university of groningen

faculty of mathematics and natural sciences

Modeling a Planning Domain for Smart Offices

Bachelor's thesis

August 2013

Student: Angèle Croes

Daily supervisor: ir. Ilche Georgievski

Primary supervisor: Prof. dr. ir. Marco Aiello

Secondary supervisor: Prof. dr. ir. Paris Avgeriou

Abstract

The planning of services for smart offices can be done using the Hierarchical Task Network planning technique. A state-based HTN planner to be more exact. This kind of planner needs task decompositions and state transitions in order to compose a plan and both these things are included in the domain description. This paper focuses on the domain description in Hierarchical Planning Domain Language (HPDL). A scenario is developed to showcase how a domain is described in HPDL. Also a problem description in HPDL is developed to test the output of the planner given the domain description. Furthermore, the performance of the planner given the domain description is tested since it mostly depends on the manner the domain was modeled.
It is concluded that HPDL works well for modeling the domain and that the functionality and performance of the domain modeled for this paper was as expected.

Contents

Chapter 1

Introduction

It was only 10 years ago when people were getting used to having cell phones. Back then their main and basically only functionality was to receive and make phone calls. These basic cell phones then became smartphones. The smartphone is not only used as a phone, by now most of them are our media player, camera and GPS too. Most importantly however is that it allows its user to connect to the internet and that adds many more functionalities. Why do we call these upgraded cell phones smartphones then? The term "smart" was added to things that made our lives easier, mostly some kind of system. The reason this term was used is because the services provided gave the illusion that the system is able to think. There are already smart tv's, smart board and many other appliances connected to the internet that can be viewed as smart appliances. The next step in the evolution of smart technology is to go from smart appliances to complete smart environment.

Smart environments will make decisions and take actions based on information obtained from the users, us mere humans, either directly or indirectly. According to Ramos et al. [10], the entire purpose of such environment is "to interact with human beings in a helpful, adaptive, active and unobtrusive way" [10]. Besides improving our own comfort and ease, there are many other goals that can be achieved by building smart environments. One of these goals is reducing the energy consumption as much as possible.

The University of Groningen is currently developing a framework for Energy Smart Offices (further referenced to as EnSO) along with Eindhoven Technical University, Philips and IBM. Their aim is "to couple, for the first time, advanced research and novel techniques in ambient network technology, activity recognition and artificial intelligence planning with innovative service-oriented approaches, thus developing a truly energy-aware platform for the offices of tomorrow"[1]. The smart office envisioned for this project is a smart environment where the environment refers to offices. It must make decisions as to what actions needs to be taken to complete the goal task in the most energy efficient manner. Among all the computational techniques available to solve such decision making problems, artificial intelligence planning is the most intrinsically interesting. For the planning technique there has to be a domain and a problem that can be solved by adapting the environment in the form of a sequence of ac-

tions. For this thesis, the domain is of particular interest.The domain provides all necessary information to the planner so it can effectively do its job when given a problem description. This leads to the subject of this paper, which is the modeling of the domain for a smart office. Specifically how the domain is modeled for the EnSO project.

The next chapter is dedicated towards explaining some core concepts, which are smart offices, automated planner and HPDL. Next some attention is paid to related work. The creation of the domain model is explained using an example in chapter 4. Chapter 5 focuses shortly on the performance of the planner. The conclusions reached are featured in the chapter after that and finally some recommendations for future work related to this subject can be found in the last chapter.

Chapter 2

Background

In this chapter we provide background information on concepts and techniques necessary to understand the domain of smart offices and the way of creating office adaptations.

2.1 Smart offices

There are many kind of offices, but they usually have some kind of common ground. They all serve the purpose of being used to work in. Almost all have the somewhat the same furnitures and appliances. A *smart office* is an ordinary office that offers some extraordinary services. Services that almost all environments offer like turning the light on and some specific to the office environment such as pulling down a projection screen.
A term that goes hand in hand with smart offices is ubiquitous computing [6]. Ubiquitous computing is a fancy word used to denote the appearance of having computing everywhere. The aim of ubiquitous computing is to have natural interaction with computers. Exactly what is considered natural is up for debate. One can go as far as to say that offices need to be context-aware, meaning they should be aware of everything such as persons and objects and adapt to this context. It can also be viewed as having an interactive office, where the user can give commands to the underlying system. There is no way around the fact that smart offices would need to have some kind of system that can find out what commands to execute to reach a directly given goal or a goal derived from the context.

2.2 Automated planning

There is a field in Artificial Intelligence that concerns itself with the realization of planning for a sequence of actions that leads to the fulfillment of a certain goal, which is automated planning. Since the execution of commands can be seen as giving the system a certain goal and expecting it to be done, automated planning comes in handy.
Automated planning is done by a planner, which specifies the course of action that needs to be taken to get from the initial state to the goal state. In order

for the planner to accomplish this, it needs to know what options are available. This information is passed along in the form of a domain description.

There are many different kinds of planning techniques that get their plans in many different ways. For the EnSO project, a state-based HTN planner is developed [4]. HTN stands for hierarchical task network, which means that the planner can be provided with the primitive tasks that bring an actual change to the state and with compound task that can composed out of more tasks, primitive or compound. Furthermore, HTN planners need to be given a task as the goal objective. The state-based HTN planner developed for the EnSO project will be addressed as the Scalable Hierarchical planner, or SH planner. The SH planner needs to have a domain with all the tasks and then it can generate a list of primitive tasks as output when given a problem description as input. The planner seen as a black box is shown in figure 2.1.

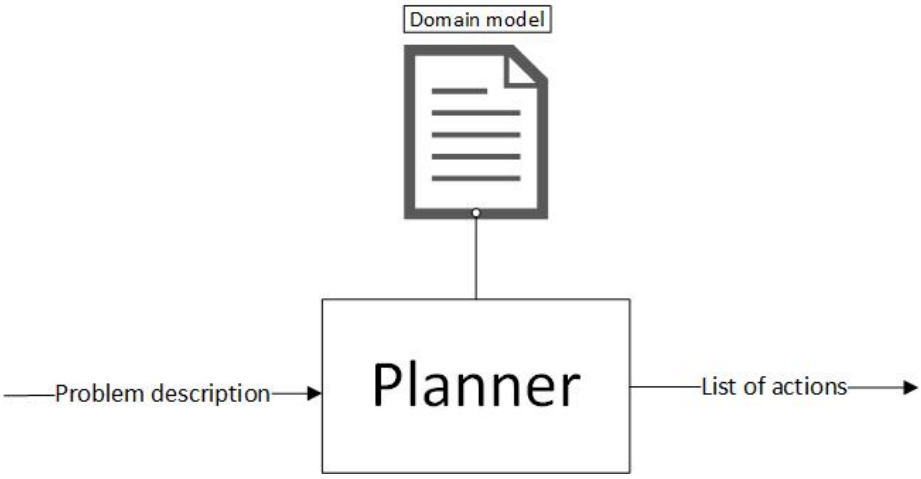

Figure 2.1: Planner as black box.

2.3 HPDL

As seen the SH planner needs a domain and a problem description. Both these two need to be in given in the Hierarchical Planning Domain Language (HPDL) [3]. The HPDL is partially based on the more popular Planning Domain Definition Language (PDDL) [8]. PDDL was not entirely suitable for HTN planners such as the SH planner, so HPDL was developed in an effort to meet more of the requirements.
HPDL describes tasks as either action, a primitive task or as a method, a compound task. It also can be extended to include typing, assignment and negative preconditions for example, however the planner needs to meet these requirements of course. The latest version of HPDL to date, and the one used during this project, is 0.2 by Georgievski and the grammar is given in appendix A. More explanation on specific parts of the grammar is given when the scenario is explained and parts of the domain and problem description is given.

Chapter 3

Related work

Many examples of other smart environment projects can be found, even smart offices. Active Badge, Monica SmartOffice, Stanford's Interactive Workspaces and Intelligent Environment Laboratory of IGD Rostock are all examples of such projects, some more ambitious than others [11, 2, 5]. The purpose of this project was not to compare the EnSO project to other existing projects, so the previously mentioned projects are not discussed as related work.

The subject of domain modeling for smart offices (or smart environments) has been less researched and documented. However Marquardt and Uhrmacher have researched using PDDL to create the planning domain in the context of the MuSAMA (Multimodel Smart Appliance Ensembles for Mobile Applications) project [9, 7]. In [7] the authors discuss using PDDL for creating AI planning domains. They do this with the help of some pre-existing scenarios from literature. The syntax of PDDL is explained through the examples, but the main product of the paper are the requirements gathered by evaluating the scenarios used. The requirements vary from very specific PDDL ones to general static state space ones. All in all the requirements can be seen as very helpful guidelines when creating a planning domain with PDDL and can be of use when creating a planning domain with HPDL.

In the paper by Plociennik et al. [9] the subject shifts from the general problem of planning domains to the specific problem of locking in problem domains. After introduction of some approaches that deal with the problem, two approaches are elaborated on. The two are the locks approach and the guarding approach. By comparing the two, the conclusion was reached that both were suitable for solving the persistant action problem. Due to the limited scope of this paper, the option of locking was forgone. When the need arises though, the two approaches should be considered since HPDL has some similarities with PDDL.

Chapter 4

Scenario: getting room ready for demonstration

Making a complete domain for an office demands exhaustive work. It is not necessary though to have the complete domain in order to demonstrate a domain model described in HPDL for the purpose of solving a planning problem in smart offices by using the SH planner. In this thesis, for the purpose of illustration, we take the preparation of a demonstration room as a working scenario. This scenario entails that there is a room somewhere that can be turned into a demonstration room. In order to be able to do this we envision it to have a computer, beamer and projection screen. The only information necessary for the task to be completed is which room and which computer. A room is said to be ready for demonstration when the computer and beamer is on and the video output of the computer is redirected to the beamer. The projection screen that the beamer is targeting also needs to be lowered and the lighting needs to be right. The lighting is considered to be right when it is dark enough near the projection area, but still light further away. As part of the project a domain was modeled to get a room ready for demonstration from all possible initial states, permitting all the hardware is set. For example the location of a computer cannot be changed by a system, the location is just a set of coordinates and that can be changed but then it would not be in accordance with the real world. Theoretically speaking the location can be moved with robots for example, but most offices do not have those and in this scenario things like location are set.

4.1 Data modeling

First step in creating a domain is to model the data. This means all the information the planner can have access to.
There are many different kinds of data. The first is data obtained from appliances directly. Each appliance knows what kind of appliance it is, it knows its location, it knows its state. For every appliance the state can mean different things. For the scenario the appliances can be roughly divided into two groups. The pullables are appliances whose state can be either pulled up or pulled down. The devices are appliances whose state can be either turned on or off. The light is a special case since it can be a normal device or it can be dimmable, in

which case its state include the intensity percentage of the light. To illustrate an appliance with a more complex state, the domain could include a printer that is not used in the scenario itself. A printer can be ready, printing or not ready. A printer also keeps track of documents in its queue. The appliances all have identifiers too so that the system can differentiate between the different instances of one appliance. There are also some appliances that can have some extra information specific to their sort. A beamer can be targeting a projection screen and a computer can be connected to a beamer.

Next comes data obtained from sensors. There are many kinds of sensors and each can provide a system with information. For this particular scenario the only sensor whose value we are interested in is a lux sensor. Lux sensors can sense the intensity of light. The lux sensor only needs to keep track of the room it is in and the value it reads.

There is also contextual data. This data is not necessarily obtained directly from a sensor of appliance. For this scenario it includes information about the room like the location and the floor it is on. These are all data set once the building was built. Contextual data does not have to be set though, it can be obtained elsewhere. This scenario requires some information about the clarity outside. This data is obtained from the appropriate meteorological service and it is assumed that they all provide one of the mentioned clarity level.

4.2 Domain in HPDL

When creating a domain, the first thing that needs to be done is some administration work.

```
(define (domain enso)
(:requirements :strips :typing :negative-preconditions :
    universal-preconditions :numeric-fluents)
```

The name is necessary since the planner can support more than one domain and a problem description will need to specify which domain needs to be used by name. The requirements are features the planner needs to support. This scenario declares strips, which is the basic STRIPS-style and is the minimum the planner needs. Typing is also declared in order to type parameters. Negative preconditions needs to declared to check for the absence of predicates in preconditions. Universal preconditions is declared to allow the use of `for all` in the preconditions, this creates the opportunity of checking a predicate with more values. Numeric fluents is the last to be declared and can be seen as the most important one since it allows for the simple arithmetic operations and the assignment of numerical values.

4.2.1 Typing

Next all the types have to defined.

```
(:types Room
        DemonstrationRoom - Room
```

```
        Appliance
        Device - Appliance
        Light Computer Monitor Printer Beamer - Device
        Blind ProjectionScreen - Appliance
        Coordinate Percentage Luxvalue- number
        Clarity
        PrinterStatus
)
```

With the data modeled in a previous stage, it is simple to derive the necessary types and how they are related. There are two existing types, those are `object` and `number`. The default type is `object` so in this domain Room, Appliance, Clarity and PrinterStatus are subtypes of `object`. The Coordinate, Percentage and Luxvalue are subtypes of `number`. The other types are subtypes of a newly defined type, like Blind is of Appliance.

4.2.2 Predicates

A state is composed out of a collection of predicates. If the predicate is in the state then it is true and else it is false. This effectively removes the need to have two predicates for something that can only have two values. The predicates that are in the state initially is given in the problem description, but the domain also makes use of predicates for preconditions and effects. The predicates that can be used have to be declared after the types.

```
(:predicates    (location ?o - Object ?x ?y - Coordinate)
                (intensity ?l - Light ?p - Percentage)
                (dimmable ?l - Light)
                (turned-on ?d - Device)
                (pulled-up ?a - Appliance)
                (targeting ?b - Beamer ?s -
                    ProjectionScreen)
                (in ?a - Appliance ?r - Room)
                (on ?r - Room ?f - Floor)
                (clarity ?c - Clarity)
                (light-in-room ?r - Room ?c - Luxvalue)
                (connected ?a1 ?a2 - Object)
                (input ?c - Computer ?d - Device)
                (output ?c - Computer ?d - Device)
                (printer-status ?p - Printer ?s -
                    PrinterStatus)
)
```

These predicates can also be derived from the data modeled. The way the predicates are made up vary. For boolean values like turned-on or pulled-up it is easy to see how it works. The predicate can also have two variables which is the case for `in`, if the predicate is true it means that Appliance x is in Room y.

4.2.3 Actions

The whole purpose of the domain is to specify operators that can change the state. In HPDL this is done by primitive tasks or actions. All the primitive actions need to declared before the compound tasks. All the actions with their parameters for the scenario are as following:

```
(adjust-light ?l - Light ?p - Percentage)
(turn-on-light ?l - Light)
(turn-off-light ?l - Light)
(turn-on-computer ?c - Computer)
(turn-off-computer ?c - Computer)
(turn-on-monitor ?m - Monitor)
(turn-off-monitor ?m - Monitor)
(turn-on-beamer ?b - Beamer)
(turn-off-beamer ?b - Beamer)
(turn-on-printer ?p - Printer)
(turn-off-printer ?p - Printer)
(pull-down-blind ?bl - Blind)
(pull-up-blind ?bl - Blind)
(pull-down-projection-screen ?s - ProjectionScreen)
(pull-up-projection-screen ?s - ProjectionScreen)
(set-output-to-beamer ?c - Computer ?b - Beamer)
```

From the list it is made clear that the actions need to be declared for each type. The action `turn-on-computer` for example does the same as `turn-on-monitor` and `turn-on-beamer` with different types. One reason for this is because type hierarchy is not yet supported by the SH planner. Another reason would be that there would be no mistake when getting the output. With a single `turn-on` action the responsability to get clear output is moved to the problem description and the need for descriptive names.

A few actions will be elaborated on in the next few paragraphs.

```
(:action turn-on-computer
        :parameters (?c - Computer)
        :precondition (not(turned-on ?c))
        :effect (turned-on ?c)
)
```

Actions have an effect on the state. If this was the first action then the initial state would not have included (turned-on ?c) and once the action is "executed" the state will contain that predicate. Predicates in their most simple forms are used in simple actions such as turn-on-computer. Proper manipulation of predicates allows for more complex actions such as adjust-light.

```
(:action adjust-light
        :parameters (?l - Light ?p - Percentage)
        :precondition (and (intensity ?l ?x) (!= ?p ?x) (
            dimmable ?l))
        :effect (and (not (intensity ?l ?x)) (intensity ?
            l ?p))
)
```

This shows how to change the intensity of a light by connecting the intensity value and the light through a predicate. The binary comparisons available in HPDL offer the opportunity to check the value of the intensity. In the real world it is only logical that one specific dimmable light can only have one value at a specific moment. The planner only recognizes predicates though and since (intensity ?l ?x) and (intensity ?l ?y) are two different predicates, it sees no problem why both cannot be true so careful attention should be paid when using predicates in this way to avoid creating real-world paradoxes.

4.2.4 Tasks

While the only way to achieve a change in the state is through an action it does not mean that everything has to be turned into an action. An action can become part of a bigger and more complex action, that is referred to as a task in HPDL. Below are all the tasks for the scenario.

```
(turn-on-computer-system ?c - Computer)
(dim-lights ?xmin ?ymin ?xmax ?ymax - Coordinate)
(pull-down-blinds ?r - Room)
(pull-up-blinds ?r - Room)
(dim-projection-area ?r - DemonstrationRoom ?b - Beamer)
(turn-on-lights ?r - Room)
(light-room ?r - Room)
(setup-beamer ?b - Beamer ?c - Computer)
(dim-room ?r - DemonstrationRoom)
(turn-to-demonstration ?r - DemonstrationRoom ?c -
    Computer)
```

There are a few things to note from the list. First is that some tasks have a Room as parameter and others have DemonstrationRoom. This is because some of the tasks are specific to DemonstrationRoom, which are the rooms with all the necessary appliances. Secondly, unlike actions the order of the tasks do matter as. When task A is composed out of task B, then task B has to be declared before task A. Having to order the tasks is a result of the implementation of the SH planner and is not in HPDL.
In the following paragraphs a few tasks are elaborated on and some features are discussed.

```
(:task turn-on-computer-system
        :parameters (?c - Computer)
(:method monitor-on
        :precondition (and (not(turned-on ?c)) (Monitor ?
            m) (connected ?m ?c)(turned-on ?m))
        :tasks (sequence (turn-on-computer ?c))
)
(:method monitor-off
        :precondition (and (not(turned-on ?c ))(Monitor ?
            m) (connected ?m ?c) (not(turned-on ?m)))
        :tasks (sequence (turn-on-computer ?c) (turn-on-
            monitor ?m))
)
```

```
)
```

The code also shows a big advantage of using tasks, which is the option to define more states that are accepted through different methods with each a different set of preconditions. In the code above the preconditions of the two methods match except for one predicate, in this case that predicate becomes the deciding factor of which method will be used. If the task is called upon only one method will be chosen even if preconditions of more than one is met. It is for this reason that methods should be used to differentiate ways of achieving the same result. The task `setup-beamer` is a good example of how all initial states are accepted. The goal of the task is to set the output of the computer to the beamer and both have to be on and the projection screen has to be pulled down. These are three appliances and each with two possible states. Going off this there are 2^3 different combinations to be considered.

Since working with only true or false predicates severely limits the actions available, HPDL has the ability to assign a value to a variable. This can be seen in the following task, where a value needs to be calculated in the precondition. The predicate with `assign` in it will always be true, since it is more or less an action. The use of assign here also showcases another feature of HPDL, which is the basic binary math operations. It can add, subtract, divide and multiply numbers.

```
(:task dim-projection-area
        :parameters (?r - DemonstrationRoom ?b - Beamer)
(:method with-blinds-up
        :precondition (and (location ?b ?xb ?yb) (assign
            ?xmin (- ?xb 4)) (assign ?ymin (- ?yb 4))
                 (assign ?xmax (+ ?xb 4)) (assign ?ymax (+
                     ?yb 4)))
        :tasks (sequence (pull-down-blinds ?r) (dim-
            lights ?xmin ?ymin ?xmax ?ymax))
)
(:method with-blinds-down
        :precondition (and (location ?b ?xb ?yb) (assign
            ?xmin (- ?xb 4)) (assign ?ymin (- ?yb 4))
                 (assign ?xmax (+ ?xb 4)) (assign ?ymax (+
                     ?yb 4)))
        :tasks (sequence (dim-lights ?xmin ?ymin xmax
            ymax))
)
)
```

Aside from assign, there are other operations that can have change a number. There are scale-up and scale-down and there are increase and decrease.

Recursion is another feature that is usable in theory and depending on how the planner is implemented within the environment it may be practical. Consider the following task where each dimmable light that is not fully on get turned on and a normal light that is not turned off gets turned off.

```
(:task turn-on-lights
```

```
        :parameters (?r - Room)
(:method there-is-dimmable-light
        :precondition (and (Light ?l)(in ?l ?r) (dimmable
            ?l) (not (intensity ?l 100)))
        :tasks (sequence (adjust-light ?l 100) (turn-on-
            lights ?r))
)
(:method there-is-not-dimmable-light
        :precondition (and (Light ?l) (in ?l ?r) (not (
            dimmable ?l)) (not(turned-on ?l)))
        :tasks (sequence (turn-on-light ?l) (turn-on-
            lights ?r))
)
(:method no-more-lights
        :precondition ()
        :tasks ()
)
)
```

Executing actions takes time and the state may not have been changed before the tasks is called again. This will result in stockpiling the same action more than once and it will fail after the first time.

Creating the domain in HPDL resulted in having 14 predicates, 17 actions and 10 tasks. Not all of them are absolutely necessary and a few were added to make the domain more complete, but most of them were needed to correctly implement the task of turning a room into a demonstration room. The complete domain can be found in appendix A.

4.3 Problem description in HPDL

Once the domain model is verified and uploaded onto the planner, a problem description is needed in order to solve a problem. Creating a problem description might be a lot of work, but it remains simple.

```
(define (problem enso1) (:domain enso)
(:requirements :strips)
```

Just like with the domain, the problem descriptions needs a name and a list of the requirements for the planner. A problem description also needs to name the domain that needs to be used to plan the action sequence.

Next is the initialization of all the predicates. The current state of the environment consists out of a collection of predicates and the initial state needs to be given in the problem description. Below is a shortened list of the predicates for an example of a problem description. The complete list is in the problem description file included in appendix C.

```
(:init
        (Percentage 0)
        (Percentage 60)
        (Percentage 80)
```

```
        (Percentage 100)
        (Room room222)
        (DemonstrationRoom room222)
        (Light light2_222)
        (location light2_222 106 102)
        (in light2_222 room222)
        (turned-on light2_222)
        (Light light4_222)
        (location light4_222 106 110)
        (in light4_222 room222)
        (clarity mostly-sunny)
        (light-in-room room222 400)
)
```

The first four predicates initialize some numbers of the type Percentage. All types need to be defined and their values initialized. This is because when the parameter of a task/action is a number it gets converted into a precondition for that task/action. So if the parameter is "?x - Percentage", the problem description needs to have "(Percentage X)" in the initial value, where X is an actual value.

The two initializations of room222 shows how to deal with type hierarchy. DemonstrationRoom was declared as a subtype of Room, but the planner does not support type hierarchy and in order to achieve the same results the variable needs to be initialized as both types.

Included in the excerpt is also the initialization of two non-dimmable lights,

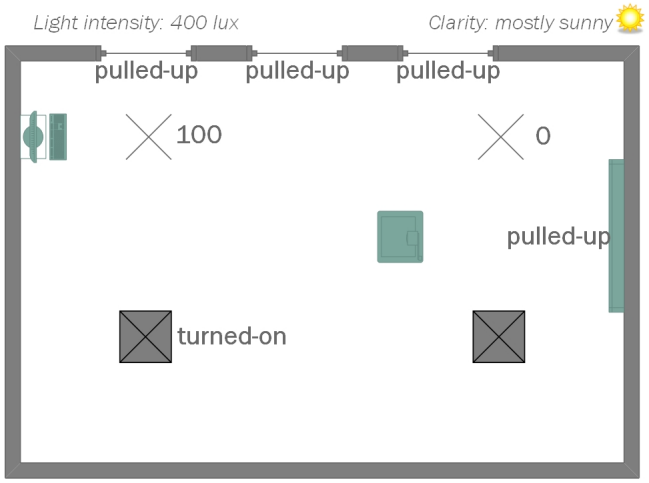

Figure 4.1: Initial state from problem description

where one is turned on and the other is turned off. Since the turned-on predicate is false when omitted, the planner recognizes light4_222 as turned off since there is not a (turned-on light4_222) in the state.
The last two predicates show information that in a real situation would be retrieved from the meteorology service and from a sensor.

A list of predicates does not give the best idea of how the initial state of the room is, figure 4.1 is the graphical representations of all the predicates.

With the initial state given the only thing the planner needs to know before getting to work is what the goal task or tasks are.

```
(:goal-tasks (sequence (turn-to-demonstration room222
    computer1_222)))
)
```

Following the syntax of the domain any of the tasks and/or actions can be specified as a goal task.

4.4 Output

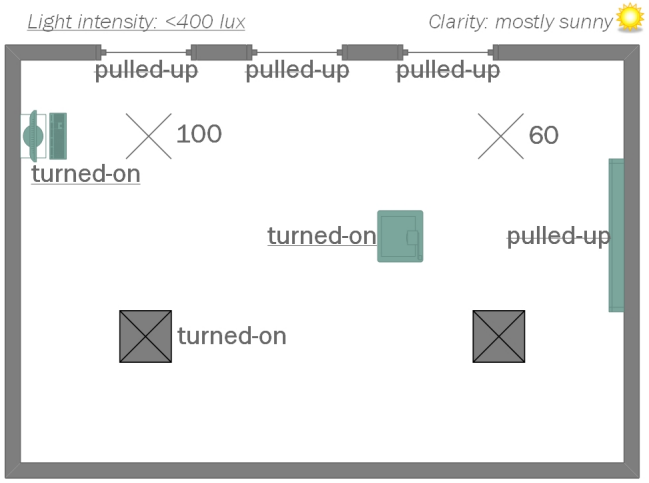

Figure 4.2: The state after the actions have been executed

With the domain and problem description provided, the planner will generate the actions needed to achieve the goal task which is to turn room222 into a demonstration room using computer1_222. The output is as follow:

```
1. pull-down-blind(blind3_222)
2. pull-down-blind(blind2_222)
3. pull-down-blind(blind1_222)
4. adjust-light(light3_222 ,60.0)
5. turn-on-computer(computer1_222)
6. pull-down-projection-screen(projection-screen1_222)
7. turn-on-beamer(beamer1_222)
8. set-output-to-beamer(computer1_222 , beamer1_222)
```

It will show exactly what actions need to be taken in the language of the domain. So naming the actions in the domain is important when deciphering the result. The state of the environment after these actions is graphically represented in figure 4.2

4.5 Requirements

Marquardt and Uhrmacher produced ten requirements in their paper on using PDDL for creating a planning domain [7]. Taking those into account together with experience gained when modeling the domain, we can produce our own list of requirements for using HPDL when creating a planning domain.

- **Actions should be kept modular.** By keeping the actions modular, two things can be achieved. First, the action sequence will be clearer.

- **Human actions should not be part of the effect of an action.** There are times when a goal can only be completed with human intervention. In our scenario we would need human action to move a device if it is not in the correct room. However, by allowing human actions to be part of the effect, the action sequence will become more like a suggestion for actions to take to complete the goal. Thus defeating the purpose of the planning algorithm of planning the execution of services. Most human actions can, or will be able to in the future, be substituted by robots. In such a scenario human action can be modeled as a service provided by the robot, but that is left out of our scope.

- **The objective must be formed only from actions and tasks contained in the domain description.** As opposed to PDDL where the objective is formed from propositions.

- **Along with syntactical mapping from HPDL to a common service description language, the plan should be enough to be executable.** This infers that no further knowledge is needed once the plan is generated.

- **Objects cannot be created.** Creation of objects would violate the static state planning of AI planning.

- **Axioms can and should be used when possible.** During the creation of the scenario for this paper, axioms were not needed. However when there are two predicates and the value of one can be derived from the value of other, axioms are favorable when considering efficiency. If we added `hasBeamer ?r - Room`, its value could be derived from `(in ?d - Device ?r - Room)` for example.

- **Locking resources should be considered.** While not considered during the creation of the domain in this paper, it is possible in HPDL and is necessary for some situations. The paper from Plociennik et al. delves deeper into some locking paradigms [9].

- **Objects should not be cast.** Static state space disallows objects of changing types. Strictly theoretically speaking though, in HPDL and the SH planner, the type is seen as just another predicate in the state that can be removed and and a new one added. Practically it has no use, since most objects cannot morph.

- **Type hierarchy should be used to its full capability.** While not implemented at the time of this paper, there is a workaround provided in section 4.3. Type hierarchy is beneficial to the efficiency of the domain.
- **Caution should be taken when introducing recursion into the domain.** Recursion may solve some problems, but can cause others at the same time. Proper locking techniques decrease the danger that accompanies recursion.
- **Numbers should be initialized along with objects.** The domain will search for initialization of number types when they are included as parameters. If not all are known at initialization, a range of numbers should be included.

Chapter 5

Performance

The scenario was an example of a domain description that the planner used. However, the performance of the planner depends on how the domain was modeled. To test the domain a test plan was made that should be enough to give an indication of how efficient the domain is. The domain is not exceptionally big and the SH planner should have no problem extracting an action sequence in mere milliseconds. Next the testing process is elaborated upon.

Setup

With the given domain there were two variables to test. The first one was the number of predicates in the initial state and the second one the number of actions it needs to execute. Both of these variables are part of the problem description.
To test the number of predicates the goal task was kept the same and just whole rooms were added, like room202 and room212. Each room added approximately 50 predicates to the initial state. The number of rooms was varied, going from 1, to 3, to 5, to 7 and to 10.
Testing the number of actions becomes trickier since some action take longer to get to. The best way to achieve reliable results was to have 10 rooms in exactly the same condition and repeat a task on a varying number of rooms.

Result

The result was plotted in a single graph as shown in figure 5.1. The graphs shows a somewhat linear increase in time. In terms of scalability it bodes well especially considering it could be exponential growth. For a building consisting of hundreds of rooms and a few goal tasks, the planner would still only need seconds and that is acceptable time. When measuring time for smart offices one should keep in mind that a normal office would have the user doing everything and can take minutes even.

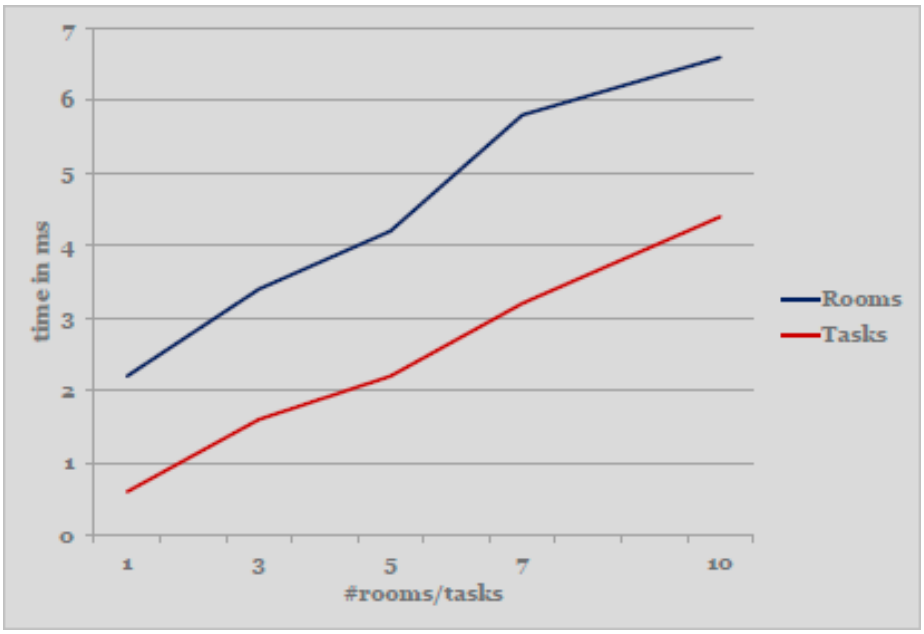

Figure 5.1: Result of performance measurements

Chapter 6

Conclusion

The purpose of this thesis was to model a planning domain for smart offices. For illustration purposes the scenario of getting a room ready for demonstration purposes was modeled. The steps followed were to first model the data and then the scenario itself. A problem description was developed to show how the domain could be used to solve a problem. A list of requirements came out of the modeling of the actions along with requirements previously mentioned in other related work [9, 7]. Most of these requirements highlight features supported by HPDL such as type hierarchy and axioms. Some requirements stem for the static state space used, while others come from HPDL or the combination with the SH planner. Furthermore the performance of the planner with the created domain was measured in order to test the efficiency of the domain. Results show that the performance is good enough and it is still a matter of milliseconds for increasing predicates in the initial state and goal tasks. The scalability deduced from the results is linear.

There is also something to be said about using HPDL instead of PDDL. At first glance there does not seem to much difference, but the scenario in this paper was never recreated using PDDL so one could not know exactly what the differences are. It is clear that since HPDL and the SH planner were both developed by the same person and used in the EnSO project it will be easier to adapt them to each other.

Future work

The purpose of this paper and the underlying project was to get an insight into the creation of a domain in HPDL and was not meant as project that should be built on since in a later stage the domain would not be hypothetical anymore. There are side projects that can be recommended as future work. The main issue was checking the syntax and semantics once the domain was finished. The advantages that programmers have when using popular programming languages and compilers are that there are good editors and error indications. The same can be applied to HPDL and the SH planner. A good interactive editor for HPDL can spot syntax errors from early on. Better error indication by the planner would make it easier to solve semantic errors later on.

As for HPDL specific issues, support for type hierarchy should be a must. In the partial domain created for this paper it was more about using the functionalities already embedded in HPDL instead of figuring out new functionalities that can be added. Because of this there are not many suggestions for the language.

Chapter 7

References

[1] Website for the enso project, 2011.

[2] Jan Borchers, Meredith Ringel, Joshua Tyler, and Armando Fox. Stanford interactive workspaces: a framework for physical and graphical user interface prototyping. *Wireless Communications, IEEE*, 9(6):64–69, 2002.

[3] Ilche Georgievski. Hierarchical planning definition language. Technical report, University of Groningen, JBI 2013-12-3, 2013.

[4] Ilche Georgievski and Marco Aiello. An overview of hierarchical task network planning. Technical report, CUniversity of Groningen, JBI 2012-12-5, 2012.

[5] Thomas Heider and Thomas Kirste. Intelligent environment lab. *Computer Graphics Topics*, 15(2):8–10, 2002.

[6] Christophe Le Gal. Smart offices. In *Smart Environments: Technologies, Protocols, and Applications*. John Wiley & Sons, Inc., 2005.

[7] Florian Marquardt and Adelinde Uhrmacher. Creating ai planning domains for smart environments using pddl. In *Intelligent Interactive Assistance and Mobile Multimedia Computing*, pages 263–274. Springer, 2009.

[8] Drew Mcdermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[9] Christiane Plociennik, Christoph Burghardt, Florian Marquardt, Thomas Kirste, and Adelinde Uhrmacher. Modelling device actions in smart environments. In *Intelligent Interactive Assistance and Mobile Multimedia Computing*, pages 213–224. Springer, 2009.

[10] Carlos Ramos, Goreti Marreiros, Ricardo Santos, and Carlos Filipe Freitas. Smart offices and intelligent decision rooms. In *Handbook of Ambient Intelligence and Smart Environments*, pages 851–880. Springer, 2010.

[11] Roy Want, Andy Hopper, Veronica Falcão, and Jonathan Gibbons. The active badge location system. *ACM Trans. Inf. Syst.*, 10(1):91–102, January 1992.

Appendix A

Grammar HPDL

A.1 Domain Description

```
<domain>          ::= (define (domain <name>)
                          <require-def>?
                          <types-def>?:typing
                          <predicates-def>?
                          <structure-def>*)
<require-def>     ::= (:requirements <require-key>+)
<require-key>     ::= See Section 3
<types-def>       ::= (:types <typed list (<name>)>)
<typed list (x)>::= x*
<typed list (x)>::=:typing x + - <type> <typed list (x )>
<type>            ::= <name>
<type>            ::= object
<predicates-def>::= (:predicates <atomic formula skeleton
    >+)
<atomic formula skeleton>::= (<predicate> <typed list (<
    variable>)>)
<predicate>       ::= <name>
<variable>        ::= ?<name>
<structure-def>   ::= <action-def>
<structure-def>   ::= <task-def>
<structure-def>   ::=:derived-predicates <derived-def>
<emptyOr (x)>     ::= ()
<emptyOr (x)>     ::= x
<action-def>      ::= (:action <name>
                          :parameters (<typed list (<
                              variable>)>)
                          <action-def body>)
<action-def body>::= :precondition <emptyOr (<pre>)>
                          :effect <emptyOr (<effect>)>
<pre>             ::= <atomic formula (<term>)>
<pre>             ::=:negative-preconditions <literal (<term>)>
<pre>             ::= (and <pre>*)
```

```
<pre>              ::=:disjunctive-preconditions (or <pre>*)
<pre>              ::=:disjunctive-preconditions (not <pre>)
<pre>              ::=:disjunctive-preconditions (imply <pre> <pre>)
<pre>              ::=:universal-preconditions (forall (<typed list
    (<variable>)>) <pre>)
<pre>              ::=:numeric-fluents <f-comp>
<atomic formula (t)>::= (<predicate> t*)
<literal (t)>     ::= <atomic formula (t )>
<literal (t)>     ::=(not <atomic formula (t )>)
<term>  ::= <name>
<term>  ::= <variable>
<term>  ::= <number>
<f-comp>          ::= (<binary-comp> <f-exp> <f-exp>)
<binary-comp>     ::= >
<binary-comp>     ::= <
<binary-comp>     ::= =
<binary-comp>     ::= <=
<binary-comp>     ::= >=
<f-exp>           ::=:numeric-fluents <number>
<f-exp>           ::=:numeric-fluents (<binary-op> <f-exp> <f-
    exp>)
<f-exp>           ::=:numeric-fluents (<multi-op> <f-exp> <f-exp
    >+)
<f-exp>           ::=:numeric-fluents (- <f-exp>)
<f-exp>           ::=:numeric-fluents <f-head>
<binary-op>       ::= <multi-op>
<binary-op>       ::= -
<binary-op>       ::= /
<multi-op>        ::= *
<multi-op>        ::= +
<f-head>          ::= (<function-symbol> <term>L)
<f-head>          ::= <function-symbol>
<function-symbol>::= <name>
<effect>          ::= (and <c-effect>L)
<effect>          ::= <c-effect>
<c-effect>        ::=:conditional-effects (forall (<typed list (<
    variable>)>) <effect>)
<c-effect>        ::= <p-effect>
<p-effect>        ::= (not <atomic formula (<term>)>)
<p-effect>        ::= <atomic formula (<term>)>
<p-effect>        ::=:numeric-fluents (<assign-op> <f-head> <f-
    exp>)
<assign-op>       ::= assign
<assign-op>       ::= scale-up
<assign-op>       ::= scale-down
<assign-op>       ::= increase
<assign-op>       ::= decrease
<task-def>        ::= (:task <name>
```

```
                        :parameters (<typed list (<
                            variable>)>)
                        <task-def body>)
<task-def body> ::= <method>+
<method>        ::= (:method <name>
                        :precondition <emptyOr (<pre>)
                        :tasks <task-list>)
<task-list>     ::= (<ordering> <task-atom>)
<ordering>      ::= sequence
<ordering>      ::= unordered
<task-atom>     ::= (<name> <term>*)
<derived-def>   ::= (:derived <atomic formula skeleton> <
    pre>)
<name>  ::= <letter> <any char>*
<letter>        ::= a..z | A..Z
<any char>      ::= <letter>
<any char>      ::= <digit>
<any char>      ::= -
<any char>      ::= _
<number>        ::= <digit>+ <decimal>?
<digit>         ::= 0..9
<decimal>       ::= .<digit>+
```

A.2 Domain Description

```
<problem>       ::= (define (problem <name>)
                        (:domain <name>)
                        <require-def>?
                        <object-declaration>?<init>
                        <goal-tasks>)
<require-def>   ::= (:requirements <require-key>+)
<require-key>   ::= See Section 3
<object-declaration>::= (:objects <typed list (<name>)>)
<init>          ::= (:init <init-el>*)
<init-el>       ::= <literal (<name>)>
<atomic formula (t)>::= (<predicate> t*)
<literal (t)>   ::= <atomic formula (t )>
<predicate> = <name>
<goal-tasks>    ::= (:goal-tasks <task-list>)
<task-list> ::= (<ordering> <task-atom>*)
<ordering>      ::= sequence
<ordering>      ::= unordered
<task-atom>     ::= (<name> <term>*)
<name>          ::= <letter> <any char>*
<letter>        ::= a..z | A..Z
<any char>      ::= <letter>
<any char>      ::= <digit>
<any char>      ::= -
<any char>      ::= _
```

```
<number>          ::= <digit>+ <decimal>?
<digit>           ::= 0..9
```

A.3 Requirements

A table of currently supported requirements by the SH planner. If a domain specifies no requirements, it is assumed a requirement `:strips`.

```
:strips                          Basic STRIPS-style adds
   and deletes
:typing                          Allow types in
   declarations of variables
:negative-preconditions          Allow not in precondition
    descriptions
:disjunctive-preconditions       Allow or in precondition
   descriptions
:universal-preconditions         Allow forall in
   precondition descriptions
:numeric-fluents                 Allow use of effects
   using assignment operators and arithmetic
   preconditions
:conditional-effects             Allow use of effects with
    forall operator
:derived-predicates              Allows predicates whose
   truth value is defined by a formula
```

Appendix B

Domain: enso

```
(define (domain enso)
(:requirements :strips :typing :negative-preconditions :
   numeric-fluents)
(:types          Room
                 DemonstrationRoom - Room
                 Appliance
                 Device - Appliance
                 Light Computer Monitor Printer Beamer -
                    Device
                 Blind ProjectionScreen - Appliance
                 Coordinate Percentage Luxvalue- number
                 Clarity
                 PrinterStatus
)
(:predicates     (location ?o - Object ?x ?y - Coordinate)
                 (intensity ?l - Light ?p - Percentage)
                 (dimmable ?l - Light)
                 (turned-on ?d - Device)
                 (pulled-up ?a - Appliance)
                 (targeting ?b - Beamer ?s -
                    ProjectionScreen)
                 (in ?a - Appliance ?r - Room)
                 (on ?r - Room ?f - Floor)
                 (clarity ?c - Clarity)
                 (light-in-room ?r - Room ?c - Luxvalue)
                 (connected ?a1 ?a2 - Object)
                 (input ?c - Computer ?d - Device)
                 (output ?c - Computer ?d - Device)
                 (printer-status ?p - Printer ?s -
                    PrinterStatus)
)

(:action adjust-light
        :parameters (?l - Light ?p - Percentage)
```

```
        :precondition (and (intensity ?l ?x) (!= ?p ?x) (
            dimmable ?l))
        :effect (and (not (intensity ?l ?x)) (intensity ?
            l ?p))
)

(:action turn-on-light
        :parameters (?l - Light)
        :precondition (and (not (turned-on ?l))(not (
            dimmable ?l)))
        :effect (turned-on ?l)
)

(:action turn-off-light
        :parameters (?l - Light)
        :precondition (and (turned-on ?l)(not (dimmable ?
            l)))
        :effect (not (turned-on ?l))
)

(:action turn-on-computer
        :parameters (?c - Computer)
        :precondition (not(turned-on ?c))
        :effect (turned-on ?c)
)

(:action turn-off-computer
        :parameters (?c - Computer)
        :precondition (turned-on ?c)
        :effect (not (turned-on ?c))
)

(:action turn-on-monitor
        :parameters (?m - Monitor)
        :precondition (not(turned-on ?m))
        :effect (turned-on ?m)
)

(:action turn-off-monitor
        :parameters (?m - Monitor)
        :precondition (turned-on ?m)
        :effect (not (turned-on ?m))
)

(:action turn-on-beamer
        :parameters (?b - Beamer)
        :precondition (not(turned-on ?b))
        :effect (turned-on ?b)
)
```

```
(:action turn-off-beamer
        :parameters (?b - Beamer)
        :precondition (turned-on ?b)
        :effect (not(turned-on ?b))
)

(:action turn-on-printer
        :parameters (?p - Printer)
        :precondition (not (turned-on ?p))
        :effect (and (turned-on ?p) (printer-status ?p
           ready))
)

(:action turn-off-printer
        :parameters (?p - Printer)
        :precondition (and (turned-on ?p) (not(printer-
           status ?p printing)))
        :effect (turned-on ?p)
)

(:action pull-down-blind
        :parameters (?bl - Blind)
        :precondition (pulled-up ?bl)
        :effect (not (pulled-up ?bl))
)

(:action pull-up-blind
        :parameters (?bl - Blind)
        :precondition (not (pulled-up ?bl))
        :effect (pulled-up ?bl)
)

(:action pull-down-projection-screen
        :parameters (?s - ProjectionScreen)
        :precondition (pulled-up ?s)
        :effect (not(pulled-up ?s))
)

(:action pull-up-projection-screen
        :parameters (?s - ProjectionScreen)
        :precondition (not (pulled-up ?s))
        :effect (pulled-up ?s)
)

(:action set-output-to-beamer
        :parameters (?c - Computer ?b - Beamer)
        :precondition (and (turned-on ?c) (turned-on ?b)
           (connected ?b ?c))
        :effect (output ?c ?b)
)
```

```
(:task turn-on-computer-system
        :parameters (?c - Computer)
(:method monitor-on
        :precondition (and (not(turned-on ?c)) (Monitor ?
           m) (connected ?m ?c)(turned-on ?m))
        :tasks (sequence (turn-on-computer ?c))
)
(:method monitor-off
        :precondition (and (not(turned-on ?c ))(Monitor ?
           m) (connected ?m ?c) (not(turned-on ?m)))
        :tasks (sequence (turn-on-computer ?c) (turn-on-
           monitor ?m))
)
)

(:task dim-lights
        :parameters (?xmin ?ymin ?xmax ?ymax - Coordinate
           )
(:method there-is-dimmable-light
        :precondition (and (Light ?l)(location ?l ?xl ?yl
           ) (> ?xl ?xmin) (> ?yl ?ymin)
                (< ?xl ?xmax) (< ?yl ?ymax) (dimmable ?l)
                     (not (intensity ?l 60)))
        :tasks (sequence (adjust-light ?l 60) (dim-lights
            ?xmin ?ymin ?xmax ?ymax))
)
(:method there-is-not-dimmable-light
        :precondition (and (Light ?l) (location ?l ?xl ?
           yl) (> ?xl ?xmin) (> ?yl ?ymin)
                (< ?xl ?xmax) (< ?yl ?ymax) (not (
                    dimmable ?l)) (turned-on ?l))
        :tasks (sequence (turn-off-light ?l) (dim-lights
           ?xmin ?ymin ?xmax ?ymax))
)
(:method no-more-lights
        :precondition ()
        :tasks ()
)
)

(:task pull-down-blinds
        :parameters (?r - Room)
(:method there-is-blind
        :precondition (and (Blind ?b) (in ?b ?r) (pulled-
           up ?b))
        :tasks (sequence (pull-down-blind ?b) (pull-down-
           blinds ?r))
)
(:method no-more-blinds
```

```
        :precondition ()
        :tasks ()
)
)

(:task pull-up-blinds
        :parameters (?r - Room)
(:method there-is-pulled-down-blind
        :precondition (and (Blind ?b) (in ?b ?r) (not (
            pulled-up ?b)))
        :tasks (sequence (pull-up-blind ?b) (pull-up-
            blinds ?r))
)
(:method no-more-blinds
        :precondition ()
        :tasks ()
)
)

(:task dim-projection-area
        :parameters (?r - DemonstrationRoom ?b - Beamer)
(:method with-blinds-up
        :precondition (and (location ?b ?xb ?yb) (assign
            ?xmin (- ?xb 4)) (assign ?ymin (- ?yb 4))
                 (assign ?xmax (+ ?xb 4)) (assign ?ymax (+
                     ?yb 4)))
        :tasks (sequence (pull-down-blinds ?r) (dim-
            lights ?xmin ?ymin ?xmax ?ymax))
)
(:method with-blinds-down
        :precondition (and (location ?b ?xb ?yb) (assign
            ?xmin (- ?xb 4)) (assign ?ymin (- ?yb 4))
                 (assign ?xmax (+ ?xb 4)) (assign ?ymax (+
                     ?yb 4)))
        :tasks (sequence (dim-lights ?xmin ?ymin xmax
            ymax))
)
)

(:task turn-on-lights
        :parameters (?r - Room)
(:method there-is-dimmable-light
        :precondition (and (Light ?l)(in ?l ?r) (dimmable
             ?l) (not (intensity ?l 100)))
        :tasks (sequence (adjust-light ?l 100) (turn-on-
            lights ?r))
)
(:method there-is-not-dimmable-light
        :precondition (and (Light ?l) (in ?l ?r) (not (
            dimmable ?l)) (not(turned-on ?l)))
```

```
        :tasks (sequence (turn-on-light ?l) (turn-on-
            lights ?r))
)
(:method no-more-lights
        :precondition ()
        :tasks ()
)
)

(:task light-room
        :parameters (?r - Room)
(:method light-naturally
        :precondition (and (light-in-room ?r ?x) (< ?x
            320) (or (clarity sunny) (clarity mostly-sunny
            )))
        :tasks (sequence (pull-up-blinds ?r))
)
(:method light-artificially
        :precondition (and (light-in-room ?r ?x) (< ?x
            320))
        :tasks (sequence (turn-on-lights ?r))
)
)

(:task setup-beamer
        :parameters (?b - Beamer ?c - Computer)
(:method set-everything
        :precondition (and (not(turned-on ?c)) (not (
            turned-on ?b)) (ProjectionScreen ?s) (
            targeting ?b ?s)
                (pulled-up ?s))
        :tasks (sequence (turn-on-computer ?c) (pull-down
            -projection-screen ?s) (turn-on-beamer ?b) (
            set-output-to-beamer ?c ?b))
)
(:method set-everything-but-computer
        :precondition (and (turned-on ?c) (not (turned-on
             ?b)) (ProjectionScreen ?s) (targeting ?b ?s)
                (pulled-up ?s))
        :tasks (sequence (pull-down-projection-screen ?s)
             (turn-on-beamer ?b) (set-output-to-beamer ?c
            ?b))
)
(:method set-everything-but-screen
        :precondition (and (not(turned-on ?c)) (not (
            turned-on ?b)) (ProjectionScreen ?s) (
            targeting ?b ?s)
                (not (pulled-up ?s)))
        :tasks (sequence (turn-on-computer ?c) (turn-on-
            beamer ?b) (set-output-to-beamer ?c ?b))
```

```
)
(:method set-everything-but-beamer
        :precondition (and (not(turned-on ?c)) (turned-on
            ?b) (ProjectionScreen ?s) (targeting ?b ?s)
                (pulled-up ?s))
        :tasks (sequence (turn-on-computer ?c) (pull-down
           -projection-screen ?s) (set-output-to-beamer ?
           c ?b))
)
(:method set-only-computer
        :precondition (and (not(turned-on ?c)) (turned-on
            ?b) (ProjectionScreen ?s) (targeting ?b ?s)
                (not(pulled-up ?s)))
        :tasks (sequence (turn-on-computer ?c) (set-
           output-to-beamer ?c ?b))
)
(:method set-only-screen
        :precondition (and (turned-on ?c) (turned-on ?b)
           (ProjectionScreen ?s) (targeting ?b ?s)
                (pulled-up ?s))
        :tasks (sequence (pull-down-projection-screen ?s)
            (set-output-to-beamer ?c ?b))
)
(:method set-only-beamer
        :precondition (and (turned-on ?c) (not (turned-on
            ?b)) (ProjectionScreen ?s) (targeting ?b ?s)
                (not(pulled-up ?s)))
        :tasks (sequence (turn-on-beamer ?b) (set-output-
           to-beamer ?c ?b))
)
(:method only-set-output
        :precondition (and (turned-on ?) (turned-on ?b) (
           ProjectionScreen ?s) (targeting ?b ?s)
                (not(pulled-up ?s)))
        :tasks (sequence (set-output-to-beamer ?c ?b))
)
)

(:task dim-room
        :parameters (?r - DemonstrationRoom)
(:method from-light
        :precondition (and (light-in-room ?r ?x) (> ?x
           320) (Beamer ?b) (in ?b ?r))
        :tasks (sequence (dim-projection-area ?r ?b))
)
(:method from-darkness
        :precondition (and (light-in-room ?r ?x) (< ?x
           320) (Beamer ?b) (in ?b ?r))
        :tasks (sequence (light-room ?r) (dim-projection-
           area ?r ?b))
```

```
)
)

(:task turn-to-demonstration
    :parameters (?r - DemonstrationRoom ?c - Computer)
(:method turn-to-demonstration-only-case
        :precondition (and (in ?c ?r) (Beamer ?b) (
            connected ?b ?c))
        :tasks (sequence (dim-room ?r) (setup-beamer ?b ?
            c))
)
)

)
```

Appendix C

Problem descriptions

```
(define (problem enso1) (:domain enso)
(:requirements :strips)
(:init
        (Percentage 0)
        (Percentage 60)
        (Percentage 80)
        (Percentage 100)
        (Coordinate 100)
        (Coordinate 101)
        (Coordinate 102)
        (Coordinate 103)
        (Coordinate 104)
        (Coordinate 105)
        (Coordinate 106)
        (Coordinate 107)
        (Coordinate 108)
        (Coordinate 109)
        (Coordinate 110)
        (Coordinate 112)
        (Coordinate 112)
        (Room room222)
        (DemonstrationRoom room222)
        (location room222 100 100)
        (Computer computer1_222)
        (location computer1 102 101)
        (in computer1_222 room222)
        (Light light1_222)
        (location light1_222 102 102)
        (in light1_222 room222)
        (dimmable light1_222)
        (intensity light1_222 100)
        (Light light2_222)
        (location light2_222 106 102)
        (in light2_222 room222)
```

```
        (turned-on light2_222)
        (Light light3_222)
        (location light3_222 102 110)
        (in light3_222 room222)
        (dimmable light3_222)
        (intensity light3_222 0)
        (Light light4_222)
        (location light4_222 106 110)
        (in light4_222 room222)
        (Monitor monitor1_222)
        (location monitor1_222 101 101)
        (in monitor1_222 room222)
        (connected monitor1_222 computer1_222)
        (Beamer beamer1_222)
        (location beamer1_222 104 108)
        (in beamer1_222 room222)
        (connected beamer1_222 computer1_222)
        (ProjectionScreen projection-screen1_222)
        (location projection-screen1_222 104 112)
        (in projection-screen1_222 room222)
        (pulled-up projection-screen1_222)
        (targeting beamer1_222 projection-screen1_222)
        (Blind blind1_222)
        (location blind1_222 100 103)
        (in blind1_222 room222)
        (pulled-up blind1_222)
        (Blind blind2_222)
        (location blind2_222 100 106)
        (in blind2_222 room222)
        (pulled-up blind2_222)
        (Blind blind3_222)
        (location blind3_222 100 109)
        (in blind3_222 room222)
        (pulled-up blind3_222)
        (clarity mostly-sunny)
        (light-in-room room222 400)
)
(:goal-tasks (sequence (turn-to-demonstration room222
    computer1_222)))
)
```