An integration of manipulation and action planning

Diplomarbeit

Sebastian Trüg

Albert-Ludwigs-Universität Freiburg
Fakultät für Angewandte Wissenschaften
Institut für Informatik

February 2006

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe.

Freiburg, im Februar 2006 Sebastian Trüg

Contents

Chapter 1

Introduction

Today, when people talk about robot automatization, they mostly refer to some kind of machine which works on the basis of deterministic programs. The robot may *understand* a certain set of deterministic commands which it can perform in a very simple domain. Take, for example, a construction robot in a car factory which has to weld parts of the car body. It must perform a distinct and very restricted task. For every car body arriving at the station of the robot, it is exactly the same. The robot does not have to care about its surroundings, which remain unchanged. It can be controlled by a deterministic program which is triggered by an event like the arrival of the next car body. There is little need for artificial intelligence. This robot, although it seems to do certain things in an intelligent fashion, is just a plain and simple machine, comparable in its usage to a lawn mower.

In this thesis, we will discuss an approach to make a robot more intelligent. This may sound impressive and even futuristic to some, but it should be clear that the goal cannot be to create a near-human being as in science fiction. This thesis will definitely not pave the way to our very own Sonny[1]. Our goal is to lay the foundation for a robot which can perform multiple tasks that are not hard-wired, a robot which can adapt to its surroundings (at least to a certain level).

It is a typical household: a married couple, two kids, a boy and a girl, a dog, maybe a cat and a hamster for the little daughter. Both the man and the woman are working hard to feed the kids and the hamster. One has to go shopping for groceries almost every day. And every day, there is the same problem: who will empty the shopping basket? The parents are much too exhausted and just want to get some rest, the boy is playing soccer with his friends, and the little girl cannot reach the top shelves anyway. This is where

[1]Sonny is the name of the first robot which has the freedom to break the three basic rules of robotics in the movie adaption of Isaac Asimov's *"I, Robot"*

an *intelligent* robot could take over. It could empty the shopping basket, put everything where it belongs, the perishable goods into the fridge, the cereals onto the top shelf (that which the little girl cannot reach), and the ice-cream into the freezer.

Solving this rather complex problem involves a lot of subtasks like analyzing the surroundings of the robot, classification of the items, determining of the order in which to put them away, finding free surfaces, moving through the kitchen without crashing into things, or actually grasping the items and putting them away.

In this thesis, we will be concerned with one of these subtasks, namely the grasping of items. The question we would like to answer in a situation like the above is: "In which pose does the robot have to grasp an object in order to put it in a different place without colliding with other objects?" We will present a new approach based on AI planning techniques that differs from the approaches previously examined. A reduced exemplar based on the "kitchen domain" mentioned above will be used to prove the feasibility of our approach.

The main focus in this thesis will be on the implementation of the required extensions to classical AI planning techniques. In chapter 2, we will present the basics of these techniques, describe our main problem, and discuss related work. Chapters 3 and 4 will then deal with the extensions and provide the basis for discussion of the actual problem in chapter 5.

Finally, we will present some experimental results in chapter 6, before making some concluding remarks and discussing ideas for further research in chapter 7.

Chapter 2

Problem Description and Related Work

In this chapter, we will give a short overview of planning in robotics, and provide a motivation for as well as a more formal definition of the problem we are dealing with in this thesis. Finally, we will present related work and outline in what way other approaches differ from ours.

2.1 Planning in Robotics

In general, there are four main research fields concerned with planning in the context of robotics. *Path* and *motion planning*, *navigation planning*, and *manipulation planning*.

Path planning refers to the purely geometrical task of finding a collision-free path for a robot among static obstacles. Motion planning elaborates on this by introducing the element of time, thus also handling the dynamics of the robot like acceleration and speed. The goal of motion planning is to find a control trajectory along the path. Motion and path planning are without doubt two of the most researched planning problems in robotics [Kavraki, 1999; Latombe, 1999; Barraquand and Latombe, 1991]. In particular, the introduction of probabilistic roadmap techniques by Kavraki *et al.* [Kavraki *et al.*, 1994] provided the foundation for some of the most successful approaches. Motion and path planning can be regarded as fairly mature today.

Motion planning is often used, too, as a subtask in higher level robotic planning problems such as navigation planning. Navigation planning involves perception handling through sensor input as well as self-localization in a constantly updated world model.

The main task in manipulation planning is to manipulate movable objects among obstacles. A robot moves towards an object, grasps it, and then transports this object to a different location.

The most common approach to manipulation planning is to create a manipulation graph which connects valid free grasp positions. The idea of a manipulation graph was introduced by Alami *et al.* [Alami *et al.*, 1990]. They define the set FC of all possible robot and object configurations that are free, i.e. in which no two bodies (robot, object, or obstacle) collide. Then $P \subset FC$ is defined as the set of valid object configurations, i.e. those which comply with the physical principles valid in the domain (e.g. an object is not allowed to float in mid-air). Also, $G \subset FC$ is defined as the set of configurations in which robot and object are aligned in a way that allows the robot to grasp the object. A *transfer-path* is a path in FC on which the robot moves one object, while a *transit-path* denotes a path in FC along which the configurations of all objects except the robot's are static.

The set of manipulation graph nodes, then, is made up of the connected components of $P \cap G$[1]. In case of a discrete number of placements and grasps, this set is finite. Otherwise, one has to select a subset representing the actual graph nodes which, for instance, by using a probability distribution. There are two types of edges: transfer and transit edges. Two nodes are connected by a transfer (transit) edge if a transit-path (transfer-path) exists between two configurations of the associated connected components. Solving the manipulation planning task then consists of building the manipulation graph and finding in it a path from the initial to a goal configuration.

This approach, however, has one disadvantage.

Imagine a box of water bottles which need to be put away into the kitchen cabinet. Being in the box, the bottles can only be grasped at the top. So a robot presented with this task would hold a bottle at the top after having taken it from the box. The shelf in the cabinet may just provide enough space to fit in the bottle, with little space left above it. So the robot would not be able to place the bottle into the cabinet using the current grasp pose. It would have to put the bottle down (or use a second hand if it has one) and use another grasp pose to put the bottle on the shelf.

The manipulation graph does not explicitely include the notion of regrasping an object in order to change the grasp pose. Thus, in order to allow as free a handling of the grasp poses as possible, we will follow a different approach here.

[1]The conneced components can be computed using the Collins decomposition. For further reading, we refer to [Latombe, 1991, Chapter 11].

2.2 Introducting: Action-Planning

In this thesis, we will approach the problem of manipulation planning based on classical domain-independent action planning techniques which allow the usage of general-purpose planning languages (such as PDDL [McDermott and others, 1998]) and planning systems. Domain-independent planning seeks to exploit commonalities to all forms of planning by relying on abstract, general models of actions. The state of the world is described by a finite set of state variables. Actions modify the values of these variables. In the most basic form of action-planning, the state variables are logical symbols and an action a is defined as as a tuple $a = (pre(a), add(a), del(a))$ where the preconditions $pre(a)$, the "add-list" $add(a)$, and the "delete-list" $del(a)$ are sets of variables. The action can be applied if the current state S fulfills all preconditions $pre(a)$ ($pre(a) \subset S$). The effect of the action, then, is defined as $apply(S, a) = S \cap add(a) \setminus del(a)$. In chapter 4, we will give a more formal definition of an action planning task.

The input of a domain-independent planner is the domain specification and knowledge about the specific task. Several planning languages have been developed to provide a formalism for domain and problem-task specifications. Two of the most important are ADL, the **A**ction **D**efinition **L**anguage [Pednault, 1989] and its successor PDDL (see [McDermott and others, 1998] and chapter 3) which has become the *de facto* standard language for describing planning domains. Both languages are based on the STRIPS style of actions [Fikes and Nilsson, 1971]. UCPOP [Penberthy and Weld, 1992] was the first planner to fully support ADL and was followed by several planning systems which mostly supported the ADL extensions through PDDL [Dean and Boddy, 1988; Chapman, 1987; Peot and Smith, 1992; Hoffmann and Nebel, 2001].

Using domain-independent planning in combination with manipulation planning, we can encode the grasp poses directly into the problem. Thus, in our interpretation of the problem, the action of grasping an object also involves choosing an appropriate grasp pose. In the context of this thesis we will regard the collision-free motion between grasping and putting down the object as an atomic action.

We use a simplified robot hand which is simply a long bar grasping objects by touching their surface. In other words, our hand is a kind of magnetic robot gripper. This simplification does not change the original problem of the grasp poses while leaving aside things like opening and closing the hand or the question if an object is too big for the hand. All these details we will not consider here.

Let us define our specialized version of manipulation planning formally.

Definition 2.1 (Planning Task) *A simplified manipulation planning task is a tuple*

$$\mathcal{T} = (\mathcal{B}, \mathcal{M}, \mathcal{I}, \mathcal{G})$$

where $\mathcal{B}$ is the set of static objects, $\mathcal{M}$ is the set of movable objects, and the initial state $\mathcal{I}$ and the goal state $\mathcal{G}$ are sets of mappings $\mathcal{M} \mapsto \mathcal{B}$ specifying which movable object is placed on top of which static object.

We define two operators *grab* and *putdown*. An operator o has three parameters: the object to be manipulated $m(o)$, the static object involved $b(o)$, and the grasp pose used $p(o)$. The state resulting from the application of an operator o to a state s is denoted by $result(o, s)$. A valid solving plan is thus defined as follows.

Definition 2.2 (Plan) *A plan solving a planning task $\mathcal{T} = (\mathcal{B}, \mathcal{M}, \mathcal{I}, \mathcal{G})$ is a sequence of grab and putdown operators*

$$\mathcal{P} = (g_0, p_0, g_1, p_1, \ldots, g_n, p_n)$$

such that $result(p_n, \text{result}(g_n, \ldots result(p_0, result(g_0, \mathcal{I})) \ldots)) = \mathcal{G}$ and $\forall i \in [0, n] : p(g_i) = p(p_i)$ and g_i and p_i can be executed collision-free.

In order to handle this specialized version of manipulation planning, we need to encode the following information into our domain:

1. The robot gripper: We need to know if it holds an object and if so in what pose.

2. Geometric objects in the three dimensional space $\mathbb{R}^3$: We need to encode information about the shape and dimensions of objects.

3. The relative position of two objects: We need to be able to express if an object sits on top of another. It is not necessary to know the absolute position of every object in the domain as we do not handle motion of the robot between them.

4. Collision detection between objects: We need to determine if certain grasp poses are usable in combination with certain objects.

Including the information necessary for three dimensional collision detection into the domain could be done by introducing numerical variables denoting the coordinates of points of triangles. Triangles could then belong to objects through the usage of a predicate like `is_part_of`. It is obvious though that this encoding would result in a very big domain description

```
(action putdown
  (parameters c - movable s - base p - pose)
  (preconditions
    (forall st - triangle
      (imply is_part_of(st,s)
        (forall t - triangle
          (imply is_part_of(t,c)
            (and
              (p1_y(t) > p1_y(st))
              (p2_y(t) > p1_y(st))
              (p3_y(t) > p1_y(st))
              (p1_y(t) > p2_y(st))
              (p2_y(t) > p2_y(st))
              (p3_y(t) > p2_y(st))
              (p1_y(t) > p3_y(st))
              (p2_y(t) > p3_y(st))
              (p3_y(t) > p3_y(st))
            )
          )
        )
      )
    )
  )
)
```

Figure 2.1: Encoding collision detection directly into the domain

which would not be solvable with current planning systems in reasonable time. Just testing if one object is located above another would be a difficult task as the excerpt from a PDDL action template in figure 2.1 shows.

With this already quite complex precondition we would not even be close to solving the collision detection problem. We would have to check a lot more constraints of this kind to "implement" full collision detection into the domain description and would probably end up with an encoding which is not only very large but also very slow.

This means that just using plain action planning cannot be the answer for a complex domain like this. We need a way of simplifying the domain while preserving the quality of the resulting plan.

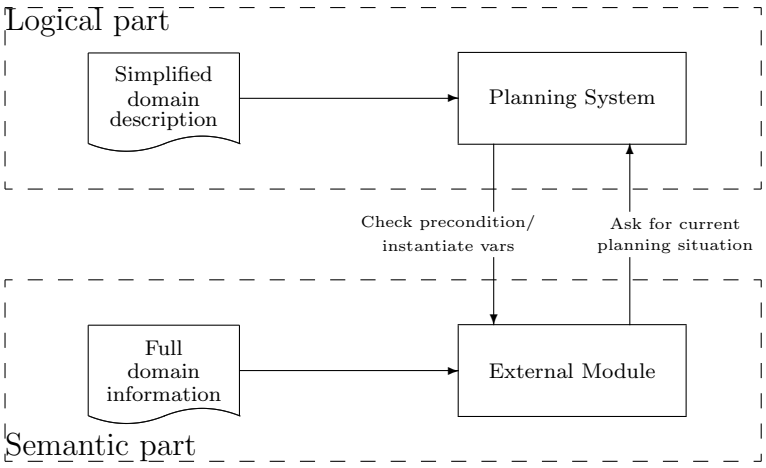


Figure 2.2: Architecture of Planning with external modules

2.3 Decomposing the Problem

The idea of the approach we will present here is to keep all geometrical information out of the planning domain to simplify a complex problem to a symbolic representable subset. In this subset of the problem, a question like *Is there a collision-free way to move from point a to point b?* is represented by a single symbol whose logical value is determined by an external instance. This external instance can have an arbitrary complexity, which is hidden from the planning system. So whenever the planning system accesses the logical value of this symbol, it actually invokes the external instance.

Thus we decompose the work that is to be done into two main parts: The sometimes expensive constraint checking and effect calculation based on the full domain knowledge including geometric information on the one hand, and the purely symbolic planning process on the other. The domain as seen by the planning system is reduced in complexity while not necessarily losing accuracy at the same time. We call these external instances *modules*. Figure 2.2 shows how an external module provides a "black box" for the planning system in order to generate certain facts in the course of the planning process. The external module has full access to all aspects of the domain while the planning system only has to deal with a simplified version of it.

To realize the concept of external modules as described above we need "procedural attachments" in our planning language. Procedural attachments refer to possibly complex code associated with an entity in the domain. Whenever the entity is used, an attached procedure is called to evaluate its status. In the context of action planning, these procedures are attached to logical symbols as mentioned above.

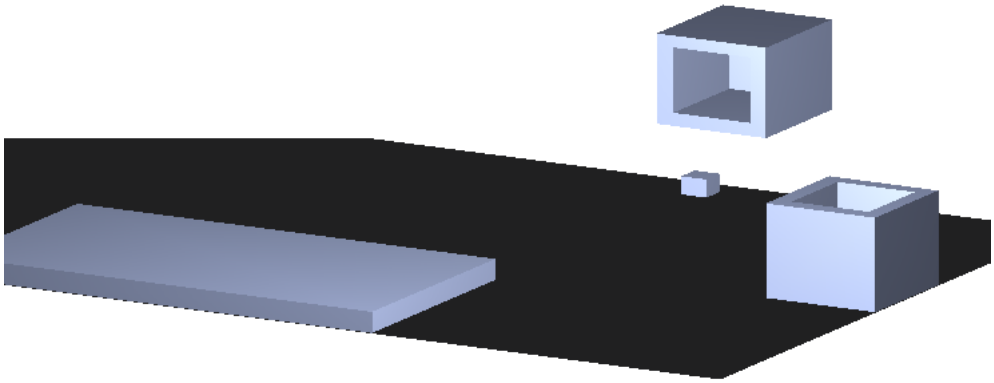


Figure 2.3: Example configuration

This leads us to the two main goals of this thesis: extending the PDDL to support procedural attachments and implementing a reference planning system to support these extensions. Thus, we create a solid foundation for the action planning approach to manipulation planning.

Throughout this thesis, we will use one very simple example problem configuration consisting of four objects: a box, a table (which is nothing more than a thick plate), and a shelf (which is in fact a rotated box), as well as a little cube serving as the movable object. Figure 2.3 shows this simple setup.

2.4 Related Work

Most research in manipulation planning deals with the creation of the manipulation graph and extraction of a manipulation path from it [Alami *et al.*, 1995; Siméon *et al.*, 2004; Nielsen and Kavraki, 2000; Koga and Latombe, 1994; Amato and Wu, 1996].

There have already been approaches to extract complexity from domain-independent planning problems. With their hybrid planning system STAN4 [Fox and Long, 2001], Fox and Long try to identify automatically subproblems from a domain which are better solved with specialized algorithms. In particular, they extract path planning subtasks from the domain description based on information gathered by the automated domain analysis tool TIM [Fox and Long, 1998]. The domain is then abstracted from these subproblems and solved seperately by a forward planner which uses an estimation of the path planning subtasks in its heuristic. The path planning problems

are then reintegrated once an action has been chosen for addition into the resulting plan.

Srivastava [Srivastava, 2000] proposes a decoupling of resource reasoning and planning task. His approach, implemented in the RealPlan planning system, is to identify resources in a planning task and then to create a partially-ordered solution plan for the resource-abstracted variant of the planning task which ignores resource usages, i.e. treats resources as if they were always available. From this plan, a number of constraints is derived, most of which are based on resource spans provided by the plan. A resource span defines a pair of actions which occupy and release a certain resource (For example, in the case of a transport robot, the actions of loading and unloading would form a resource span.). To each span belongs a set of variables defining, among other things, the resources used and the level of the action in the final plan. These yield constraints on the ordering of actions and on the resource usage. These contrainsts[2] form a CSP[3] which is solved with a standard CSP solver. In the final step, the resource-abstracted plan and the policies derived from the CSP problem are combined in a solution plan.

On the one hand, the automatic separation of subproblems in STAN4 and RealPlan is more general than ours since they automatically determine subproblems. On the other hand, however, our approach using external modules allows the specification of arbitrary additional information not restricted to a specialized subproblem such as path planning. This is because the extraction of subproblems is done in the design phase of the domain.

Procedural attachments have already been developed for search engines. In [Jónsson, 1996], Jónsson formally defines a framework, which, based on general definitions, defines how search engines and procedures interface.

Procedural attachments in combination with planning languages have also been mentioned. In [Golden, 2003], Golden introduces the DPADL[4] planning language. It is designed explicitly for Data Processing; actions are not allowed to change the world except by creating new objects (input/output design). DPADL is an object oriented planning language which supports procedural attachments through the embedding of Java code into constraints and action effects. Although with some minor changes, DPADL might also be used for general purpose planning domains, and already sup-

[2]For the sake of simplicity we do not discuss all constraints which are used in the actual RealPlan algorithm.

[3]CSP – **C**onstraint **S**atisfaction **P**roblem: A CSP consists of a finite set of variables, each with a finite range of possible values, and a set of constraints. A solution to a CSP is an assignment of all variables satisfying the contraints.

[4]DPADL – **D**ata **P**rocessing **A**ction **D**escription **L**anguage

ports the extensions we need in its present form, it is only implemented by the IMAGEbot system which only acts in one specialized domain. What we need is not only a general purpose planning language but also a non-specialized planner supporting it.

Chapter 3

PDDL/M - Procedural Attachments in PDDL

PDDL, the **P**lanning **D**omain **D**efinition **L**anguage, first published in [McDermott and others, 1998] is the *de facto* standard language for describing planning domains.

In this context, planning is the task of transferring a certain initial state into a goal state using a fixed set of actions.

A PDDL definition consists of two parts: the domain and the problem-task definitions. The domain definition provides the predicates used to describe the facts that form a state of the domain, and the actions which alter these facts, thus changing the state. The problem-task definition basically contains the initial and goal states. PDDL is derived from STRIPS [Fikes and Nilsson, 1971] which in its basic form only allows a state description consisting of simple logical facts as described in section 2.2. Actions only set or negate these facts. PDDL, however, extends beyond STRIPS by adding quantified conditions and conditional effects forming the ADL [Pednault, 1989] action definitions. In addition to that, PDDL provides many more features, one of which is of interest in the current context namely that numerical state variables can be used in addition to symbolic facts.

It is important for the understanding of the following to have a basic knowledge of PDDL. We refer to [McDermott and others, 1998] and [Edelkamp and Hoffmann, 2003] for a detailed introduction to PDDL.

In this chapter, we describe an extension to PDDL 2.2 , which allows the usage of procedural attachments (called *external modules* in the following) to provide the planner with additional information during the planning process. These modules are intended to be independent from the planning system but tightly coupled with the problem definition. This means that in PDDL/M a planning problem does not only consist of the domain and task specifications

but also defines an arbitrary number of external modules. These modules are implemented separately from the planner but can be loaded dynamically and accessed through a well-defined API[1].

It is not sufficient for the module to only know everything about the domain in general. It must also know about the current planning situation. If, for example, we would like to put down a bottle on a table, we need to know which objects are already on the table. This might be represented symbolically in the domain by the use of a predicate like *ontop(table, bottle)*. The module in some way should be able to access all *ontop* instances in the current planning situation in order to make a useful decision. Otherwise, we would have to encode all necessary information into the parameters of the action. This would create inconveniently big action definitions while the information accessible to the module would still be restricted in complexity by the fixed number of parameters. What we need, then, is a planner interface to be used by the PDDL/M modules to obtain information about the current planning situation from the planning system.

The extension described here consists of two main parts. In section 3.2, we describe the syntax changes to the PDDL, and in section 3.3, we introduce the plug-in API used to implement the external modules. First, however, we will give a quick overview of the module structure in section 3.1.

Section 3.4 deals with implementation. We will present a support library which can be used to implement support for external modules in planners.

3.1 External Modules

As mentioned above, the general idea of an external module is to provide procedural attachments in PDDL. An external module is a piece of software which can be accessed by the planner during the planning process to answer certain queries like constraint checking. One can think of a module as a *plug-in* for PDDL/M planning systems.

Every module implements the method *init* which can be used by the planning system to initialize the module for first usage. A typical task for the initialization would be to load a configuration file. The parameter of this method is an array of options. We will explain this in detail later on. In addition to the *init* method, every module implements a method specific to its type (the types of modules will be defined in section 3.2.1) which is invoked by the planner to answer a query from this module.

[1] Application Programming Interface

To support PDDL/M modules, a planning system needs to implement a *callback* method which provides access to the current planning state for the modules. In section 3.3, we will define the layout of this API in detail.

3.2 Syntax

PDDL/M introduces the idea of external modules into classical PDDL. To achieve this, we introduce new syntax in the domain and problem-task definitions. In this section, we will describe this new syntax. See appendix A for a complete description of the new syntax in BNF.

We introduce here a new requirement to be used in domains which depend on external modules as described below: *module*. See Figure 3.1 for an example.

```
(:requirements :strips :typing :module)
```

Figure 3.1: Example: Usage of the *module* requirement

When using the `module` requirement a new section: `modules` is allowed in the domain specification. In the `modules` section, the modules actually used in the domain are specified. It consists of several module specifications. Each of them consist of three mandatory parts:

1. A unique identifier to reference the module anywhere in the domain or, as we shall see in the problem-task definition.

2. The type of module. Currently, two types of modules are defined, as described in section 3.2.1.

3. The binary which implements the external module. This may be an absolute path to the specific dynamically loadable library file or simply the name of a library, thus leaving resolution of the name to the dynamic linker.

Moreover, an arbitrary number of parameters may be specified after the module identifier. These parameters may include variables in the same manner as for a predicate definition in classical PDDL, which will be instantiated in an action. In case of the effect module, this is the place to specify the function symbols defining the numerical variables modified by this module (see section 3.2.1 for details). See Figure 3.2 for an example of the definition of two modules `module1` and `module2`.

```
(:modules
    (module1 ?var1 ?var2 conditionchecker @modulelib)
    (module2 ?var1 ?var2 function1() effect @modulelib)
)
```

Figure 3.2: Syntax of the *modules* section

In this example, the domain uses two external modules, one of the type `conditionchecker` and one of the type `effect`, both to be loaded from the library `modulelib`. The modules will be referred to as `module1` and `module2`. Both modules have two parameters, and the effect module `module2` changes the value of the function symbol `function1`.

3.2.1 Module Types

As mentioned above, every external module used in a PDDL/M domain has a type which determines its usage in the domain. PDDL/M introduces two different module types: the *Condition Checker* and the *Effect* module. We will now describe these module types in greater detail.

Condition Checker

A *Condition Checker* module provides a new type of contraint to be used in an action precondition in case the domain description itself does not provide enough information to handle the condition symbolically.

For a *Condition Checker* module to be used in an action, it needs to be specified just like a classical precondition. The only difference is that the module identifier enclosed in square brackets replaces the predicate name, so that the planner knows it is dealing with a module condition rather than a classical (symbolical or numerical) one.

Figure 3.3 shows an example of a Condition Checker module in a domain description. The module *puthelper* is used to determine if a certain combination of parameters is suitable for the action *putdown* or not.

Effect

An *Effect* module provides a new means of modifying the values of numerical state variables via an action effect. The *Effect* module only modifies the values of numerical functions and not those of symbolic facts. This limitation can easily be resolved if modification of symbolic facts using an external

```
(:modules
     (puthelper
      ?o - object
      ?p - place
      conditionchecker
      @libputhelper)
)

(:action putdown
     :parameters ( ?o - object
                   ?p - place
     :precondition ( ([puthelper] ?o ?p) )

[...]
```

Figure 3.3: Example: Usage of the Condition Checker module

PDDL/M module should ever prove desireable. However using module conditions in an action effect would produce the same effect.

The syntax of the *Effect* module follows that of a classical symbolic effect except that the module identifier enclosed in square brackets replaces the predicate name. As the functions whose values are modified by the module effect are fixed and have already been specified in the module definition, they are omitted here.

Figure 3.4 shows an example of an effect module with two parameters o and p which modifies the values of two numerical functions a and b.

3.2.2 Module Options

Each module can be provided with a set of initial options via the PDDL problem-task description file. In the initialization phase of the planner, these options will be passed on to the initialization method described in section 3.3.

To specify options for specific modules, we introduce a new section to be used in the problem-task file, namely *moduleoptions*. This section contains one or more entries. Each entry consists of the identifier of the module and an arbitrary number of options separated by commas. Each option is encoded by its name, followed by a "=" sign, and its value.

Figure 3.5 shows an example of how to use the new syntax in a PDDL problem-task file.

```
(:modules
     (grabhelper
      ?o - object
      ?p - place
      a()
      b()
      effect
      @libgrabhelper)
)

(:action grab
     :parameters (?o - object ?p - place)
[...]
     :effects ([grabhelper] ?o ?p)
)
```

Figure 3.4: Example: Usage of the Effect module

```
(:moduleoptions
     (module1 option1=value1,option2=value2)
)
```

Figure 3.5: Example: Usage of the module options

The external module referred to as `module1` will be initialized with two options, `option1` and `option2`. They will be assigned the values `value1` and `value2` respectively.

3.3 The Module API

In this section, we define the interfaces for the different module types. These can be used to implement module support in a planner and to create new modules.

We define an interface in *C* here but bindings for other programming languages should be easy to implement.

The module interface is very simple. It consists of three methods and some simple data structures. Each module has an initialization method *init* which needs to be called before using the module. Although this method might have no effect with some modules, it must be implemented. The init method has two parameters: the number of options and a list of option strings (each option string consists of option name and option value, separated by a comma). This list of options has to be created by the planner using the values in the *moduleoptions* section in the problem-task file, as described in section 3.2.2.

In addition to the init method, the API defines the actual module methods *checkcondition* and *applyeffect*. Depending on the type of module, only one of these needs to be implemented. Both methods have as parameters the name of the action, a list of parameters, and a pointer to a planner call-back method which can be used to access the current planning state. A planner supporting external modules has to implement this call-back method.

The parameters of the call-back method are a list of predicates (facts) and a list of numerical variable values, both of which the planner must fill if the call-back method is used by the module.

See appendix B for a complete description of the API and all the data structures.

3.4 PDDL/M Support Library

The PDDL Module Support Library consists of *C* declarations of the data structures and interfaces used in the Module API defined in section 3.3, and of methods to load and unload PDDL modules by name.

A module is represented by the structure defined in figure 3.6 and consists of an internal handle used by the PDDL Module Support Library to load and

```
typedef struct _Module {
  void* handle;

  int type;

  int (*init)( int argc, char** argv );
  int (*checkcondition)( const char*,
                         ParameterList*,
                         modulecallback* );
  int (*applyeffect)   ( const char*,
                         ParameterList*,
                         double* values,
                         modulecallback* );
} Module;
```

Figure 3.6: PDDL Module *C* interface

unload the module, of the type of the module, and of pointers to the *init*, *checkcondition* and *applyeffect* methods.

Depending on the the type of the module, not all methods need to be implemented. The `checkcondition` method is only implemented by *conditionchecker* modules, while the `applyeffect` method is only implemented by *effect* modules. Methods which are not implemented have NULL pointers in the `Module` structure.

The type of module is defined as a simple enumeration, as shown in figure 3.7.

```
typedef enum _ModuleType {
  CONDITION_CHECKER,
  EFFECT
} ModuleType;
```

Figure 3.7: PDDL Module *C* Type

The PDDL Module Support Library provides two methods for loading and unloading PDDL modules by name. The loading method returns a pointer to an instance of the *Module* structure, or a NULL pointer in case no module with the provided name could be found. The unloading method completely removes the *Module* instance from memory.

```
Module* loadModule( const char* name );
void unloadModule( Module* );
```

Figure 3.8: PDDL Module loading interface

In addition it implements methods for memory management in combination with the different data structures, i.e. construction and destruction methods for Parameters, Predicates, Functions, and lists of all three.

Chapter 4

FF/M - An Implementation of PDDL/M

An extension to a planning language is useless without a planner supporting the new syntax. In this chapter, we present an implementation of the PDDL/M syntax as an extension to the award-winning FF planning system by Jörg Hoffmann [Hoffmann and Nebel, 2001] or, more precisely, to its big brother Metric-FF which extends FF to numerical state variables [Hoffmann, 2002]. So in the following, when we talk about FF, what we really mean is Metric-FF.

First, we will give an overview of how FF works and talk about some implementation details. We then describe how the external module extension has been realized in FF.

4.1 Anatomy of a Planning System

FF is a forward planning system which relies on forward state space search, using a heuristic that works with a relaxed version of the planning problem to provide fast estimates for goal distances. Although FF supports full ADL, it compiles the ADL domain description down to a much simpler form internally by instantiating all parameters and eliminating quantifiers as well as negative symbolic preconditions. The result is a STRIPS planning task combined with numerical variables. In the following, we will use a slightly simplified version of this so-called propositional normal form, which is close enough to the original to explain the internal proceedings of FF.

We start by defining the formal concepts used, which are all based on a set of propositions P and a set of numerical variables $V = \{v_1, \ldots, v_n\}$.

Definition 4.1 (State) *A state S is a pair*

$$S = (facts(S), values(S))$$

where $facts(S) \subset P$ is a finite set of logical propositions, and $values(S) : V \mapsto \mathbb{Q}$ is a function which assigns a value to each numerical variable.

Definition 4.2 (Numerical Constraint) *A numerical constraint nc is a triple*

$$nc = (exp(nc), comp(nc), exp'(nc))$$

where $exp(nc)$ and $exp'(nc)$ are arithmetical expressions over V and the rational numbers, and $comp(nc) \in \{<, \leq, =, >, \geq\}$.

A numerical constraint nc is fulfilled in a State S ($S \models nc$) iff the value of $exp(nc)$ in S stands in relation $comp(nc)$ to the value of $exp'(nc)$ in S.

Definition 4.3 (Condition) *A Condition c is a pair $c = (p(c), nc(c))$ where $p(c) \subseteq P$ is a set of propositions and $nc(p)$ is a set of numerical constraints.*

A condition c is fulfilled in a state S ($S \models p$) iff $p(c) \subseteq facts(S)$ and $\forall n \in nc(c) : S \models n$.

Definition 4.4 (Numerical Effect) *A numerical effect ne is a triple*

$$ne = (var(ne), ass(ne), exp(ne))$$

where $var(ne) \in V$, $ass(ne) \in \{:=, + =, - =, * =, / =\}$, and $exp(ne)$ is an arithmetical expression over V and the rational numbers.

A numerical effect ne is applied to state S by updating $values(S)(var(ne))$ using the operator $ass(ne)$ and the value of $exp(ne)$ in S.

Definition 4.5 (Action) *An action o is a tuple*

$$o = (pre(o), add(o), del(o), ne(o))$$

where $pre(o)$ is a condition, the "add-list" $add(o) \subseteq P$ and the "delete-list" $del(o) \subseteq P$ are sets of propositions, and $ne(o)$ is a set of numerical effects.

The result of applying an action o to a state S is defined as follows:

$$Result(S, o) = \begin{cases} S' & S \models pre(o) \\ undefined & otherwise \end{cases}$$

where $facts(S') = (facts(S) \cup add(o)) \setminus del(o)$ and $values(S')$ is gained from $values(S)$ by applying all effects in $ne(o)$.

Definition 4.6 (Planning Task) *A planning task $\mathcal{T}$ is a tuple*

$$\mathcal{T} = (\mathcal{P}, \mathcal{V}, \mathcal{O}, \mathcal{I}, \mathcal{G})$$

where $\mathcal{P}$ is a set of propositions, $\mathcal{V}$ is a set of numerical variables, $\mathcal{O}$ is a set of actions, $\mathcal{I}$ is a state (the initial state), and $\mathcal{G}$ is a condition (the goal).

FF converts the planning task into a Linear Normal Form, defined as follows.

Definition 4.7 (Linear Normal Form) *A planning task*

$$\mathcal{T} = (\mathcal{P}, \mathcal{V}, \mathcal{O}, \mathcal{I}, \mathcal{G})$$

is in Linear Normal Form iff the following holds.

1. *Any expression used in the left hand-side of a numerical constraint and in the right-hand side of a numerical effect is of the form*

$$\sum_{i \in X} c_i * v_i$$

   where $c_i > 0$ and $X \subseteq \{1, \ldots, n\}$.

2. $\forall o \in \mathcal{O} : \forall n \in nc(pre(o)) : comp(n) \in \{>, \geq\}$

3. $\forall o \in \mathcal{O} : \forall n \in ne(o) : ass(n) \in \{:=, + =\}$

The translation into a Linear Normal Form is for the most part straightforward, using inverse variables to eliminate negative weights and replacing expressions like $exp = exp'$ with $exp \leq exp'$ and $exp \geq exp'$ or replacing $v- = exp$ with $v+ = (-1) * exp$ and so on. This is explained in detail in [Hoffmann, 2002]. The important point here is that every planning task can be converted into LNF.

Now, we can take a look at the heuristic of FF[1], which solves a relaxed planning task to provide a fast and optimistic estimate on goal distances. A relaxed planning task "ignores" the delete lists of actions and the numerical assignment effects.

[1]The heuristic presented here is not the only one used in FF. But presenting all of FF's internals would go beyond the scope of this thesis. We refer to [Hoffmann and Nebel, 2001] for a complete discussion of FF's inner workings.

Definition 4.8 (Relaxed Action) *Let* $o = (pre(o), add(o), del(o), ne(o))$ *be an action in Linear Normal Form. For the relaxed version of* o *then applies*

$$relaxed(o) = (pre(o), add(o), \emptyset, \{n \in ne(o) \mid ass(n) = +=\})$$

Definition 4.9 (Relaxed Planning Task) *Let* $\mathcal{T} = (\mathcal{P}, \mathcal{V}, \mathcal{O}, \mathcal{I}, \mathcal{G})$ *be a planning task in Linear Normal Form. The relaxation* $\mathcal{T}'$ *of* $\mathcal{T}$ *is then defined as* $\mathcal{T}' = (\mathcal{P}, \mathcal{V}, \mathcal{O}', \mathcal{I}, \mathcal{G})$ *where*

$$\mathcal{O}' = \{relaxed(o) \mid o \in \mathcal{O}\}$$

FF runs a GRAPHPLAN [Blum and Furst, 1995] algorithm on the relaxed planning task. A plan graph is a layered graph with two iterating types of layers: fact layers and action layers. The first layer contains all facts valid in the initial state of the planning task. Action layer i contains all actions whose preconditions are met in fact layer i (this includes no-op actions which propagate facts from one fact layer to the next) and yields fact layer $i+1$ by applying all effects of these actions. If the task is solvable, the creation of the plan graph is finite and ends with fact layer n in which all goal conditions are met. The GRAPHPLAN algorithm then searches backwards through the graph to extract a solution plan[2].

The most important feature of the relaxed planning task is its monotonicity: due to the absence of negative symbolical effects and the fact that we only evaluate numerical effects which increase values of numerical variables, any condition which is true in state S is also true in every superstate S' ($S \subseteq S'$). This leads to two major simplifications in the GRAPHPLAN algorithm. Building the plan graph is polynomial in $|\mathcal{O}|$ because every action level introduces at least one new action. Once each action has been selected for insertion in the graph, the next fact layer contains all reachable facts and thus also the goal (if the task is solvable). Furthermore, since no action in the relaxed planning task removes any fact or decreases the value of any numerical variable, no two actions can ever be mutually exclusive[3]. So the algorithm will never backtrack and thus a relaxed solution $\langle O_0, \ldots, O_m \rangle$ can be extracted in polynomial time. This yields the goal distance heuristic of FF

$$h(S) := \sum_{i=0,\ldots,m} |O_i|$$

[2]We present a simplified variant of the GRAPHPLAN algorithm for a better understanding of FF's heuristic. For a detailed discussion we refer to [Blum and Furst, 1995].

[3]In a plan graph two actions are mutually exclusive if one of them negates a precondition of the other one or if one negates a fact which is in turn set by the other one.

The base search algorithm of FF is an *enforced* variation of hill-climbing, combining local and systematic search to select always one best successor to the current state. Enforced hill-climbing differs from classical hill-climbing in that it performs a complete breadth-first search starting from the current state S instead of being restricted to a local search. Thus, it will always find the next best successor state S'. It then adds the path from S to S' to the plan and iterates the search in S'. Due to the often simple structure of the search space, the next best state is usually only a few steps away. During this search, a state S is evaluated using the GRAPHPLAN heuristic described above.

Should the enforced hill-climbing algorithm fail to find a solution plan, FF will fall back to a best-first search.

4.2 Extending FF to External Modules

Now, we will see how FF has been extended to support external module preconditions (conditionchecker modules) and effect modules. In a domain using external modulesm an action is extended as follows:

Definition 4.10 (Condition/M) *A condition with external module support p is a triple $c = (p(c), nc(c), mc(c))$ where $p(c) \subseteq P$ is a set of propositions, $nc(p)$ is a set of numerical constraints, and $mc(c)$ is a set of module conditions.*

A condition c is fulfilled in a state S ($S \models p$) iff $p(c) \subseteq facts(S)$, $\forall n \in nc(c) : S \models n$, and $\forall m \in mc(c) : S \models m$ where $S \models m$ iff the call to `checkcondition` *with parameters determined from S in the external module belonging to m returns a value $\neq 0$.*

Definition 4.11 (Action/M) *An action with external module support o is a tuple*

$$o = (pre(o), add(o), del(o), ne(o), me(o))$$

where $pre(o)$ is a condition, $add(o) \subseteq P$ and $del(o) \subseteq P$ are sets of propositions, $ne(o)$ is a set of numerical effects, and $me(o)$ is a set of module effects.

The result of applying an action o to a state S is defined as

$$Result(S, o) = \begin{cases} S' & S \models pre(o) \\ undefined & otherwise \end{cases}$$

where $facts(S') = (facts(S) \cup add(o)) \setminus del(o)$, and $values(S')$ is gained from $values(S)$ by applying all effects in $ne(o)$ and calling `applyeffect` *in all modules in $me(o)$ with the parameters from S.*

The module effects which can be regarded as a different form of assignment (:=) effects are simply ignored in the relaxed planning task. The latter is a straightforward extension of the heuristic in [Hoffmann, 2002] which also ignores numerical assignment effects. The module conditions, however, can be treated in different ways. We could simply ignore them, too, and assume they were always true (as in the current implementation of FF/M). This would mean a relaxed planning task with external modules would not differ from that in definition 4.9.

The other possibility is to treat module conditions like symbolic facts. This can be achieved by introducing artificial facts for every module condition in the relaxed planning task. These artificial facts are all false in the beginning of the GRAPHPLAN algorithm, which means that none of them is included in the first fact layer of the plan graph. Now, imagine the creation of action layer i, and an action with module condition mc whose symbolic conditions and numerical constraints are all met in fact layer i. In order to decide if this action can be applied, we check if the artificial fact corresponding to mc is included in fact layer i. If this is the case the action can be applied. If fact layer i does not include the artificial fact for module condition mc, we call the appropriate `checkcondition` function in the external module belonging to mc. If the function returns a value $\neq 0$ and the module condition is thus met, we include the artificial module fact corresponding to the module condition in the next fact layer and apply the action[4].

So the basic idea is that in the relaxed planning task, a module condition is met on all levels of the plan graph that follow the layer in which the condition was met for the first time. Algorithm 4.1 shows this in detail, assuming a relaxed plan does exist. The artificial fact corresponding to a module condition mc is depicted by $pseudofact(mc)$.

As in [Hoffmann, 2002], one can show that algorithm 4.1 fails iff the relaxed task is unsolvable. The proof given there is not changed by the introduction of the artificial module facts. In fact, they are treated in the exactly in the same way as classical facts.

It is, however, not as easy to extend the time analysis from FF to our algorithm. The original GRAPHPLAN algorithm from FF terminates in polynomial time in the size of the task and the number of time steps (i.e. graph layers) built. If we treat a module call as an atomic operation of

[4]It should be made clear though, that we probably need a "relaxed" version of the *checkcondition* module since the module may depend on certain characteristics of the domain which are lifted in its relaxed version. An example is a module which checks the space left on a table based on the objects already on the table. In the relaxed version of this task an object will stay on the table once it has been put there. Thus, the module would fail even if, in "reality", objects have been removed from the table.

Algorithm 4.1 Create relaxed plan graph for State S

$P_0 := facts(S)$
for $i := 0$ to n **do**
  $max_0^i := values(S)(v_i)$
end for
$t := 0$
while $p(\mathcal{G}) \not\subseteq P_t$ or ($(v_i, \geq [>], c) \in nc(\mathcal{G})$ and $max_t^i \not\geq [>]c$) **do**
  $O_t := \emptyset$
  $P_{t+1} := P_t$
  for $i := 0$ to n **do**
    $max_{t+1}^i := max_t^i$
  end for
  for all $o \in \mathcal{O}$ **do**
    if useAction(o, t) **then**
      $O_t := O_t \cup \{o\}$
      $P_{t+1} := P_{t+1} \cup add(o)$
      for all $(v_i, +=, c) \in ne(o)$ **do**
        $max_{t+1}^i = max_{t+1}^i + c$
      end for
    end if
  end for
  $t := t + 1$
end while

Algorithm 4.2 *useAction*: Determine if operator o should be used in step t (P_i denotes the facts true in time step i)

if $p(pre(o)) \subseteq P_t$ and $\forall (v_i, \geq [>], c) \in nc(pre(o)) : max_t^i \geq [>]c$ **then**
  for all $m \in mc(pre(o))$ **do**
    if $pseudofact(m) \notin P_t$ **then**
      if module(m).checkcondition(P_t, $max_t^0, \ldots, max_t^n$) **then**
        $P_{t+1} := P_{t+1} \cup \{pseudofact(m)\}$
      else
        return false
      end if
    end if
  end for
  return true
else
  return false
end if

time complexity $O(1)$, this proposition still holds[5]. Yet in fact, a module call can have arbitrary complexity making a complete time analysis virtually impossible.

4.3 Improving FF/M

One of the most important steps in the preprocessing of FF is the full instantiation of all predicates and actions, and the creation of an integer representation of the domain. This reduces the memory and computation overhead during the actual planning to a minimum.

To accomplish this, FF fully instantiates[6] all predicates and actions, thus creating a simplified but probably larger representation of the domain. Each entity in this simplified representation including action preconditions and effects is assigned an integer. The numerical part is converted into an integer representation of the LNF as described above. Thus, the domain is converted into the simplest version possible: a collection of atomic facts and actions modifying these facts, all represented by arrays of integers.

We extend the integer representation of the planing domain in FF in an equally simple manner. Each module precondition will be completely instantiated and assigned an integer. Effect modules will be integrated into the representation of the LNF as described above.

A straightforward improvement in the external module support of FF/M would be to base the communication between FF/M and the external modules entirely on the internal integer representation of FF. Each module would be informed about the integer representation in the preprocessing phase (e.g. through the use of the `init` function). This would increase speed by avoiding many string copy operations. The disadvantage is of course that the implemented modules would be dependent on FF/M and could not be used anymore in combination with another planner that uses a different internal representation.

[5]Treating a module call as an atomic action would be like using an oracle in complexity theory.

[6]FF does actually not work on the full instanciation of all predicates and actions but minimizes this set to those that are actually used in the problem. One example is that predicates (here we talk about the symbolical instanciated ones) that can never be true because no action exists to make them true are ignored completely.

Chapter 5

The Geometric Domain

In chapter 2, we have shown that plain PDDL is not well-suited to model geometric domains and have introduced the idea of an oracle-like or black-box extension to PDDL using procedural attachments.

In chapters 3 and 4, we have described this extension, defined the new syntax, and provided a reference implementation in the form of an extension to the award-winning FF planning system.

Coming back to our original problem of handling grasp poses, we will now put the new PDDL syntax to use. We intended to design a robot gripper which is able to grasp objects using different grasp poses. A bottle, for example, may be grasped at the top (when getting it out of a box) or from the side (to put it into the fridge). The grasp pose cannot be changed while the gripper is holding the object in a fixed grasp. In order to change the pose, the robot gripper needs to put down the object and pick it up again using the new pose. So the main issue we need to address is a mechanism to test if a certain pose is "compatible" with a certain object and a certain action.

We will approach this problem in two steps. First, we will use discretized poses which allow to solve part of the problem using plain PDDL planning. We then will replace these discretized poses by free poses defined by angles in three-dimensional space. Before treating these two different pose encodings in detail, we will outline the basic PDDL domain description which they are based on, as well as the encoding of the additional domain information not available to the planner (referred to as the full domain description in figure 2.2).

It should be noted that the domains presented here are merely examples providing a proof of concept, a feasibility study of the usefulness of PDDL/M for the handling of geometric problems in planning domains. By this, we do

not claim to solve the problem exhaustively, but intend to present a promising approach.

5.1 Basic domain description

The basic layout of the domain description is the same with both types of pose encoding. Since we would like to pick up objects and put them down again, we need two object types: those which can be picked up and moved around and those which serve as surface. Each movable object needs to be either on top of a surface object or in the grasp of the robot gripper at all times. This is described by two predicates.

```
(ontopof ?client - object ?server - base)
(in_hand ?o - object)
```

We also introduce a predicate which is true if *(in_hand)* is false for every object in the domain.

```
(hand_free)
```

The problem necessitates the use of two very simple actions in our geometric domain: The *grab* action is used to pick up a movable object o from a base object b. This requires the gripper to be free and o to be on top of b. As a result of the *grab* action the gripper is not free anymore, and o is not on top of b anymore, but *in_hand*. See figure 5.1 for the basic layout of the grab action.

The *putdown* action is used to put down a movable object o currently grasped by the gripper onto a base object b. Here the precondition is slightly simpler, as mentioned above: o must be in the robot gripper. As a result the gripper is free again, and o is no longer *in_hand* but on top of b. See figure 5.2 for the basic layout of the putdown action.

5.2 World Model

For the design of a geometric domain one element is indispensable, namely the world model describing the geometric objects used in the domain. In our case, these objects are described as sets of triangles in three-dimensional space. For the sake of simplicity, we associate with each object a set of triangles parallel to the x-y-plane. In the case of a movable object these triangles define its base area, while for a base object they define the surface on which movable objects can be placed. Each object has a unique name corresponding to the name used in the domain description.

```
(:action grab
   :parameters (
       ?client - object
       ?server - base
   )
   :precondition (and
       (ontopof ?client ?server)
       (hand_free)
   )
   :effect (and
       (not (ontopof ?client ?server))
       (not (hand_free))
       (in_hand ?client)
   )
)
```

Figure 5.1: Basic layout of the grab action

The following definitions may seem trivial but they will simplify later discussion of the objects.

Definition 5.1 (Triangle) *A triangle t is a triple*

$$t = (p1(t), p2(t), p3(t))$$

where $p1(t), p2(t), p3(t) \in \mathbb{R}^3$ are the points defining the triangle in three-dimensional space.

Definition 5.2 (Movable Object) *A movable object o is a pair*

$$o = (tris(o), floor(o))$$

where $tris(o)$ is the set of all triangles in the object and $floor(o)$ is the set of triangles in the base area.

Definition 5.3 (Base Object) *A base object o is a pair*

$$o = (tris(o), sur(o))$$

where $tris(o)$ is the set of all triangles in the object and $sur(o)$ is the set of surface triangles.

Thus, an arbitrary set of objects may be defined to be used in an instance of the geometric domain.

One particular movable object which must be defined is the robot gripper. Since it is encoded implicitly into the domain, its name is fixed.

```
(:action putdown
   :parameters (
       ?client - object
       ?server - base
   )
   :precondition
       (in_hand ?client)
   :effect (and
       (hand_free)
       (not (in_hand ?client))
       (ontopof ?client ?server)
   )
)
```

Figure 5.2: Basic layout of the putdown action

5.3 Discretized Poses

A discretization of grasp poses allows us to handle them using symbolic planning constructs. Therefore, we introduce a new object type *pose* and extend the *in_hand* predicate to describe the pose that is currently used to hold the object.

```
(in_hand ?o - object ?p - pose)
```

We also introduce a new predicate stating that a certain pose can be used with a certain movable object[1]

```
(supports_pose ?o - object ?p - pose)
```

Now every possible grasp pose has an equivalent object in the domain. These poses will be used in the *grab* action when checking if the object can be grasped with the pose in question. We must thus add another parameter to the action representing the pose used to grasp the object. By adding the *supports_pose* predicate to the precondition of the action we can make sure that the an object can actually be grasped with a certain pose.

The *putdown* action is modified in the same way. We also add to it a third pose parameter. Evidently, the object must be grasped in the pose that is used to put it down. This is ensured by the extended *in_hand* predicate which includes the pose used.

[1]Although the information provided by the *supports_pose* predicate could also be encoded implicitly into the external module, testing the pose with an additional symbolic construct may help to optimize the planning process further.

These extensions, however, do not quite solve the problem yet (and incidentally do use PDDL/M). We have ensured that only poses "compatible" with an object are used to grasp it but we have not addressed the actual geometric problem yet. So far, our encoding does not take into account if it is actually possible to grasp an object in the current situation with regard to its geometric surroundings. This is where an external PDDL/M module comes in.

We will now design an external PDDL/M module to handle the geometric part of our domain. This means that the module will be used to test if it is actually possible to use a certain pose with a certain object in the current geometric situation. Let us first take a look at the module from the planner's point of view and assume we already have such a module and only need to integrate it into our domain. (We will not discuss the loading of the module here. For the complete domain description, see appendix C.1. Let us assume the module has been loaded into our domain using the name *grabhelper*.)

All we have to do to integrate the external module into our domain is to add the following module condition to the precondition sections of both actions.

```
([grabhelper] ?client ?server ?p)
```

In the case of the *grab* action we need to know if it is possible to grasp the movable object *client* from the base object *server* using the pose p. In the case of the *putdown* action we need to know if it is possible to put down the movable object *client* onto the base object *server* if the robot gripper holds it using pose p. In fact, both queries are identical. We need to know if the two objects involved and the gripper collide in any way (except for the trivial collisions when holding the movable object and putting it down onto the base object). In this thesis, we simplify this problem and will only check if the gripper collides with the base object when it puts down the movable object. So we assume that the movable object always "fits" onto the base object.

Having integrated the condition module into our domain and defined what is has to do we will now take a look at the implementation.

We will restrict the discretized poses used here to *horizontal* and *vertical*. In our simple setup, these poses are fixed in the implementation of the module[2] and translated into a horizontal or vertical orientation of the gripper (using one fixed direction vector $(0, 0, 1)$ or $(0, -1, 0)$ respectively).

[2]It is of course possible to create a module which handles arbitrary discretized poses by use of some conversion procedure. For our purposes, however, it is sufficient to have two fixed poses.

The implementation of the external module is straightforward. The *check-condition* function as defined in section B.2 will be called with a combination of base object b, movable object m, and grasp pose p. It first calculates the translation t of the movable object onto the base object. Due to the restrictions on the objects and to their simple structure, this calculation is simply based on the *floor* and *surface* properties of the objects: $t = center(surface(b)) - center(floor(m))$. Following the translation of the movable object, the gripper needs to be positioned according to p. Since the default orientation of the gripper corresponds to the *vertical* orientation, it only needs to be rotated if $p = horizontal$. Then, the bounding box of the movable object $bb(m)$ is determined. If bc is the center point of the top side (if $p = vertical$) or the front side (if $p = horizontal$) of $bb(m)$, the translation of the gripper can be described by $bc - center(floor(gripper))$.

Once both m and the gripper have been positioned correctly, a collision query between b and the gripper answers the question if pose p is usable with m and b. The actual collision query is done using PQP, the **P**roximity **Q**uery **P**ackage[3].

5.4 Free Poses

The discretized poses presented above have one major disadvantage. Let us assume we intended to create automatically a model of the surrounding world by scanning all the objects and classifying them according to an ontology. If we were restricted to discretized poses, we would either have to encode in its own symbol every possible pose of every object, or restrict ourselves to a small set of poses. In the first case, we would most likely end up with a large number of discretized poses which would massively increase the size of the domain description. In the latter case, we would of course lose accuracy since we would have to "force" predefined poses onto the objects or merge similar poses into one. So none of the approaches is really feasible.

One solution to this problem is free poses as presented in the following. A free pose, in contrast to a discretized one, is not defined by a single logical symbol but by numerical variables. These variables directly (or indirectly, using some mathematical function) describe the grasp pose. Thus, a free pose naturally yields the translation and rotation values of the robot gripper necessary to grasp the object in question. While the number of grasp poses describable through discretized poses is limited to the number of symbols, free poses allow an arbitrary number of poses to be described with the same set of numerical variables.

[3]PQP Home page: `http://www.cs.unc.edu/~geom/SSV/`

One possible definition of a free pose is the following, which is no more than a placement of the gripper in three-dimensional space.

Definition 5.4 (Free Pose) *A free pose p is a tuple*

$$p = (rot_x(p), rot_y(p), rot_z(p), tr(p))$$

where for $i \in (x, y, z)$, $rot_i(p) \in \mathbb{R}$ is the rotation of the robot gripper around the i axis, and $tr(p) \in \mathbb{R}^3$ is the respective translation vector.

This is the definition we are going to use in this thesis. Nevertheless, we will only work in our implementation with the three rotation values, while the translation $tr(p)$ will be calculated through the size of the object to be grasped and the rotation values, as illustrated in the following.

Lez us suppose we wanted to calculate the translation of our gripper for the rotation values of pose p and movable object o. Let $center(T) \in \mathbb{R}^3$ be the center point of a set of triangles T (for the simple forms we are using, it is sufficient to determine only the mean point of all points in the object: $center(o)_i = (max_{p \in tris(o)}(p_i) - min_{p \in tris(o)}(p_i))/2$ for $i \in (x, y, z)$ where $p \in tris(o)$ iff $\exists t \in tris(o) : p \in \{p1(t), p2(t), p3(t)\}$). Let $dir \in \mathbb{R}^3$ be the direction vector of our gripper after the rotation. By direction of the gripper we mean its intuitive orientation $dir = center(floor(gripper)) - center(sur(gripper))$. The straight line $s(t) = center(o) + t * dir$ then intersects o in two points. (We assume every object to be convex.) Of these, we are interested in the one that lies "further down" the straight line, i.e. if $s(t_1)$ and $s(t_2)$ are the two points of intersection, the one we focus on is $q = s(min(t_1, t_2))$. If we think of the gripper stabbing through the object, q is the point of entry. The translation corresponding to pose p then is simply $t(p) = q - center(floor(gripper))$.

We will now extend our geometric domain to support free poses, including a modified domain description and a more elaborate external module.

Unlike for the discretized pose domain, we will not select one specific pose to be used in each action. Instead, we will maintain a set of *possible poses* stating which poses *could* be used in the current situation. Here, *current situations* refers to the state of the world between actions. A current situation could be the gripper holding an object. In the subsequent execution of the plan, any of the poses in the associated set could be used, so the final grasp poses may be selected after the planning process. Such post-processing may include the prevention of poses that force the robot gripper into a problematic position. A problematic position, for example, could be one straining the hardware. It is unnecessary to select specific grasp poses during the planning process since this will not change the resulting plan.

Thus, postponing the selection of a final pose can only increase the quality of the final plan.

5.4.1 PDDL/M Domain

We will use three numerical intervals representing the possible rotations along all three axes:

```
(:functions
   (pose_rot_x_min)
   (pose_rot_x_max)

   (pose_rot_y_min)
   (pose_rot_y_max)

   (pose_rot_z_min)
   (pose_rot_z_max)
)
```

These three intervals define the set of poses that *could* be used in the current situation. They each define a range of rotation angles valid in the current planning situation. Thus a pose p is valid iff for $i \in (x, y, z)$, $(pose\_rot\_i\_min) \leq rot_i(p) \leq (pose\_rot\_i\_max)$.

This is a rather simple pose set encoding, which has one major disadvantage. It cannot describe arbitrary sets of poses. This is due to the mutual dependancy of the intervals[4] and the restriction to only one interval for every angle. Therefore, it is not possible to describe a set like $\{(0, 0, 0), (0.5, 0, 0)\}$ because this would also include all x-rotation values between 0 and 0.5.

We will use the pose encoding despite these limitations since they do not influence the main problem. We will discuss some ideas of how to improve the encoding in chapter 7.

Now that we know how to handle a single grasp pose, let us take a look at the PDDL/M domain description, which is again an extension of the basics defined in section 5.1.

Having added the six numerical variables defining the grasp poses currently possible, we will now equally extend the *grab* and the *putdown* actions with two external module calls. As with the discretized poses, we add a

[4] An x-rotation value x_1 may allow for a much larger interval of y-rotation values than x_2. Nonetheless, we must choose the smaller one since any combination of rotation values from the intervals has to describe a valid grasp pose.

```
(:action grab
   :parameters (
      ?client - movable
      ?server - base
   )
   :precondition (and
      (ontopof ?client ?server)
      (hand_free)
      ([grabhelper] ?client ?server)
   )
   :effect (and
      (not (ontopof ?client ?server))
      (not (hand_free))
      (in_hand ?client)
      ([putty] ?client ?server)
   )
)
```

Figure 5.3: *grab* action for free poses

module condition to the precondition section of both actions. Furthermore, we add a module effect which is used to update the three pose intervals as described below. Figure 5.3 shows this extension using the *grab* action as an example. See appendix C.2 for the full PDDL/M domain description.

The three intervals defined above are only modified by these module effects. They are not actually used in the PDDL domain. The planner only drags them along so that the external modules can access their current values.

We should probably clarify one issue here. If looked at from a semantic point of view, the "effect" of setting the pose angles is not really an effect in the strict sense. It sets a value which normally would be required before execution of the action. The robot needs to know the grasp pose before being able actually to grasp the object. One could think of the *grab* action as a command not only to grasp the object but also to determine the proper grasp pose. Thus one could understand the setting of the pose intervals as a proper effect of the action and solve the semantic problem (which is not really a problem but merely a minor imprecision without influence on the solution).

5.4.2 Module Implementation

Although we use identical module calls with both actions, we need the modules to perform different tasks depending on which action was used to call them. In case of the *grab* action, we can assume that the gripper is currently free, i.e. no currently usable poses are set and the condition checker module has to find only one pose usable in the current situation. When a pose has been found, the action can be executed (provided that all symbolic conditions hold) and the effect module is called to determine all poses usable in this situation and to update the three intervals in the planner accordingly.

Now the gripper holds the object and a range of usable poses has been saved in the current planning state. (Remember that we do not fix the grasp pose in the planning process, but always maintain an interval of possible values.) The next action to be executed is the *putdown* action. Again, a condition checker module has to search for a usable pose to put down the object. This time, however, the module may only choose from the restricted set of intervals defined by our six functions. The reason is simple: The gripper cannot change pose while holding the object, so the only poses that can be used to put it down are those which were usable when grasping it. If the module condition holds, the action may be executed.

The module effect in the *putdown* action does not influence the planning process but is important for the resulting plan, as it restricts the set of currently usable poses to those usable for putting down the object.

To illustrate this, let P be the set of all free poses, and let $grasp(o, b, P) \subseteq P$ be the set of poses usable to grasp a movable object o, lifting it from a base object b. We also define $putdown(o, b, P) \subseteq P$ as the set of poses usable to put down a movable object o onto a base object b. Thus, the poses usable in a plan grasping object o from base object b_1 and putting it onto base object b_2 can be described as $putdown(o, b_2, grasp(o, b_1, P))$ or $grasp(o, b_1, P) \cap putdown(o, b_2, P)$.

This is the set which the external modules need to determine.

In the external module, there are four cases to be handled: condition checking for the *grab* and the *putdown* actions as well as effect execution for both.

Condition Checking

When checking if the gripper can grasp or put down a movable object, we only need to search for one usable pose. We perform this search in the simplest way possible. By running through three nested loops (one for each rotation axis), we generate all possible pose configurations and check in each case if

the gripper collides with the base object. This collision query is done in the same way as in section 5.3, except that the gripper is positioned according to the definition above. The term "all possible pose configurations" is not quite exact since, theoretically, there is an infinite number of configurations. Thus, we introduce a *step* size defining the increment which is used to traverse the pose configuration loops. (The step value can be configured through the module options. See appendix C.2.2.)

In case of the *grab* action, the loop boundaries are only physical, i.e. all three loops run from 0 to 360 degrees (0 to 2π).

In case of the *putdown*, action the pose intervals saved in the current planning state are used as boundaries.

Once a usable pose has been found (i.e. a collision query has returned no collision), the loop is stopped and the condition is considered met.

Effect execution

As stated above, our effect module has to determine all poses usable with a combination of base object and movable object. This is slightly more complicated than simply searching for one of them, as in the case of condition checking. Due to the limitation of the "three interval" approach, we cannot simply determine all possible combinations of rotation values but have to search for the maximal intervals $[x_{min}, x_{max}]$, $[y_{min}, y_{max}]$, and $[z_{min}, z_{max}]$ with $\forall p \in \{(x, y, z) \in \mathbb{R}^3 \mid x \in [x_{min}, x_{max}] \wedge y \in [y_{min}, y_{max}] \wedge z \in [z_{min}, z_{max}]\}$: base object and gripper do not collide under pose rotation p.

Algorithm 5.1 calculates these intervals. The main idea behind this is to determine intersections as follows. Imagine fixed x and y rotation values. Now let z_i be the smallest rotation value resulting in a non-colliding pose and z_j be the smallest value with $j > i$ resulting in a colliding pose. We assign the interval $Z_{x,y} = [z_i, z_{j-1}]$ to the fixed combination of x and y.

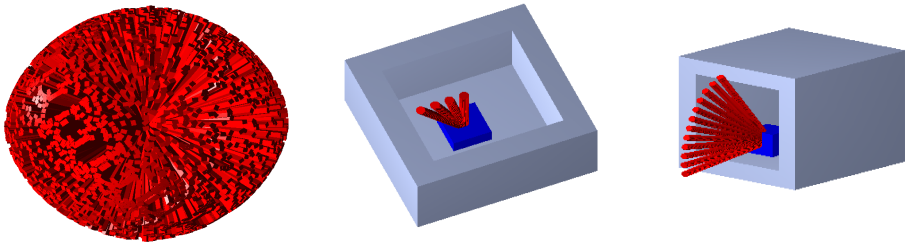
For each x rotation value, we get $(y_{max} - y_{min})/\,\text{STEP}$ intervals $Z_{x,y}$ with valid z rotation values, some of which may be empty (which would mean that this combination of x and y values is not usable at all). If, for a fixed x rotation value, y_i is the smallest y rotation value with $Z_{x,y_i} \neq \emptyset$, and y_j is the smallest y rotation value with $j > i$ and $Z_{x,y_j} = \emptyset$, we define $Y_x = [y_i, y_{j-1}]$.

Equally, we define $X = [x_i, x_{j-1}]$ where x_i is the smallest x rotation value with $Y_{x_i} \neq \emptyset$, and x_j is the smallest value with $j > i$ and $Y_{x_j} = \emptyset$.

Now, the resulting intervals are

$$[x_{min}, x_{max}] = X$$

$$[y_{min}, y_{max}] = \bigcap_{x \in X} Y_x$$



(a) **(b)** **(c)**

Figure 5.4: All possible poses discretized with a step value of 0.5 (a), and poses determined as free by algorithm 5.1 for the box (b) and the shelf (c)

$$[z_{min}, z_{max}] = \bigcap_{x \in X} \bigcap_{y \in Y_x} Z_{x,y}$$

To determine these three pose rotation intervals, algorithm 5.1 updates their initial values using three nested loops. It maintains three boolean variables inX, inY, and inZ stating if the algorithm is currently working on the boundaries of X, Y_x, and $Z_{x,y}$ respectively. It computes the intersections implicitly by changing the interval boundaries each time a new rotation value has been tested. Thus, the lower boundary is set once a non-colliding pose has been found while the upper boundary is set a colliding pose has been found. The in variables are used to memorize if these updates have to be performed or not. For example, if inY is true, the lower boundry of the y-interval has already been set and the algorithm is currently seaching for the upper boundry.

As with the condition checking, the algorithm starts with different min and max values depending on the action.

The pose value intervals generated by the algorithm translate into gripper positions as illustrated in figure 5.4 (b) and (c). It is obvious that the calculated pose intervals do not cover all possible intervals. This is caused by the limitations of the pose encoding which we already stated.

Algorithm 5.1 Determine the usable pose rotations from the initial intervals $[x_{min}, x_{max}]$, $[y_{min}, y_{max}]$, and $[z_{min}, z_{max}]$.

```
inX := false
inY := false
inZ := false
x := x_min
y := y_min
z := z_min
while x ≤ x_max do
  haveIntervalInX := false
  inY := false
  y := y_min
  while y ≤ y_max do
    haveIntervalInY := false
    inZ := false
    z := z_min
    while z ≤ z_max do
      if collision( x, y, z ) then
        if inZ then
          z_max := z - STEP
        end if
      else
        haveIntervalInX := true
        haveIntervalInY := true
        if ! inX then
          inX := true
          x_min := x
        end if
        if ! inY then
          inY := true
          y_min := y
        end if
        if ! inZ then
          inZ := true
          z_min := z
        end if
      end if
      z += STEP
    end while
    if ! haveIntervalInY then
      if inY then
        y_max := y - STEP
      end if
    end if
    y += STEP
  end while
  if ! haveIntervalInX then
    if inX then
      x_max := x - STEP
    end if
  end if
  x += STEP
end while
```

Chapter 6

Results

We will now present some experimental results from the geometric domains described in chapter 5. We will use the exemplary configuration outlined in chapter 2 as a basis.

Initially, the cube lies inside the box. The goal is to put it onto the shelf. To provide a means of comparison, we test this domain setup with different numbers of cubes, boxes, and tables. Each of the cubes is sitting in one of the boxes and has to be moved onto the shelf using one of the tables as a temporary position to change the grasp pose.

The problem instances with multiple cubes may seem simple since all movable objects have the same shape and need to be manipulated in an equal manner. Apparently, we only need to solve the problem once and "copy" the solution as many times as necessary. For the planning system, however, this is irrelevant since it does not know anything about the shapes of the objects and does not see a difference in calling the external module for an object called *cube* or for one called *bottle*. So it cannot reuse solutions to subtasks for other equal subtasks (or those appearing equal to a human planner).

As said in section 5.3, with the discretized pose encoding, we only use the two poses *horizontal* and *vertical*. This restriction does not have any influence on the performance of the external module since it will always perform a collision check using one specific pose. A greater number of discretized poses, however, would increase the size of the instanciated domain in FF/M, resulting in a larger state space, and thus in a loss of performance. Since it is free pose encoding which is of interest here, we will not investigate the exact effect of a greater number of discretized poses.

Figure 6.1 shows a plan for a discretized pose planning problem with four cubes, two boxes, and four tables, as generated by FF/M.

```
 0: GRAB CUBE1 BOX1 VER
 1: PUTDOWN CUBE1 TABLE4 VER
 2: GRAB CUBE1 TABLE4 HOR
 3: PUTDOWN CUBE1 SHELF HOR
 4: GRAB CUBE2 BOX2 VER
 5: PUTDOWN CUBE2 TABLE4 VER
 6: GRAB CUBE2 TABLE4 HOR
 7: PUTDOWN CUBE2 SHELF HOR
 8: GRAB CUBE3 BOX3 VER
 9: PUTDOWN CUBE3 TABLE4 VER
10: GRAB CUBE3 TABLE4 HOR
11: PUTDOWN CUBE3 SHELF HOR
12: GRAB CUBE4 BOX4 VER
13: PUTDOWN CUBE4 TABLE4 VER
14: GRAB CUBE4 TABLE4 HOR
15: PUTDOWN CUBE4 SHELF HOR
```

Figure 6.1: FF/M planning result (discretized poses, four cubes, two boxes, and four tables)

Figure 6.2 shows a visualization of the core part of the solution plan: One cube has to be moved from the box to the shelf.

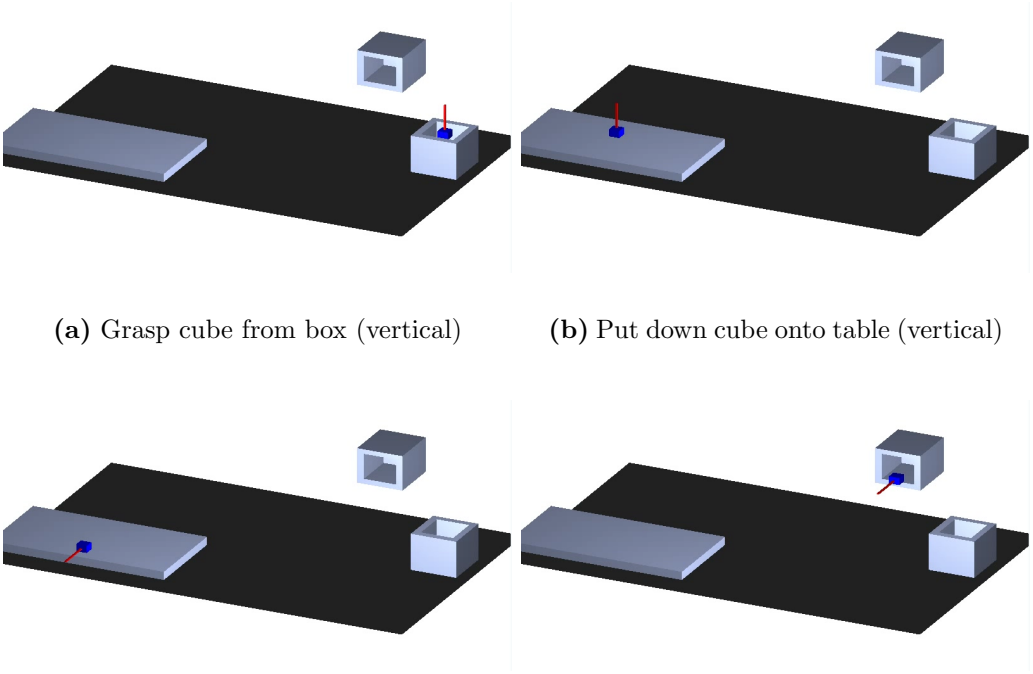
Figure 6.3 shows the plan for the simple configuration with one cube and free poses, as generated by FF/M with a STEP value of 0.1. For each action, the resulting pose intervals are given which could be used in the corresponding planning situation. It is easy to see that the poses after the *putdown* actions are subsets of the poses after the corresponding *grab* actions, as defined in section 5.4.2. The intervals in the plan steps 0 and 3 generated by FF/M correspond to the grasp poses shown in figure 5.4 (b) and (c) respectively.

We will now take a look at the overall performance of FF/M when it solves instances of our geometric manipulation planning domain. Table 6.1 shows the results of the different runs of FF/M in our example domain. The tests were performed on a standard desktop computer[1] with different STEP values for the free pose encoding. The values are based on the internal timing mechanism of FF and give the amount of time (in seconds) it took FF to solve the entire problem instance.

[1]Used computer testing system: AMD Athlon(tm) XP 2000+ CPU (1667 MHz), 512 MB of RAM

			Free Poses			Discretized
Cubes	Boxes	Tables	Step 0.5	Step 0.2	Step 0.1	
1	1	1	0.01	0.04	0.2	< 0.01
4	1	1	0.1	0.3	1.2	0.1
6	1	1	0.1	0.5	2.6	0.1
8	1	1	0.3	1.0	5.0	0.5
10	1	1	0.45	1.78	8.46	2.50
12	1	1	0.70	2.80	12.80	12.90
1	4	4	0.03	0.11	0.58	0.01
1	6	6	0.04	0.15	0.82	0.02
1	8	8	0.05	0.21	1.06	0.02
1	10	10	0.06	0.25	1.31	0.02
1	50	50	0.30	1.17	6.27	0.14
4	4	1	0.3	1.2	5.3	0.3
6	4	1	1.19	4.09	17.04	12.67
6	6	1	2.68	9.37	36.55	98.91
8	8	1	14.22	42.40	158.60	-
12	12	1	163.36	422.92	1444.90	-
2	1	2	0.03	0.14	0.81	0.01
4	1	4	0.33	1.61	9.08	0.23
6	1	6	2.27	9.44	49.21	67.72
7	1	7	121.23	502.75	2454.45	-
2	10	10	0.72	3.84	22.95	0.12
4	10	10	10.94	38.13	171.01	67.06
5	10	10	204.81	752.35	-	-
2	2	2	0.04	0.18	1.03	0.02
3	3	3	0.63	3.13	17.42	0.08
4	2	4	0.49	2.23	12.05	0.45
4	4	4	1.0	3.9	19.5	1.4
4	4	6	1.80	7.34	37.15	3.42
4	8	1	1.4	4.5	17.9	2.6

Table 6.1: Stocking cubes onto a shelf using free poses with different STEP values as well as discretized poses. All values give the amount of time (in seconds) which it took FF/M to solve the problem. A '-' indicates that the planning process was stopped after an hour, before completion. (Note that the memory was exhausted in these cases and the system had to swap to the harddisk a lot.)



(a) Grasp cube from box (vertical)

(b) Put down cube onto table (vertical)

(c) Grasp cube from table (horizontal)

(d) Put down cube onto shelf (horizontal)

Figure 6.2: Visualization of a plan with discretized poses *vertical* and *horizontal* for a single cube

Generally, planning problems using discretized poses are solved much faster[2]. This is no surprise, as the implementation of the external module is much simpler in this case and, thus queries are answered much faster. The module used for the free pose domain performs multiple collision queries in each call to determine the set of usable poses.

The results in the second section of table 6.1 illustratea that the number of base objects is virtually irrelevant if the problem only contains a single object to be manipulated. Even with 100 base objects, FF/M will find a solution in just a few seconds. The reason for this is the small solution plan. As described in section 4.1, FF performs an enforced hill-climbing search in

[2]Some of the results pointing to a quicker solution of the free pose problem are misleading. Due to the larger number of action templates generated by FF/M for the discretized pose encoding, the system memory was exhausted quickly, which resulted in massive hard-disk swapping.

```
0: GRAB CUBE BOX
   POSE_ROT_X_A(0.000000)
   POSE_ROT_X_B(0.400000)
   POSE_ROT_Y_A(1.400000)
   POSE_ROT_Y_B(3.200000)
   POSE_ROT_Z_A(0.000000)
   POSE_ROT_Z_B(0.000000)

1: PUTDOWN CUBE TABLE
   POSE_ROT_X_A(0.000000)
   POSE_ROT_X_B(0.400000)
   POSE_ROT_Y_A(1.400000)
   POSE_ROT_Y_B(3.200000)
   POSE_ROT_Z_A(0.000000)
   POSE_ROT_Z_B(0.000000)

2: GRAB CUBE TABLE
   POSE_ROT_X_A(0.000000)
   POSE_ROT_X_B(1.900000)
   POSE_ROT_Y_A(1.200000)
   POSE_ROT_Y_B(3.400000)
   POSE_ROT_Z_A(0.000000)
   POSE_ROT_Z_B(0.000000)

3: PUTDOWN CUBE SHELF
   POSE_ROT_X_A(0.600000)
   POSE_ROT_X_B(1.900000)
   POSE_ROT_Y_A(1.200000)
   POSE_ROT_Y_B(3.400000)
   POSE_ROT_Z_A(0.000000)
   POSE_ROT_Z_B(0.000000)
```

Figure 6.3: FF/M planning result (free poses, STEP value 0.1)

the state space. If the plan length is small, then so is the number of states explored.

However, if the number of objects to be manipulated increases, the performance drops radically. Here, it becomes evident that the heuristic used in FF/M to estimate the distance from the current planning state to the goal state has not been extended to external modules yet. It would be interesting to see how big a difference an improvement of the heuristic would make.

The situations in the third section of table 6.1 are comparable to the problem of the box of water bottles mentioned earlier. While FF/M can empty a small box of Volvic water bottles (which always come in boxes of six bottles), emptying a box of beer (normally 24 bottles) presents a real problem.

The last section in the table shows some mixed examples with a small number of objects. These results suggest that an optimized planner will be able to solve mid-sized problem instances in reasonable time.

Chapter 7

Conclusion and Future Work

In this thesis, we have introduced an extension to PDDL allowing the use of procedural attachments in action planning. We have seen how the award-winning FF planning system can be modified to support this superset of PDDL in a straightforward manner. We have presented a new approach to manipulation planning based on these extensions and have shown that it is feasible.

However, we have not optimized algorithms or encodings yet. In section 4.3, we have already pointed out ways to enhance PDDL/M support in FF/M through a non-standardized interface based on the internal domain encoding used in FF. Another possible optimization concering to the interaction between the planning system (FF/M) and the external modules would be to integrate part of the heuristic into the external modules. Each external module could provide a simplified version of its interface which could then be used to get optimistic approximations of the actual answer for use in the heuristic of the planner. In FF/M, this approach could be combined with the extension of relaxed actions presented in section 4.2 to obtain a fast and reliable heuristic which takes external modules into account.

In addition to the optimization of the planner, the domain encoding and external modules used in the geometric domain in chapter 5 still allow many improvements. The results from chapter 6 illustrate the decisive impact of the STEP value on the performance of the planning system for the free poses domain. This experimantally proves the apperent fact that improving the performance of the external modules has a significant impact on the performance of the whole system.

The free pose encoding based on three rotation intervals from section 5.4 is far from optimal since it can only encode a very small subset of the poses actually usable. One possible approach would be to extend PDDL/M further for it to support sets of numerical values or sets of intervals, thus allowing the

encoding of the full set of poses. One could also further exploit the fact that the numerical variables used to encode the rotation intervals are never actually used by the planner but only stored inside the current planning state. If this information was only used and understood by the external modules, it would be possible to embed arbitrary information into the planning state. The external modules support in PDDL/M could be extended with a special kind of data type which could be used by external modules to store information inside the internal structures of the planner. With regard to the pose encodings, one could imagine a data structure describing all usable poses in an arbitrary way. This approach may even be used to store an entire geometrical world model of the current planning situation, which is manipulated by the external modules, inside the planner.

In chapter 5, we checked whether a certain pose is usable only by performing a collision query for the initial position (or the final position in case of the putdown action). It would probably be better to determine a collision-free grasping path using an external path planner. Thus, we would increase the accuracy of the pose planning and provide a solution for the sub-task of the actual grasping at the same time. Equally, additional sub-tasks could by integrated into our approach, so that we could create a kind of meta-planner for the whole robot control.

Since we focused on PDDL/M and FF/M in this work and used the problem of manipulation planning and pose encodings merely as an example to prove the feasibility of our approach, we did not compare it to other works, which are based on manipulation graphs. This comparison is still to be drawn.

Appendix A

PDDL/M - BNF Description

The following BNF description of the new PDDL syntax introduced in this document is based on the one in [Edelkamp and Hoffmann, 2003]. We do not present the complete BNF description here but only the parts that have been changed in comparison to PDDL 2.2.

Domains

Domains are defined exactly as in PDDL 2.2, except that we now allow the definition of external modules.

```
<domain>       ::=  (define (domain <name>)
                        [<require-def>]
                        [<modules-def>]:module
                        [<types-def>]:typing
                        [<constants-def>]
                        [<predicates-def>]
                        [<functions-def>]:fluents
                        <structure-def>*)
<modules-def>  ::= (:modules <module-entry>+)
<module-entry> ::= (<name> <typed list (variable)> module-fnkt* (<module-type> @<filename>)
<module-fnkt>  ::= <name>()
<module-type>  ::= conditionchecker|effect
<filename>     ::= A library name usable to the dynamical loader
```

Actions

Actions are defined exactly as in PDDL 2.2, except that we now allow external module preconditions as well as module effects.

```
<GD>          ::=:module <modulecall>
<c-effect>    ::=:module <modulecall>
<modulecall>  ::= (<modulename> <term>*)
<modulename>  ::= A module name as specified in the
                  domain file enclosed in square brackets.
```

Problems

Problems are defined exactly as in PDDL 2.2, except that now we allow the definition of options for the external modules defined in the domain file.

```
<problem>            ::= (define (problem <name>)
                             (:domain <name>)
                             [<require-def>]
                             [<module-options-def>]:module
                             [<object declaration>]
                             <init>
                             <goal>
                             [<metric-spec>]
                             [<length-spec>])
<module-options-def> ::= (:moduleoptions <options-entry>+)
<options-entry>      ::= (<name> <option-list>)
<option-list>        ::= <option><suppl option>*
<option>             ::= <name>=XXX
<suppl option>       ::= ,<option>
```

Requirements

In addition to the requirements in PDDL 2.2 we introduced the new `:module` requirement.

Requirement	*Description*
`:module`	The domain requires external modules to be loaded.

Appendix B

The complete PDDL/M Module API

We now present the complete PDDL/M Module API as discussed in section 3.3.

B.1 PDDL/M Module Data Structures

The module API defines the necessary data structures and some usable helper methods.

```
typedef struct _Parameter {
  char* type;
  char* name;
  char** values;
} Parameter;

typedef struct _ParameterList {
  Parameter* parameter;
  struct _ParameterList* next;
} ParameterList;

typedef struct _Predicate {
  char* name;
  ParameterList* parameters;
} Predicate;

typedef struct _PredicateList {
  Predicate* predicate;
```

```
  struct _PredicateList* next;
} PredicateList;

typedef struct _Function {
  char* name;
  ParameterList* parameters;
  double value;
} Function;

typedef struct _FunctionList {
  Function* function;
  struct _FunctionList* next;
} FunctionList;


Parameter* new_Parameter();
ParameterList* new_ParameterList();
void free_Parameter( Parameter* );
void free_ParameterList( ParameterList* );

Predicate* new_Predicate();
PredicateList* new_PredicateList();
void free_Predicate( Predicate* );
void free_PredicateList( PredicateList* );

Function* new_Function();
FunctionList* new_FunctionList();
void free_Function( Function* );
void free_FunctionList( FunctionList* );
```

B.2 PDDL/M Module Interface

The interface an external PDDL/M module has to provide is defined as follows. Condition Checker modules implement the *checkcondition* method while Effect modules implement the *applyeffect* method.

```
typedef int modulecallback( PredicateList*, FunctionList* );

void init( int argc, char** argv );
int checkcondition( const char*,
```

```
                    ParameterList*,
                    modulecallback* );
int applyeffect( const char*,
                 ParameterList*,
                 double* values,
                 modulecallback* );
```

Appendix C

The complete geometric PDDL/M domain

C.1 Discretized Poses

C.1.1 The domain description

```
(define (domain geometric-domain-v1)

   (:requirements :strips :module :typing)

   (:types pose movable base)

   (:modules
      (grabhelper ?o - movable ?b - base ?p - pose
                  conditionchecker @libmodule_v2.so)
   )

   (:predicates
      (hand_free)
      (in_hand ?o - object ?p - pose)
      (ontopof ?client - movable ?server - base)
      (supports_pose ?o - movable ?p - pose)
   )

   (:action grab
      :parameters (
         ?client - movable
         ?server - base
         ?p - pose
      )
      :precondition (and
         (ontopof ?client ?server)
         (hand_free)
         (supports_pose ?client ?p)
         ([grabhelper] ?client ?server ?p)
      )
      :effect (and
         (not (ontopof ?client ?server))
         (not (hand_free))
```

```
         (in_hand ?client ?p)
      )
   )

   (:action putdown
      :parameters (
         ?client - object
         ?server - base
         ?p - pose
      )
      :precondition (and
         (in_hand ?client ?p)
         ([grabhelper] ?client ?server ?p)
      )
      :effect (and
         (hand_free)
         (ontopof ?client ?server)
         (not (in_hand ?client ?p))
      )
   )
)
```

C.1.2 An example problem

```
(define (problem fact1-v1)

   (:domain geometric-domain-v1)

   (:moduleoptions
      (grabhelper itemfile=items1-v2)
   )

   (:objects
      box1 box2 table - base
      cube - movable
      hor ver - pose
   )

   (:init
      (ontopof cube box1)
      (hand_free)
      (supports_pose cube ver)
      (supports_pose cube hor)
   )

   (:goal
      (and
         (hand_free)
         (ontopof cube box2)
      )
   )
)
```

C.2 Free Poses

C.2.1 The domain description

```
(define (domain geometric-domain-v2)

   (:requirements :strips :module :typing :fluents)

   (:types movable base)

   (:modules
      (grabhelper ?o - movable ?b - base
                  conditionchecker @libmodule_v3.so)
      (putty ?o - movable ?b - base
             pose_rot_x_a() pose_rot_x_b()
             pose_rot_y_a() pose_rot_y_b()
             pose_rot_z_a() pose_rot_z_b()
             effect @libmodule_v3.so)
   )

   (:predicates
      (hand_free)
      (in_hand ?o - movable)
      (ontopof ?client - movable ?server - base)
   )

   (:functions
      (pose_rot_x_a)
      (pose_rot_x_b)

      (pose_rot_y_a)
      (pose_rot_y_b)

      (pose_rot_z_a)
      (pose_rot_z_b)
   )

   (:action grab
      :parameters (
         ?client - movable
         ?server - base
      )
      :precondition (and
         (ontopof ?client ?server)
         (hand_free)
         ([grabhelper] ?client ?server)
      )
      :effect (and
         (not (ontopof ?client ?server))
         (not (hand_free))
         (in_hand ?client)
         ([putty] ?client ?server)
      )
   )

   (:action putdown
      :parameters (
         ?client - movable
         ?server - base
      )
      :precondition (and
```

```
         (in_hand ?client)
         ([grabhelper] ?client ?server)
      )
      :effect (and
         (hand_free)
         (ontopof ?client ?server)
         (not (in_hand ?client))
         ([putty] ?client ?server)
      )
   )
)
```

C.2.2 An example problem

```
(define (problem fact1-v2)

   (:domain geometric-domain-v2)

   (:moduleoptions
      (grabhelper itemfile=items1-v2,step=0.1,debug=off)
   )

   (:objects
      box1 box2 table - base
      cube - movable
   )

   (:init
      (ontopof cube box1)
      (hand_free)
   )

   (:goal
      (and
         (hand_free)
         (ontopof cube box2)
      )
   )
)
```

Bibliography

[Alami *et al.*, 1990] Rachid Alami, Thierry Simeon, and Jean-Paul Laumond. A geometrical approach to planning manipulation tasks. the case of discrete placements and grasps. In *The fifth international symposium on Robotics research*, pages 453–463, Cambridge, MA, USA, 1990. MIT Press.

[Alami *et al.*, 1995] R. Alami, J. P. Laumond, and T. Siméon. Two manipulation planning algorithms. In *WAFR: Proceedings of the workshop on Algorithmic foundations of robotics*, pages 109–125, Natick, MA, USA, 1995. A. K. Peters, Ltd.

[Amato and Wu, 1996] N. Amato and Y. Wu. A randomized roadmap method for path and manipulation planning, 1996.

[Barraquand and Latombe, 1991] Jérîme Barraquand and Jean-Claude Latombe. Robot motion planning: a distributed representation approach. *Int. J. Rob. Res.*, 10(6):628–649, 1991.

[Blum and Furst, 1995] Avrim Blum and Merrick Furst. Fast planning through planning graph analysis. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI 95)*, pages 1636–1642, 1995.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artif. Intell.*, 32(3):333–377, 1987.

[Dean and Boddy, 1988] T. Dean and M. Boddy. Reasoning about partially ordered events. *Artif. Intell.*, 36(3):375–399, 1988.

[Edelkamp and Hoffmann, 2003] Stefan Edelkamp and Jörg Hoffmann. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical Report 194, Albert-Ludwigs-Universität, Institut für Informatik, Freiburg, Germany, 2003.

[Fikes and Nilsson, 1971] R. E. Fikes and N. J. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. In *Proc. of the 2nd IJCAI*, pages 608–620, London, UK, 1971.

[Fox and Long, 1998] M. Fox and D. Long. The automatic inference of state invariants in tim. *Journal of AI Research*, 9:367–421, 1998.

[Fox and Long, 2001] M. Fox and D. Long. Hybrid stan: Identifying and managing combinatorial optimisation sub-problems in planning. In B. Nebel, editor, *Proceedings of IJCAI*, pages 445–452. Morgan Kaufmann, 2001.

[Golden, 2003] Keith Golden. A domain description language for data processing, 2003.

[Hoffmann and Nebel, 2001] Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

[Hoffmann, 2002] Jörg Hoffmann. Extending FF to numerical state variables. In F. Van Harmelen, editor, *Proceedings of the 15th European Conference on Artificial Intelligence (ECAI-02)*, pages 571–575, Lyon, France, July 2002. Wiley.

[Jónsson, 1996] A. Jónsson. Procedural reasoning in constraint satisfaction, 1996.

[Kavraki *et al.*, 1994] Lydia Kavraki, Petr Svestka, Jean-Claude Latombe, and Mark Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. Technical Report CS-TR-94-1519, 1994.

[Kavraki, 1999] L. E. Kavraki. Algorithms in robotics: The motion planning perspective. In *Frontiers of Engineering Publication*, pages 90–93. National Academy of Engineering, 1999.

[Koga and Latombe, 1994] Y. Koga and J.-C. Latombe. On multi-arm manipulation planning. pages 945–952, 1994.

[Latombe, 1991] Jean-Claude Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

[Latombe, 1999] J. Latombe. Motion planning: A journey of robots, molecules, digital actors, and other artifacts, 1999.

[McDermott and others, 1998] Drew McDermott et al. *The PDDL Planning Domain Definition Language*. The AIPS-98 Planning Competition Committee, 1998.

[Nielsen and Kavraki, 2000] C. Nielsen and L. Kavraki. A two level fuzzy PRM for manipulation planning. Technical Report TR2000365, Rice University, 2000.

[Pednault, 1989] Edwin P. D. Pednault. Adl: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning*, pages 324–332, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.

[Penberthy and Weld, 1992] J. Scott Penberthy and Daniel S. Weld. UCPOP: A sound, complete, partial order planner for ADL. In Bernhard Nebel, Charles Rich, and William Swartout, editors, *KR'92. Principles of Knowledge Representation and Reasoning: Proceedings of the Third International Conference*, pages 103–114. Morgan Kaufmann, San Mateo, California, 1992.

[Peot and Smith, 1992] M. Peot and D. Smith. Conditional nonlinear planning. In James Hendler, editor, *Proceedings of the First International Conference on AI Planning Systems*, pages 189–197, College Park, Maryland, June 15–17 1992. Morgan Kaufmann.

[Siméon *et al.*, 2004] Thierry Siméon, Jean-Paul Laumond, Juan Cortés, and Anis Sahbani. Manipulation planning with probabilistic roadmaps. In *International Journal of Robotics Research, Vol.23, N7-8*, pages 729–746, 2004.

[Srivastava, 2000] Biplav Srivastava. Realplan: Decoupling causal and resource reasoning in planning. In *AAAI/IAAI*, pages 812–818, 2000.