Design and Implementation of an Object-Oriented Planning Language

Master of Science Thesis at the Albert-Ludwigs-Universität Freiburg

Faculty of Engineering
Research Group Foundations of Artificial Intelligence
Prof. Dr. Bernhard Nebel

Andreas Hertle

2011

Declaration

I hereby declare, that I am the sole author and composer of my Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

Abstract

Semantic attachments improve the applicability of symbolic planning systems to real world problems, by incorporation low-level algorithms directly into the high-level reasoning process. The Temporal Fast Downward (TFD) planning system implements this concept by providing a generic interface for data exchange between its internal state and external modules. However, due to its generic nature, the interface is inefficient and cumbersome to use.

In this Thesis we present the Object-oriented Planning Language. We demonstrate how a domain-specific module interface can be generated from a planning task described in OPL. We show, how the domain-specific interface is integrated into the planning system, without compromising the domain-independence of the TFD. We conduct experiments to show compare the efficiency of our novel approach to the current TFD module interface.

Abstrakt (German)

Semantische Anhänge erhöhen den Nutzen von Symbolischen Planungssystemen beim Einsatz in realistischen Planungsproblemen, indem sie spezialisierte low-level Algorithmen direkt in die symbolische Planung mit einbeziehen. Das Temporal Fast Downward (TFD) Planungssystem implementiert dieses Konzept: es stellt eine Schnittstelle zur Verfügung, über die externe Prgramme mit dem Planner kommunizieren können. Jedoch muss die Schnittstelle allgemeingültig sein und ist daher ineffizient und kompliziert in der Verwendung.

In dieser Arbeit stellen wir die Object-oriented Planning Language (OPL) vor. Wir demonstrieren, wie aus einer Problembeschreibung in OPL eine and das Problem angepasste Schnittstelle automatisch generiert werden kann. Wir zeigen, wie diese Schnittstelle in das TFD Planungssystem integriert werden kann, ohne dessen Allgemeingültigkeit in Mitleidenschaft zu ziehen. Wir führen experimente durch, um die Effizienz unseres neuartigen Ansatzes mit der bestehenden Implementierung zu vergleichen.

Contents

1 Introduction

In recent years, autonomous robots have become capable of executing increasingly complex tasks. As the task complexity grows, it becomes more difficult to foresee every possible situation and encode it into the control software of the robot. Finite state automate, a popular choice to encode the mission of an autonomous robot, are difficult to maintain and to modify, once a certain level of complexity is reached. A more dynamic system is required, that can easily be extended and adopted according to the mission specification. The techniques employed in automated planning systems could offer a solution.

The artificial intelligence community made significant progress in the past decade. International planning competitions have fueled the development of efficient planning algorithms that produce plans of high quality. The efficiency of such planing systems results from a highly abstract representation of the planning task. However, the high level of abstraction hinders the application of AI planning systems to real world planning tasks.

For instance, consider an autonomous robot with the task to fetch an item from a different room. A symbolic planning system can easily determine the order of actions necessary to fulfill the task: move to the other room, find requested item, pick it up and return to starting room. However, it is impossible for the planning system to determine, whether a valid path can be found to the specified room. It just assumes, that the robot is able to find such a path, when the plan is executed. Therefore, the quality of the plans produced by a symbolic planning system is often insufficient for real world applications.

The concept of semantic attachments promises to bridge the gap between high level planning systems and real world applications. Semantic attachments allow a high-level planning system to solve sub-problems using a specialized low-level algorithm during the planning process. In the example discussed above, the planning system can invoke the path planning algorithm of the robot during the planning process to determine whether a valid path exists to the specified room. This increases the quality of generated plans significantly and reduces the need for re-planning.

To demonstrate the benefits of semantic attachments, they were incorporated into the Temporal Fast Downward (TFD) planning system. The TFD is capable of solving planning problems in temporal and numeric domains. In order to preserve domain-independence, the TFD implements a generic, domain-independent interface, which is responsible for the exchange of data between the internal planning state and external applications. However, the generic nature of the interface imposes restrictions on its usability: the exchange of state information between planning system and an external application is inefficient and the interface is difficult to implement to correctly.

In this Thesis we propose a solution that improves the efficiency and usability of the module interface. We introduce the Object-oriented Planning Language (OPL). The main feature of OPL is an automated generation of a domain-specific module interface, based on the definitions in the planning domain. The interface provides external applications

with a type safe and efficient access to the internal state of the planning system. We integrate OPL into the TFD, without compromising the domain-independent nature of the underlying planning system. We provide tools to translate planning task definitions in OPL into the Planning Domain Definition Language (PDDL), the de facto standard language of the AI planning community. Therefore, OPL can be easily integrated into a planning system capable of parsing PDDL; an OPL parser is not necessary. In addition, we believe the object-oriented syntax of OPL lowers the learning curve for application developers who have no prior experience with symbolic planning.

This Thesis is structured as follows: First, we briefly discuss related work in chapter two. Section three gives a basic introduction to planning task descriptions for classical and temporal domains. The syntax of the Domain Definition Language is introduced with a focus on the integration of external algorithms into the task description. In section four we take a look at the Temporal Fast Downward planning system, with focus on the internal representation of the planning task and the integration of external modules into the search process. Section five presents our contribution, the Object-oriented Planning Language. In section six we discuss the generation of the domain specific module interfaces for the TFD planning system. We present experimental results of our proposed module interface and compare them to the current module interface of the TFD. In section eight, we draw conclusion and discuss future work.

2 Related Work

In this section we briefly examine work related to the field of the combination of symbolic planning with external algorithms.

2.1 PDDL

- The Planning Domain Definition Language (PDDL) is considered the standard language of the AI planning community. The first version of PDDL was released in 1998 by [Drew McDermott, 1998], as an attempt to create a standard language for specifying planning tasks. From the beginning PDDL was developed with flexibility and extensibility in mind: individual features can be enabled as required. Planning systems are not required to implement the full PDDL specification, a specialization is possible. Initially, only Boolean predicates were supported. Moved by considerable interest in AI planning solutions for real world problems, PDDL 2.1 was introduced for the third International Planning Competition held in 2002 to represent temporal and numeric planning tasks. For the IPC in 2008 PDDL 3.1 added concise finite-domain state variables ([Malte Helmert, 2009]), which increase the performance of planning systems significantly.

- To move AI planning methods closer to real world planning tasks, semantic attach-

ments were proposed by [Dornhege et al., 2009]. Symbolic planning systems rely on high-level abstractions to increase the efficiency of the planning process. However, the resulting plans often fail in the real world. Using semantic attachments, low-level algorithms can be incorporated into the high-level planning planning process. For instance, external algorithms consider geometrical constraints, which can not be represented in the symbolic planning systems. The work in Thesis aims to improve the method of integration between the symbolic planning system and external algorithms.

2.2 Planning Systems

- The Fast Downward (FD) planning system is a classical planner, based on heuristic forward search. Initially, it was developed according to the specification of PDDL 2.2 by [Malte Helmert, 2006]. The FD combines hierarchical problem decomposition with a causal graph heuristic. Starting a the top of the causal graph a problem is decomposed recursively, until the resulting sub-problems are basic graph search tasks. The FD proved its approach to be successful by winning the propositional, non-optimization track of IPC 2004.

- The Temporal Fast Downward planning system is the adaptation of the classical FD planning system to temporal domains by [Eyerich et al., 2009]. The TFD is capable of performing a search in temporal search space using the context-enhanced additive heuristic. According to the rating scheme of the IPC 2008, the TFD out performs all other temporal planning systems. In 2009 semantic attachments were successfully integrated into the TFD. The work in this Thesis aims to improve the current implementation of semantic attachments, by improving the efficiency of information exchange between the internal planning state of the TFD and external applications.

2.3 Applications

- [Wurm et al., 2010] present a solution to multi-robot exploration tasks with teams of marsupial robots. They combine a classical path planner with a temporal symbolic planner using semantic attachments. The work in this Thesis reduces the effort required for the integration of external low-level algorithms with a symbolic planner.

- [Kleiner and Dornhege, 2009] solve mobile manipulation tasks, by decomposing the manipulation problem into a geometric and a symbolic part. For the symbolic part the TFD planning system is used, while a probabilistic road-map planner is employed to solve the geometrical part. Both parts are tightly integrated using semantic attachments. The work in this Thesis reduces the effort required for the integration of geometric planner with symbolic planner, by providing tools to generate an interface

for efficient data exchange.

3 Domain-Independent Planning

In this chapter a short overview is given of how planning problems can be represented in symbolic logic. Section 3.1 defines the basic concept of Symbolic Planing. Section 3.2 shows how these concepts are realized in the Planning Domain Definition Language (PDDL) and defines the terminology used throughout this Thesis. Section 3.3 explains the benefits of including external algorithms into a symbolic planning system and how this is handled in PDDL.

3.1 Planning Tasks

In this section the elements of a planning task are described. An example of a planning task is defined, to help a reader unfamiliar with symbolic planing understand the concepts.

A planning task consist of a domain and a problem definition. The domain definition contains general knowledge relevant to the planning scenario; it contains facts and rules:

- Facts describing properties of objects are called *predicates*. The state of our example task contains two predicates:

  ```
  persons are either outside or inside
  doors are either open or closed
  ```

- Rules describing how facts change are called *action*s. An action consist of a condition and an effect. If the condition is fulfilled by the state, the action can be applied. Once an action is applied to the state, it changes predicates according to its effect. In our example task, we define two actions:

  ```
  a person moves inside through a door
    condition: the door is open and the person is outside
    effect: the person is inside
  a person opens a door
    condition: the door is closed
    effect: the door is open
  ```

The problem definition specifies one particular instance of the planning task; it contains an initial state and a goal condition.

- The *initial state* describes the values of facts before the planning process begins. In our example, we create a person called Anthony and a door named FrontDoor. Anthony is outside and the FrontDoor is closed.

```
Antony is a person
FrontDoor is a door
Antony is outside
FrontDoor is closed
```

- The *goal condition* describes the desired values some facts must have to solve the planning task successfully. In our example we want Anthony to be inside.

```
Antony is inside
```

A valid plan consist of a sequence of actions that transforms the initial state into a state satisfying the goal condition. For this example there is only one valid plan, as shown below, but in general, there are many valid plans for one planning task.

```
Anthony opens FrontDoor
Anthony moves inside through FrontDoor
```

So far, we have no notion of time in our planning tasks. This is called Classical Symbolic Planning. However, planning tasks in the real world often require temporal information to find a good plan. To incorporate temporal information into our plan description we add a duration to each action. The effects of our actions can now be applied at the start of an action or at the end: the temporal qualifier *at start* and *at end* specify when each effect occurs. Conditions can even have a third qualifier: *over all* signifies, that a condition must hold while the action is in progress. The following listing shows the two actions from our planning task with specified durations.

```
a person moves inside through a door
  duration: 10
  condition: at start: the door is open and
     at start: the person is outside
  effect: at end: the person is inside
a person opens a door
  duration: 30
  condition: at start: the door is closed
  effect: at end: the door is open
```

In the temporal domain, a valid plan specifies the start time of each action. The duration of a plan is called *make-span*. It is a measure for the quality of the plan. For instance, the plan shown in the following listing is valid but not optimal:

```
0:  Anthony opens FrontDoor
60: Anthony moves inside through FrontDoor
```

Anthony moves inside after 60 time units, while the open door action finished after 30 time units. In the optimal plan, Anthony would move inside as soon as the door is open, resulting in a make-span of 40 time units.

In temporal domains actions can also be executed in parallel. To demonstrate this we add a second person to our initial state:

```
Charlie is a person
Charlie is outside
```

In the goal condition we want both persons to be inside. Neither Charlie nor Anthony can get inside while the door is still closed. But as soon as Anthony opens the door, both can go inside at the same time. The corresponding plan with a make-span of 40 time units is shown in the following listing:

```
0:  Anthony opens FrontDoor
30: Anthony moves inside through FrontDoor
30: Charlie moves inside through FrontDoor
```

This concludes the introduction to planning tasks in temporal domains.

3.2 Planning Domain Definition Language

The Planning Domain Definition Language (PDDL) was originally designed in 1998 [Drew McDermott, 1998] as a standard language for describing planning tasks. PDDL is based on predicate logic. The original version supported Boolean predicates and user defined types. Since then many additional features were added. Today, PDDL supports Boolean, numeric and object fluents[Malte Helmert, 2009]. Custom object types can be specified, which can extend other types. Temporal information[Maria Fox and Derek Ling, 2003] can be included into actions by specifying the action duration. With the temporal information, it is possible to schedule actions in parallel. The additional features can be enabled individually and it is up to the planning system to implement them correctly. For the remainder of this Thesis we use the term PDDL as synonym for PDDL version 3.1, with numerical and object fluents, durative actions and external modules.

A planning task is defined in two separate files. The domain file describes relevant statements and operators. A problem file describes one specific instance of the planning task

by defining the initial state and the goal condition. Usually, multiple problem files are associated with one domain file. Throughout this section PDDL examples are loosely based on the Transport Domain, which was part of the International Planning Competition (IPC) in 2008.

The domain definition begins by specifying the name of the domain. The domain file specifies which PDDL extensions are required for this planning task, as shown in the following PDDL listing:

```
(define (domain transport)
  (:requirements :typing :durative-actions
    :numeric-fluents :modules)
```

Custom types are defined in the *type* section. A type can extend an another type. If no super type is specified, the built-in type *object* is extended. The following PDDL example shows various type definitions: *location*, *target* and *locatable* extend the built-in *object* type, while *vehicle* and *package* extend the *locatable* type.

```
  (:types location target locatable - object
        vehicle package - locatable)
```

Predicates express Boolean facts about certain objects or relations between objects; for instance whether a certain vehicle is green, or whether there is a road from location A to location B. Predicates can have multiple parameters. Each parameter has a specific type.

```
  (:predicates
    (road ?l1 ?l2 - location)
    (at ?x - locatable ?y - location)
    (green ?v - vehicle)
  )
```

Functions express numerical facts or relations; for instance the size of a package or the distance between two locations. Functions without a type specification are numeric or the type *number* can be used to define a numeric function explicitly.

```
  (:functions
    (road_length ?l1 ?l2 - location)
    (package_size ?p - package) - number
  )
```

Functions with a specified type represent object facts. Object facts allow to encode the relation between objects in a more restrictive manner. They can be thought of as finite, multivalued attributes. For instance the objects *green* and *red* of the type *color* are defined and an object fact *color_ of* of the same type *color*, as shown in the following PDDL listing:

```
(:functions
  (color_of ?v - vehicle) - color
)
```

or *red* as its color fact but not both at the same time. If two predicates, *green* and *red* were defined instead, both of them could be true at the same time for one object.

In this Thesis, predicates, numerical functions and object functions are referred to as Boolean, numerical and object fluents.

Actions define how fluent values can change. To incorporate temporal information, actions are defined with the keyword *durative-action*. A durative action contains four lists, which define the parameters of the action, the action's duration, a condition that needs to be met before the action can be executed and an effect that specifies the fluents to be changed. The following PDDL listing shows the action *drive*:

```
(:durative-action drive
  :parameters (?v - vehicle ?l1 ?l2 - location)
  :duration (= ?duration (road_length ?l1 ?l2))
  :condition (and
    (at start (at ?v ?l1))
    (at start (road ?l1 ?l2)))
  :effect (and
    (at start (not (at ?v ?l1)))
    (at end (at ?v ?l2)))
)
```

The *drive* action has three parameter, one of the type *vehicle* and two of the type *location*. The duration of the action is determined by the length of the road between the two locations. The condition term contains two terms encapsulated by the Boolean function *and*, so both of them must hold the action to be applied. The temporal function *at start* signifies, that the contained predicate must hold at the start of the action. Analogous, the temporal function *at end* signifies the and of the action, as can be seen in the effect statement. The drive action can only be applied, when the *vehicle v* is at the location *l1* and there is a road between the *locations l1* and *l2*. As soon as the action starts, the *vehicle v* leaves the *location l1*. When the action finishes, the *vehicle v* arrives at the *location l2*. In this Thesis the word action is a synonym for durative-action; all actions are durative in temporal planning.

That covers all elements of a domain file.

As mentioned above, a problem file describes one specific instance of the planning task. It instantiates objects, defines the initial state and sets a goal condition. The problem definition begins with the name of the problem and the name of the corresponding domain:

```
(define (problem p01_5nodes_2trucks_2packages)
  (:domain transport)
```

Next, objects are instantiated. They can have any *type* specified in the domain file. In the example below five *location*s, two *vehicle*s and two *package*s are instantiated:

```
  (:objects
    loc1 loc2 loc3 loc4 loc5 - location
    truck1 truck2 - vehicle
    package1 package2 - package
  )
```

The *init* statement assigns values to fluents. All fluents should be initialized, as not doing so, leaves the initial value to be decided by the planning system implementation. Therefore, implicit fluents initialization may vary across different implementations. The following example only show exemplary the initialization of the two Boolean fluents *at* and *road* and the numerical fluent *road_ length*.

```
  (:init
    (at package1 loc3)
    (road loc1 loc3)
    (= (road_length loc1 loc3) 23)
  )
```

The goal condition uses a similar syntax as the condition of an action. However, temporal functions are not permitted in the goal condition. The example below shows, that *package1* is supposed to be delivered to *loc2* and *package2* to *loc3*, for this problem instance to be considered solved.

```
  (:goal (and
    (at package1 loc2)
    (at package2 loc3))
  )
```

In this section a brief overview was given over the structure and syntax of PDDL. With this section as reference, even readers with no prior experience with PDDL should have knowledge to understand PDDL listings throughout the Thesis.

3.3 Semantic Attachments in PDDL

The efficiency of symbolic planning results for high level abstractions of the planning problem. However, real world planning tasks often require explicit low level knowledge, to be solved successfully. Geometric relations and algorithms can not be expressed with predicate logic. For such cases, a PDDL extension was developed by [Dornhege et al., 2009] to integrate external algorithms into the domain-independent planing process via modules.

A module is defined with a name, a list of parameters, the module type and the names of a function and a library, as shown below for the *can_load* module:

```
(can_load ?v - vehicle ?p - package
  conditionchecker can_load@libTransport.so)
```

The *can_load* module has two parameters, a *vehicle v* and a *package p*. The module type is *conditionchecker*, which signifies, that the module is called in a condition statement of an action. Analogous, the module type *cost* signifies the duration statement and the the module type *effect* signifies the effect statement. The library string *can_load@libTransport.so* specifies the function *can_load* from the library *libTransport.so*. It is important to note, that while PDDL in general is case insensitive, the function and library names are an exception: here the case matters.

To distinguish module calls from fluents in PDDL domain files, a module call is encapsulated by square brackets, as shown in the load action below (other conditions and effects are omitted in this example):

```
(:durative-action pick_up
  :parameters (?v - vehicle ?l - location ?p - package)
  :duration (= ?duration 1)
  :condition (and
     (at start ([can_load ?v ?p]) ... )
  )
  :effect (and ...
  )
)
```

4 Temporal Fast Downward

In this section we take a look at a domain-independent planning system for temporal problems. The Temporal Fast Downward (TFD) planning system was developed by [Eyerich et al., 2009]. The TFD is able to generate low make-span plans by performing

a heuristic search in temporal search space. It supports PDDL version 3.1 with numeric, object fluents and modules. Based on the benchmark of the International Planning Competition (IPC) in 2008, the TFD was able to produce better plans than other participants. Its success is attributed to the context-enhanced additive heuristic, adapted to temporal and numeric planning. TFD is based on the classical Fast Downward planning system developed by [Malte Helmert, 2006].

The TFD planning system consist of three separate programs. The Translate program parses the PDDL domain and problem files and defines the variable mappings of the internal state used in the search. The Preprocess program initializes the heuristic for the search. The actual planning takes place in the search program, which performs a best first search.

The focus of this Thesis lies on the interaction between the planning system and external applications via the module interface. Therefore, we take a closer look at the Translate program in section 4.1 and the Search program in section 4.2. In section 4.3, we examine the module interface of the TFD.

4.1 Translate

The Translate program parses a planning task description and translates it from PDDL into the SAS^+ form as described by [Christer Bäckström and Bernhard Nebel, 1995]. It requires the path to a domain file and a problem file as arguments. The supplied domain and problem files are parsed and checked for inconsistencies. Then begins the actual work of the translate program: allocating variables for fluents, actions and modules.

First, the translate program grounds all fluents and modules by substituting their parameters with all available objects of the corresponding types. Each substitution is mapped to a different variable. For instance, the domain contains a Boolean fluent *is_reachable* with two parameters of the type *location*. Furthermore the problem file specifies two *location* objects: *loc1* and *loc2*. In that case the translate program produces the following four grounded terms and assigns each term to a separate variable:

```
variable1: (is_reachable loc1 loc1)
variable2: (is_reachable loc2 loc1)
variable3: (is_reachable loc1 loc2)
variable4: (is_reachable loc2 loc2)
```

Each variable is provided with a translation table that encodes its content as a real value. For Boolean fluents the value *0* represents *true* and all other values *false*. Numeric fluents do not need a table, their numeric value is used directly. Object fluents enumerate all objects of the specified type: *0* for the first object, *1* for the second, *n-1* for the *n*th object and all other values represent an invalid object.

The actions are grounded by the same schema. Additionally, the conditions and effects of the actions are converted to the variable mapping. For instance, an action *drive* has in its condition term the predicate *is_ reachable*, so that the action is only applicable, if *is_ reachable* has the value *true* for the *start* and *dest* location, as shown in the following PDDL listing:

```
(:durative-action drive
  (:parameters ?start - location ?dest - location)
  [...]
  (:condition
    (and (at start (is_reachable ?start ?dest)) [...])
  )
  [...]
)
```

of the action with *loc2* as the value of *start* and *loc1* as the value of *dest*.

In the next step, the Translate program iterates through the effects of all grounded actions. Any fluents that are not found in any effect are constants and their values can not change during the planning process. Then, the conditions of all actions are examined whether they contain constants with the value *false*. These actions can not be applied during the search, thus they are eliminated. Should any actions have constants with the value *true* in its condition across all grounded instances, then the constant is removed from the condition, since that does not change the applicability of that action. All these steps are executed to reduce the search space of the planning instance and therefore improving the search performance.

When the Translate program is finished with the search space reduction, the results mapping is written into a file. The output file contains the variable mappings for all grounded terms, the initial state and the goal condition. The output file also contains the function and library name for each module. Additionally, the names of the objects, fluents and actions are written into that file, although the preprocess and the search program work solely with the variables and ignore the names. The names are used to convert the plan found by the search program into a human readable format. The names are also necessary during module calls: when a module requests the value of a fluent, the name of the fluent and their parameters are used to look up the value of the corresponding variable.

4.2 Search

The actual planning process takes place in the Search program. It expects numerous arguments, which adjust timeouts and heuristic strategies.

During the initialization phase, the Search program reads the output files of the Translate and Preprocess programs. As specified in these files, it allocates a vector of *double* variables. This vector represents the internal state throughout the planning process. The values are initialized according to the initial state section in the Translate output file.

Next, operators are instantiated, one operator per PDDL action. The condition of an operator is encoded by pairing the index of a vairable with a value. The translation table produced by the Translate program specifies the required values for a set of variables must have, before the action can be applied. Effects of operators work analogous: the translation table specifies the new values for variables, when the operator is applied to the state vector.

An operator can also contain a module call to let an external program determine the value of a condition, an effect or the duration variable. Instead of accessing a value in the internal state vector, the search program calls the function specified in the Translate output file.

After the initialization process is complete, the search begins. The search program implements the best first search algorithm. The details of the search algorithm and the heuristic functions are beyond the scope of this Thesis.

When a valid plan is found, in most instances the plan is far from optimal. Therefore the search program can be started in anytime mode: even when a valid plan was found the search continues for a predefined amount of time or until the search space is completely explored.

4.3 TFD Module Interface

In this section we take a look at the handling of module calls in the TFD Search program.

Module calls are defined in PDDL domain files. A module definition consist of a name, an arbitrary number of arguments and the specification of a function and library name. The domain file is read by the Translate program and the module specification is passed along unchanged through the Preprocess program. The Search program then obtains a function pointer to the specified library function.

Since the TFD is domain-independent, the module interface is generic: the types and names of the arguments are passed to the module as strings. It is up to the module developer to parse these strings to retrieve the desired information. In addition to the module parameters a module often requires data from the internal state. Callback function pointer are provided to obtain values of fluents from the search engine. Again, these functions are generic: to obtain values from state, a module developer must pass the correct fluent names as strings to the callback function and parse the returned string lists for the desired fluent names.

To better understand the TFD module interface, we go through the process of implementing a module step by step. For the remainder of this section, we imagine a robotic manipulation scenario. We want to create a module that decides, whether the manipulator can reach a certain pose in three dimensional space.

First, we take a look at the PDDL domain file. We need to handle poses in three dimensions, which we represent by a position vector (*x, y, z*) and an orientation quaternion (*qx, qy, qz, qw*). We want our module to have two parameters: the current pose of the manipulator and the destination pose. In the domain file, we define the type pose3d, numerical fluents for the coordinates and the module call, as shown in the following PDDL listing:

```
(:types pose3d - object)
(:modules
  (can_reach ?current - pose3d ?dest - pose3d
    conditionchecker can_reach_module@lib_manipulation.so)
)
(:functions
  (x ?pose - pose3d) - number
  (y ?pose - pose3d) - number
  (z ?pose - pose3d) - number
  (qx ?pose - pose3d) - number
  (qy ?pose - pose3d) - number
  (qz ?pose - pose3d) - number
  (qw ?pose - pose3d) - number
)
```

With the domain specification, we can begin the module implementation. We declare a function named *can_reach_module* according to the specification of the TFD module interface. The following C++ code listing shows the function signature:

```
double can_reach_module(const ParameterList& parameterList,
   predicateCallbackType predicateCallback,
   numericalFluentCallbackType numericalFluentCallback,
   objectFluentCallbackType objectFluentCallback,
   int relaxed);
```

is a list of module parameters, as specified in the PDDL domain file. The type, name and value of each parameter is represented by a string. The second, third and fourth argument are function pointers, which allow to look-up fluent values in the internal planner state. The last argument can be used to indicate, whether a precise solution is required or an approximate solution is sufficient. This is useful, when the heuristic value for an action is computed. In that case, an approximate result is sufficient. A module can implement a simplified approximation function to preserve computational resources.

Before we can perform the 3d trajectory calculations, we need to obtain the numerical values for the two poses via the *numericalFluentCallback* function pointer. The *parameterList* argument contains the list of parameters for the module. From the domain file we know that the parameter list for the *can_ reach_ module* will have the two entries *current* and *dest*, both of type *pose3d*. With the value of the *dest* parameter, we fill the *ParameterList* of the fluent are interested in. In the example listing below we can see the code to request the value of the x coordinate of the *dest* pose. We repeat these steps for every coordinate of each pose, before using the callback function. The callback function will populate our *NumericalFluentList* with the requested values.

```
string dest_value = parameterList.at(1).value;
NumericalFluentList* nfl = new NumericalFluentList();
ParameterList pl;
pl.push_back(Parameter("pose", "pose3d", dest_value));
nfl->push_back(NumericalFluent("x", pl));
[...]
numericalFluentCallback(nfl);
```

After obtaining the numerical values, the algorithms to perform the calculations can be executed. After the calculation finishes, we return the value 0, in case the pose can be reached successfully. Otherwise, we return any value greater than 0 to signify failure.

5 Object-Oriented Planning Language

In this section we present a new language for describing planning tasks: the Object-oriented Planning Language (OPL). OPL is developed around three core concepts:

1. Simplifying module development: Improving the integration of external algorithms is the main motivation behind the development of OPL. Therefore, OPL provides tools to generate a C++ module interface based on the definitions in the OPL domain file.

2. Compatibility to PDDL: a planning task description in OPL can be converted to PDDL 3.1 syntax. Therefore, planning systems based on PDDL can be adopted to OPL, without implementing an OPL parser.

3. Object-oriented syntax: the object-oriented syntax of OPL lowers the learning curve for application developers that have no prior experience with symbolic planning. Also, the object-oriented representation helps to generate an efficient and clean C++ module interface.

The following sections illustrate each concept in detail. Section 5.1 introduces the object-oriented features of OPL. Section 5.2 explains how compatibility to PDDL is achieved.

Section 5.3 shows the advantages of the OPL module interface. The syntax and grammar of OPL is defined in section 5.4.

5.1 Object Orientated Planning Task Description

The type definitions in PDDL are used to restrict the grounding of fluents: instead of grounding a fluent against all objects in the domain, it is only grounded against all objects of a specific type.

OPL places far greater importance on the specification of custom types than PDDL. OPL types can be regarded analogous to C++ classes. An OPL type has its own naming scope. Fluents and actions can be defined in the scope of a type. These fluents and actions within types are called member fluents and member actions. Of course, the declaration of global fluents and actions is still possible: global fluents and actions are defined in the naming scope of the domain.

Similar to PDDL, types can inherit from other types. OPL defines the following rules for inheritance: a type can have only a single base type, multiple inheritance is not allowed. If no base type is specified, it defaults to *Object*, a built-in abstract type. Since each type has its own naming scope, member fluents of different types can share the same name. OPL defines the following naming scope tree:

- The root scope is the domain scope. The names of any types, global fluents and global actions belong to the domain scope.

- Each type has its own scope. The names of their respective member fluents and actions belong to that scope. These scopes are contained in the domain scope.

- Each action has its own scope. The names of their arguments belong to that scope. These scopes are either contained in the domain scope (in case the are global actions) or in a type scope (in case they are member actions).

- The task has its own scope. The names of instantiated objects belong to this scope. It is contained in the domain scope.

A name must be unique within a scope, e.g. a type must not define a fluent and an action with the same name. When referencing a fluent name, the look-up process begins in the scope where the reference is located. If the fluent is not found within that scope, the search continues in its super scope, until the domain scope is reached.

The scope of a type is a special case. Before extending the name look-up to the domain scope, the scope of the super type is searched. This allows sub types to access the inherited fluents of their super type. Therefore, when inheriting from a base type, a type must not define fluents or actions with a name already defined in its base type.

The dot . acts as the scope operator in OPL. Using the scope operator a specific scope can be addressed: a member fluent of a certain type can be referenced by obtaining a reference to an object fluent of that type and then applying the scope operator with the name of the fluent. It is possible to chain scope operations, if the referenced member fluent happens to be an object fluent. The key word *this* can be used to address the scope of the current object. The global scope can be addressed by prefixing the fluent name with the scope operator.

For instance, we have the type *Robot* with the Boolean member fluent *busy* and the member action *drive*. The *drive* action has as precondition, such that the robot must not be busy. To reference the *busy* fluent, the condition of the *drive* action uses the following term:

```
this.busy
busy
```

This will obtain a reference to the *busy* fluent for the current instance of the *Robot* type. Part of this term can be omitted: simply writing *busy* without *this* will obtain the same reference. If we have a global Boolean fluent *busy*, it can be referenced by writing:

```
.busy
```

The look up process would begin at the global scope in that case.

5.2 Compatibility to PDDL

OPL is intended as an extension to a PDDL planning system. Therefore it is based on the same concepts as PDDL. A domain file defines types, fluents and actions. A problem file defines an initial state and a goal condition. This similarity in structure allows an OPL domain to be easily converted to a PDDL domain. With the conversion tools provided with OPL, a PDDL planning system can be modified to understand OPL with little effort.

The conversion process adheres to the following rules:

First, each capital letter in any name is converted to lower case and suffixed with a minus -, to account for case insensitivity in PDDL. The minus sign was chosen, because OPL does not allow minus signs in its names, so accidental name collision cannot occur. For instance assume we have a type with the name *Robot*. After conversion the PDDL type has the name *r-obot*.

PDDL does neither have member fluents nor actions. In OPL these have their own naming scope. In order to convert from OPL to PDDL, the name of each member fluent

and action is prefixed with the name of the type they belong to, followed by an underscore _ . For instance we assume the type *Robot* from the example in the previous section has a Boolean member fluent *busy* and a member action *drive*. When converted, the names change to *r-obot_ busy* and *r-obot_ drive*. The same reasoning is applied to the scopes of actions. Assuming the *drive* member action has one argument *destination*, the name of that argument is converted to *r-obot_ drive_ destination*. Analogous, the names of objects declared in a problem are prefixed with *problem_* when converted. If we have two instances of the type *Robot*, *rob1* and *rob2*, they are converted to *problem_ rob1* and *problem_ rob2*.

Furthermore, member fluents and actions belong to one specific object. If we have two instances of the type *Robot* – *rob1* and *rob2* – the values of the member fluent *busy* have individual values for each object instance. When converting from OPL to PDDL this is resolved by adding one parameter to each member fluent and action: the object it belongs to. In our example the *busy* member fluent has no arguments. After the conversion, it will have one argument with the name *this* and the type *r-obot*. Analogous, the *drive* member action has one *destination* argument, it will have two arguments after the conversion: first one of the type *r-obot* followed by the *r-obot_ drive_ destination* argument.

Finally, some function names in OPL differ from their PDDL counterpart. The temporal functions *at start*, *at end* and *over all* are written as *atStart*, *atEnd* and *overAll* in OPL. This renaming was done for reasons of consistency: all OPL functions consist of a single word.

After the conversion the PDDL files might be more difficult to read for the human eye. However, this is not a disadvantage, since the converted files are intended only to be read by a PDDL parser.

5.3 Domain-Specific Module Interface

The TFD planning system is domain-independent; it can be applied to any planning task defined in PDDL. Therefore, the TFD module interface can not incorporate domain-specific information. As a result of its generic nature, the module interface is cumbersome to implement for a module developer. The information exchange between the module and the internal state of the TFD is encoded in plain strings. Obviously, the compiler can not be utilized to mismatched names in strings. Figure 1 illustrates the situation.

OPL provides a solution to this problem. When a OPL domain is translated, a domain-specific module interface is generated. The generated interface acts as an adapter between the domain-specific module implementation and the domain-independent TFD interface. Figure 2 visualizes the concept. Throughout the remainder of this Thesis, we call modules implementing the generic TFD interface *TFD modules* to distinguish them from *OPL modules*, which implement the generated OPL interface.

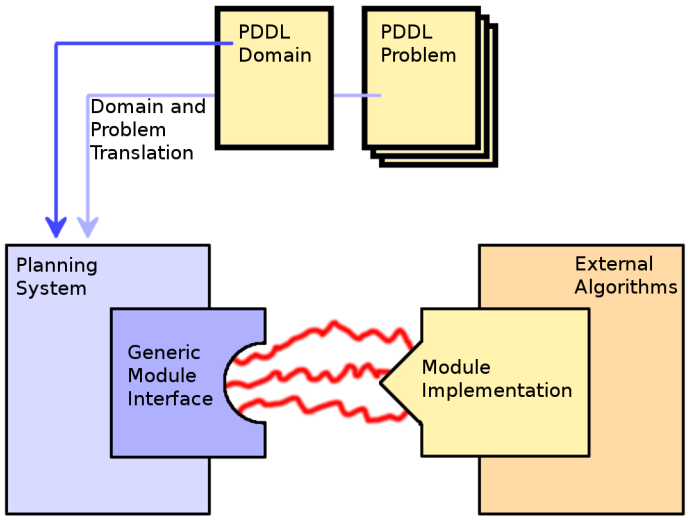


Figure 1: The domain-specific content (orange) of the PDDL planning task specification and module implementation transfer state information via the domain-independent (blue) module interface of the TFD. The exchange process (red) is complex and error prone.

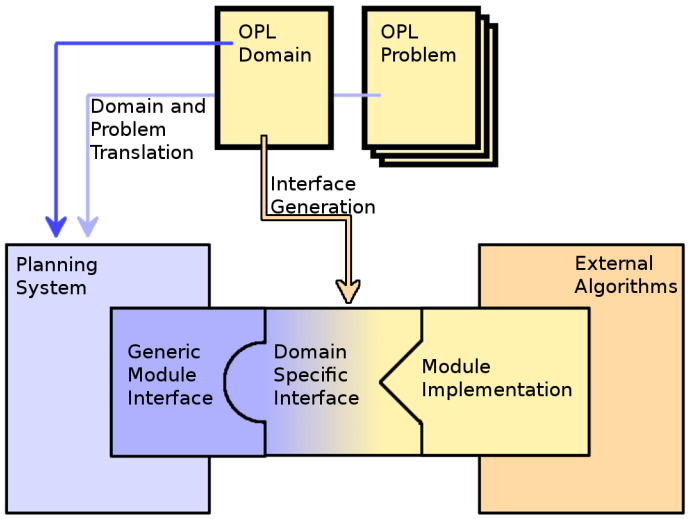


Figure 2: A domain-specific interface is generated from an OPL domain. It bridges the gap between the domain-independent (blue) TFD interface and the domain-specific (orange) module implementation. The generated OPL module interface provides efficient and safe state access to the module.

The generation process takes advantage of the structural similarities between OPL and C++. For each type defined in the domain, a corresponding class is generated. These classes provide methods to request fluent values for every member fluent of the associated type. In the internal state of the TFD, all fluents are represented by double variables. The generated methods convert these values into the appropriate data type, depending on the fluent. A *State* class is generated, which provides access to global fluents in the same manner. In addition, the state class provides methods to iterate through all objects of a type that were specified in the problem file. Each module definition in the domain file produces a skeleton C++ implementation for the module. With this interface accessing the planner state in an type safe manner becomes as easy as calling the provided methods. A module developer can focus his full attention on the functionality of the module.

The details of the generation process are presented in chapter 6.1.

5.4 Syntax of OPL

When comparing the syntax of OPL to PDDL, little resemblance is visible. Instead, the syntax is intended to resemble object oriented programming languages, like C++ or Java. The similarity is supposed to lower the learning curve for application developer who have no prior experience with PDDL.

In this section, we define each element of OPL with an EBNF representation, followed by a brief explanation and several syntax examples. The EBNF representation is intended to help understand the OPL syntax structure; some of the formalism might be not completely accurate.

- Names in OPL are defined as shown in the following EBNF listing:

  ```
  name = letter,{digit|letter|"_"}*;
  number = digit[.{digit}*];
  ```

  where a *letter* stands any lower or upper case letter and *digit* stands for a natural number from 0 to 9. All expressions in OPL are case sensitive. Additionally, a valid name must not be one of the following key words: *Domain*, *Type*, *Task*, *Object*, *boolean*, *number*, *DurativeAction*, *CostModule*, *ConditionModule*, *EffectModule*, *and*, *or*, *not*, *atStart*, *atEnd*, *overAll* or any other PDDL function name.

- `domain = "Domain", name, "{", {type | fluent | action | module}*, "}";`

  The *Domain* element encompasses any fact types, object types, relations and operators available in the world. It must be the root element in the domain file. Domain elements can not be contained in any other element and only one domain element is allowed per file. It has two identifiers: first the key word *Domain* followed by the name of the domain. A *Domain* element can contain the following elements: *Type*,

Fluent, *Module* and *DurativeAction* elements. The example below defines a domain with the name *RoomScanning*:

```
Domain RoomScanning
{
  [...]
}
```

- `type = "Type", name, [":", name], "{", {fluent | action | module}*, "}";`

  Type elements specify a certain type of object in the world. They can inherit from other *Type* elements. If no super type is specified, the super type defaults to the abstract type *Object*. The super type must be specified in the same Domain element as the derived type. *Type* elements can only be contained in the body of a *Domain* element. *Type*s that inherit from the base type *Object* are defined with the key word *Type* followed by the name of the type. *Type*s that inherit from other types have two additional identifiers: the inheritance operator colon, followed by the name of the inherited type. A *Type* element can contain *Fluent*, *Module* and *DurativeAction* elements. The following example defines the type *Pose* that inherits directly from the built-in type *Object* and a second type *Target* that inherits from *Pose*:

```
Type Pose
{
  [...]
}
Type Target: Pose
{
  [...]
}
```

- `fluent = ("boolean" | "number" | name), name, optional arguments, ";";`
  `optional arguments = ["(", argument,{",", argument}*, ")"];`

  Fluent elements represent facts in the world. They come in three flavors: predicates to represent Boolean facts, numerical fluents to represent numbers and object fluents to represent the relation to another object in the world. *Fluent* elements can be contained in a domain element, in which case they are considered global. In addition, they can be contained in *Type* elements, in which case they are member fluents. *Fluent* elements of two identifiers: the key word *boolean* for predicates, *number* for numerical fluents and the name of the type for object fluents, followed by the name of the fluent. *Fluent* elements may contain *Arguments*. The following example defines a Boolean fluent *busy* without arguments, a Boolean fluent *isNear* with one argument, a numerical fluent x and an object fluent of type *Pose* named *currentPose*:

```
boolean busy;
boolean isNear(Pose p);
```

```
number x;
Pose currentPose;
```

- `argument = name, name;`

  Argument elements specify a reference to an object. These elements can only be contained in *Fluent*, *Module* and *DurativeAction* elements. They consist of two identifiers: the name of the object type followed by the name of the reference. The following example shows the Boolean fluent *isNear* with an argument of the type *Pose* named *p* and the numerical fluent *pathLength* with two arguments of type *Pose* named *startPose* and *endPose*:

  ```
  boolean isNear(Pose p);
  number pathLengt(Pose startPose, Pose endPose);
  ```

- ```
  action = "DurativeAction", name, optional arguments,
     "{", duration, condition, effect, "}";
  duration = "Duration", "{", number | symbol, ";", "}";
  condition = "Condition", "{", function, ";", "}";
  effect = "Effect", "{", function, ";", "}";
  ```

  DurativeAction elements describe actions that can be executed, when a certain condition is met and can change fluents as specified in their effect statement. *DurativeAction* elements have two identifiers: the *DurativeAction* keyword followed by the name of the action. They may have optional arguments. *DurativeAction* elements can be contained in Domain or in Type elements. *DurativeAction* elements contain three sub-elements:

  - The *Duration* specifies the temporal duration of the action. It may either contain a *Number*, a *Symbol* element pointing to a numerical fluent or a *Symbol* element pointing to a duration module.

  - The *Condition* specifies the values for certain fluents in the state, before this action can be applied. It must contain a *Function* element.

  - The *Effect* specifies new values for certain fluents in the state. The new values are applied as specified by temporal functions. It must contain a *Function* element. The following example defines a *DurativeAction* named *scan* with one argument.

    ```
    DurativeAction scan(Target t)
    {
      Duration{40;}
      Condition{[...]}
      Effect{[...]}
    }
    ```
    ```

- `function = name, "(" function | symbol, ")";`

  Function elements are used to represent the logical relation of conditions or effects. They have one identifier: the function name. They have at least one argument, either another *Function* element or a *Symbol* element. Functions have the same names and functionallity as in PDDL except for the temporal functions *at start*, *at end* and *over all*. For consistency they are replaced with *atStart*, *atEnd* and *overAll* respectively. The following example shows the Boolean function *and*, which contains further nested functions:

  ```
  and(
    atStart(not([...])),
    atStart(not(equals([...], [...])))
  )
  ```

- `symbol = "this" | (name,["(", symbol, ")"]),[".", symbol];`

  Symbol elements provide means to reference fluents, objects or modules in the domain. They have a single identifier: the name of the referenced fluent or module. If the referenced symbol is an object fluent or an *Argument* element the scope operator dot can be used to reference member fluents defined in the corresponding *Type* element. The key word *this* can be used instead of a name to reference the current object. When referencing *Fluent* or *Module* elements, *Symbol* elements must contain additional *Symbol* elements as arguments. The number and type of the arguments must be identical to the arguments defined in the referenced fluent or module. The following example shows a symbol named *isNear* with another symbol *destination* as argument. In the second line, three symbols are shown concatenated with the scope operator:

  ```
  isNear(destination)
  this.currentPose.x
  ```

- `module = ("ConditionModule" | "DurationModule", name, ";") |`
  `"EffectModule", name, "{", {symbol, ";"}* "}";`

  Module elements provide the means to integrate external applications into the planning process. There are three type of modules: duration modules compute the duration of an action. Condition modules allow to check, whether an action is applicable in the current state. Effect modules allow to change specified numerical values in the state. Module elements have two identifiers: one of the key words *ConditionModule*, *CostModule* or *EffectModule* followed by the name of the module. Module elements can have optional arguments. *EffectModule* elements must also contain at least one *Symbol* element. The symbols specify, which fluents can be changed by the effect module. If a symbol points to a numerical fluent, that fluent may be changed. If a symbol points to a object fluent, all numerical member fluents of the corresponding type are allowed to be changed. It is not necessary to

specify a function and library name for the module call, since the library and the function will be generated automatically. The following example defines a condition module named *reachable* with two arguments, a duration module named *moveDuration* also with two arguments and an effect module named *findGraspingPose* with one argument. The effect module specifies that all numerical member fluents of the *gripperPose* argument can be changed by the module.

```
ConditionModule reachable(Pose startPose, Pose endPose);
DurationModule moveDuration(Pose startPose, Pose endPose);
EffectModule findGraspingPose(Pose gripperPose)
{
  gripperPose;
}
```

- `problem = "Problem", name, "(", name, ")",`
  `"{", {object | initialization}*, goal, "}";`

The *Problem* element defines a specific scenario for the planner to solve. It specifies initial values for fluents, instantiates objects and specifies a goal condition. The *Problem* element must be the root element in the problem file. It has two identifiers: the key word *Problem* followed by the name of the problem. It must have the name of the corresponding OPL domain as argument. It can contain arbitrary many *Object* and *Initialization* elements and exactly one *Goal* element. The example below defines a problem named *Scenario1* for the *RoomScanning* domain:

```
Problem Scenario1(RoomScanning)
{
  [...]
}
```

- `object = name, name, ";" | "{", {initialization}*, "}";`

Object elements allocate objects of the specified type. They have two identifiers: the name of the *Type* element followed by the name of the object. The *Object* element can contain *Initialization* elements to assign an initial value for the member fluents of this object. The example below shows the object *r1* of the type *Robot* without initializations and the object *p1* of the type *Pose*, containing one initialization:

```
Robot r1;
Pose p1
{
  x = 1;
}
```

- `initialization = name | name, "=", number | name, "=", name;`

Initialization elements assign an initial value to Boolean, numerical and object fluents. Initialization for Boolean fluents only have one identifier: the name of the fluent. This will initialize the Boolean fluent to the value *true*. A numerical fluent is initialized with the assign operator. The name of an instantiated object can be assigned to an object fluent with the assign operator. If no initialization is specified for a fluent it is initialized to a default value: *false* for Boolean fluents, *0* for numerical fluents and *NULL* for object fluents. Any comparison with *equals* between the object fluent with the value *NULL* and any object has the Boolean value *false*. The following example shows the initialization of the numeric fluent x and the Boolean fluent *explored*:

```
Target t1
{
  x=10;
  explored;
}
```

- `goal = "Goal", "{", function, "}";`

  The *Goal* element specifies a state, when the current problem is considered solved. There must be only one *Goal* element per task. Analogous to the conditions of *DurativeAction* elements, the *Goal* element contains one *Function* element. The example below shows a typical goal condition:

```
Goal
{
  and(target1.explored,
    target2.explored,
    target3.explored);
}
```

The complete EBNF representation of OPL is shown below as reference:

```
name = letter,{digit|letter|"_"}*;
number = digit[.{digit}*];
domain = "Domain", name, "{", {type | fluent | action | module}*, "}";
type = "Type", name, [":", name], "{", {fluent | action | module}*, "}";
fluent = ("boolean" | "number" | name), name, optional arguments, ";";
optional arguments = ["(", argument,{",", argument}*, ")"];
argument = name, name;
action = "DurativeAction", name, optional arguments,
   "{", duration, condition, effect, "}";
duration = "Duration", "{", number | symbol, ";", "}";
condition = "Condition", "{", function, ";", "}";
```

```
effect = "Effect", "{", function, ";", "}";
function = name, "(" function | symbol, ")";
symbol = "this" | (name,["(", symbol, ")"]),[".", symbol];
module = ("ConditionModule" | "DurationModule", name, ";")
   | "EffectModule", name, "{", {symbol, ";"}* "}";

problem = "Problem", name, "(", name, ")",
   "{", {object | initialization}*, goal, "}";
object = name, name, ";" | "{", {initialization}*, "}";
initialization = name | name, "=", number | name, "=", name;
goal = "Goal", "{", function, "}";
```

6 OPL Module Interface

As was stated in section 4.3, the search program needs a generic module call interface. The search engine must be domain independent. Nevertheless, from the perspective of a module developer, that generic interface is cumbersome to use. OPL provides a solution to this problem. When a OPL domain file is translated to PDDL, a domain specific Module interface is generated. This interface adds a layer on top of the TFD interface. It provides type save access to all OPL objects and fluents, without the necessity to refrain to strings based description of the desired values.

In the following sections we take a look at the C++ code generation process and the improved variable look-up mechanisms.

6.1 Interface Generation

In this section the code generation procedure for the OPL module interface is explained.

OPL was designed to work on top of existing TFD module interface. Only a few lines of code, for the initialization of the OPL interface, were added in the search program. When the search program executes a module call, it uses the original interface, as described in section 4.3. OPL provides a translation from the original module interface to the generated OPL interface. Before explaining how the translation works, we show what classes are generated.

Some of the required functionality of the module interface is domain independent. There is no need to have this functionality in the generated code. Instead, it is implemented in various classes located in the *opl::interface* namespace.

- The *OplObject* class represents the abstract base class for any custom type defined in a OPL domain. It provides a public method to get the object id, which is unique

for every object instantiated during the planning process. The object id is read form the object name defined in the OPL task file.

- The *AbstractState* class exposes the internal state of the search program to a module. The state variables can only be read from the module, but not written. The *AbstractState* class stores mappings from fluent names and object ids to the corresponding state variables. It also provides methods to obtain references to all instantiated *OplObject*s. For each domain, a concrete *State* class is generated, which subclasses the *AbstractState*. Most of the methods of *AbstractState* are intended to be only called by the generated classes and are therefore not accessible from the module.

- The *AbstractStateFactory* class initializes the fluent mappings in a *State* object. The initialization process is executed only once per planning process. Since it is based on the abstract factory design pattern, the actual instantiation of the *State* object is left to a concrete, domain specific *StateFactory* subclass. The concrete subclass also handles instantiation of *OplObject*s.

Each domain creates its own namespace *opl::DomainName*, where *DomainName* stands for the name specified in the OPL domain file. This prevents naming conflicts between the generated classes and classes included from other sources.

- A class is generated for every OPL type defined in the domain file. The class is named as specified in the file. The generated class extends either *OplObject* or another OPL type, depending on the specification in the domain file. Member fluents of the OPL type produce corresponding getter method in the generated classes.

- A *State* class is generated, which subclasses the *AbstractState* class. In addition to the functionality of the base class, the generated *State* class provides access to object lists, one for each OPL type defined in the domain. Also, global fluents produce corresponding getter method.

- Each fluent defined in the domain file generates a function with the same name. Global fluents produce getter methods in the *State* class. Member fluents produce getter methods in the corresponding type class. These functions are declared constant using the C++ key word *const*, i.e. they can not change attributes of their class. The return value of these functions depends on the fluent: functions for Boolean fluents return *bool* values, functions for numerical fluents return *double* values and functions for object fluents return a *const* pointer to the corresponding type class.

- A *StateFactory* class is generated, which subclasses the *AbstractStateFactory* class. The *StateFactory* class handles the instantiation of a *State* object, the instantiation of *OplObjects* for each OPL type. The instantiation is executed once per planning process. Objects are instantiated as specified in the OPL task file. The instantiated

objects are placed in the corresponding list in the *State* class. Objects are also recursively added to the list of their super type; the list for *OplObject*s therefore contains all instantiated objects.

Now that we know the classes generated for the interface, we take a look at the actual translation from the original TFD interface to the new OPL interface.

As is explained in section 5.2, module declarations in OPL do neither specify a function nor a library name. The module function name is derived from the module name specified in the OPL domain file after the following pattern: *Scope_ moduleName*, where *Scope* stands for the OPL type name, in case of a member module or is an empty string in case of a global module. The library name is derived from the name of the domain as *libtfd_ opl_* , followed by the domain name, followed by the file ending *.so*. All of the generated classes and functions explained in this section are compiled into that library.

For each cost and condition module specified in the OPL domain file, three files are generated:

1. A header file is generated, declaring the OPL module interface function to be implemented by the module designer. The file name is equal to the function name with the C++ header suffix *.h*. The header declares a single function with the module name as discussed above. The function is located in the *opl::DomainName* namespace. The return value of the function depends on the module type; Boolean for condition modules and double for cost modules. The function declares a number of arguments: first, a constant pointer to a *State* object. If the module is a member of an OPL type, the next argument is a constant pointer to an object of the corresponding type class. Then follow the arguments of the module as specified in the domain file, with constant pointers to objects of the corresponding type classes. Finally, the last four arguments are the same as in the original TFD module interface. Their purpose is not related to OPL, therefore they are not discussed here.

2. A second header file is generated, declaring the TFD module interface function to be called by the search program. The function name consist of the module function name, followed by the suffix *_plannerCall*. Analogous, the file is named as the function with the C++ header file extension *.h*. The function is located in a *extern "C"* environment, which is necessary when calling a function via a function pointer. As the function implements the TFD module interface, it has the generic argument list and the fluent callback functions as arguments.

3. A *.cpp* file is generated, which implements the TFD module interface function. In this file, the translation form the generic TFD module interface to the domain specific OPL module interface takes place. First, a pointer to the *State* object is obtained. Then the parameter names are extracted from the generic parameter list. Pointers to the corresponding objects are obtained via the *State* object; the fluent callback pointers are ignored. Finally, the OPL module interface function is called, with the pointer to the *State* object and the other OPL type object pointers as

arguments. The return value of the OPL module function is converted to double and returned, according to the specification of the TFD module interface.

In this section we saw the code generation process explained in detail. The actual mechanisms to access the internal state of the search program is shown in the following section.

6.2 Planner State Access

In this section we examine how the OPL module interface exposes the internal planner state to a module.

The internal state of the Search program is represented by a double vector, as is explained in section 4.2. The Search program works solely on the double vector. The Translate program maps each fluent, object and module defined in the domain file to a variable in this vector. The mapping is constant throughout the search process. The OPL module interface takes advantage of the constant mapping. Before the search is started, a mapping table is initialized, which translates from a fluent name to the corresponding variable in the internal state vector.

The responsibility of tracking the fluent mappings is shared among the *VarVal* class, the *AbstractState* class and the *AbstractStateFactory* class. These classes are located in the *opl::interface* namespace.

- The *VarVal* class tracks the value of a single mapping. The Translate program determines how many mappings a fluent requires. It can also occur that a fluent is constant. In that case it will be eliminated from the state vector. However, a module might still request the value of that variable. The *VarVal* class accounts for both cases. Should the fluent have mappings, a *VarVal* object stores the index of each corresponding variable in the internal state vector. Otherwise the *VarVal* object stores the constant value of the fluent.

- The *AbstractState* class manages the fluent mapping table. A fluent mapping depends on the values of the fluent's parameters. Therefore, the key for the mapping table is created from the fluent name and the names of its parameter objects. The values in the mapping table are pointer to *VarVal* objects. A second mapping table is necessary to retrieve the correct object for a requested object fluent. The object retrieval table maps from an index and the corresponding value to a *OplObject* pointer. The *AbstractState* class also provides methods to retrieve *VarVal* objects from the table and to compute the table keys. Finally, it also stores a pointer to the internal state vector. When the internal state changes between subsequent module calls, the pointer is redirected to the new state without additional copy operations.

- The *AbstractStateFactory* class initializes the fluent mappings. The mapping information is extracted from a file created by the translate program, which specifies

all mappings for variable fluents and the values for constant fluents. After a *State* object is allocated, its mapping table is filled with the mappings according to the file. This process is executed only once per planning process, as the mapping does not change during the search.

The methods provided by these three classes are not intended to be used by a module developer, and thus are only exposed to the generated classes. As explained in section 6.1, the generated *State* class and the type classes provide getter methods to look up fluents. Inside these methods, the look-up process is implemented in three steps:

1. In the first step, a key for the mapping table is composed. The key consist of the fluent name and the names of all parameter objects passed to this fluent. If the fluent is a member of a type, the first parameter is the name of the object it is called on.

2. In the next step, the corresponding *VarVal* object is retrieved from the fluent mapping table in the *AbstractState* class using the composed key.

3. In the last step, the fluent value is retrieved using the *VarVal* object. The *AbstractState* class provides different methods to retrieve values for Boolean, numerical and object fluents. Thus the encoded double value from the internal state vector is converted to correct data type. In case of object fluents, the *AbstractState* the object pointer is obtained from the object retrieval table. With the object name a *OplObject* pointer is returned, which is cast to the concrete class of the object fluent, before returning the pointer to the module.

As is explained above, the number of fluent mappings for each fluent depends on the parameters of the fluent. Therefore, if a fluent has no parameters, only one mapping is created. In that case the fluent look-up process becomes simpler: the corresponding *VarVal* object is retrieved during the initialization. Throughout the search process, the first two steps of the look-up process are omitted when the fluent value is requested.

The same caching strategy can be applied to member fluents. Usually, when a member fluent has no parameters, it is still dependent on the name of its object. That means, one fluent mapping is created for each instantiated object. Thus, each object caches the corresponding *VarVal* object at initialization.

The following C++ listing shows the default retrieval process for the Boolean fluent *in*, member of the type *Package*. The fluent has a parameter of the type *Vehicle*. To retrieve the value of *in* for a specific vehicle, a key is created consisting of the id of the *Package* object and the id of the *Vehicle* object. The key is used to retrieve the corresponding *VarVal* object. In the final step, the VarVal object is interpreted as a Boolean value.

```
bool Package::in(const Vehicle* v) const
{
  const VarVal* inVariable;
```

```
  vector<string> inArguments;
  inArguments.push_back(getObjectID());
  inArguments.push_back(v->getObjectID());
  string inKey = ObjectLookupTable::instance
        ->createKey("Package_in", inArguments);
  inVariable = ObjectLookupTable::instance
        ->getVariable(inKey);
  return ObjectLookupTable::instance
        ->getBooleanValue(inVariable);
}
```

A cached fluent value retrieval is shown in the following listing. The numerical fluent *size*, member of the type *Package*, has no parameters, therefore the *VarVal* object can be fetched during initialization. Only the interpretation of the *VarVal* object as a numeric value is required during the search.

```
double Package::size() const
{
  return ObjectLookupTable::instance
        ->getNumericValue(sizeVariable);
}
```

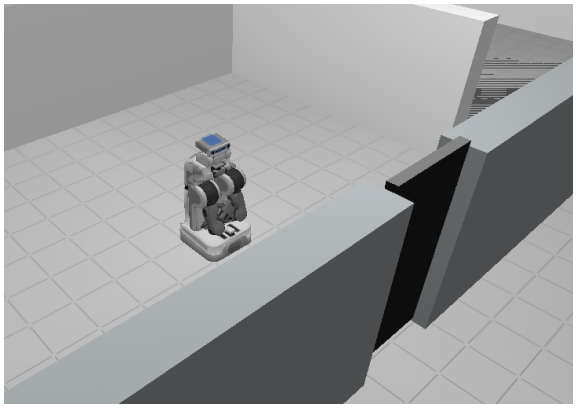
The following listing shows the *initialize* method of the *Package* class. It contains the code to retrieve the VarVal object for the *size* fluent.

```
void Package::initialize()
{
  vector<string> sizeArguments;
  sizeArguments.push_back(getObjectID());
  string sizeKey = ObjectLookupTable::instance
        ->createKey("Package_size", sizeArguments);
  sizeVariable = ObjectLookupTable::instance
        ->getVariable(sizeKey);
}
```

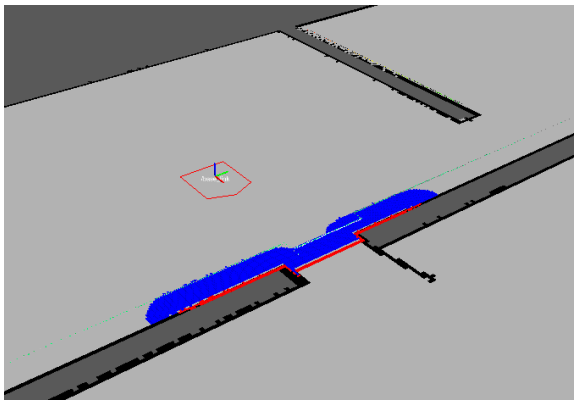
6.3 Room Scanning Domain

This section demonstrates, how real world data is integrated into planning process using OPL modules.

For this demonstration, we let an autonomous robot drive through an office like environment. The robot has to search a number of rooms for items. Some rooms might initially



(a) Gazebo Simulator: A simulated PR2 robot approaches a door in order to open it. The simulation is accurate enough to let the robot use the same motion control algorithms as a physical PR2 robot.



(b) Navigation Stack visualization: A metric 2D map represents the environment. Sensor readings of the robot are accumulated to detect obstacles (red) and calculate unsafe areas (blue) to be avoided by the robot.

Figure 3: Two views of the Room Scanning planning task: Figure (a) shows a screenshot from the Gazebo Simulator. Figure (b) depicts the same scene visualizing the map representation of the ROS Navigation stack.

be inaccessible due to closed doors. The robot knows the layout of the environment and the initial state of the doors a priori. We use symbolic planning to determine the optimal sequence of actions for the robot, to search all rooms in a minimum amount of time.

We chose ROS as the operating system for the robot in this demonstration. ROS provides tools and algorithms to simplify the development of robot applications. The components of ROS are organized in packages called *stack*s, which expose their services to other components.

The Gazebo Simulator is integrated with ROS. It is capable of simulating the sensors and actuators of a robot accurately in a 3D environment. As a result, a simulated robot can use the same algorithms as its physical counterpart to navigate in the simulated world. This is useful for developing and testing new algorithms, before deploying them on a physical robot.

For this demonstration, a simulated PR2 robot from Willow Garage is deployed in the Gazebo simulator. Figure 3 shows the PR2 robot in the simulated environment.

To complete the task, the PR2 needs to be able to navigate to specified locations and to open doors. The Navigation stack provides services to localize the robot in the environment and to compute safe paths between two poses. The pr2_doors stack provides services to enable a PR2 robot to open doors.

In the following section we iterate through the integration process. In section 6.3.1 we create an OPL domain for our planning task. In section 6.3.2, we explain the implementation of an OPL module. In section 6.3.3 we compare the implemented OPL module to a

TFD module with the same functionality to highlight the advantages of the OPL module interface.

6.3.1 Room Scanning OPL Domain

We create the Room Scanning OPL domain according to our needs for the planning task. We want to represent a robot moving through the environment, searching for items. The environment consist of multiple rooms or corridors. The robot knows the environment and the coordinates for good scan locations in every room a priori. The goal of the robot is to scan every room, but the way to some rooms is blocked by closed doors.

From this requirements, we define the following types in our domain :

- *Pose*: This type represents a 3D position and orientation in the environment. It has three numerical fluents *x, y* and *z*, to specify the position vector, and four more *qx, qy, qz* and *qw* to specify the orientation quaternion. Furthermore, a condition module is defined, to check whether this pose can be reached from another given pose. Below the OPL definition of the *Pose* type is shown:

```
Type Pose
{
  number x; number y; number z;
  number qx; number qy; number qz; number qw;
  boolean isReachableFrom(Pose origin);
}
```

- *Target*: This type represents a scan position in the environment. It extends the type *Pose* and adds a Boolean fluent *explored* to track, whether a scan was conducted at the specified position. The OPL definition below shows the *Target* type:

```
Type Target : Pose
{
  boolean explored;
}
```

- *Door*: This type represents a door in the environment. The Boolean fluent *open* to track the state of the door. The object fluent *approachPose* of the type *Pose* specifies a position, from where a robot can manipulate the door.

```
Type Door
{
  Pose approachPose;
```

```
  boolean open;
}
```

- *Robot*: This type represents a robot. The position of the robot is tracked with the object fluent *currentPose* of the type *Pose*. The Boolean fluent *busy* prevents the robot form executing two actions at the same time. The duration module *driveDuration* performs service calls to the ROS navigation stack, in order to compute a accurate path length between the current pose of the robot and a specified destination. Additionally, the *Robot* type defines three actions that are explained in detail further below.

```
Type Robot
{
  Pose currentPose;
  boolean busy;
  DurationModule driveDuration(Pose destination);
  DurativeAction scan(Target poi)
  {...}
  DurativeAction openDoor(Door door1)
  {...}
  DurativeAction drive(Pose destination)
  {...}
}
```

- *scan*: This action lets the robot explore one target. To execute this action the robot must not be busy with another action, it must be located at the target pose and the target must not be explored yet. When the action is started, the robot is flagged as busy, to prevent other actions from interfering. After the action finishes, the robot stops being busy and the target is flagged as explored.

```
  DurativeAction scan(Target poi)
  {
    Duration{20.0;}
    Condition
    {
      and (atStart (not (busy)),
           atStart (equals(currentPose, poi)),
           atStart (not (poi.explored)));
    }
    Effect
    {
      and (atStart (busy),
           atEnd (not (busy)),
           atEnd (poi.explored));
```

```
  }
}
```

- *openDoor*: This action lets the robot open one door. Analogous to the *scan* action, the *busy* fluent prevents the robot from executing other actions, while the *openDoor* action is in progress. To execute the action the robot has to be at the *approachPose* of the door and the door must be closed. When the action finishes, the door is open.

```
DurativeAction openDoor(Door door1)
{
  Duration {100.0;}
  Condition
  {
    and (atStart (not (busy)),
         atStart (equals (currentPose, door1.approachPose)),
         atStart (not (door1.open)));
  }
  Effect
  {
    and (atStart (busy),
         atEnd (not (busy)),
         atEnd (door1.open));
  }
}
```

- *drive*: This action lets the robot move from its current pose to a new destination. Again, the *busy* fluent prevents the robot from executing other actions, while the *drive* action is in progress. The duration of the drive action is computed via the module *driveDuration*. The module request a path calculation from the ROS navigation stack, which is a costly operation; getting the result may take several seconds. Therefore, the *isReachable* module tracks the state of doors and prevents the *drive* action from execution, in cases where the destination is blocked by a closed door.

```
DurativeAction drive(Pose destination)
{
  Duration {driveDuration(destination);}
  Condition
  {
    and (atStart (not( equals(currentPose, destination))),
         atStart (not (busy)),
         atStart (destination.isReachableFrom(currentPose)));
  }
  Effect
  {
```

```
      and (atStart (busy),
          atEnd (assign (currentPose, destination)),
          atEnd (not (busy)));
    }
  }
```

A problem file for the Room Scanning OPL domain can instantiate any number of robots, targets and doors. However, the domain was designed with one robot in mind and is not suitable for multi-robot planning tasks.

Below we show how to instantiate each object type in a problem file:

- *Target*: For each room to be scanned, the problem file must specify the coordinates of the scan pose. By omitting the *explored* fluent, it is initialized with the value *false*. The OPL code is shown below:

```
Target target1
{
  x = 10.0; y = 0.3; z = 0;
  qx = 0; qy = 0; qz = 0; qw = 1;
}
```

- *Door*: To add a door to the planning task, the coordinates of the door need to be specified as a *Pose* object and assigned to the *approachPose* fluent of the door.

```
Pose door1Pose
{
  x = 5.0; y = 0.3; z = 0;
  qx = 0; qy = 0; qz = 0; qw = 1;
}
Door door1
{
  approachPose = door1Pose;
}
```

- *Robot*: To add a robot to the planning task, the starting pose of the robot needs to be specified and assigned to the *currentPose* fluent of the robot.

```
Pose robot1StartPose
{
  x = 1.0; y = 0.3; z = 0;
  qx = 0; qy = 0; qz = 0; qw = 1;
}
```

```
Robot robot1
{
  currentPose = robot1StartPose;
}
```

Finally, the problem file also specifies a goal condition. In the listing shown below, we want *target1* to be explored:

```
Goal
{
  and (target1.explored);
}
```

With the Room Scanning OPL domain specified, we continue with the OPL module implementation in the following section.

6.3.2 OPL Module Implementation

In section 6.3.1 we defined the Room Scanning OPL domain, which contained two module definitions. In this section we exemplary show the implementation of the duration module *driveDuration*.

First, we translate the OPL domain with the *opl_translate_domain* program. The program produces a PDDL domain equivalent to the OPL domain and C++ module interfaces for each module defined in the domain. The header generated for the *driveDuration* module is shown below:

```
#ifndef RoomScanning_Robot_driveDuration_H_
#define RoomScanning_Robot_driveDuration_H_
#include "State.h"

namespace opl
{
namespace RoomScanning
{
double Robot_driveDuration(const State* currentState,
    const Robot* thisRobot,
    const Pose* destination,
    int relaxed);
}
}
#endif
```

After the usual include guard, the namespaces for the Room Scanning domain are declared. The module interface function returns a *double* value, since it is generated for a duration module. In the OPL domain file, the module is defined as member of the *Robot* type. Thus, the name of the function starts with *Robot_* followed by the name of the module. The first argument is a constant pointer to a *State* object. As described in section 6.1, the *State* pointer provides read only access to all objects and global fluents defined in the domain. The second argument is a constant pointer to a *Robot* object. This argument is generated because the module is defined as member of the *Robot* type in the domain file. It points to the robot that currently tries to execute the *drive* action to expose all member fluents of the *Robot* to the module. The third argument is a constant pointer to a *Pose* object according to *Pose* parameter in the module definition. The last argument indicates, whether the value returned by the module call must be precise or can be approximate. The latter case occurs, when the duration is required to compute a heuristic value, while the former case occurs, when the action is actually added to the plan.

We begin the module implementation by creating a cpp file. We include the module interface header and implement the declared function. The C++ code listing below shows the first half of the implementation. For clarity, some parts of the implementation are omitted; in particular debug code and ROS functions, if they are not related to the module interface.

```
double Robot_driveDuration(const State* currentState,
   const Robot* thisRobot,
   const Pose* destination,
   int relaxed)
{
  if (serviceClient == NULL)
  {
    initializeClient();
  }
  const Pose* robotPose = thisRobot->currentPose();
  // query path via service
  nav_msgs::GetPlan srv;
  setPose(srv.request.start.pose, robotPose);
  setPose(srv.request.goal.pose, destinationPose);
  [...]
```

We start by initializing the ROS service client, if it was not yet initialized in a previous module call. The ROS service client provides the means to request a path planning operation from the ROS Navigation stack. We call the *currentPose* method of the *thisRobot* object to obtain a reference to the robot pose. The *GetPlan* message contains the details of our path planning request. We call the *setPose* function to copy the coordinates of the robot pose and the destination pose into the message. The following listing shows the implementation of the *setPose* function:

```
void setPose(ros::geometry_msgs::Pose& rosPose,
   const Pose* oplPose)
{
  rosPose.position.x = oplPose->x();
  rosPose.position.y = oplPose->y();
  rosPose.position.z = oplPose->z();
  rosPose.orientation.x = oplPose->qx();
  rosPose.orientation.y = oplPose->qy();
  rosPose.orientation.z = oplPose->qz();
  rosPose.orientation.w = oplPose->qw();
}
```

The *setPose* function basically translates between the *Pose* class generated from the OPL domain and the *Pose* message class required for ROS service calls. The coordinate values are obtained via the fluent getter functions and assigned to the corresponding field in the message pose.

We continue in the module interface function. The listing below shows the second half of the implementation:

```
  [...]
  if (serviceClient.call(srv))
  {
    if (!srv.response.plan.poses.empty())
    {
      double pathLength;
      // compute path length:
      // sum of waypoint distances
      [...]
      return pathLength;
    }
    [...]
  }
  return modules::INFINITE_COST;
}
```

After filing the message with the coordinates of the robot and the destination, we send the service request. Further execution is blocked until the response arrives. If the path planning process succeeded, the response message contains a number of waypoints. We compute the path length by summing up the distances between the waypoints. We do not consider the maximum velocity of the robot and simply return the path length as action duration. If the path planning process failed, we return infinite cost.

This concludes the implementation of the OPL module.

6.3.3 TFD Module Comparison

To highlight the benefits of the OPL module interface, we compare the module implemented in section 6.3.2 to a TFD module implementation with the same functionality.

Before beginning the implementation, we create a PDDL version of the Room Scanning domain. The PDDL listing below shows the parts to module implementation:

```
(:types
  robot pose - object
)
(:modules
  (drive_duration ?rob - robot  ?destination - pose
     cost driveDuration@libtfd_RoomScanning.so)
)
(:functions
  (current_pose ?rob - robot) - pose
  (x ?p - pose) - number
  (y ?p - pose) - number
  (z ?p - pose) - number
  (qx ?p - pose) - number
  (qy ?p - pose) - number
  (qz ?p - pose) - number
  (qw ?p - pose) - number
)
```

For the module implementation we only care about the types *robot* and *pose*. The drive duration module is defined with two parameters; the first of the type *robot*, the second of type *pose*. The object fluent function *current_pose* specifies at which pose object a robot is located. To represent the coordinates of pose objects, we need seven numerical fluents: three for the position vector and four more for the orientation quaternion.

With the domain definitions above, we can begin the module implementation. The signature of the TFD module function is explained in section 4.3.

As in its OPL counterpart, we omit parts of the implementation and focus on the interaction between module and planner state. The following listing shows the first half of the module interface function:

```
double driveDuration(
   const ParameterList& parameterList,
   predicateCallbackType predicateCallback,
   numericalFluentCallbackType numericalFluentCallback,
```

```
   objectFluentCallbackType objectFluentCallback,
   int relaxed)
{
if (serviceClient == NULL)
  {
    initializeClient();
  }
  string robotName = parameterList.at(0).value;
  string destinationName = parameterList.at(1).value;
  ObjectFluentList* ofl = new ObjectFluentList();
  ParameterList robotPL;
  robotPL.push_back(Parameter("rob", "robot", robotName));
  ofl->push_back(NumericalFluent("current_pose", robotPL));
  if (!objectFluentCallback(ofl))
  {
    printf("Error in current pose request!");
    exit(0);
  }
  string robotPoseName = ofl->at(0).value;
  delete ofl;
  // query path via service
  nav_msgs::GetPlan srv;
  setPose(srv.request.start.pose, robotPoseName,
     numericalFluentCallback);
  setPose(srv.request.goal.pose, destinationName,
     numericalFluentCallback);
  [...]
```

After the initialization of the ROS service client, we obtain the name of the robot object from the parameter list. The parameter list is sorted according to the module definition. Therefore, the robot parameter is found on the index 0 and the destination pose on index 1. We create an object fluent with the name *currentPose* with our robot as parameter. The object fluent is inserted in to the object fluent list. We use the object fluent callback function, to request the value of the of the current pose of the robot from the internal planner state. The module call (and the whole planning process) is aborted, if something went wrong during the callback. Finally, we have obtained the name of the pose object, at which the robot is currently located.

Next, we create the ROS service message. The *setPose* function initializes the poses in the service request message. The following listing shows the *setPose* function:

```
void setPose(ros::geometry_msgs::Pose& rosPose,
   const String& poseName,
   modules::numericalFluentCallbackType numericalFluentCallback)
{
```

```
  NumericalFluentList* nfl = new NumericalFluentList();
  ParameterList pl;
  pl.push_back(Parameter("p", "pose", poseName));
  nfl->push_back(NumericalFluent("x", pl));
  nfl->push_back(NumericalFluent("y", ParameterList(pl)));
  nfl->push_back(NumericalFluent("z", ParameterList(pl)));
  nfl->push_back(NumericalFluent("qx", ParameterList(pl)));
  nfl->push_back(NumericalFluent("qy", ParameterList(pl)));
  nfl->push_back(NumericalFluent("qz", ParameterList(pl)));
  nfl->push_back(NumericalFluent("qw", ParameterList(pl)));
  if (!numericalFluentCallback(nfl))
  {
    printf("Error in setPose function!");
    exit(0);
  }
  rosPose.position.x = nfl->at(0).value;
  rosPose.position.y = nfl->at(1).value;
  rosPose.position.z = nfl->at(2).value;
  rosPose.orientation.x = nfl->at(3).value;
  rosPose.orientation.y = nfl->at(4).value;
  rosPose.orientation.z = nfl->at(5).value;
  rosPose.orientation.w = nfl->at(6).value;
  delete nfl;
}
```

To obtain the coordinate values of the pose, we have to populate the numerical fluent list. All the fluents we are interested in have the same parameter. Therefore, the parameter list is copied for every fluent in the list. When the list is filled with all seven fluent specifications, we use the numerical fluent callback function to request the values from the internal planner state. If everything was specified correctly, the returned values are assigned to the message pose.

The second half of the module interface function is identical to the OPL implementation; thus, it is omitted here.

The disadvantages of the TFD module interface are obvious. Fluent requests are specified using strings, making them complicated, error prone and inefficient. Multiple lines of code are required for every request. Errors can not be detected by the compiler.

The OPL module interface addresses these problems. A single function call retrieves the value from the internal planner state. The generated classes provide type safety, errors are detected at compile time. A module developer can concentrate on implementing the actual module algorithms instead of requesting fluent values.

7 Experimental Results

In this chapter we perform experiments to evaluate various aspects of the OPL module interface. The computational overhead caused in the planning system by module calls is examined in section 7.1. In section 7.2, we compare the efficiency of accessing the internal planner state from a TFD module and an OPL module. Section 7.3 shows results of an integration of the planning system with an simulated autonomous robot.

7.1 Crew Planning Domain

This section summarizes the results of experiments with the Crew Planning Domain.

The Crew Planning Domain is part of the temporal satisficing track of the 6th International Planning Competition 2008. The domain is based on astronauts working on a space ship. A certain number of research projects has to be completed until a specified deadline. Multiple crew member are available. Each crew member can do only one task at a time. Each crew member has a daily routine: every day he must sleep, eat and exercise. However, some activities require additional equipment, which places restrictions on how many crew member can do the corresponding activity at the same time.

For IPC 2008, the Crew Planning Domain presented the contenders with 30 problem files with increasing complexity. The first problem has only one crew member and workload of 60% for one day, while the 30th problem has three crew members working for three days at 100% load.

The Crew Planning Domain is represented solely with Boolean predicates; it does not require external modules. That makes it the perfect candidate to measure the overhead of external module calls. As state above, a crew member can only execute a single action at a time. This is enforced by the predicate *available*. Before a crew member starts a new action, the *available* predicate must have the value *true* for him. Once an action begins, the predicate is set to *false*. When the action finishes, the crew member is available once again.

For this experiment, a condition module named *emptyModule* was added to the domain. Every action that originally had the *available* predicate in its condition, now additionally calls the *emptyModule*. The module is very simple: it always returns *true*. Thus, it does not change the applicability of actions.

Three different domains were evaluated against each other, in order to determine the overhead of the OPL module interface. The original Crew Planning Domain from IPC 2008 acts as the base line. The Crew Planning Module Domain has additionally the *emptyModue* as described above, to show the overhead of the TFD module interface. The Crew Planning OPL Domain was created, containing equivalent types, fluents and

actions as the Crew Planning Module PDDL domain. This shows the overhead of the OPL module interface. For each of the 30 problem files 100 test runs were conducted. Table 1 shows the results of this experiment.

As can be seen in the table, the OPL module interface has in most experiments a significant lower overhead than the TFD module interface. For some problems the overhead increases the search time insignificantly compared to the original domain. However, due to an elusive bug in the search program, no solution was found for problem 25 of the OPL domain in any of the test runs. Even when removing the module calls from the OPL domain, the bug persisted. So far, the phenomenon seems to be restricted to this one problem.

7.2 Transport Domain

This section summarizes the results of experiments with the Transport Domain.

The Transport Domain is part of the temporal satisficing track of the 6th International Planning Competition 2008. The domain is based on a delivery company. A city consist of locations connected by roads. Multiple packages require deliverance to various locations in the city. A number of transport vehicles needs to be coordinated, in order to pick up and deliver every package to the specified destination in a minimum amount of time.

For this experiment, eight problems of the original 30 from IPC 2008 were selected. The complexity scales from the first problem with five locations, two vehicles and two packages up to the last problem with 45 locations, four vehicles and 18 packages.

During their work on [Dornhege et al., 2009], the authors created a TFD module. The module performs volumetric calculations to determine, whether a package fits into the vehicle. In order to do so, the module requests the sizes of all packages to be loaded into the vehicle and the vehicle's capacity from the internal planner state. We created an OPL domain and a corresponding module implementing the same algorithm for this Thesis. In this experiment, the performance of these two modules is compared, to determine the computational overhead of requesting fluent values from the internal planner state with OPL modules and TFD modules.

For each of the eight problem files 100 test runs were conducted. Table 2 shows the results of the experiment.

Apart from problem 3 and 4, OPL modules outperform TFD modules significantly: for instance, for problem 6 the planning time with the OPL module was more than three times as low. Especially for the complex problems with long search durations (problems 6, 7 and 8), the time advantage is enormous. For the problems 3 and 4 the situation is reversed: the OPL module produces longer search durations. The reasons for the reversal could not yet be determined.

	crew count	day count	work load	Original mean	Original std	Module mean	Module std	OPL mean	OPL std
1	1	1	40	3	4	8	4	5	5
2	1	1	60	7	5	13	5	9	5
3	1	1	80	2	4	9	5	5	5
4	1	1	100	18	4	25	5	22	4
5	2	1	40	41	3	55	5	50	4
6	2	1	60	73	6	89	4	85	5
7	2	1	80	96	5	119	4	120	4
8	2	1	100	87	5	110	5	95	6
9	3	1	40	152	6	186	5	172	6
10	3	1	60	265	6	322	6	275	6
11	3	1	80	238	5	291	6	304	6
12	3	1	100	286	6	351	6	348	6
13	1	2	60	32	5	42	4	37	5
14	1	2	80	48	5	60	4	54	6
15	1	2	100	36	5	45	5	35	5
16	2	2	60	266	5	328	6	289	6
17	2	2	80	314	7	395	6	337	6
18	2	2	100	388	6	417	6	372	8
19	3	2	60	1,040	10	1241	10	960	10
20	3	2	80	1,060	10	1290	10	1,220	10
21	3	2	100	1,350	10	1640	20	1,390	10
22	1	3	60	80	4	101	6	90	6
23	1	3	80	63	5	81	5	73	6
24	1	3	100	104	5	133	6	105	5
25	2	3	60	646	7	784	8	-	-
26	2	3	80	799	8	970	10	812	8
27	2	3	100	787	9	960	20	875	9
28	3	3	60	1,770	10	2,120	10	1,770	10
29	3	3	80	2,950	30	3,540	20	2,890	30
30	3	3	100	2,840	20	3,430	20	2,840	20

Table 1: Results of the Crew Planning experiment. The first column shows the problem number. Columns two to four indicate the complexity of the problem with the number of crew members, number of days and the amount of work. Column five shows the mean time required to find the first valid plan for each problem for the original Crew Planning Domain in milliseconds. Column six contains the corresponding standard deviations. The columns seven and eight show mean run time and standard deviation for the Crew Planning Module Domain. The columns nine and ten show show mean run time and standard deviation for the Crew Planning OPL Domain.

	location count	vehicle count	package count	TFD mean	TFD std	OPL mean	OPL std
1	5	2	2	9	3	3	5
2	10	2	4	119	6	39	4
3	15	3	6	524	8	1,016	7
4	20	3	8	1,310	10	2,100	10
5	25	3	10	3,360	30	2,890	20
6	30	4	12	87,500	400	17,910	160
7	35	4	14	140,900	400	68,000	1000
8	45	4	18	54,400	200	29,200	100

Table 2: Results for the Transport experiment. The first column shows the problem number. Columns two to four indicate the complexity of the problem with the number of locations, number of vehicles and number of packages. Column five shows the mean time required to find the first valid plan for each problem for the Transport TFD Module Domain in milliseconds. Column six contains the corresponding standard deviations. The columns seven and eight show mean run time and standard deviation for the Transport OPL Module Domain.

7.3 Room Scanning Domain

This section summarizes the results of experiments with the Room Scanning domain.

The Room Scanning Domain is based on an autonomous robot searching for items in various rooms. The environment consist of multiple rooms or corridors. The path to some rooms is blocked by closed doors. The robot has a metric map of the environment and knows the coordinates for good scan locations in every room. The robot wants to scan all target locations in a minimum amount of time.

For the experiment we created eight problems with varying complexity. The complexity ranges from two scan targets without any door in first problem, up to eight scan target and two doors in the last. The Room Scanning domain and the corresponding module implementation is explained in sections 6.3.1 and 6.3.2.

The planning process and the plan execution were evaluated separately. For each of the problems 250 planing iterations were conducted. The planning system was configured in *anytime* mode: the search continues after the first plan was found, until either the search space is completely explored or a predefined amount of time has passed. For this experiment the timeout was set to 180 seconds.

Table 3 shows the results of the planning experiment. As expected, the time required to find the first valid plan increases with problem complexity. However, for the best plan,

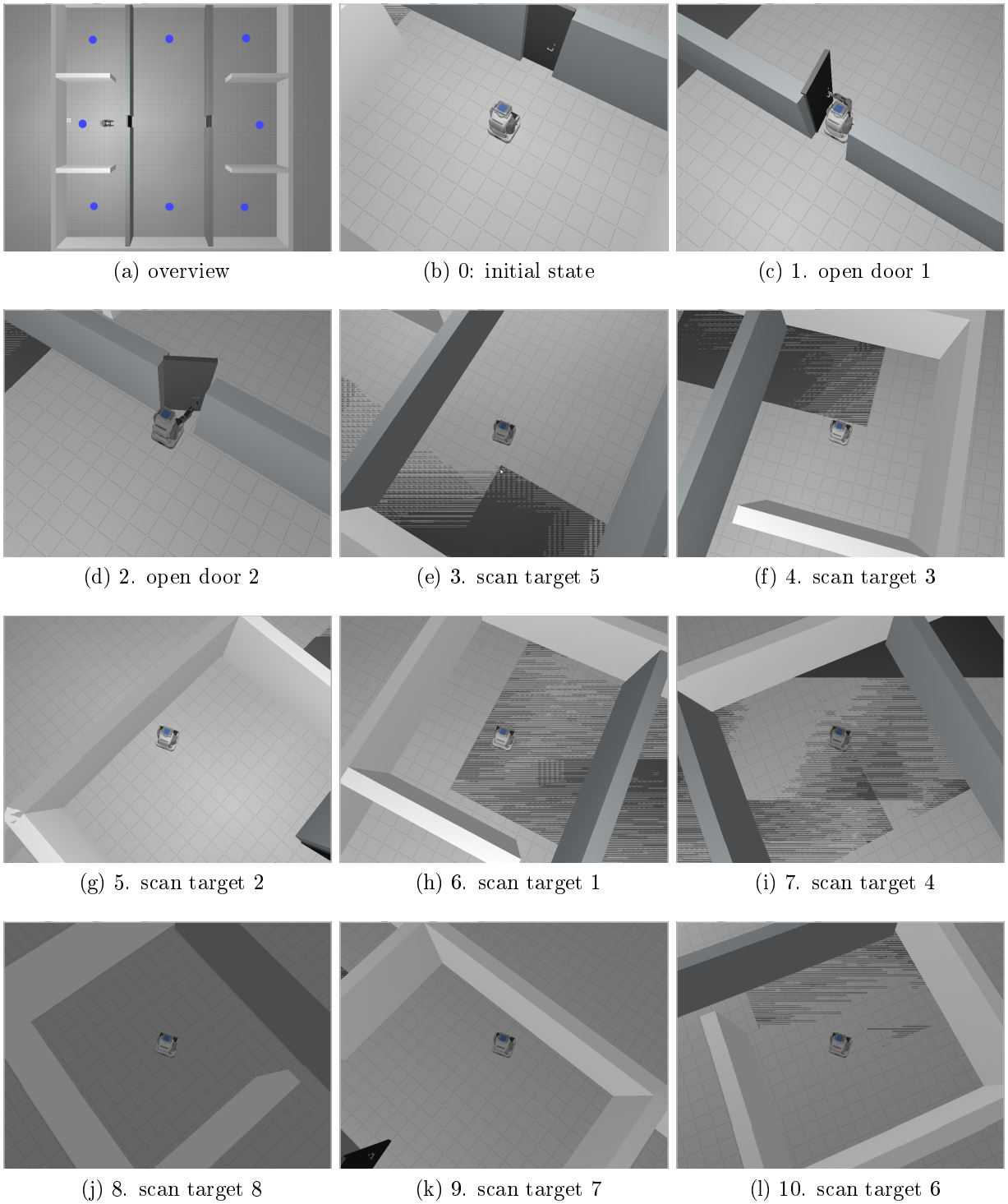
	target count	door count	first plan mean	first plan std	best plan mean	best plan std	plan improvements
1	2	0	13.8	0.3	13.8	0.3	1
2	3	1	17.9	0.3	18.7	0.3	2
3	4	1	28.5	0.4	48.0	0.5	4
4	5	1	33.4	0.5	40.7	0.6	5
5	5	2	48.9	0.6	86.2	0.8	6
6	6	2	57.5	0.7	160.0	20.0	10
7	7	2	64.3	0.7	95.0	2.0	7
8	8	2	74.2	0.8	173.0	8.0	21

Table 3: Results for the Room Scanning experiment. The first column shows the problem number. Columns two and three indicate the complexity of the problem with the number of scan targets and number of doors. Column four shows the mean time required to find the first valid plan for each problem for the Room Scanning OPL Domain in seconds. Column five contains the corresponding standard deviations. Column six shows the time required to find the best plan, with the standard deviation in column seven. Column seven indicates how often the plan was improved after the first plan was found.

problem 6 and 8 show a substantially higher standard deviation than the other problems. We suspect the high deviation is related to the timeout: for some iterations the plan was improved shortly before the timeout occurred, while for other iterations the search was aborted, before a better plan could be discovered. Instead, the previously discovered plan was used for the statistics, resulting in high time differences between iterations.

The number of plan improvements increases with problem complexity. Thus it can be argued, that with increased problem complexity the quality of the initial plan decreases.

The plan execution was evaluated exemplary. Once a valid plan is found, the commands sent to corresponding services in the ROS Navigation stack. The success of the execution only depends on the interactions between the Gazebo simulator and the various ROS stacks. Thus, an in-depth evaluation is beyond the scope of this Thesis. Figure 4 shows screenshots of the simulation throughout the execution of a plan that was generated for problem eight.



(a) overview (b) 0: initial state (c) 1. open door 1

(d) 2. open door 2 (e) 3. scan target 5 (f) 4. scan target 3

(g) 5. scan target 2 (h) 6. scan target 1 (i) 7. scan target 4

(j) 8. scan target 8 (k) 9. scan target 7 (l) 10. scan target 6

Figure 4: Execution of a valid plan for problem 8, with two doors and eight scan targets (blue circles).

8 Conclusion

Autonomous robotic applications become increasingly complex in recent years. Hard-coded mission control, for instance via finite state machines, does not scale well with complexity and will reach its limit soon. Methods of AI planning have the potential to bridge the gap: AI planning systems offer efficient and flexible high level reasoning. However, for real world applications the precision of low level reasoning is crucial. So far, only few attempts were made to integrate both, efficient high level reasoning and precise low level reasoning, into a single planning system.

One such attempt is implemented in the Temporal Fast Downward planning system. It offers a module interface to integrate external algorithms into the planning process. The module interface is domain independent, thus, no changes in the planner software are required for new domains. However, due to its generic nature, the coupling between a module and the internal planner state is inefficient, error prone and counter-intuitive. We believe an efficient and intuitive interface is key to widespread use of planning systems for real life applications.

In this Thesis we present the Object-oriented Planning Language (OPL), as a successor of PDDL. OPL offers a C++ like syntax, which reduces the learning curve for application designer unfamiliar with PDDL. More importantly, it provides tools, to automatically generate a domain specific module interface for the TFD planning system. The generated C++ classes provide type-safe and efficient access to the internal planner state. The generated interface integrates with the planner on top of the existing domain independent interface. Thus, the planning system remains domain independent. With the generated interface, developers can concentrate on algorithm design instead of wasting time on planning system integration. We demonstrate the improved module performance in various experiments.

The potential of OPL is not yet fully realized. So far, modules can only read the internal planner state, but not change the state. In the next step, we will include effect modules into the code generation process to provide a way for two-ways information exchange between planning system and external applications.

Furthermore, we believe the performance of the planning system can be improved significantly through means of automated module result caching. In most cases a module only depends on a sub set of the planning state. The dependency can be discovered by tracking, which part of the state is accessed via the module interface.

Finally, two more points of interaction exist between the planning system and an external application, besides the module interface. The specification of the initial state and the goal condition is currently handled via problem files. Once the planning process is completed successfully, the plan is transferred to the application via a plan file. To avoid these inefficient and error prone information exchange methods, additional domain specific interfaces could be generated.

References

[Christer Bäckström and Bernhard Nebel, 1995] Christer Bäckström and Bernhard Nebel (1995). Complexity Results for SAS+ Planning. *Computational Intelligence*, 11:625–655.

[Dornhege et al., 2009] Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., and Nebel, B. (2009). Semantic attachments for domain-independent planning systems. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 114–121. AAAI Press.

[Drew McDermott, 1998] Drew McDermott, Malik Ghallab, A. H. (1998). PDDL – The Planning Domain Definition Language – Version 1.2. Technical report, Yale Center for Computational Vision and Control.

[Eyerich et al., 2009] Eyerich, P., Mattmüller, R., and Röger, G. (2009). Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS)*, pages 130–137. AAAI Press.

[Kleiner and Dornhege, 2009] Kleiner, A. and Dornhege, C. (2009). Operator-assistive mapping in harsh environments. In *Proc. of the IEEE Int. Workshop on Safety, Security and Rescue Robotics (SSRR)*, pages 1–6, Denver, USA. (Best Paper Award Finalist).

[Malte Helmert, 2006] Malte Helmert (2006). The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246.

[Malte Helmert, 2009] Malte Helmert (2009). Concise finite-domain representations for PDDL planning tasks. *Artificial Intelligence*, 173:503–535.

[Maria Fox and Derek Ling, 2003] Maria Fox and Derek Ling (2003). An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20:61–124.

[Wurm et al., 2010] Wurm, K. M., Dornhege, C., Eyerich, P., Stachniss, C., Nebel, B., and Burgard, W. (2010). Coordinated exploration with marsupial teams of robots using temporal symbolic planning. In *Proceedings of the 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2010)*.