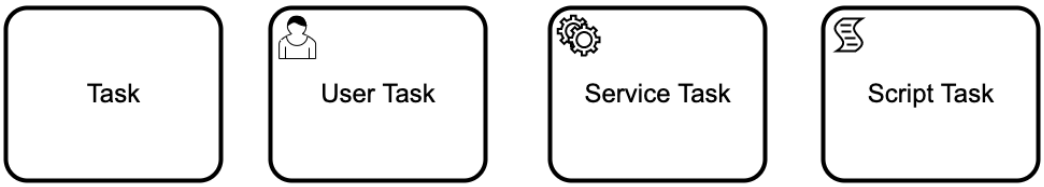

Figure 4. BPMN Task types.

is used to execute an external code which could be a web service or a Java class. For example, it could show a video to the user on the phone or check the tire pressure from the car via Bluetooth. Service tasks are indicated by a wrench icon on the top left corner (Figure 4) [8].

Gateways are control nodes which are represented with diamond shapes. They are connected to activities by arcs and determine the execution paths of the process. There are different types of gateways and one of them is AND gateway (also known as parallel gateway) [8].

The parallel gateway is marked with a plus sign in the diamond shape (Figure 2) and is used in two following situations.

1. When the process path splits into multiple paths at the gateway, then the next paths will be executed in parallel [8]. For example, in Figure 2, after the user has parked the car, the tire pressure is checked and at the same time information is displayed to the user.

2. When multiple process paths join at the gateway, then these paths need to be finished before the process can continue [8]. For example, in Figure 2, the tire pressure has to be checked, and the information to the user has to be displayed before the process can end.

The parallel flows are conceptually present in partial order planning (Section 2.1.2). Because of the partially ordered plan, some tasks could be done at the same time, and therefore it could be modelled as a parallel flow in a BPMN model.

Connecting Objects. Flow objects are connected by Connecting Objects. The main connecting object is a **Sequence Flow**. They are used to show in what order Activities are going to be performed. Sequence flows are indicated by an arrow, as can be seen in Figure 2 [8].

2.3.2 BPMS and Activiti

Business Process Management System (BPMS) is a software suite that helps with automating and the execution of the business process. The tool contains an execution engine (also called process engine), process modelling tool, worklist handler and administration and monitoring tools [8].

One of the business process management systems is Activiti, which is an open-source software platform that can execute and manage business processes defined in BPMN 2.0. It is written in Java and is supported by a team of individuals and companies. The project's leading sponsor is Alfresco, and the project was created in 2010 [17].

The process engine in the BPM systems is responsible for executing a business process model. Instead of using a cloud-hosted process engine, it could also be embedded in the mobile application. For example, WiseWare [22] is using the Activiti process engine, which has been modified to run on Android OS.

The benefits of using a mobile-embedded execution engine are that it allows an offline execution of the deployed process models and storing the collected data on smart devices [30]. Therefore, the application can be used in remote environments where the network is not always available. Furthermore, storing the data in the device allows the data to be private and protected.

2.3.3 BPM for Mobile and IoT

Business process management can significantly benefit from the use of mobile applications. For example, application domains, which need to collect large amounts of data, can reduce human errors by digitalizing the process [30].

Using IoT devices can reduce the need for manual tasks even further. Smart devices have available sensor data which can be used to complete physical and digital tasks such as opening a curtain in a smart home and getting temperature data from a smart thermostat. Using IoT leads to efficiency gains and more accurate data. Smart devices produce a large amount of data that can also be used to analyze and optimize the business process [15, 18].

As suggested in [15], IoT tasks in the business process can be modelled in two ways:

1. BPMN Script Task. The task would contain a script that is executed by the process engine.

2. BPMN Service Task. The task would execute an external Java Delegate class that includes an implementation of the IoT service.

2.4 Android

Android is an open mobile software platform which was developed by Google and Open Handset Alliance [27]. The first official version 1.0 of Android was released on September 28, 2008 [5]. The latest version is Android 10, with API level 29[4]. By December 2019, Android is the leading mobile operating system worldwide by having 74.13 percent of the market [33].

Applications for this software platform can be written in Kotlin, Java and C++. All application code, along with any data and resource files, are compiled with Android SDK tools into an APK, which is an Android package. This package contains all the contents of an Android app and is used to install the app in Android-powered devices [7].

The official Integrated Development Environment for Android is Android Studio. It is developed by JetBrains and is based on the IntelliJ platform [34]. Android projects are built using Gradle[5] build automation tool.

2.5 Related Works

This section gives an overview of the related frameworks that benefit from the use of workflow management (business process management) and automated planning to create web, mobile and IoT applications.

Questionnaire Application. In an article [30], the authors discussed how business processes, which collect a lot of data, benefit from the use of mobile applications. An advanced mobile service that contains a process engine was developed, and it is capable of processing the logic of data collection. This mobile service can be used to create a mobile application that shows to the user a questionnaire, which the user fills out and the app collects the data.

The article introduces fundamental requirements and a generic architecture for this system. Some of the given requirements are relevant to the framework developed as a part of this thesis as well. First is "Integrate sensors". In the article, the process engine has to allow for the integration of sensors (e.g. heart rate sensors). This applies to this framework as well, which has to allow integration of different Internet of Things devices and sensors. The second requirement is "Provide customizable user interfaces". In the article, the process engine should dynamically create the user interface based on its model when running on a mobile device. This also applies to this framework which has to create user interfaces based on process engine tasks dynamically.

The mobile service, which was developed as a part of the article, only focuses on data collection and questionnaire applications. The developed framework in this

[4]Android SDK Platform Release Notes - https://developer.android.com/studio/releases/platforms

[5]Gradle - https://gradle.org/

thesis focuses on having the capability of being used for multiple different scenarios, discovering different IoT devices and adapting to the user's context.

WiseWare. Another article that discusses using business process management and mobile applications together is [22]. The authors focused more on creating an IoT based BPM system that can continue their execution when the device does not have a reliable internet connection. The reason why they used a BPM system was that existing IoT smart devices are often produced by different manufacturers and therefore have different capabilities, standards and protocols. Integrating them into a single software system is complicated. When IoT devices are used as services, then the best option is to use Service Oriented Architecture (SOA)-based middleware and the prominent approach for realizing service composition is Workflow Management Systems (WfMS).

The authors proposed a system design where the business processes can be migrated from device to device while they are being executed, achieving a continuous execution. Furthermore, by using IoT sensors, the system can react to events when they occur and later provide a detailed history of the execution process. The implemented system uses open-source Activiti BPM software which has been adapted to run on Android OS.

This framework will use the same software for its business process execution engine. The WiseWare system cannot be easily used for different scenarios since the mobile application user interface has to be changed every time. The developed framework in this thesis focuses more on supporting different IoT scenarios and dynamically creating user interfaces based on the scenario.

ECo-IoT. In [1], the authors discuss how the IoT world is dynamic and smart devices can suddenly appear or disappear in the network. Therefore the framework should adapt to the changing environment and user's needs.

They proposed a framework called ECo-IoT that queries the user's goal and suggests a list of smart devices to use to complete that goal. The framework monitors the devices while the user is completing their goal.

The framework uses automated planning to describe user goals and find a suitable plan. They use a JavaFF planner which takes in two PDDL files. The prototype used manually defined PDDL files, but in the future the authors want to implement a Context2PDDL which creates PDDL files by translating the user's goals and the context of IoT devices.

When system monitors the IoT devices, while the user is completing their goal, and when the system sees that, for example, some device is gone offline or the battery is too low, then the planner generates a new plan with another suitable device and suggests it to the user.

The created framework is only for monitoring the IoT devices and not for creating mobile applications based on user's preferences. The developed framework as part of this thesis, is using automated planning with business process engine to create mobile

applications based on the select application scenario and user's preferences.

E-Tourism Application. In article [24], a similar framework to the one developed as part of this thesis was created. The authors were interested in composing and executing web services based on the different domains. The system was created using a tourism scenario, where the user fills in what they want to do in the city they visiting (for example, eat in a fancy restaurant) and the system proposes a plan on when and where to do go.

With the user's query and a list of service descriptions, the framework would use AI-planning to find a sequence or parallel plan. This plan would then be transformed into a BPMN model which would be executed using on a BPMN engine.

The framework's uses Fluent Calculus to describe the planning tasks, and FLUX (Fluent Executor) planning method to find a plan. The found plan is converted into BPMN model based on the Prolog language.

The created framework focuses more on web services while the framework developed as part of this thesis, focuses more on creating user-specific applications specifically on mobile and supporting the large scale of IoT devices.

3 System Design and Implementation

This section gives an overview of the system design and implementation. First, the main functional requirements are listed down for the system being developed. Second, a general overview of the system architecture is given, and then in subsections, every part of the architecture is discussed in detail.

3.1 Functional Requirements

The first objective of this thesis was to investigate and develop a framework that can take a template for an IoT application described as a planning problem and use it to generate a mobile application. It is done by using automated planning and business software management while taking into account the user's preferences and mobile device capabilities. Here is listed the main functional requirements for this system:

1. The framework must create a PDDL domain and problem file from any given scenario.
2. The framework must filter out planning actions that the user cannot use based on their preferences.
3. The framework must allow the user to choose the metric for the planner (Section 2.1.1).
4. The framework must be able to map any PDDL plan to an executable BPMN plan based on a repository of PDDL to BPMN actions.
5. The framework must execute the generated BPMN plan as a mobile application.
6. The framework must allow offline execution of the BPMN plan (Section 2.3.2).
7. The framework must dynamically create user interfaces based on the process engine task.
8. The framework shall allow for the integration of IoT sensors (Section 2.5).

Discussion. The reasons for choosing some of the functional requirements are talked about in further detail next.

The second functional requirement is to filter out planning actions based on user's preferences. As mentioned in Section 2.1, the main challenge in planning is scalability in terms of the planning model size. The planner can have a lot of smart devices to choose from because of the large scale of IoT devices, mentioned in Section 2.2.1. When

filtering out the devices, which cannot be used based on the user's needs, this reduces the planning model size significantly.

The fourth requirement is for the framework to use the repository, where multiple organizations provide their actions and map the PDDL plan to an executable BPMN plan. The seventh functional requirement is to dynamically create user interfaces based on Android activity classes attached to BPMN service tasks (Section 2.5 and Section 2.3.1).

3.2 System Architecture

The system consists of three separate components - database, mobile application and process planner. These components communicate with each other and make up one framework. A general overview of the framework can be seen in Figure 5.

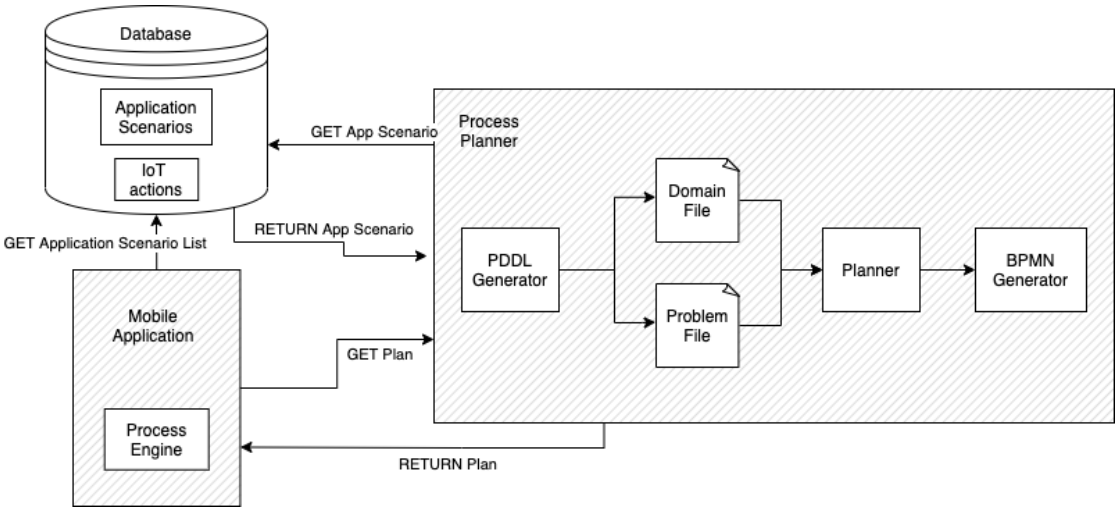

Figure 5. System architecture.

The first component is the **Database**, which consists of different application templates that are used for creating user-specific apps. A template defines what type of app it is (for example tire change, smart city or package delivery application) by the problem definition and the list of actions. The problem definition describes the PDDL planning problem, and the PDDL planning domain is put together using actions. Each action has a name and list of preconditions and effects, which are used to create a PDDL action. The action also has BPMN snippet attached to it, which is used in transforming the plan into a valid BPMN model. The snippet can contain Java class code paths that are used in the mobile application. In Section 3.3, an in-depth description of the database and the application templates are given.

The database also contains a list of available standalone actions that are created by various IoT device vendors. These actions, while having a sub-goal (for example,

turning on a light), also include device-specific implementation details (for example, turning on a specific Bluetooth light bulb) in the Java class attached to the BPMN snippet. When creating application templates, the creator can choose actions from the list of these standalone IoT actions. The database is public, and the templates and actions can be added by an HTTP POST request.

The second component is the **Mobile application**. When creating an application that uses this framework, a developer needs to set up the basic Android mobile application project with some minimal development. The app must know the application template identifier in order for the process planner to get the template from the database. When the user opens the app, they need to fill out a questionnaire about their preferences. Asking to fill a questionnaire is optional and needs to be created by the developer. The preferences can be about anything regarding this application. For example, in the motivating tire change scenario, the user needs to fill out what type of car they are using, if they allow the use of Bluetooth and whether the plan should be optimized to save phone's battery or not. The mobile application can also gather information about the system configuration. The app can check the battery levels or the OS version and combine this information with the user's preferences.

The mobile app sends the application template identifier, list of user preferences and a PDDL planning metric to the process planner by an HTTP POST request. Sending the metric is optional. The process planner responds with a BPMN model which is then executed by the process engine. The engine tells the application what task to do next and what view to show to the user when the BPMN task has a Java class attached to it.

The third component is the **Process Planner**, which consists of three separate modules and is responsible for generating user-specific BPMN model for the application. This can also be thought as the framework's backend. The first module, **PDDL Generator**, uses an application template identifier to get the template data from the database. While taking into account the user's preferences, the template's list of actions is filtered by removing which are not needed. For example, when the user specified not to use Bluetooth, all actions that require Bluetooth are filtered out. The PDDL Generator creates two files for the automated planner - a PDDL planning problem and domain file.

The PDDL generator feeds the problem and domain file into the second module - the **Planner**. The automated planner finds the most optimal plan by using the metric specified by the mobile application. If the app did not specify a metric, then by default, it uses minimizing plan's cost metric.

The third module is the **BPMN Generator**, which is responsible for creating a business process plan. It uses the output of the automated planner, and for every PDDL action in the plan, the generator gets a BPMN snippet from the application template. All of the snippets are put together as a complete BPMN model. This BPMN file is sent back to the application as an HTTP POST response.

The developed framework is accessible from two the Bitbucket repositories - the

process planner[6] and the mobile application with motivating scenario[7].

3.3 Database

The database contains application templates and a list of standalone available IoT actions. For the developed prototype of the framework, the database was mocked as a JSON file and only included the list of application templates. To implement this database, one could use, for example, Firebase Realtime Database[8], which is a NoSQL cloud database and where the data is stored in JSON format. The proposed structure of the database is given in the next section. Furthermore, the developed database uses PDDL predicates, which are written in a specific format and described in detail in the last section.

3.3.1 Database Structure

The general structure of the file can be seen in Listing 4. The JSON contains an attribute *apps* that contains the list of different application templates. In the developed framework, the process planner contains the database JSON file and reads it in when the mobile application asks for a user-specific plan.

```
{
    "apps": [
        {
            "id": "check_tire_pressure",
            "name": "Check tire pressure",
            "problem": { ... },
            "actions": [ ... ]
        }, ...
    ],
}
```

Listing 4. General structure of the application templates file.

The **application template** object can be seen in Listing 4. For the examples, the modified motivating scenario (mentioned in Section 2.1.1) is used, because the motivating scenario's JSON file is quite large and the scenario does not use all of the framework's features. The template object contains four attributes - *id*, *name*, *problem* and *actions*. The mobile app sends the template's *id* to the process planner, which will get the correct application template from the JSON file using this *id*. The attribute *name* is used by the app to show the generated plan's name.

The attribute ***problem*** describes the application template's problem definition. It consists of three attributes - *objects*, *init* and *goal* (Listing 5). They are used to create

[6]Process Planner Repository - https://bitbucket.org/kelian/thesismobilebeproject/src/master/

[7]Mobile Application Repository - https://bitbucket.org/kelian/thesismobile project/src/master/

[8]Firebase Realtime Database - https://firebase.google.com/docs/database

a PDDL problem file. *Objects* is an attribute that may contain zero to multiple keys inside and their value is a string list. The attribute *init* describes the initial state, and *goal* describes the goal state, and they contain a string list of predicates. The predicates in the database are written in a specific format. The format is described in detail in the next section.

```
"problem": {
    "objects": {
        "car": ["audi"]
    },
    "init": ["has_car audi"],
    "goal": ["tire_pressure_checked"]
}
```

Listing 5. Problem object for check tire pressure scenario in JSON file.

In Listing 5, an example of the problem attribute for the check tire pressure scenario (Section 2.1.1) is given. The *object* attribute contains one type of object called "car" which has one instance - "audi". The attribute *init* describes the initial state - the user has an Audi car. The *goal* of the scenario is to finish checking the car's tire pressure.

The attribute ***actions*** contains a list of actions which are needed to create planning actions for planning domain file in the PDDL Generator and later in BPMN Generator to get BPMN snippet for every planning step. The action object has seven attributes - *name*, *duration*, *conditions*, *effects*, *totalCost*, *requirements* and *bpmn* (Listing 6).

The *name* is used to give the action a name in the PDDL file and since the planner does not handle action names with spaces then snake case is preferred. The attributes *duration*, *conditions* and *effects* are used to create the PDDL action. The last two contain a string list of predicates. The *totalCost* defines the action's cost and in PDDL Generator it is converted into an effect predicate (Section 3.6). The attribute *requirements* is used by PDDL Generator when filtering the actions (Section 3.6). The object also has attribute *bpmn* which contains the BPMN snippet for this action. The snippet is a XML code which should be executable by Activiti engine. It is used in the BPMN Generator which is discussed in detail in Section 3.7.1.

```
{
    "name": "park_car",
    "duration": 1,
    "conditions": ["has_car ?car-audi"],
    "effects": ["car_is_parked"],
    "totalCost": 1,
    "requirements": [],
    "bpmn": "..."
},
{
    "name": "check_tire_pressure_bluetooth",
    "duration": 1,
```

```
    "conditions": ["car_is_parked"],
    "effects": ["tire_pressure_checked"],
    "totalCost": 2,
    "requirements": ["has_bluetooth"],
    "bpmn": "..."
},
{
    "name": "check_tire_pressure_manually",
    "duration": 4,
    "conditions": ["car_is_parked"],
    "effects": ["tire_pressure_checked"],
    "totalCost": 1,
    "requirements": [],
    "bpmn": "..."
}
```

Listing 6. Action objects for check tire pressure scenario in JSON file.

In Listing 6 an example of the action attribute for the check tire pressure scenario is given. The action named "check_tire_pressure_bluetooth" can only be done if car is parked and if requirement "has_bluetooth" is satisfied. After the action is done then tire pressure has been checked.

The list of standalone IoT actions in the database would follow the same format as *actions* attribute for application template.

3.3.2 Predicate Format

Predicates are used in multiple places in the application template. They are written in a specific format because this makes it easier for the PDDL Generator to parse them and actions written by different people would follow the same style. An example of predicate format can be seen in Figure 6.

```
name ?paraType-paraName ...
```

Figure 6. Predicate format example.

The *name* is replaced with the predicate's name. This has to be written in snake case because the automated planner does not handle predicates that have spaces in their names. For example in Listing 6 the action has a condition predicate *car_is_parked*.

The predicate can also have zero to many parameters and they are written after the predicate name and separated by space. Every parameter starts with a question mark, followed by the parameter's type name (*paraType*) and the parameter's name is separated with a hyphen (*paraName*). For example, in Listing 6, the action named *park_car* has condition *has_car ?car-audi*. The predicate *has_car* has a parameter named *audi* with a type *car*.

3.4 Mobile Application

A basic Android mobile application needs to be set up by the developer with some minimal development. The application must include the business process engine and a networking layer to communicate with the database and process planner.

When the user opens up the app, the mobile application gathers some data about the user context and it can be done in two ways. First, the app can ask the user to fill in a questionnaire about their preferences, which can be anything regarding the desired scenario. For example, in the motivating tire change scenario (Section 1.1), the user needs to fill in what kind of car they are using. The second way to gather information is to check the system configuration. The application can check the device's battery levels or if the Bluetooth is turned on. The user can also choose what kind of metric is used to create the user specific app, but this is optional.

The mobile application makes an HTTP POST request to the process planner and sends the application template's id, the questionnaire answers and system configuration data together as user's preferences and the metric, if the user specified one.

Mobile Application Prototype. The developed framework's mobile application supports two scenarios to showcase that the framework and the application can handle different scenarios. First is an example Hello World scenario and the other is motivating tire change scenario (Section 1.1) as can be seen in Figure 7.

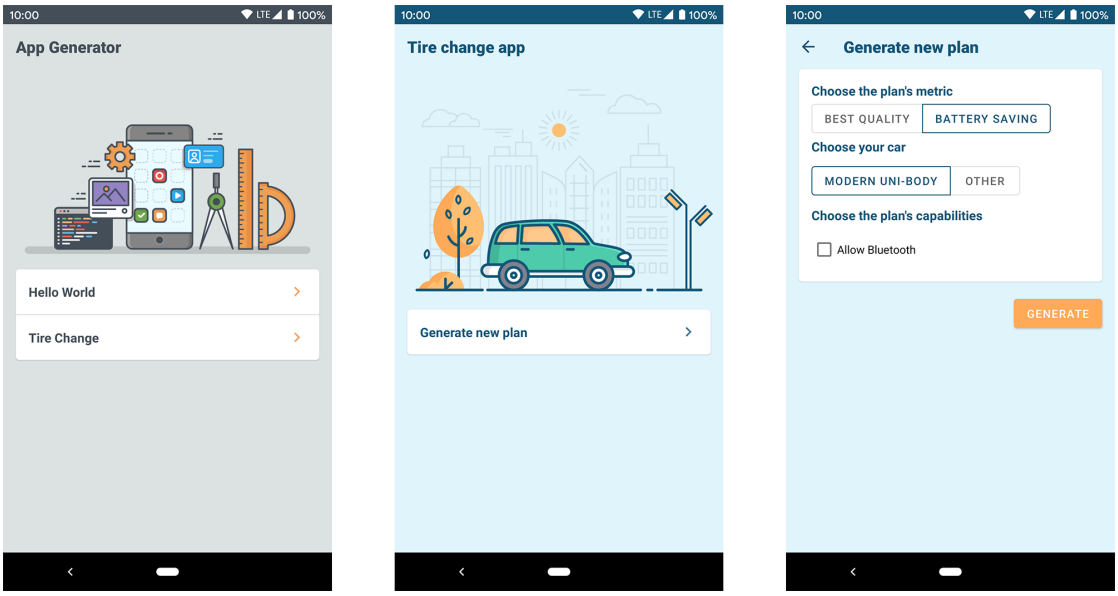

Figure 7. Screenshots of the mobile application prototype.

The user starts their flow by opening the app and choosing one of these two scenarios.

After, for example, choosing the tire change scenario, the user can generate a new plan. The mobile application asks the user their preferences about what kind of car they have and if they would like a battery-saving plan or best quality plan (Figure 7). The battery-saving plan means using a metric to minimize the cost of the plan. The best quality plan is using a metric to minimize the duration of the plan.

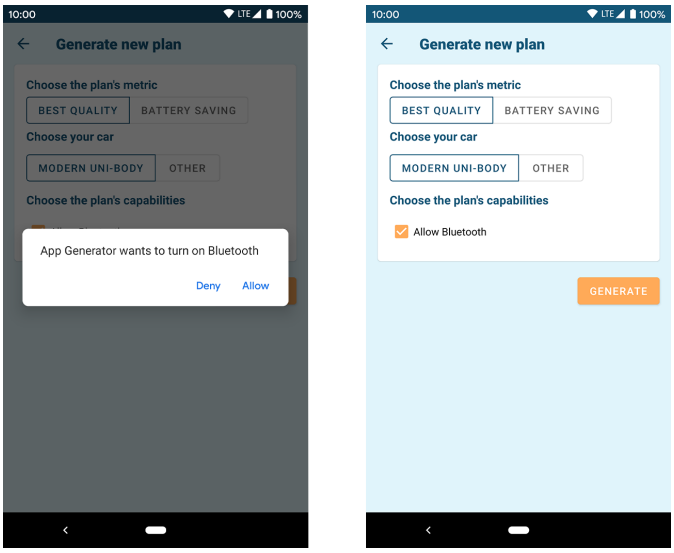

Figure 8. Screenshots of selecting the Bluetooth capability.

The user can also select if they allow the application to use Bluetooth as can be seen in Figure 8. When the device has not Bluetooth available, the application asks from the user if the Bluetooth could be turned on. If the user allows then checkbox is selected.

After the user selects the button "Generate", the mobile application makes a request to the process planner and waits for an answer. If the request is not successful, then error is shown to the user. If the request is successful, then the view is closed and the user is taken back to previous screen. The logic of showing the generated mobile application is discussed in detail in Section 3.8.

The Android Activity class that is responsible for asking user's preferences and getting a generated plan from the process planner is extending a class called *GeneratePlanActivity*. This Activity class is responsible for making the request to process planner and handling the response. When another developer is creating their own application, they can extend this same class.

3.5 Process Planner

The process planner is a Java Spring application that is running on Docker[9] as a containerized app. The perks of using a Docker container is that all of the code and its packages are packed together as a standard unit of software and therefore it runs on any environment quickly and reliably.

```
1 FROM openjdk:8-jdk-alpine
2 VOLUME /tmp
3 COPY build/libs/be-0.0.1-SNAPSHOT.jar be-0.0.1-SNAPSHOT.jar
4 COPY tflap /tflap
5 RUN apk update && apk add make && apk add g++ && mkdir files && cd
      tflap/src && make clean && make all
6 ENTRYPOINT ["java", "-jar", "/be-0.0.1-SNAPSHOT.jar"]
```

Listing 7. Process Planner Dockerfile.

The Dockerfile used to create a Docker container can be seen in Listing 7. The container is built on top of OpenJDK 8[10], which is an official open-source implementation of the Java Platform. The Docker container also uses volume, where all of the temporary files are generated. The process planner project is called "be" and it is packaged into a jar file that is copied to the container together with the TFLAP source code. The Docker container runs the commands to add dependencies "make" and "g++" for the planner. Next, it will build the TFLAP project. Lastly, the container is configured to run as a Java Spring Boot executable.

The process planner project consists of five modules - *src*, *common*, *PDDLGenerator*, *BPMNGenerator* and *tflap*. The *src* module contains a RESTful Web Service, which is implemented by using Java Sprint Boot[11]. The *common* module contains data models that used throughout the process planner.

The current prototype of the framework implements HTTP POST request, which can be seen in Figure 9. The request body consists of *appId*, which is application template's id, *requirements*, which is a list of user's preferences, and *metric*, which is used in automated planning and this is a nullable field.

The HTTP request's response body consists of *bpmnResourceName*, which is the name of the generated model, and *bpmnFile*, which is the BPMN model.

When the mobile application makes the HTTP POST request, the process planner starts by reading in the mocked database JSON file (Section 3.3.1) and tries to find the application template using the *appId*. If the template is not found, then the HTTP POST request responds with an error.

When the planner finds the correct application template, it will continue with generating the plan based on the user preferences. First it will run the PDDL Generator

[9]Docker - https://www.docker.com/

[10]OpenJDK Docker Image - https://hub.docker.com/_/openjdk

[11]Java Spring Boot - https://spring.io/projects/spring-boot

```
@POST("plan")
Request:
{
    "appId": "tire_change",
    "requirements": ["has_bluetooth"],
    "metric": null
}
Response:
{
    "bpmnResourceName": "Tire Change",
    "bpmnFile": "<?xml..."
}
```

Figure 9. HTTP POST request for getting the plan.

that creates the PDDL files for the planner. Then planner is executed with the generated files. Lastly, BPMN Generator turns the planner's generated planning solution into a BPMN model. If any time during this flow something goes wrong, the POST request will respond with an error.

3.6 PDDL Generator and Planner

The PDDL Generator takes in as input the application template, user's preferences and the planner metric and returns a PDDL solution, as can be seen in Listing 8.

```
1  public PDDLGenerator() { }
2  public String create(ApplicationTemplate app, List<String>
       preferences, String metric) throws Exception {
3
4      List<Action> fa = new ActionFilter().create(app.getActions(),
           preferences);
5
6      PlanningTask pt = new PlanningTask();
7      pt = new DomainGenerator().create(pt, fa);
8      pt = new ProblemGenerator().create(pt, app.getProblem());
9
10     String domain = new DomainFileWriter().write(app.getId(), pt);
11     String problem = new ProblemFileWriter().write(app.getId(), pt,
           metric);
12     return new PlanGenerator().create(domain, problem);
13 }
```

Listing 8. PDDLGenerator Java class.

First, the planner filters out unnecessary application template's actions based on the user's preferences. Then it generates the planning task's domain and problem. After that, it creates the PDDL domain and problem file. Finally, the *PlanGenerator* executes the TFLAP planner with the created files and returns the received plan.

Action Filtering. As mentioned in Section 2.1, the main challenge in planning is scalability in terms of the planning model size. The planner can have a lot of smart devices to choose from because of the large scale of IoT devices, mentioned in Section 2.2.1. When filtering out the devices, which cannot be used based on the user's needs, this reduces the planning model size significantly.

Every action in the application template has a list of requirements (Section 3.3.1). The mobile application asks the user to fill out a questionnaire about their preferences. These preferences together with some system configuration data are sent to the process planner in the same structure as they are used in action's list of requirements. If the the action has a requirement that is not in the user's preferences, this action is filtered out.

For example, the action *check_tire_pressure_bluetooth* in Listing 6 in Section 3.3.1 has a requirement *has_bluetooth*, which means this action needs access to Bluetooth in order to complete its job. The mobile application checks if the Bluetooth is turned on or asks the user to turn it on, as mentioned in Section 3.4. If the user allows using Bluetooth, the application sends *has_bluetooth* as user preference and the action is not filtered out. If the user does not allow Bluetooth, the preference is not sent and the action is filtered out.

Domain Generator. After the application template's actions have been filtered, a planning task is put together. First, the planning domain definition is generated.

Every template action in the filtered list is analyzed. If the template action has no conditions specified, the generator adds a predicate called *no_conditions* to action's conditions list. The reason for it is explained later in this section in the planner paragraph.

The template action's conditions and effects are a list of predicates that are analyzed. The generator adds every predicate found in the conditions and effects to the list of planning task predicates. If the predicate has parameters, the generator adds them to the planning action's parameters list. The parameter's type is added to the planning task types list and the type's object to planning task objects list.

Problem Generator. After the domain data for the PDDL file is generated, the Problem Generator creates the data for the problem file by using the problem definition (Section 3.3.1) of the application template. The generator adds the problem's objects, initial state and goal state to the planning task data.

PDDL File Writer. After the planning task data has been generated, the PDDL Generator creates the domain and problem files for the planner.

First, the domain file is created. The planning domain's name is application template's id joined with "_domain". The planning task domain's requirements are strips, typing, fluents, action costs and durative actions (Section 2.1.1). After that planning task's types and predicates are added to the domain file. Domain file writer also adds the action cost function (Section 2.1.1) to the domain file.

All of the actions in the file are created as durative actions. All of the condition predicates will be done at the start of the action's execution and effect predicates will be done at the end of the action's execution. The file writer also adds the increase of total cost to the list of effect predicates. The writer adds the necessary brackets around the predicates and if there are more than one predicate for the conditions, effects or goal state, the writer adds the "and" keyword in front of them.

After the domain, the problem file is created. The planning problem's name is application template's id joined with "_problem". The problem file writer adds planning task's objects and the initial state. The generator also adds to the initial state that plan's total cost is zero.

After the planning task's goals are added to the file, the file writer adds the metric based on the user's preference. If no metric was specified or the metric is "min_cost" then the planer will try to minimize the total cost. If the user specified the metric to be "min_duration" then the planner will try to minimize the duration. If the metric is "min_both", the planner will try to minimize the sum of duration and total cost.

```
1  (define (domain check_tire_pressure_domain)
2      (:requirements :strips :typing :fluents :action-costs :durative-
           actions)
3
4      (:types car - object
5      )
6
7      (:predicates
8          (has_car ?car - car)
9          (car_is_parked)
10         (tire_pressure_checked)
11     )
12
13     (:functions (total-cost) - number)
14
15     (:durative-action park_car
16         :parameters(?car_audi - car)
17         :duration(= ?duration 1)
18         :condition(at start(has_car ?car_audi))
19         :effect(and (at end(car_is_parked)) (at end(increase (total-
               cost) 1)))
20     )
```

```
21
22      (:durative-action check_tire_pressure_manually
23          :parameters()
24          :duration(= ?duration 4)
25          :condition(at start(car_is_parked))
26          :effect(and (at end(tire_pressure_checked)) (at end(increase
                (total-cost) 1)))
27      )
28  )
```

Listing 9. Check tire pressure generated problem file.

```
1  (define (problem check_tire_pressure_problem)
2      (:domain check_tire_pressure_domain)
3
4      (:objects
5          audi - car
6      )
7
8      (:init
9          (= (total-cost) 0)
10         (has_car audi)
11     )
12
13     (:goal
14         (tire_pressure_checked)
15     )
16
17     (:metric minimize (total-cost))
18
19 )
```

Listing 10. Check tire pressure generated domain file.

The generated domain and problem file for the check tire pressure JSON file mentioned in Section 3.3.1 can be seen in Listing 9 and Listing 10.

Lastly, the PDDL Generator creates a folder in the root called "files", where the domain and problem files are written.

Planner. After the PDDL files have been generated, the process planner runs the TFLAP planner on the command line in the Docker container. The TFLAP takes as an input domain, problem and solution file path.

TFLAP planner has few quirks that the PDDL Generator handles. The planner does not accept plans that have no metric and actions with no conditions. Therefore the generator adds the predicate "no_condition" to the actions that have no conditions defined. This predicate is also added to the initial state so these actions can be executed immediately as they would have no conditions.

The process planner waits for five seconds and checks if there are any solution files generated by the TFLAP planner. This time limit is considered as the maximum time the user would have to wait. If the planner did not generate any solution files, the PDDL Generator throws an exception and the HTTP request responds with an error. If the planner generated solution files, the last solution file is read in.

```
0.002: (park_car audi) [1.000]
1.010: (check_tire_pressure_manually) [4.000]
;Makespan: 5
;Actions:  2
;Planning time: 0.11
;Total time: 0.11
;2 expanded nodes
```

Listing 11. Check tire pressure generated solution file.

The solution for the generated domain and problem file can be seen in Listing 11. The read in solution file is sent to the BPMN Generator, and the created domain and problem file and the generated solution files are deleted.

3.7 BPMN Generator

The BPMN Generator takes in the generated plan and maps it into BPMN plan using BPMN snippets which are attached to the application template's actions. First, an in-depth view of the BPMN snippet and the requirements of creating it is given. Then, the BPMN Generator's implementation is discussed.

3.7.1 BPMN Snippet Requirements

Each action in application template has attributes that describe the PDDL action and a BPMN snippet, as mentioned in Section 3.3.1. In order to map the planner's output easily into a BPMN model, the snippet has some requirements:

1. The snippet must be created using Activiti BPMN XML standards[12].
2. The snippet must be a valid BPMN plan.
3. The snippet must have only one Start and End Event.

For this thesis, all of the snippets were created using Activiti Designer plugin for Intellij Idea[13]. The framework supports all BPMN elements that are supported by the Activiti Android process engine.

[12] Activiti XML representation - https://www.activiti.org/userguide/#bpmnFirstExampleXml
[13] Activiti Designer plugin for Intellij - https://plugins.jetbrains.com/plugin/7429-actibpm

3.7.2 BPMN Generator

The BPMN Generator takes in as input the generated plan and the application template, and returns a BPMN model, as can be seen in Listing 12.

```
1  public BPMNGenerator() { }
2
3  public String create(String solution, ApplicationTemplate app) throws
       Exception {
4
5      Map<Float, List<String>> pddlPlan = new PDDLParser().getPDDLPlan(
           solution);
6
7      Map<Float, List<BPMN>> actionsList = new PlanActionsBPMNParser().
           getActionsBPMN(app, pddlPlan);
8
9      BPMN planBPMN = new PlanBPMNFileGenerator().create(app,
           actionsList);
10
11     return new BPMNXMLWriter(planBPMN).create();
12 }
```

Listing 12. BPMNGenerator Java class.

First, the generator parses the TFLAP planner's output into a map that contains a list of PDDL actions for every planning step. Then, the generator maps the PDDL actions to BPMN actions. Third, the list of BPMN actions are joined into a one BPMN model. Finally the BPMN model is converted into executable Activiti XML format.

Mapping PDDL Action to BPMN Action. After the generator has parsed the planner's output into a map that contains a list of PDDL actions for every step, the map is given as an input together with application template to the "PlanActionsBPMNParser".

For every PDDL action, the generator tries to find the action's BPMN snippet from the application template. If no snippet is found, this PDDL action is ignored. The BPMN snippet is parsed into a BPMN data model. After all of the actions have been mapped into a BPMN model, the list of models is turned into a one unified BPMN model.

Creating a BPMN Model. The "PlanBPMNFileGenerator" takes as an input the application template and the map that contains a list of BPMN models for every step. The generator starts by parsing a hard-coded base BPMN model.

The base model consists of only two events - Start and End event. The Start Event is always with the id 2 and End Event with id 3. After reading in the base model, the generator adds the list of BPMN models between the base model's two events.

The unified BPMN model's id and name are created by using application template's id and name and adding the list of preferences and metric in order to differentiate generated

plans in the mobile application (Section 3.8).

If a step in the BPMN map has a list that contains more than one BPMN model, then these are added to the plan in parallel. The generator adds a Parallel Gateway before and after the BPMN actions.

In order to prevent the unified BPMN model having elements with the same id, for every BPMN model in the list, the generator changes the id's by adding a prefix that is incremented for every model. Furthermore, all of the x and y coordinates for every item in the BPMN models are recalculated. This means that the developed framework supports the graphical representation of the generated process as well.

Converting BPMN into XML Format. After the list of BPMN models are unified into one model, the generator converts the model into XML format.

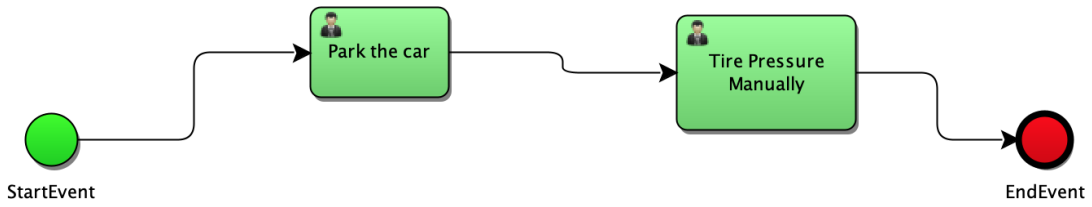

Figure 10. Generated BPMN model from the planner's output.

The graphical representation of the generated BPMN model using the planner's output showed in Listing 11 can be seen in Figure 10. After the BPMN model has been created, it is sent as a HTTP POST response to the mobile application.

3.8 Executing the Plan

The mobile application receives the business process as a BPMN model from the process planner by making a HTTP POST request. After the received model is deployed to the mobile-embedded process engine, the user can start using their generated mobile app. For that, the process engine starts the deployed business process.

The developed framework's mobile application is an extension of WiseWare [22] app which uses Activiti process engine that has been modified to run on Android OS. WiseWare extended the engine by using database tables and making the process engine migration-capable.

The WiseWare project was modified to suit the needs of this thesis. To keep the process engine code separate from the main module, it was moved to *activiti* module.

This also includes the networking layer for accessing the Process Planner and some custom Android Activity and Java Delegate classes to make the creation of different scenarios easier.

The main module contains classes that the application's developer creates. The mobile application prototype's main module contains a main view, which shows the two different scenarios to the user, and view classes for the two scenarios.

Starting the Business Process. The Hello World and Tire Change Activity classes extend *AppActivitiActivity*, which handles the process engine service. This class can be used, when creating new scenarios. When the app receives the BPMN model from the process planner, the *AppActivitiActivity* deploys the model to the process engine. By clicking on the deployed process, the *AppActivitiActivity* starts the process engine with the chosen deployed process.

When the user leaves the app in the middle of the ongoing process, then when the user comes back to the app, they see an ongoing process in the main view and can continue from there.

Running IoT and Android BPMN Tasks. The motivating scenario uses service tasks that include the implementation of the IoT service (Section 2.3.1). To showcase the capability of the application, the task *check_tire_pressure_bluetooth* implements reading data from BeeWi Smart Temperature and Humidity Sensor. The Java Delegate class contains the Android implementation of the sensor found in a Github project[14].

The generated application's user interface is also handled by service tasks. The Java Delegate classes extend *TaskActivitiDelegate*, which handles creating a new Android Activity and telling the process engine when the task is completed. It does this by accessing the Android app's context from the Application class and having a global static Boolean variable *androidActivityInProgress*. The *TaskActivitiDelegate* checks the Boolean after every millisecond in order to know if the user has completed the task. The delegate class only has to override the method called *setActivityClass* and specify what Activity class the delegate is using.

The Activity, which the Java Delegate is using, has to override the *TaskActivitiActivity*, which overrides the back button to do nothing. The process engine does not easily support going back to the previous process and therefore the back navigation is disabled. *TaskActivitiActivity* also contains a click listener which tells the delegate class that the task is completed. In the mobile application prototype, the click listener was tied to a button labeled "Done".

The mobile application prototype, the scenario Java Delegate and Activity classes are bundled with the Android app. The classes can also be dynamically fetchable by using

[14]BeeWi Android Implementation on Github - https://github.com/enrimilan/BeeWi-BBW200-Reader/

dynamic class loading approaches. One example would be to use a library called Grab'n Run [9].

3.9 Discussion

A framework prototype was developed, while taking into account the list of functional requirements discussed in Section 3.1. The framework can take an IoT planning problem and use it to generate an IoT mobile application using automated planning and business software management. The application is created by taking into account user's preferences and mobile device capabilities.

The framework proposes an unified way to access implementations of IoT devices by having a database of standalone smart actions. These actions can be reused when creating an application template. The developed framework does not need any context specific development when changing the scenario because the automated planning can take any planning domain and problem and generate a desired plan.

The usage of BPMN fosters good extensibility of the framework and speedy implementation of additional scenarios, because the BPMN snippets can be reused across multiple scenarios. The framework handles mapping the PDDL solution plan to BPMN plan so it is executable by the mobile process engine. Having the process engine embedded in the mobile application allows offline execution of the deployed processes. This means that the application can be used in scenarios where there is no access to the network.

The framework supports plans with parallel flows by using a partial-order planning and mapping the partially ordered flows into parallel BPMN flows, which makes the application more time efficient. The framework also keeps the graphical representation of the BPMN model so the developers have a chance to analyze the generated model.

The prototype also comes with some limitations, which can be considered as a future work as well. The first limitation of the prototype is that database is mocked as a JSON file and only includes the list of application templates as mentioned in Section 3.3. The author of this thesis proposes the database to be implemented as a Firebase Realtime Database and also include a list of available IoT actions.

Furthermore, the prototype's process planner is running locally on a Docker machine instead of using a hosted service. One option is to use Heroku[15] platform which supports Dockerized applications.

The automated planner can only make decisions based on the initial information it gets before it can generate a plan. Therefore, the required data (for example, battery level and NFC capability) needs to be collected before creating a plan. The framework prototype does not react to any changes or errors happening while executing the plan, for example, when the smart device is becomes suddenly unavailable. One solution is to

[15]Heroku with Docker - https://www.heroku.com/deploy-with-docker

handle the error logic in the BPMN snippet. The other option is to extend the framework and add functionality that monitors the smart devices and proposes a new plan when the context changes, as it was done in ECo-IoT [1] framework mentioned in the Section 2.5.

The thesis did not focus on collecting and storing data that is generated by the mobile application and IoT devices, and therefore the developed prototype does not support it. This is one area that can be extended in the future, and some inspiration can be taken from the Questionnaire application [30] mentioned in the Section 2.5.

4 Evaluation

In this section, the finished framework prototype is evaluated. First, the automated planner is tested with two scenarios and its performance and scalability is evaluated. Second, the framework's PDDL Generator is evaluated by testing how filtering the actions in the generator affects the automated planner's performance. Furthermore, it is discussed if five seconds is enough for the user to wait to receive a quality plan. Third, tests are run on the whole framework with the example scenario, and a breakdown of the different parts of the framework is given on how much time they take to produce a solution. Lastly, the key parts of the results are discussed.

4.1 Automated Planner Evaluation

One of the objectives of this thesis was to analyze which type of planning algorithm to use for this framework. In Section 2.1.3, during the analysis, the TFLAP algorithm was chosen.

During the implementation of the framework prototype, it was decided to set five seconds as a maximum time limit the user has to wait for the TFLAP planner. After the time limit was up, the framework chose the last solution out of all the produced plans. The planner needs to be evaluated to confirm whether the five seconds is suitable time limit or should it be higher or lower. Furthermore, it is evaluated if the last plan out of all the produced solutions is the best one based on the used metric.

The evaluation is done by taking inspiration from the International Planning Competition, which is organized in the context of the International Conference on Planning and Scheduling (ICAPS). In 2018 the competition was divided into three different tracks - classical, probabilistic and temporal [16]. The three tracks are divided into more specific tracks, and each one of them has a given list of tasks (planning problems) to solve. TFLAP planner competed in the competition in the temporal track. The planner will be tested again to evaluate if this planner suits the needs of this framework. The evaluation is done by taking inspiration from these two tracks, which are from classical and temporal tracks:

- **Agile track.** The best planner is the one that discovers the plan the fastest. The cost of the plan is ignored. The track has 8GB memory and 5-minute time limit.
- **Satisficing planning track.** The best planner is the one that discovers plans with the lowest cost. The planner can discover multiple plans, but the plan with the lowest cost counts. The track has 8GB memory and 30-minute time limit.

The agile track was chosen because it is important to receive at least any solution as fast as possible so the user can achieve what they want to do. The satisficing planning track was chosen to test how much time it takes to wait for the best quality plan. The

experiments are run on a guest machine in a docker container, and the Docker is using 1 CPU and 8GB memory. The host machine's CPU is 2.2 GHz Quad-Core Intel Core i7 (I7-4770HQ). The container is created using a Dockerfile, which can be seen in Listing 13.

```
1 FROM openjdk:8-jdk-alpine
2 RUN apk update && apk add make && apk add g++ && mkdir files
3         && cd src && make clean && make all
```

Listing 13. TFLAP Dockerfile.

The container is built on top of OpenJDK 8, just like the framework prototype is as mentioned in Section 3.5. The Docker container runs the commands to add dependencies for "make" and "g++" for the planner. Lastly, the container builds the TFLAP project. The Dockerfile is located in the TFLAP project's source folder. The planner is run in the Docker container by using a command:

```
./tflap domain_file.pddl problem_file.pddl solution_file.pddl
```

The parameters `domain_file.pddl` and `problem_file.pddl` are replaced by the planning problem's domain and problem file paths. The `solution_file.pddl` is the path where the planner outputs the files which contain the plan's output. The problem and domain files used in the following experiments and the solution files are accessible from the framework's source code in a folder called "evaluation".

4.1.1 Blocks-world scenario

The planner is firstly evaluated by running experiments with the Blocks World planning problem, which was also used in Section 2.1.3, for both tracks. To evaluate the planner's performance and scalability then for both tracks the list of tasks is a list of Blocks World planning problems where the number of blocks is incremented in every step.

Agile Planning Track. In the agile track, experiments are run on the planner to see how long it takes to receive a first plan while increasing the planning problem complexity.

While increasing the number of blocks in the Blocks World planning problem, the planner outputs the first result under five seconds with less than 23 blocks as can be seen in Figure 11. This planner fits perfectly for plans with small and medium sizes, while for bigger plans it is worth to increase the waiting time.

Satisficing Planning Track. In satisficing track experiments are run on the plan to see how long it takes to receive the solution with the lowest cost while increasing the planning problem complexity.

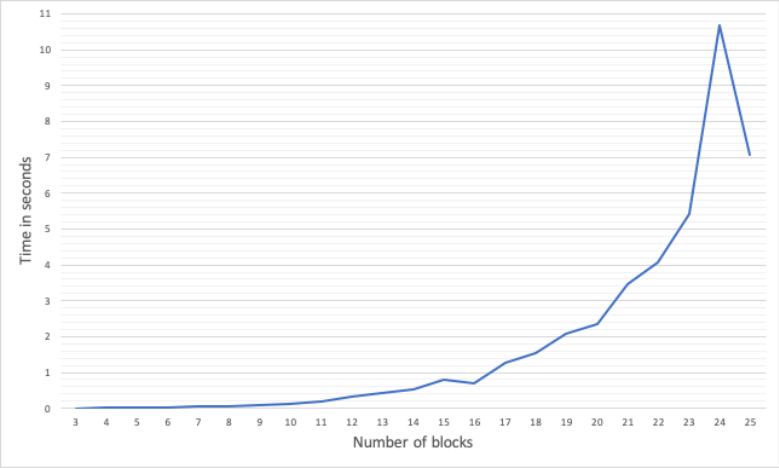

Figure 11. Time when receiving the first solution.

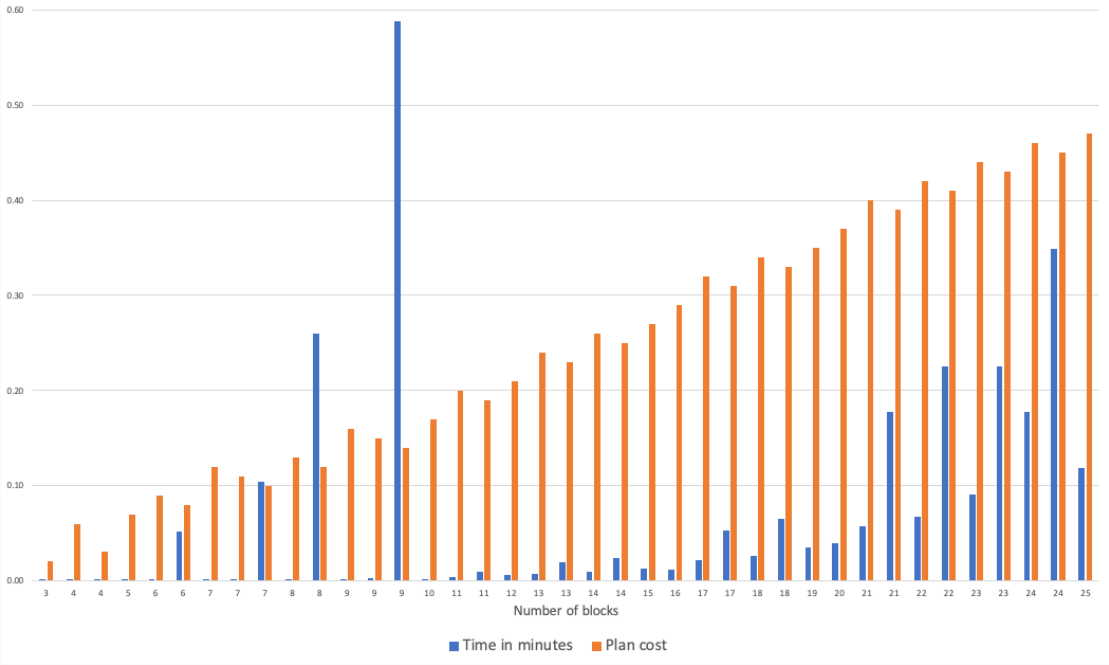

Figure 12. Time when receiving the solutions and their cost.

The Figure 12 shows all of the solutions the planner generated with blocks-world planning problem where the number of blocks was increased from 3 to 25. The Figure shows the time it took to get the solution with blue bars and the solution's cost with orange bars. For some planning problems TFLAP planner gave out multiple solutions

and the latest was with the lowest cost. Most of the solutions were received under a minute except for planning problems with 10 to 14 blocks which also gave out a solution after around one to five minutes. These five solutions were left out of the graph for readability purposes.

The Figure shows that while planner gives out multiple solutions, the differences in plan cost can be very small even though the difference in waiting time can be high. For example, for planning problem with 9 blocks the planner gave the first two solutions under one second while the last solution was received around half a minute later. Therefore it might not be worth it to wait for the best plan possible.

4.1.2 Modified Example Scenario

One of the objectives of this thesis was to find a planning algorithm that fits the motivation scenario the best. To evaluate how the chosen planner performs and scales, experiments are run with a synthetic modified example scenario.

The motivating scenario planning problem relies on defining action specific duration and cost. The tire change plan is usually linear and does not have steps that can be done in parallel. The user also has a choice to choose which metric the planner uses for the plan - either it tries to find a plan with the lowest duration or cost. The Blocks World scenario consists of actions which can be done in parallel, use the same cost and don't have duration specified. The planner tries to find a plan with the lowest cost.

The motivating scenario cannot be scaled very well, because the scenario has a finite number of tasks the user can do. In order to test the planner with planning problems which have motivating scenario's requirements (a long list of actions, support for durative actions and action costs), a synthetic motivating scenario is put together.

The scenario consists of steps - in order to do next step, previous steps have to be completed. To complete a step, the planner has four different actions to choose from. The actions have different duration, action cost and two of them have a application template requirement (Section 3.3) as can be see in Table 2.

	Action 1-1	Action 1-2	Action 1-3	Action 1-4
Duration	1	2	3	4
Action cost	2	1	4	3
Has requirement	No	No	Yes	Yes

Table 2. Example scenario planning actions for one step.

When the planner has only one step to complete, the planning domain consists of four actions. When there are two steps to complete, the domain consists of eight actions and so on.

For the motivating scenario, the user could choose between two metrics and to test out the planner's performance specifically for the framework and the scenario, the experiments are done using both of them.

Evaluation with minimum duration metric. The planner tries to find a plan which takes the least time to execute.

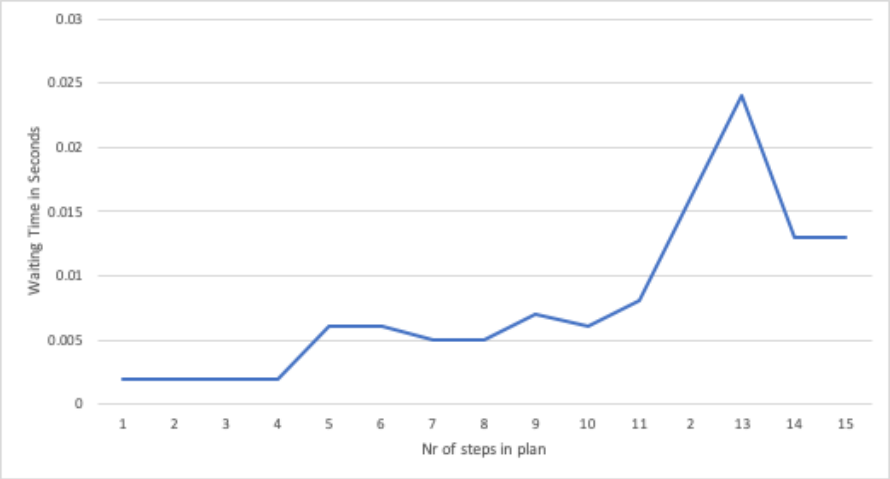

Figure 13. Time when waiting for the solution.

As can be seen in Figure 13, the planner found only one solution for the scenario every time. When inspecting the planning task and planner's output, the given solutions were the best possible solutions.

Evaluation with minimum cost metric. The planner tries to find a plan which costs the least to execute.

As can be seen in Figure 14, the planner gives out multiple plans for this planning problem. Surprisingly, the planner gave out the best plan first and the solutions after that were not better. One of the reasons why the planner is producing different results than with Blocks World planning problem is because there are two difference between these two scenarios. Tire change uses durative actions and for tire change the complexity comes from having so many actions to go through, for blocks world is the amount of elements in domain file.

4.2 PDDL Generator Evaluation

The framework's PDDL Generator filters out the actions which user cannot use based on their preferences. This makes the planning domain and problem size smaller. To evaluate

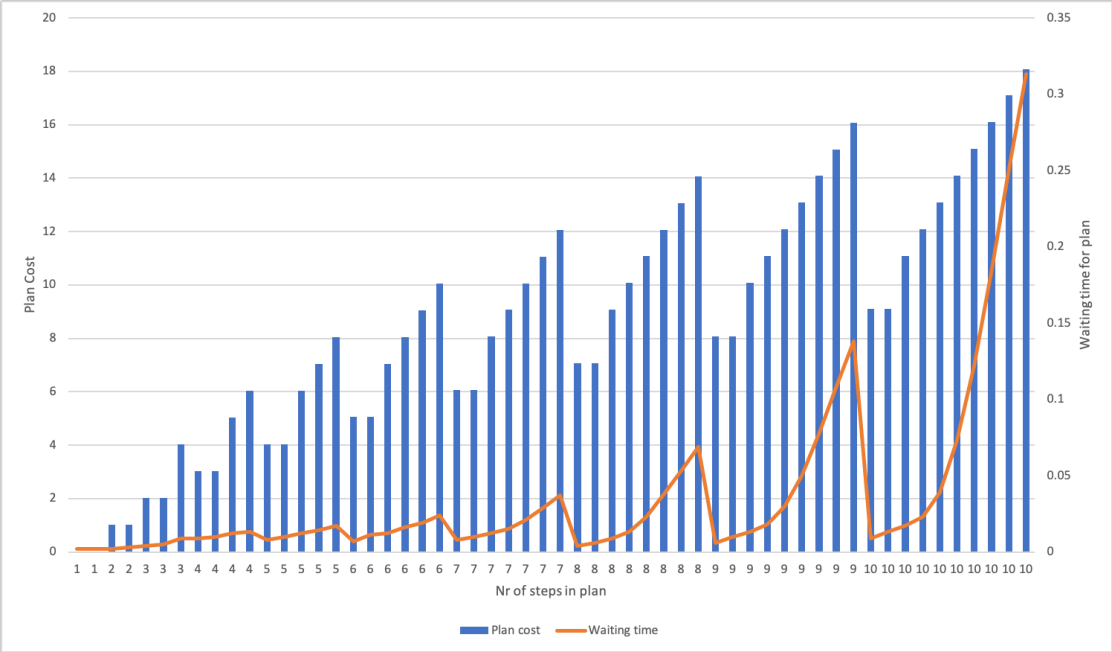

Figure 14. Time when receiving the solutions and their cost.

if generator's filtering has an impact on the planner's performance and scalability, two experiments are run on the planner - first getting the planning solutions with filtering turned off, the second time with filtering turned on. Lastly, it is discussed if five seconds is enough for the user to wait to receive a quality plan.

4.2.1 Filtering Evaluation

To test out if filtering out not needed actions makes a difference on planner's performance, the modified example scenario introduced in previous section is used.

For each step, the planner had four actions to choose from. Two of those actions should only be suggested to the user when they have filled out the requirement. When filtering is not turned off, the planner does not filter these requirements out and might suggest them to the user. When the filtering is turned on, these two actions are filtered out which means the planner has only two instead of four actions to choose from to complete a step.

As can be seen in Figure 15 and Figure 16 the filtering does make a difference in planner's performance significantly. This shows that when the framework's performance and scalability is important then it is worth to focus on improving the action filtering.

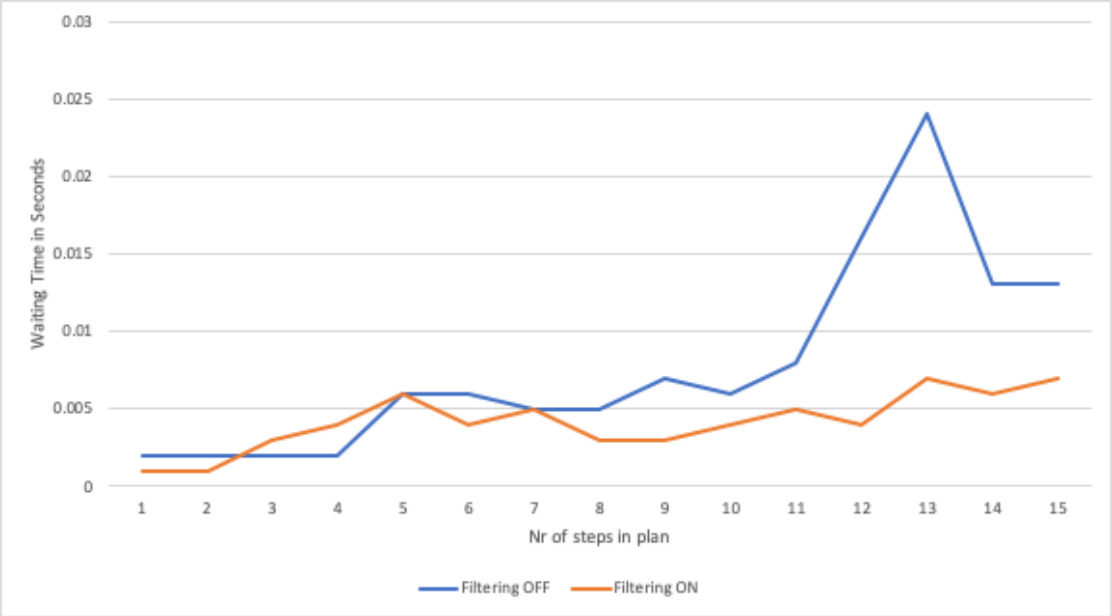

Figure 15. Time when receiving a solution with minimum duration metric.

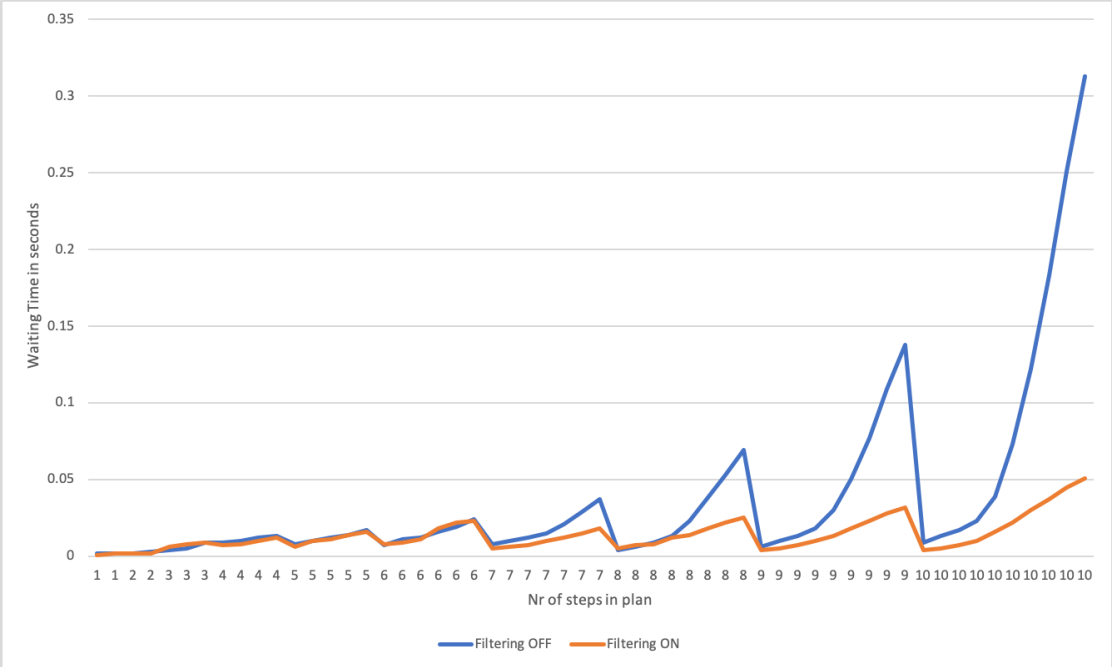

Figure 16. Time when receiving a solution with minimum cost metric.

4.2.2 Best Solution Waiting Time

During the implementation of the framework prototype, it was decided to set five seconds as maximum time limit the user has to wait for the TFLAP planner.

As can be seen from the previous figures, the planner usually gives the first plan under one second and therefore the author of this thesis proposes to improve the planner by setting the time limit to one second.

4.3 Framework Evaluation

The whole framework is evaluated by testing out how much time each part of the framework takes to produce a solution. The experiments are run with two scenarios that were used to test out the developed framework - Hello World and example tire change scenario. The Hello World scenario's goal is to complete two actions which can be done in either order. For the tire change, the framework was tested out with four different versions. They differed from what metric was used and if the user has specified any requirements.

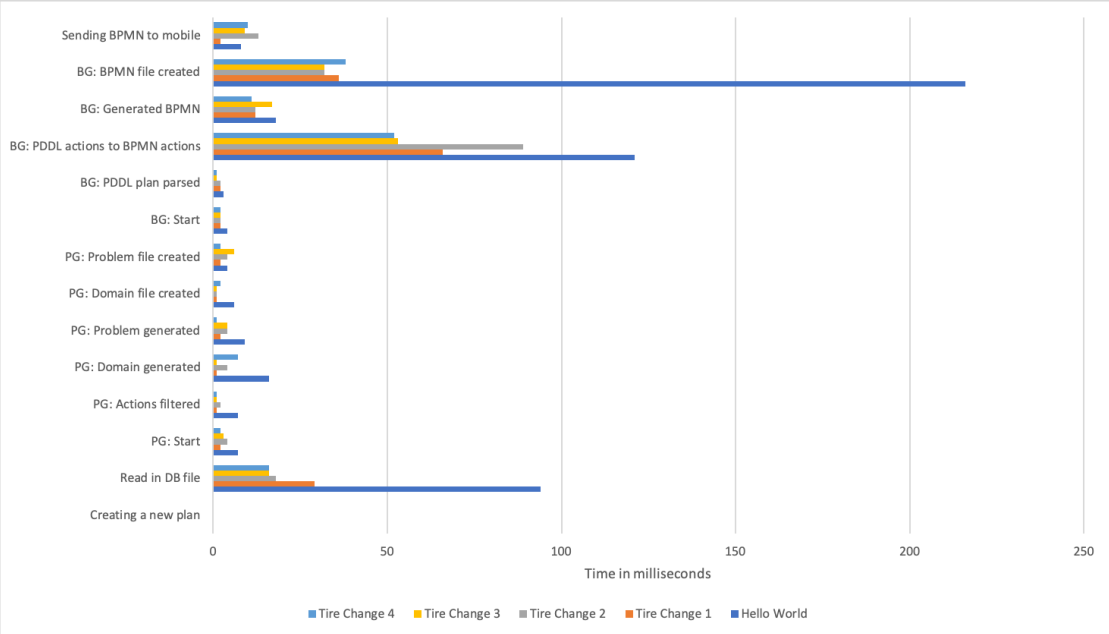

Figure 17. Breakdown of the framework on how long it takes to receive a solution.

The breakdown of the whole framework and how much time those parts take to find a solution for the planning problem can be seen in Figure 17. The figure is not showing the time the framework waited for the planner because this is always five seconds. All parts

of the developed framework besides the planner took less than 250 milliseconds. The terms PG means PDDL Generator and BG means BPMN Generator. The two longest parts of the framework besides the planner is creating a BPMN file and mapping PDDL actions to BPMN actions. Reading in database (DB) file for the Hello World scenario took a longer time as well compared to tire change scenarios.

4.4 Discussion

The different parts of the framework were evaluated and the results showed the TFLAP planner finds a solution for a medium sized plan under one second. Therefore having five seconds as a time limit for the user to wait for the planner is too much. It was proposed to change this time limit to one second.

Furthermore, the results showed that filtering out not needed IoT actions significantly improved the waiting time for the planner. When the framework's performance and scalability is important then it is worth to focus on improving the action filtering.

Lastly, the whole framework was evaluated by performance. All parts of the developed framework besides the planner took less than 250 milliseconds, which is a very good result.

5 Conclusion and Future Work

The final chapter concludes this thesis, discusses what was done and what were the results and finally presents the ideas for future work.

5.1 Conclusion

This thesis presented a framework can take a template for an IoT application described as planning problem and use it to generate a mobile application using automated planning and business software management while taking into account user's preferences and mobile device capabilities.

A state of the art review was given about automated planning, Internet of Things, business process management and Android. Different planning algorithms and their implementations were introduced, and a suitable planner TFLAP was chosen out of three planners from the International Planning Competition. As a foundation for this thesis, four similar systems were presented and it was discussed how the developed framework differs from these systems.

In the third chapter, the suggested framework's architecture was discussed. As a part of this thesis, a framework prototype was implemented and it consists of two projects - a mobile application and a process planner. The prototype was developed while taking into consideration that it could be easily extended to use any application context. This was achieved by using automated planning and mobile-embedded process engine. To showcase the framework, the mobile application supports two scenarios - the motivating tire change and Hello World scenario. An in-depth description of the prototype's implementation was given. The framework proposes a unified way accessing the implementations of the IoT device. The prototype can generate an IoT mobile application from the application templates.

Lastly, different parts of the framework were evaluated and the results showed that filtering out not needed IoT actions significantly improved the waiting time for the planner. Furthermore, the TFLAP planner found a solution for a medium sized plan under one second and therefore it was suggested to change the user's maximum waiting time for the planner from five seconds to one.

5.2 Future work

The implemented framework has some limitations such as not reacting to events happening in the middle of the plan. For example, the smart devices can suddenly disappear from the network and the framework should be able to detect and propose a new plan with different smart device.

The framework could be extended by looking into ways to store and collect data generated by the mobile application and IoT devices. This is especially beneficial, when

the application is used offline. Furthermore, the developed prototype can be improved by implementing a Firebase database and hosting the process planner in the cloud.

References

[1] Fahed Alkhabbas, Romina Spalazzese, and Paul Davidsson. ECo-IoT: An Architectural Approach for Realizing Emergent Configurations in the Internet of Things. In Carlos E. Cuesta, David Garlan, and Jennifer Pérez, editors, *Software Architecture*, volume 11048, pages 86–102. Springer International Publishing, Cham, 2018. Series Title: Lecture Notes in Computer Science.

[2] Parvaneh Asghari, Amir Masoud Rahmani, and Hamid Haj Seyyed Javadi. Internet of Things applications: A systematic review. *Computer Networks*, 148:241–261, January 2019.

[3] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, October 2010.

[4] J Benton, Amanda Coles, and Andrew Coles. Temporal planning with preferences and time-dependent continuous costs. In *Twenty-Second International Conference on Automated Planning and Scheduling*, 2012.

[5] Android Developers Blog. Announcing the Android 1.0 SDK, release 1, September 2008. https://android-developers.googleblog.com/2008/09/announcing-android-10-sdk-release-1.html (last accessed May 13, 2019).

[6] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. POPF2: A forward-chaining partial order planner. In *The 2011 International Planning Competition*, pages 65–70, 2011.

[7] Android Developers. Application Fundamentals. https://developer.android.com/guide/components/fundamentals (last accessed May 13, 2019).

[8] Marlon Dumas, Marcello La Rosa, Jan Mendling, and Hajo A. Reijers. *Fundamentals of Business Process Management*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2018.

[9] Luca Falsina, Yanick Fratantonio, Stefano Zanero, Christopher Kruegel, Giovanni Vigna, and Federico Maggi. Grab'n Run: Secure and Practical Dynamic Code Loading for Android Applications. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Los Angeles, CA, December 2015.

[10] M. Fox and D. Long. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124, December 2003.

[11] Hector Geffner. Computational models of planning. *WIREs Cognitive Science*, 4(4):341–356, 2013.

[12] Hector Geffner and Blai Bonet. A Concise Introduction to Models and Methods for Automated Planning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 7(2):1–141, June 2013.

[13] Object Management Group. Business Process Model And Notation, December 2011. https://www.omg.org/spec/BPMN/2.0/ (last accessed May 2, 2019).

[14] Naresh Gupta and Dana S. Nau. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2-3):223–254, August 1992.

[15] Faruk Hasic and Estefania Serral Asensio. Executing IoT Processes in BPMN 2.0: Current Support and Remaining Challenges. In *2019 13th International Conference on Research Challenges in Information Science (RCIS)*, pages 1–6, Brussels, Belgium, May 2019. IEEE.

[16] International Planning Competition 2018. https://ipc2018.bitbucket.io/ (last accessed August 7, 2019).

[17] Alfresco Software Inc. Activiti. https://www.activiti.org/ (last accessed April 20, 2018).

[18] Christian Janiesch, Agnes Koschmider, Massimo Mecella, Barbara Weber, Andrea Burattin, Claudio Di Ciccio, Avigdor Gal, Udo Kannengiesser, Felix Mannhardt, Jan Mendling, Andreas Oberweis, Manfred Reichert, Stefanie Rinderle-Ma, Wen-Zhan Song, Jianwen Su, Victoria Torres, Matthias Weidlich, Mathias Weske, and Liang Zhang. The Internet-of-Things Meets Business Process Management: Mutual Benefits and Challenges. *arXiv:1709.03628 [cs]*, September 2017. arXiv: 1709.03628.

[19] Daniel L Kovacs. Complete BNF description of PDDL 3.1, 2011. https://helios.hud.ac.uk/scommv/IPC-14/repository/kovacs-pddl-3.1-2011.pdf (last accessed August 1, 2020).

[20] Somayya Madakam, R. Ramaswamy, and Siddharth Tripathi. Internet of Things (IoT): A Literature Review. *Journal of Computer and Communications*, 03(05):164–173, 2015.

[21] Andrea Marrella. What Automated Planning Can Do for Business Process Management. In *Business Process Management Workshops*, Lecture Notes in Business Information Processing, pages 7–19. Springer International Publishing, 2018.

[22] J. Mass, C. Chang, and S. N. Srirama. WiseWare: A Device-to-Device-Based Business Process Management System for Industrial Internet of Things. In *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 269–275, December 2016.

[23] Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. Pddl-the planning domain definition language, 1998.

[24] P. Na-Lumpoon, M. Fauvet, and A. Lbath. Toward a framework for automated service composition and execution. In *The 8th International Conference on Software, Knowledge, Information Management and Applications (SKIMA 2014)*, pages 1–8, December 2014.

[25] Adriana Neagu. Figuring the costs of mobile app development, June 2017. https://www.formotus.com/blog/figuring-the-costs-of-custom-mobile-business-app-development (last accessed April 22, 2019).

[26] Allen Newell, John C Shaw, and Herbert A Simon. Report on a general problem solving program. In *IFIP congress*, volume 256, page 64. Pittsburgh, PA, 1959.

[27] Andy Rubin. Where's my Gphone?, November 2007. https://googleblog.blogspot.com/2007/11/wheres-my-gphone.html (last accessed May 13, 2019).

[28] O Sapena, Eliseo Marzal, and E Onaindia. Tflap: a temporal forward partial-order planner. In *IPC 2018 – Temporal Tracks*, pages 4–6. IPC, 2018.

[29] Oscar Sapena, Alejandro Torreño, and Eva Onaindía. Parallel heuristic search in forward partial-order planning. *The Knowledge Engineering Review*, 31(5):417–428, November 2016.

[30] Johannes Schobel, Rüdiger Pryss, Marc Schickler, and Manfred Reichert. A Lightweight Process Engine for Enabling Advanced Mobile Applications. In Christophe Debruyne, Hervé Panetto, Robert Meersman, Tharam Dillon, eva Kühn, Declan O'Sullivan, and Claudio Agostino Ardagna, editors, *On the Move to Meaningful Internet Systems: OTM 2016 Conferences*, Lecture Notes in Computer Science, pages 552–569. Springer International Publishing, 2016.

[31] John A. Stankovic. Research Directions for the Internet of Things. *IEEE Internet of Things Journal*, 1(1):3–9, February 2014.

[32] Forecast number of mobile users worldwide 2019-2023 | Statistic, January 2019. https://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/ (last accessed April 20, 2019).

[33] Mobile OS market share 2019, January 2020. https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/ (last accessed August 6, 2020).

[34] Eugene Toporov. IntelliJ IDEA is the base for Android Studio, the new IDE for Android developers, May 2013.

Appendix

I. Licence

Non-exclusive licence to reproduce thesis and make thesis public

I, **Kelian Kaio**,

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

   reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

   Generating Process-based Mobile Applications for the Internet of Things using Automated Planning ,

   supervised by Jakob Mass.

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Kelian Kaio
10/08/2020