Planning with Semantic Attachments: An Object-Oriented View

Andreas Hertle and **Christian Dornhege** and **Thomas Keller** and **Bernhard Nebel**[1]

Abstract. In recent years, domain-independent planning has been applied to a rising number of real-world applications. Usually, the description language of choice is PDDL. However, PDDL is not suited to model all challenges imposed by real-world applications. Dornhege et al. proposed semantic attachments to allow the computation of Boolean fluents by external processes called modules during planning. To acquire state information from the planning system a module developer must perform manual requests through a callback interface which is both inefficient and error-prone.

In this paper, we present the Object-oriented Planning Language OPL, which incorporates the structure and advantages of modern object-oriented programming languages. We demonstrate how a domain-specific module interface that allows to directly access the planner state using object member functions is automatically generated from an OPL planning task. The generated domain-specific interface allows for a safe and less error-prone implementation of modules. We show experimentally that this interface is more efficient than the PDDL-based module interface of TFD/M.

1 INTRODUCTION

In recent years, domain-independent planning has been applied to a rising number of real-world applications, including battery charging [8], space applications [5], robotics [15, 21] or control of hybrid systems [16]. In contrast to pre-scripted solutions as, for example, finite state automatae, automated planning enables a flexible solution that can easily be adapted to changing task specifications. It also allows to be used in dynamic systems with the need to react intelligently to unforeseen situations. This makes planning attractive to researchers and application developers from other areas.

The most common description language for planning tasks is the Planning Domain Definition Language (PDDL), which is mostly suited to describe planning problems on an abstract symbolic level. This is often not sufficient for the challenges imposed by real-world planning applications. Subproblems that, for instance, involve geometric computations like object manipulation or navigation cannot be modeled with PDDL, and are thereby beyond the scope of symbolic planners. A way to solve complex real-world planning problems is to decompose it into subtasks, and use a hierarchical combination where specialized planners refine the high-level symbolic plan. The assumption that the description given to the symbolic planning system is on an abstraction level that permits a successful execution of any generated plan is often not true, though. Instead of such a top-down approach, hierarchical composition can also be achieved in a bottom-up manner, where all information possibly relevant to the symbolic planner is precomputed by the lower level reasoners. This, however, is usually too costly for practical application.

Alternatively, Dornhege et al. introduced an approach that integrates high- and low-level planning more tightly: Semantic attachments compute the semantics of Boolean fluents by an external process during planning [4]. They are realized as modules that implement a generic interface in a user-provided library. PDDL/M is the slightly modified version of PDDL that allows to attach such a module to a Boolean fluent. TFD/M is a version of the Temporal Fast Downward (TFD) planning system [6] which implements a domain-independent interface that calls a module to compute the semantics of the Boolean fluent it is attached to. In the implementation of a module it is usually necessary to access the current planning state. The generic nature of the interface imposes restrictions on its usability: The exchange of state information between the planning system and an external module is based on manual requests that are inefficient and error-prone to implement.

In this paper we propose a solution that improves both efficiency and usability of module interfaces. We introduce the Object-oriented Planning Language (OPL), which incorporates structure and advantages of modern object-oriented programming languages like Java or C++. We use this structure to automatically generate domain-specific module interfaces based on the definitions in the planning domain. We prepare class definitions for module developers along the lines of the OPL description, and objects as instances of these class definitions reflecting the current internal planning state. This provides module developers with type-safe and efficient access to the internal state of the planning system, and is compatible with the generic module interface of TFD/M. Figure 1 illustrates the integration of the automatically derived interface between the planner's generic interface and a user-defined implementation.

Furthermore, we aim to create a system that lowers the learning curve for application developers. We believe an object-oriented syntax to be a step towards this goal. In contrast to PDDL, OPL is centered around object definitions, which contain member fluents and actions and allow a natural domain representation. However, it is imperative that OPL can be used with state of the art planning systems to be valuable. For this reason, we show the translation of OPL tasks to PDDL and provide the necessary tools, which allow easy integration of OPL into any planning system capable of parsing PDDL.

2 RELATED WORK

Moved by considerable interest in planning solutions for real-world problems, several steps were taken towards more realistic domain descriptions: PDDL 2.1 was introduced in 2003 [7], extending the basic STRIPS formulation by the possibility to express temporal and nu-

[1] University of Freiburg, Germany, email: {hertle, dornhege, tkeller, nebel}@informatik.uni-freiburg.de

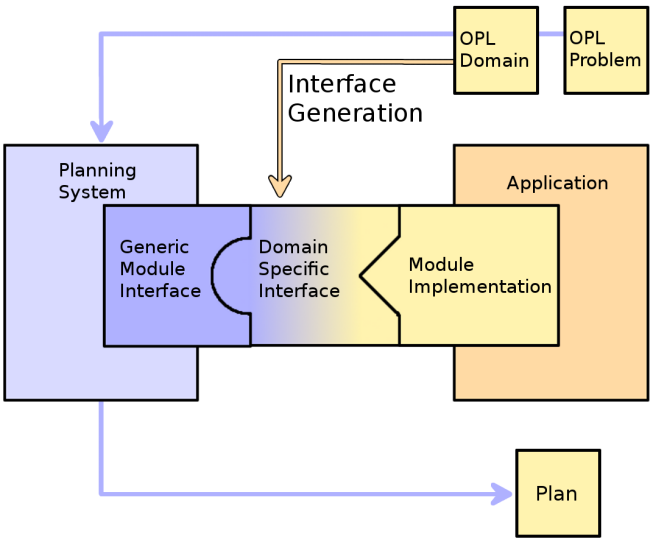

Figure 1. This figure gives an overview of how OPL task descriptions including semantic attachments are integrated with a planning system. OPL domain and problem are converted to PDDL and a domain-specific interface is automatically generated. The interface works as an adapter between the generic module interface in the planner and the module implementation.

meric features; SAS+ [1] allows the use of variables in finite-domain representation (FDR); and the functional STRIPS formulation even permits functional expressions to refer to objects [9], a mechanism finally adopted in PDDL 3.1 as *object fluents*. Independent from PDDL, some planning systems model their internal state with variables in FDR, which are achieved by performing an invariant synthesis [11]. We consider this process, which led to the introduction of object-fluents and variables in FDR, as first steps towards an object-oriented language. While the last step has previously been taken by Vaquero et al. [19] and Simpson et al. [18], our approach combines object-orientation with semantic attachments and draws advantages of the overlap these topics entail.

Our approach covers a wide area of possible applications, most notably scenarios demanding a model of the planning task that is abstract enough to be represented in a language that domain-independent planning systems understand and can solve efficiently, but at the same time not abstracting away necessary constraints that allow a plan to be executed in the real world. A common field are space applications as, for example, the CASPER project [3, 5], or autonomous robots, which have become more and more capable of fulfilling complex tasks in real-world scenarios like mobile manipulation [10, 15, 20] or multi-robot coordination [21]. An area that is often mentioned in the literature dealing with this problem is the *integration of task and motion planning*.

Earlier approaches link symbolic representations and geometric representations directly in a specialized planner. The aSyMov system [2], for example, additionally uses Metric-FF [13] to compute heuristics on a symbolic abstraction of the task. The approach by Kaelbling and Lozano-Perez [15] computes new ground atoms during the planning process by suggesters and verifies collision-free robot paths by external procedures. They also follow a hierarchical planning approach that plans "in the now", meaning that they execute actions immediately when a plan prefix has reached the deepest level of the hierarchy and becomes executable. This allows them to plan abstractly for later tasks in the plan and thus they can reach deep planning horizons.

Dornhege at al. introduce semantic attachments that provide a generic interface to integrate external reasoning procedures as modules in domain-independent planning systems [4]. They implement this concept in Hoffmann and Nebel's Fast Forward planner [14] and in TFD of Eyerich et al. [6], which performs forward-chained search in timestamped state-space using the context-enhanced additive heuristic. The work by Wurm et al. [21] adds cost modules that also allow to compute an action's cost or duration by an external procedure. In this paper we use TFD/M's generic domain-independent interface to implement modules based on OPL.

3 THE OBJECT-ORIENTED PLANNING LANGUAGE

In PDDL, type definitions are used to restrict the usage of objects in parameter lists, thereby restricting the grounding of predicates and operators. OPL places far greater importance on the specification of custom types than PDDL. In addition to global fluents and actions, an OPL task contains types with member fluents and actions, similar to object-oriented programming languages' class declarations. OPL task definitions are separated into a domain and a problem description in the same way PDDL tasks are defined. Like PDDL, types can inherit from other types, but will also inherit the super types' member fluents and actions.

OPL uses single inheritance. If no base type is specified, it defaults to `Object`, a built-in type with no members. The syntax appears similar to such languages as Java or C++ and names are case-sensitive, which simplifies the generation of module interfaces for case-sensitive programming languages. Consider this excerpt from the OPL description for the ROOMSCANNING domain:

```
Domain RoomScanning {
  Type Pose {number x; number y; number th;}
  Type ScanTarget : Pose {boolean scanned;}
  Type Door {Pose approachPose;
             boolean open;}
  ... }
```

A type *Pose* is defined that contains numerical member fluents describing a pose in 2D space. The subtype *ScanTarget* inherits this pose and defines an additional Boolean fluent *scanned*.

The dot `.` acts as the structure access operator in OPL: a member fluent of a type can be referenced by obtaining an object (or object fluent) of that type and then applying the `.` operator with the name of the fluent. The result of the expression will have the type of the referenced fluent. It is possible to chain such operations, if the referenced fluent is an object fluent. The keyword *this* is used to address the current object in member actions.

Like PDDL actions, OPL actions have a name and parameters and define a (pre-)condition and an effect. For temporal planning, durative actions also need a duration. The condition and effect statements contain nested logical formulas that are semantically identical to their PDDL counterparts with a slightly changed syntax. Prefix-notation is used with a function-style syntax consisting of a name and a comma separated parameter list enclosed in parentheses. The *equals* keyword compares two numerical or object fluents.

In contrast to PDDL/M, semantic attachments are easily integrated into OPL domains by simply declaring the type of semantic attachment, its name and parameters. There is no need to specify an explicit

library call for PDDL/M as that will be generated automatically with the domain-specific interface. They can be used as any other fluent.

The ROOMSCANNING domain from above also defines a type *Robot* that has the robot's *currentPose* as a member fluent. The *Robot* type has a *drive* action with one parameter *dest* that moves the robot from its current pose to a destination. Additionally a cost module *driveCost* with one parameter *dest* is defined that is used as the operator's cost instead of a numerical fluent. The action also uses a condition module defined at global scope: *pathExists* will check if there is a path *from* one pose *to* a destination.

```
Domain RoomScanning {
 ...
 ConditionModule pathExists(
     Pose from, Pose to);

 Type Robot {
  Pose currentPose;
  CostModule driveCost(Pose dest);
  Action drive(Pose dest) {
   Cost {driveCost(dest);}
   Condition {
    and(not(equals(this.currentPose, dest)),
        pathExists(this.currentPose, dest));}
   Effect {assign(this.currentPose, dest);}}
```

Like PDDL problem files, OPL problems define the initial state and goal. This example defines the problem *Scenario1* for the ROOMSCANNING domain.

```
Problem Scenario1(RoomScanning) {
  Pose p1;
  Pose p2 { x = 5; y = 1; th = 0.5; }
  Target t1 { x = 0; y = -13; th = 0.5; }
  Robot r1 { currentPose = p1; }
  Goal { and(equals(r1.currentPose, p2),
      t1.explored); } }
```

Type instantiation and initialization is combined. The goal is simply stated as a formula like in PDDL. For a more precise definition of OPL we refer to the work by Andreas Hertle [12].

4 TRANSLATION TO PDDL

The previous section illustrated the syntax and elements of an OPL planning task. OPL is intended to be used by PDDL planning systems. Therefore we show how an OPL task can be converted to a PDDL planning task here. In fact, the semantics of OPL is given by a translation to PDDL – so this translation specifies the meaning and is at the same time the way to enable planning in OPL by employing a PDDL planner.

We need to convert member fluents and actions to PDDL fluents and actions. Member fluents and actions are specific to an object instance. When converting from OPL to PDDL this is resolved by adding an additional parameter named *?this* as the first parameter to each member fluent and action representing the object it belongs to. Additionally the name is prefixed by the type for uniqueness. The OPL description of the ROOMSCANNING domain will be translated to the following PDDL predicates and functions:

```
(:types
  Door Pose - object
  ScanTarget - Pose)
(:predicates
  (ScanTarget_scanned ?this - ScanTarget)
  (Door_open ?this - Door) ...)
(:functions
  (Pose_x ?this - Pose) - number
  (Pose_y ?this - Pose) - number
  (Pose_th ?this - Pose) - number
  (Door_approachPose ?this - Door) - Pose
  ...)
```

Finally, formulas in conditions and effects are replaced by their PDDL analogues in prefix-notation. When using only global fluents the translation is straight-forward. However, as we allow to chain expressions over member fluents using the **.**-operator to access structure elements, we need to translate such expressions to PDDL. Chained expressions using structure element access are translated recursively. Variables and ground names are translated by their identity. When referring to the member fluent of an object *?o* with the name *m*, we translate this to `(m ?o <parameters>)`, where `<parameters>` is the recursive translation of the fluent's parameters. Note that such a chained expression might contain object fluents as parameters. See for example the `openDoor` action from the ROOMSCANNING domain. A global predicate `inRange` defines if two objects of type `Pose` are directly reachable for the robot. The `openDoor` action does not have any `Pose` parameter. Instead the condition refers to the member object fluents of type `Pose` from the robot (*this*) and the *door*.

```
boolean inRange(Pose p1, Pose p2);
Type Robot {
 Action openDoor(Door door) {
 Condition {and(
  inRange(this.currentPose,door.approachPose),
  not(door.open));}
 Effect {door.open; } } }
```

The OPL action `openDoor` leads to the following PDDL action:

```
(:predicates
  (inRange  ?p1 - Pose  ?p2 - Pose))
(:action Robot_openDoor
  :parameters
    (?this - Robot ?door - Door)
 :condition (and
   (inRange (Robot_currentPose ?this)
            (Door_approachPose ?door))
   (not (Door_open ?door)))
 :effect (Door_open ?door)
```

5 AUTOMATIC GENERATION OF THE DOMAIN-SPECIFIC MODULE INTERFACE

The implementation of semantic attachments in TFD/M provides a generic domain-independent module interface. As a result of its generic nature, the module interface is inefficient, and cumbersome and error-prone to implement for a module developer. Information is exchanged via object names and the current planner state is accessed via callback functions. Requests are manually created based on the fluent name and its parameters, which have to be supplied by the module implementation. OPL provides a solution to this problem by automatically generating a domain-specific interface from an OPL domain description that acts as an adapter between TFD's generic

interface and a domain-specific module implementation. Figure 1 illustrates this concept.

The generated interface provides a safe and efficient way to implement modules. OPL objects are represented as classes, i.e. objects in an object-oriented programming language. For TFD/M, this is realized in C++. Module calls receive object instances as parameters rather than object names. The generated classes provide member functions for each member fluent to access the planner state. Therefore, performing an error-prone, work-intensive manual callback based on object and fluent names as part of the actual module implementation is not necessary anymore. The name mappings and the state access are efficiently encapsulated in the automatically generated interface.

To demonstrate the generation process, consider the ROOMSCANNING domain from Section 3, where the cost module *driveCost* is defined in the *Robot* type. The following call stub is automatically generated:

```
double Robot_driveCost(
  const State* currentState,
  const Robot* thisRobot,
  const Pose* dest);
```

The first argument is the *State* object that allows access to global fluents and also contains lists that allow to access all objects of each type. If the module is a member of an OPL type, the next argument is an object of the corresponding type class. Then follow the arguments of the module as specified in the domain. Condition modules return Boolean values and cost modules return floating point numbers.

For each of the types in the OPL domain a corresponding class is created. If a type extends a base type in the domain, the generated class is derived from the base type's generated class. Otherwise it is derived from *OPLObject*, a generic base class. Each member fluent of the OPL type will generate a member function with the same name that retrieves the fluent's value in the current state. Boolean fluents lead to *bool* return values and numerical fluents return *float* values. Object fluents return a pointer to the class that was generated for the corresponding OPL type.

As an example consider the OPL type `Door` from Section 4 that contains an object fluent approachPose of type `Pose` and a Boolean fluent open. This leads to the following class declaration in the domain-specific interface:

```
class Door : public OPLObject {
  public:
    const Pose* approachPose() const;
    bool open() const; };
```

We will now describe how implementations for such member functions acquire a fluent's value from the planner's internal state. The process requires a planner specific implementation of a *StateMapping* class. Such a state mapping represents the value of one ground fluent and is able to acquire the current value from a pointer to the planner's internal state,

First, two mapping tables are created during initialization: Fluent mappings map from each ground fluent name to a *StateMapping* object and have to be provided by the planner as it depends on the encoding of the planner's internal state. For Boolean and numerical fluents they are used to acquire values directly. For object fluents, object mappings are created when *OPLObjects* are instantiated. They map from a *StateMapping* to the *OPLObject* instance with the same name as the object the *StateMapping's* object fluent value points to. When the current state changes between subsequent module calls, the state pointer is updated without requiring additional copy operations.

In TFD/M, where a state s is a single vector of floating point numbers representing a task in FDR, the *StateMapping* implementation stores a tuple (var, val). When a Boolean fluent is requested $s[var]$ is compared to val. For a numerical fluent $s[var]$ is returned.

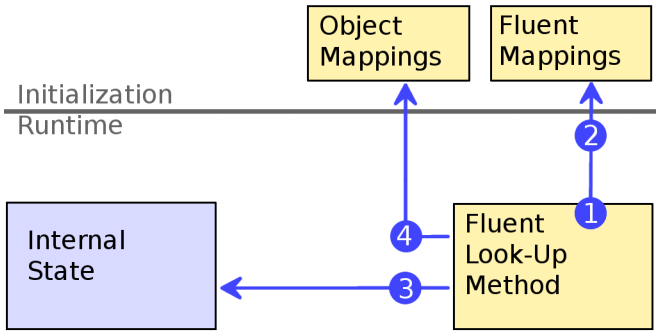

Figure 2. This figure shows the state look-up process. A ground name is constructed (1) that is used to retrieve a fluent mapping (2). The fluent mapping determines its value from the internal state (3). For object fluents the object mappings map this value to an *OPLObject* (4).

Using these mapping tables the methods for accessing member and global fluents can be generated. The required steps are illustrated in Figure 2.

1. A key for querying the fluent mapping table is composed. The fluent name is appended with the names of the parameters forming the ground name of the fluent. Member fluents insert the object's name as the first parameter.
2. Next, the corresponding *StateMapping* is retrieved from the fluent mapping table.
3. Now, the fluent value is retrieved from the internal state using the *StateMapping* object. For Boolean and numerical fluents their value is returned.
4. In case of object fluents, the *StateMapping* requests the planner state in the same way, but is handed to the object mapping table to retrieve an *OPLObject* pointer that is cast to the specific return type.

If a member fluent does not have any parameters (besides the implicit *this*), the *StateMapping* does not change for this fluent. Therefore we can move step 1 and 2 to the initialization phase and store the retrieved *StateMapping* in the object instance. During planning these steps are then skipped and thus no look-ups are performed for accessing Boolean and numerical fluents, and only the object mapping look-up is performed for object fluents.

By using this automatically generated interface module developers now directly deal with objects instead of performing manual requests for fluent values that are work-intensive and error-prone. This functionality is now performed by the interface. Additionally, in some cases, we can move look-ups to the initialization phase leading to a more performant implementation.

6 EXPERIMENTAL RESULTS

We conducted three experiments using OPL domains including semantic attachments. The performance improvement of OPL can only

be measured in the presence of semantic attachments as otherwise a translated OPL task is a PDDL task and thus behaves the same. All experiments are conducted with the TFD/M planner developed by Dornhege et al. [4]. We denote results achieved with the original generic module interface as TFD/M, and the combination of TFD/M with our automatically generated domain-specific module interface as OPL.

The first experiment shows the computational overhead of TFD/M and OPL compared to a base line planner not using semantic attachments at all. We show representative results from the CREWPLANNING domain in the temporal-satisficing track of the International Planning Competition (IPC) 2008. The predicate *available* occurs in most conditions of the domain. We add a condition module to the domain that is additionally called whenever *available* is used. The module simply acquires the predicate from the planner state either with the TFD/M generic module interface or via OPL's domain-specific interface and returns its truth value. Thus the applicability of the operator is not changed and no additional computations are performed. To ensure comparative results independent of the actual search procedure, we measured runtimes of testing operator applicability for each planner on an identical set of states that was derived from the closed list of the base line planner.

From the 30 problems of IPC 2008 we pick the last 20 problems as the size of the closed list generated for the first ten problems is too small to yield discriminative results. We conduct 150 test runs for each problem and compute the overhead of TFD/M and OPL compared to the base line. The results in Table 1 show that the runtime scales well for both approaches, and is lower for OPL in all problems instances.

#	Base [s]	TFD/M [s]	[%]	OPL [s]	[%]
11	0.22	0.23	7.09	0.22	2.20
12	0.24	0.26	7.19	0.25	2.51
13	0.02	0.02	10.06	0.02	3.25
14	0.03	0.04	10.13	0.03	1.79
15	0.02	0.02	10.78	0.02	2.85
16	0.20	0.22	5.56	0.21	1.47
17	0.24	0.26	5.76	0.25	2.60
18	0.34	0.35	3.19	0.34	1.11
19	0.68	0.71	3.89	0.69	1.61
20	0.86	0.89	3.78	0.87	1.54
21	1.00	1.04	4.64	1.02	2.37
22	0.05	0.06	10.79	0.06	3.28
23	0.04	0.05	14.40	0.04	4.09
24	0.06	0.07	10.61	0.06	3.47
25	0.56	0.59	5.09	0.57	2.08
26	0.56	0.58	4.69	0.57	1.75
27	0.61	0.63	4.09	0.62	2.08
28	1.83	1.87	2.03	1.84	0.66
29	2.04	2.11	3.05	2.08	1.71
30	2.11	2.19	3.94	2.14	1.66

Table 1. This table shows the results of the CREWPLANNING experiment. Runtimes in seconds are averaged over 150 trials. The computational overhead in percent is given in comparison to the base line without modules.

The second experiment is based on the TRANSPORT-NUMERIC domain from the satisficing track of IPC 2008 and is designed to show the impact of acquiring fluent values from the planner state. The domain models a logistics task where packages are transported by trucks. We replace the simple volume based condition check to see if another package can be loaded by a semantic attachment that checks if the package fits geometrically into the truck given the current cargo. The module requests the sizes of all packages to be stored in the vehicle and the vehicle's capacity from the internal planner state. We use the TFD/M module implementation as described by Dornhege et al. [4] and created an equivalent OPL domain together with an OPL module that implements the same algorithm.

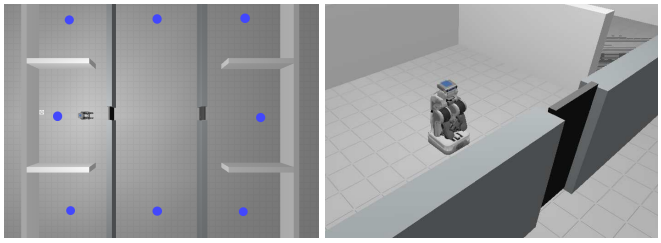

Figure 3. The figure illustrates the ROOMSCANNING domain. The robot needs to explore a building and might need to open doors to reach different rooms. The left side shows an overview of such a task, the right side an example scene from executing a plan in the simulation environment.

#	L	V	P	TFD/M [s]	OPL [s]	Relative [%]
1	5	2	2	0.01	0.00	-61.6
2	10	2	4	0.12	0.04	-67.7
3	15	3	6	0.52	1.02	94.0
4	20	3	8	1.31	2.10	60.1
5	25	3	10	3.36	2.89	-13.9
6	30	4	12	87.53	17.92	-79.5
7	35	4	14	140.87	68.34	-51.5
8	45	4	18	54.42	29.17	-46.4

Table 2. This table shows the results of the TRANSPORT experiment. L, V, P list the number of locations, vehicles and packages for each problem. The last column gives the relative runtime of OPL compared to TFD/M.

We created eight problems with increasing complexity and compare the TFD/M module with the OPL formulation. For each of the eight problems 100 trials were conducted and we give the average runtime. The standard deviation for all trials was lower than two percent of the average runtime. The results in Table 2 show that the runtime using the OPL module interface is significantly lower in most cases than the original TFD/M interface's. This is especially visible in the more complex tasks (6 – 8) as in these cases the time for initialization is negligible. As problems three and four show, the initialization time might dominate the positive runtime effects for simple tasks.

The main objective of the third experiment is twofold. On the one hand, it demonstrates the use of OPL in a real-world environment including the application in a realistic robotics scenario. On the other hand, it shows how a complex system is built easily by integrating an existing path planner into TFD/M using the OPL planner interface.

The ROOMSCANNING domain that was used as an example in this paper models an autonomous robot searching for items in various rooms (see Figure 3). The environment is an office space with multiple rooms, corridors and doors. Some of the doors might be closed and the robot has the ability to open doors. The robot has a metric map of the environment and knows the coordinates for good scan locations in every room. The goal is to scan all target locations in a

minimal amount of time.

The geometry of the world restricts the robot's path to reach another location. We implemented a module to compute the cost for the `drive` action. The module implementation calculates the real path cost by calling the external path planner used by the navigation component of the Robot Operating System (ROS) [17]. For the experiment we created eight problems with varying complexity starting from two scan targets without closed doors in the first task, up to eight scan targets and two closed doors in the last.

#	Targets	Doors	Search time [s]	Total time [s]
1	2	0	0.09	7.69
2	3	1	0.13	11.92
3	4	1	1.05	17.23
4	5	1	1.76	23.54
5	5	2	2.37	24.44
6	6	2	4.64	33.18
7	7	2	6.17	41.90
8	8	2	11.79	55.24

Table 3. This table shows the results of the ROOMSCANNING experiment. Total time lists the time until the first valid plan was found. Search time excludes the time spent waiting for module computations.

As can be seen in Table 3, the time required to find the first valid plan increases with problem complexity. We show the total time, i.e., the time the planner needed to come up with a plan, and the search time, i.e., the total time without the runtime of the path planner, separately. The observed runtimes are still acceptable for use in a real-world robotics system and could be improved by using a more efficient path planner.

7 CONCLUSION

We presented the Object-oriented Planning Language OPL, a novel description language for planning tasks, which incorporates structure and advantages of modern object-oriented programming languages like Java or C++, allowing the design of real-world scenarios in a natural way. We furthermore support semantic attachments, a concept that was introduced to integrate external procedures to determine a fluent's semantics, and use the object-oriented structure to generate a more efficient domain-specific interface that acts as an adapter between the generic interface of a planning system and a domain-specific module implementation. The automatic generation of the domain-specific interface produces a convenient to use and type-safe implementation skeleton for external modules.

We also show how to translate OPL to PDDL, allowing any state of the art planner based on PDDL to solve OPL tasks if combined with the tools that are described in this paper. We adapted the PDDL/M interface of TFD/M to also support OPL modules and compare them experimentally to PDDL/M modules. Our evaluation shows that the automatically generated interface is more efficient than the previous implementation of semantic attachments in TFD/M due to the improved look-up process when accessing the planner state.

ACKNOWLEDGEMENTS

This work was partially supported by Deutsche Forschungsgemeinschaft (DFG) in the PACMAN project within the HYBRIS research group (NE 623/13-1) and in the Transregional Collaborative Research Center *SFB/TR8 Spatial Cognition* project R7-[PlanSpace], as well as by the German Aerospace Center (DLR) as part of the Kontiplan project (50 RA 1010).

REFERENCES

[1] C. Bäckström and B. Nebel, 'Complexity results for SAS+ planning', *Computational Intelligence*, **11**(4), 625–655, (1995).
[2] S. Cambon, F. Gravot, and R. Alami, 'A robot task planer that merges symbolic and geometric reasoning.', in *European Conference on Artificial Intelligence (ECAI)*, pp. 895–899, (2004).
[3] S. Chien, R. Knight, A. Stechert, R. Sherwood, and G. Rabideau, 'Integrated planning and execution for autonomous spacecraft', in *IEEE Aerospace Conference (IAC)*, (1999).
[4] C. Dornhege, P. Eyerich, T. Keller, S. Trüg, M. Brenner, and B. Nebel, 'Semantic attachments for domain-independent planning systems', in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 114–121. AAAI Press, (September 2009).
[5] T. Estlin, D. Gaines, F. Fisher, and R. Castano, 'Coordinating multiple rovers with interdependent science objectives', *Autonomous Agents and Multi-Agent Systems Conference (AAMAS)*, (July 2005).
[6] P. Eyerich, R. Mattmüller, and G. Röger, 'Using the context-enhanced additive heuristic for temporal and numeric planning', in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pp. 130–137. AAAI Press, (September 2009).
[7] M. Fox and D. Long, 'An Extension to PDDL for Expressing Temporal Planning Domains', *Journal of Artificial Intelligence Research*, **20**, 61–124, (2003).
[8] M. Fox, D. Long, and D. Magazzeni, 'Automatic construction of efficient multiple battery usage policies', in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, (2011).
[9] H. Geffner, *Functional Strips: a more flexible language for planning and problem solving*, 188–209, Kluwer, 2000.
[10] K. Hauser and J.C. Latombe, 'Integrating task and PRM motion planning: Dealing with many infeasible motion planning queries', in *ICAPS Workshop on Bridging the Gap between Task and Motion Planning*, (2009).
[11] M. Helmert, 'Concise finite-domain representations for PDDL planning tasks', *Artificial Intelligence*, **173**, 505–535, (2009).
[12] A. Hertle, *Design and Implementation of an Object-Oriented Planning Language*, Master's thesis, University of Freiburg, 2011.
[13] J. Hoffmann, 'Extending FF to numerical state variables', in *European Conference on Artificial Intelligence (ECAI)*, (2002).
[14] J. Hoffmann and B. Nebel, 'The FF planning system: Fast plan generation through heuristic search', *Journal of Artificial Intelligence Research*, **14**, 253–302, (2001).
[15] L.P. Kaelbling and T. Lozano-Perez, 'Hierarchical planning in the now', in *IEEE Conference on Robotics and Automation (ICRA)*, (7 May 2011).
[16] J. Löhr, P. Eyerich, T. Keller, and B. Nebel, 'A planning based framework for controlling hybrid systems', in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, (2012). To Appear.
[17] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, 'ROS: an open-source robot operating system', in *ICRA Workshop on Open Source Software*, (2009).
[18] R. M. Simpson, D. E. Kitchin, and T. L. McCluskey, 'Planning domain definition using GIPO', in *The Knowledge Engineering Review*, volume 22, pp. 117–134, (2007).
[19] Tiago Stegun Vaquero, Victor Romero, Flavio Tonidandel, and Jose Reinaldo Silva, 'itSIMPLE2.0: An integrated tool for designing planning domains', in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, (2007).
[20] J. Wolfe, B. Marthi, and S. J. Russell, 'Combined task and motion planning for mobile manipulation', in *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, (2010).
[21] Kai M. Wurm, Christian Dornhege, Patrick Eyerich, Cyrill Stachniss, Bernhard Nebel, and Wolfram Burgard, 'Coordinated exploration with marsupial teams of robots using temporal symbolic planning', in *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, (October 2010).