Semantic Attachments for Domain-Independent Planning Systems

Christian Dornhege and **Patrick Eyerich** and **Thomas Keller**
and **Sebastian Trüg** and **Michael Brenner** and **Bernhard Nebel**
University of Freiburg, Germany
Institut für Informatik
Georges-Köhler-Allee 52
79110 Freiburg, Germany
{dornhege, eyerich, tkeller, trueg, nebel}@informatik.uni-freiburg.de

Abstract

Solving real-world problems using symbolic planning often requires a simplified formulation of the original problem, since certain subproblems cannot be represented at all or only in a way leading to inefficiency. For example, manipulation planning may appear as a subproblem in a robotic planning context or a packing problem can be part of a logistics task. In this paper we propose an extension of PDDL for specifying *semantic attachments*. This allows the evaluation of grounded predicates as well as the change of fluents by externally specified functions. Furthermore, we describe a general schema of integrating semantic attachments into a forward-chaining planner and report on our experience of adding this extension to the planners FF and Temporal Fast Downward. Finally, we present some preliminary experiments using semantic attachments.

Introduction

Real-world planning problems often require several subproblems to be solved. For example, in a robotic context it is usually necessary to plan robot movements and the manipulation of objects. Furthermore, the high-level tasks the robot is supposed to perform, e.g., fetching a book from the library, must also be planned for. While the latter problem can be addressed using traditional symbolic planning approaches, navigation and path planning is beyond the scope of symbolic planners. In fact, specialized planners are available for these problems.

It makes, of course, a lot of sense to decompose a complex real-world planning problem into different simpler subtasks. However, the planners have to be combined in the right way. The usual method here is a *hierarchical combination*. On the highest level, the symbolic planner creates a symbolic plan. Then the actions are refined using the low-level planners, e.g., the path planner and the manipulation planner. The assumption here is that the symbolic description is on an abstraction level that permits a successful execution of any generated plan. However, very often this is not true. In such cases, the early commitment of the symbolic planner may lead to failures on the lower levels.

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, may be very costly if there are too many such facts. For example, the precomputation of all trajectories between all pairs of poses of a gripper at possible locations for all possible configurations of objects is too time and memory consuming. Furthermore, most of the generated information will turn out to be irrelevant to the task at hand.

Therefore, in this paper, we propose a third approach that integrates high and low-level planning more tightly and in which a low-level reasoner can provide information to the high-level planner *during* the planning process, but is only evoked if relevant to the high-level planner. Contrary to the hierarchical decomposition and combination, a particular choice on the symbolic level can lead the low-level planner to detecting failure and requesting to backtrack immediately.

To integrate information about special-purpose reasoning into symbolic planning we propose to use what we call *semantic attachments*[1] to a planning domain description. Some of the predicate symbols of the domain description can have such a semantic attachment, meaning that the truth values for corresponding atomic ground formulas are specified by an *external mechanism*. Similarly, there exist semantic attachments for effects on numerical fluents which are determined by an external mechanism as well.

Semantic attachments can easily be added to a planning language like, in our case, PDDL. Based on that, we describe a general framework for integrating these extension into forward-chaining state-space planners, which are particularly suited to our task since they search over complete world states. External modules can then access those states in order to compute conditions and effects for their special-purpose behaviors.

While similar mechanisms have been used before, in particular in domain-specific contexts (Konolige and Nilsson 1980; Orkin 2006), our work appears to be the first that extends PDDL rendering this feature available for domain-independent planners.

The rest of the paper is structured as follows. In the next

[1] *Semantic attachment* is a term coined by Weyhrauch (1980) to describe the attachment of an interpretation to a predicate symbol using an external procedure.

section, we describe a number of motivating examples. Then we specify an extension of PDDL and examine soundness and completeness of a planner *relative to semantic attachments*. Based on that, we describe our implementations of semantic attachments in the planning systems *FF* and *TFD*. Our experience with using semantic attachments from a performance point of view is reported in the experimental section. Finally, we comment on related work and close with a conclusion and outlook.

Motivating Examples

For many real-world problems, it is hard to find an abstraction suitable for symbolic planning which guarantees that for every symbolic plan an executable concretized plan will exist. In this paper, we consider two such problems, namely a logistics domain with complex truck packing problems and a robot manipulation domain with fairly realistic grasp modeling.

Transport Domain

The *logistics* domain has been a standard benchmark for several years at the International Planning Competition. It models a common logistics problem, where trucks deliver packages to different locations. In the original formulation, each truck can pick up only one package. With the introduction of numeric fluents, it became possible to model truck capacities and package sizes in the *transport-numeric* domain, allowing trucks to load multiple packages.

Although more realistic than not representing capacities at all, summing up volumes is obviously not sufficient for checking whether a set of packages can be loaded into a truck, since the package geometries are not considered. For example, Figure 1 shows that it is impossible to pack two equally sized cubes into a cube with double the volume. Moreover, it demonstrates that the volume approximation is not even close to reality.

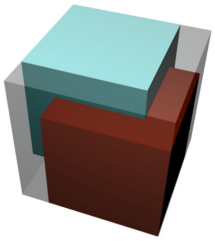


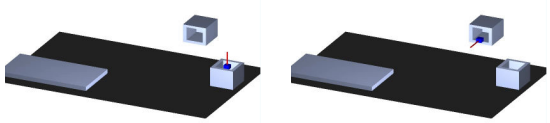
Figure 1: The two smaller cubes have half the volume of the outer one, but they obviously do not fit together into the outer cube.

Clearly, it is beyond the capabilities of a symbolic planner to solve the three-dimensional packing problem. However, there exist specialized algorithms for solving this NP-hard problem exactly or approximately. Such a reasoner could be integrated into the planner by attaching it semantically to the precondition of an action.

Robot Manipulation Domain

Similar to the *logistics* domain, the *blocks-world* domain has been a benchmark in the planning area for a long time. It is a highly abstract version of a robot manipulation problem. Nowadays such large tasks are easily solved by symbolic planners. Unfortunately, however, the domain is so abstract that it has hardly anything to do with reality. For example, gripper poses or potential collisions of the gripper with other objects do not play a role at all.

A slightly more concrete domain is depicted in Figure 2. Here we have a box which is open at the top, a shelf, a table, and a little moveable box. The gripper is simply a stick that can connect to moveable objects from any direction. Thus, depending on the continuous grasp direction, collisions can occur. In general, we want to manipulate objects, i.e., grasp them, transport them, and put them down. In particular, we want to plan for the situation that we have to grasp an object that is in the box and to place it in the shelf. This would require us to place the object on the table in order to grasp it from the side to avoid a collision of the gripper with the shelf when putting down the moveable object.



(a) (b)

Figure 2: Visualization of the manipulation domain with (a) an initial state and (b) a final state.

Again, solving such a task using only a symbolic planner is clearly impossible. Here we need a manipulation planner as a sub-component of the symbolic planner. Such an embedded planner could check the preconditions of whether a grasp or place action is possible. Furthermore, it also needs to change the internal model of the environment so that future possible collisions can be detected.

Semantic Attachments

Semantic attachments are external procedural reasoning modules (in the following just called *modules*) that may compute the valuations of state variables at planner run-time. The symbolic planner itself is mostly unaffected by this extension: instead of looking up values in a table or updating them through state transitions as usual in Strips-like languages, a function call provides the necessary information. Under the hood of the module, though, complex computations can be performed that transcend the capabilities of the planner.

In order to integrate semantic attachments into a planner we propose the architecture shown in Figure 3. Semantic attachments consist of a *declarative part* that describes their use in the planning domain, i.e., their symbolic use in preconditions and effects of planning operators. Additionally, they have a *procedural part* which is the actual algorithm for computing the value of a state variable. This part is directly included into the planner as a shared library and may

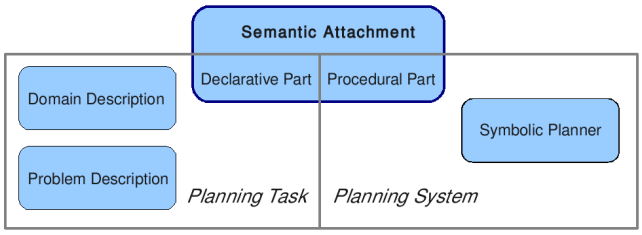


Figure 3: Extending planning tasks by modules to planning tasks with semantic attachments

access the planning state through callback functions. The next section gives more details about the implementation.

We propose two kinds of semantic attachments that can be part of operators: *Condition checker* modules test whether some complex operator precondition is satisfied. *Effect applicator* modules compute changes to any number of state variables. When speaking of the declarative part of these modules, i.e., their use as preconditions and effects of planning operators, we will speak of *module conditions* in the case of condition checker modules and *module effects* in the case of effect applicator modules.

PDDL/M In order to actually use semantic attachments in classical planning, it is necessary to extend the description language for planning tasks.

Therefore, we propose the introduction of semantic attachments to the PDDL-standard leading to PDDL/M, which is described in the following. External modules seem to be most relevant when complex *numeric* computations need to be performed during the planning process. Therefore we based our extension on the PDDL 2.1 version of the language that introduced numeric fluents (Fox and Long 2003). We call the extended language PDDL/M and add a new PDDL requirement `:modules` to indicate that a planning domain uses semantic attachments.

A PDDL/M domain may contain an additional section that declares the modules similar to the way predicates are declared in PDDL. In this section, each semantic attachment has its own entry, a *condition-checker* module consisting of three, and an *effect-applicator* module of four mandatory parts: Both start with a unique identifier to reference the module including a possibly empty list of parameters, similar to a function or predicate entry in their respective sections. Only for *effect-applicator* modules we then list any number of numerical fluents that are set by the module. Both cases then declare the type of module and finally the function and library name where the module can be found by the planning system.

For example, the *condition-checker* module used in the *transport-modules* domain is declared as follows:

```
(:modules
   (canLoad ?v - vehicle ?p - package
    conditionchecker canLoad@libTrans.so) )
```

The module is called *canLoad*, it decides whether it is possible for vehicle $?v$ to load package $?p$, and can be found in the shared library `libTrans.so` by calling the function `canLoad`.

The syntax of *effect-applicator* modules is similar, as can be seen in the following from our robot manipulation domain:

```
(:modules
  (putDown ?o - movable ?p - base ?g - grasp
    (q0) (q1) (q2) (q3) (q4) (q5) (q6)
    (x ?o) (y ?o) (z ?o)
    (yaw ?o) (pitch ?o) (roll ?o)
    effect putDown@libTrajectory.so))
```

This module sets the robot arm configuration ($q0 - q6$) and the position and orientation of object $?o$ after putting it down at $?p$ using grasp $?g$. The information is made available to the symbolic planner via the numeric fluents that are listed between the parameters and the module type.

To use a module in an operator, it has to be specified in the same way as predicates or functions. The only difference is that a module is given by enclosing its identifier and parameters in square brackets.

```
(:action put-down
 :params (?o - movable ?p - base ?g - grasp)
 :condition (... ([checkPutDown ?o ?p ?g]))
 :effect (and (on ?o ?p) (handempty)
  (not (holding ?o ?g)) ([putDown ?o ?p ?g])
```

Soundness and completeness One important question when using such semantic attachments is how these affect soundness and completeness of the planner. Since arbitrary code can be used in the modules, it cannot be guaranteed that the planner terminates when calling a *condition-checker* or *effect-applicator*. However, under some reasonable assumptions some form of soundness and completeness can still be guaranteed.

In particular, we require *condition-checkers* to always terminate and return a truth-value. Furthermore, this truth-value should be identical for identical parameter values and world states. In other words, condition checkers are nothing else than a concise representation of derived predicates.

In a similar vain, we require that *effect-applicators* always terminate and result, for identical parameters and states, in identical settings of the fluents they act on. In particular, effect applicators should not contain any mechanism for making *choices* between different outcomes, such as selecting a location for placing an object. This means that we can view effect applicators as a concise representation of a part of the deterministic transition function that is usually completely specified by the PDDL operators.

If these two conditions are met, one can analyze soundness and completeness of a planner extended by a semantic attachment mechanism *relative* to the semantic attachments used in the problem description: Assuming that the condition checkers implement the intended meaning and assuming that the effect applicators implement the intended meaning of the state transitions, are the returned plans correct and will the planner find a solution if there exists one? As we will argue below, the implementation of semantic attachments in the planners we considered guarantees this form of conditional soundness and completeness.

Implementation

A PDDL/M planner must evaluate semantic attachments at runtime, i.e., it must call the external modules and use the computed results in its planning process. The modules, on the other hand, need to access relevant parts of the planning state for their computation.

Technically, modules are implemented as dynamically loaded shared libraries. To be able to successfully load arbitrary modules, the planner and the modules need to use a common interface. In Figure 4 we present the main part of the C++-Interface used by the planner in order to support semantic attachments. Similar interfaces can be designed for other programming languages.

Besides some data types the interface defines two function types for the two kinds of semantic attachments implemented. Both function types require a `ParameterList` holding the operator's grounded parameters and two callback functions giving the module the ability to access the current planning state. There are two types of callbacks, one for logic predicates and one for numeric fluents. Additionally, both *condition-checker* and *effect-applicator* module calls can be invoked with or without a *heuristic* flag, thereby either requesting either an exact result or an approximation. Effect-applicator modules are also passed a reference to a list of numeric values which it is supposed to affect as a result of its computations.

The planning system is responsible for calling those functions with the correct parameters and at the correct time, namely during successor generation and possibly during heuristic state evaluation.

This section aims to show how this can be achieved for standard progression search planners. We present the extension of two planners: FF, a classical planner and Temporal Fast Downward, a numeric temporal planner based on a multi-valued state variable representation.

FF/M

FF (Hoffmann and Nebel 2001) is a planning system based on forward state space search, using a heuristic that works with a relaxed version of the planning problem to provide fast estimates for goal distances. A state in FF's search space is basically a valuation of state variables.

The relaxed planning task differs from the original task in that it ignores the delete lists of all actions. The most important feature of this relaxed planning task is its monotonicity: due to the absence of negative symbolical effects, any condition which is true in state S is also true in every superstate S' ($S \subseteq S'$). This results in a polynomial runtime of the GRAPHPLAN (Blum and Furst 1995) algorithm which FF uses to solve the relaxed planning task.

FF/M is an extension of FF supporting semantic attachments. A condition c of an action with a semantic attachment is satisfied in a state S iff all logical conditions are fulfilled and all condition modules referred to in c return *true*. Analogously, the application of an action with semantic attachments generates a new state by first applying all logical effects and then calling all external modules.

TFD/M

Temporal Fast Downward (TFD) (Eyerich, Mattmüller, and Röger 2009) is a domain-independent progression search planner built on top of the classical planning system *Fast Downward* (Helmert 2006). It extends the original system supporting durative actions as well as numeric and object fluents. One distinguishing feature of both systems is that the input consisting of propositional atoms is automatically translated into an encoding using *multi-valued* variables. This allows for a more concise internal state representation and enables the use of heuristics employing hierarchical dependencies between state variables, altogether resulting in a more efficient search performance.

(Temporal) Fast Downward solves a planning task in three phases: As a first step, the PDDL planning task is *translated* from its Strips-like encoding into a representation similar to SAS+ (Bäckström and Nebel 1995), using finite-domain variables instead of binary predicates. Afterwards, in a *knowledge compilation* step, some data structures utilized by the heuristic and the search component are generated. The most important of these are domain transition graphs for each variable that encode how state variables can change their values, and the causal graph that represents the hierarchical dependencies between different state variables. Finally, a best-first progression *search*, guided by a numeric temporal variant of the context-enhanced additive heuristic (Helmert and Geffner 2008), is performed.

TFD/M is an extension of TFD supporting semantic attachments. Since (in contrast to FF) the internal representation of TFD is significantly different from PDDL, enabling both the planner and the "modules" to access and manipulate the planning states is not trivial. The most significant extensions to TFD occur in the translation and search phases, which we will describe in the following.

Translation In the translation phase of Temporal Fast Downward, the task is converted into a finite-domain representation (FDR). In order to generate an appropriate FDR description from PDDL/M tasks, we adapt the method of Helmert (Helmert 2009). Roughly, this process consists of the following phases: The generation of mutual exclusion (mutex) invariants that describe which propositions may never be true at the same time; a grounding phase in which, by means of a relaxed reachability analysis, a set of propositions (instantiations of predicates) is generated that may potentially be used in the planning process; the generation of a suitable FDR based on the mutex invariants and reachable propositions.

Since module conditions are "black boxes" to the planner, during invariant generation we cannot make any assumptions about their falling into mutex groups, i.e., for each grounded module condition we introduce a distinct FDR variable that must evaluate to true in every condition where the original module condition occurred.

Module effects are "black boxes" for the planner, too; however, they cannot be ignored, since they may affect fluents that are not influenced by any non-module effects. Using the standard grounding procedure, these fluents would erroneously be compiled away as unreachable. To prevent

```
typedef int (*conditionCheckerType)(ParameterList & parameterList,
    predicateCallbackType predicateCallback, numericFluentCallbackType numericFluentCallback,
    bool heuristic);
typedef int (*effectApplicatorType)(ParameterList & parameterList,
    predicateCallbackType predicateCallback, numericFluentCallbackType numericFluentCallback,
    vector<double>& writtenVars, bool heuristic);
typedef bool (*predicateCallbackType)(PredicateList* &predicateList);
typedef bool (*numericFluentCallbackType)(NumericFluentList* &numericFluentList);
```

Figure 4: Main part of the PDDL/M C++-Interface. The two types of semantic attachments are represented by two types of functions: conditionCheckerType and effectApplicatorType. Additionally two types of callback functions are defined: One for logic predicates and one for numeric fluents.

this, the relaxed reachability analysis of TFD/M adds all numeric fluents affected by a module effect whenever the corresponding ground action is detected as reachable.

Since semantic attachments reason about PDDL, i.e., a propositional representation rather than an FDR, some care must be taken when information is shared between the planner and the semantic attachments, i.e., when using the PDDL/M interface as shown in Figure 4. Firstly, mappings between module conditions and the newly introduced corresponding FDR variables must be stored, so that the search algorithm can call the module when evaluating the truth value of a variable. Secondly, we also need to keep a mapping between the FDR and the original PDDL task so that, upon entering a callback from a module, the planner can look up the internal FDR correspondent to the PDDL fluents used by the module.

Search Temporal Fast Downward performs heuristic search in the space of so-called *time-stamped states*. The most essential information encoded in a time-stamped state $\mathcal{S}$ is a real-valued time-stamp, a valuation of all state variables, and the set of operators already started but not finished yet. The successors of such a state are those time-stamped states that can be obtained by either starting a new applicable action at the current time point or by computing the temporal progression of the current state. A solution is found as soon as a time stamped state is reached that satisfies the goal and that contains no more scheduled conditions or effects.

In order to handle external modules during search, we extended TFD to support the interface sketched in Figure 4. Compared to the implementation in FF the callback functions are less straightforward in a planner that internally uses multi-valued variables. The interface to modules was designed to be independent of planner-specific representations and therefore expects the predicates from the original PDDL/M domain. Therefore, predicate names need to be converted to multi-valued variables at runtime using the table allocated in the translation.

An additional problem in TFD is that module conditions can occur as scheduled conditions, which means that they have to be checked much more often: A scheduled module condition has to be checked whenever a time progression is performed or when its time-stamped state is checked for consistency. To minimize computation our implementation does not check module conditions before all other logical and comparison conditions have been shown to be satisfied first.

Soundness and Completeness It is fairly obvious that semantic attachments, as implemented in FF/M and TFD/M, do not affect soundness and completeness of the planning algorithms under the assumption that the external modules satisfy the requirements specified previously, i.e., assuming that they terminate and deterministically compute values that are regarded as "correct". Soundness cannot be affected, since module effects virtually define how a correct state transition looks like in the presence of semantic attachments, whereas module conditions only restrict the options of the planner, but do not alter them. Conversely, completeness can not be affected, since module conditions may only rule out possible transitions that are considered "incorrect" by the condition checker, i.e., that evaluate to *false*. Since for module effects we assume that choices are uniquely determined by the current planning state, we cannot lose possible plans through "unfortunate" effect selection in the module.

Integration in Heuristics

As depicted in the interface shown in Figure 4, *condition-checker* and *effect-applicator* functions accept a Boolean parameter `heuristic`. When a module is called with `heuristic` set to *true* it should aim at very fast computations, possibly at the expense of accuracy. The idea is that the symbolic planner can decide to call this approximate version of the module during heuristic computations.

FF/M integrates module calls into its heuristic by treating module conditions like symbolic facts in the relaxed planning task. The idea is that in the relaxed planning task a module condition is met on all levels of the plan graph that follow the layer in which the condition was met for the first time. When extracting the solution plan from the plan graph FF/M selects achieving actions for each fact in the current fact layer. After applying an action in the creation of the plan graph each module condition which is still false is tested again and if it became true the action is saved as the condition's activating action to be used in the plan extraction.

Interestingly, if a necessary condition of the module condition can be expressed in PDDL, one can avoid to call modules during heuristic computations at all. E.g., in the trans-

#	TFD	TFD/M	%	#	TFD	TFD/M	%
01	0.01	0.01	0	**16**	0.61	0.78	28
02	0.01	0.02	100	**17**	0.73	0.96	32
03	0.01	0.02	100	**18**	0.85	1.10	29
04	0.04	0.05	25	**19**	1.89	2.38	26
05	0.08	0.10	25	**20**	3.19	4.06	27
06	0.14	0.18	29	**21**	2.47	3.12	26
07	0.16	0.24	50	**22**	0.16	0.19	19
08	0.18	0.24	33	**23**	0.12	0.14	17
09	0.29	0.37	28	**24**	0.20	0.26	30
10	0.59	0.75	27	**25**	—-	—-	—-
11	0.47	0.61	30	**26**	1.50	1.89	26
12	0.58	0.76	31	**27**	—-	—-	—-
13	0.05	0.08	60	**28**	3.82	4.71	23
14	0.08	0.12	50	**29**	5.74	7.21	26
15	0.06	0.07	17	**30**	5.55	6.89	24

Table 1: Results of Experiment I (runtimes in seconds).

port domain introduced above, the condition that the package fits into the truck solely based on its volume is expressible in PDDL and is also a necessary condition for the module condition. Since the value returned by this condition should suffice as a heuristic estimate, we can safely ignore the module condition in the heuristic state evaluation. As a side effect a lot of module calls are saved during search: Whenever a necessary PDDL condition is not satisfied, the module cannot be satisfied either.

Empirical Data

In this section we present three experiments. Experiment I is an adaptation of a standard benchmark domain that does not add any new features, but provides insight on the runtime solely caused by module calls itself. Experiment II shows a new variant of the logistics domain that respects the geometry of packages when determining if a package can be loaded. In experiment III a geometric manipulation planning domain is generated in which a semantic attachment checks for possible grasping poses. While in the first two experiments TFD/M is used, in Experiment III we use FF/M.

All experiments have been run on standard desktop computers, the first two on an Intel Dual Core 6400 CPU with 2100 MHz and the third on an AMD Athlon XP 2000+ CPU with 1667 MHz, both with 1 GB of RAM.

Experiment I

The first experiment is designed to show the overhead introduced by the module calls alone. As an example we chose the *crew-planning* domain of IPC 2008. The reason is, that it contains numerous different operators, that all have one predicate in common, namely the predicate *available*, showing if a crew member is available for executing a task.

We wrote a module that resembles this predicate by executing a callback to the symbolic planner, requesting the truth value of the *available* predicate in the current state and returning it. Essentially the module does not do anything different, and does not perform any extra calculations.

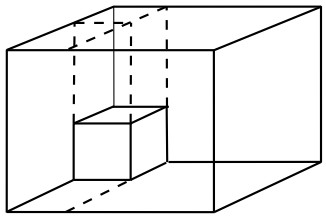


Figure 5: Recursive packing of rectangular objects: Once a package has been placed in a corner, three new rectangular containers emerge from the remaining space into which the remaining packages are recursively packed in the same manner.

In our experiment we ran TFD/M on the original version of the domain, and then compared runtimes with the modified version that adds a module call. Table 1 shows the planning time until the first plan was found (for a timeout of 30 minutes and a memory limit of 1 GB). As expected, the runtime for the module version of the domain is higher. It should be noted that the module calls do not influence the planning process itself as the same states are expanded, so results are comparable. Most importantly, it can be seen that the relative overhead is independent of the problem size, thus scaling properties of the planner are not influenced.

To judge the introduced overhead, it should be noted that in usual problems it is not the module call itself that takes a majority of the runtime, but the module's calculations. The increase in runtime is as anticipated, as we replaced a predicate check, that is usually implemented as an integer comparison, by a function call, that in turn creates a callback to the requested predicate. This clearly needs to introduce some overhead. Additionally, we chose a harsh domain for this experiment as the crew planning domain calls this module in almost every operator.

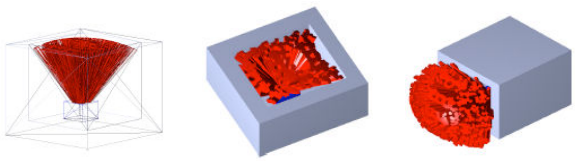
Experiment II

The second experiment presents a full implementation of a PDDL/M task that uses non-trivial semantic attachments. We follow the transport example proposed in the motivation section. Our custom domain models a classic logistics task where trucks are allowed to carry multiple packages with one crucial adaptation: One part of the *pick-up* operator's precondition is a semantic attachment implemented as a condition-checker module. The module *canLoad* is a packing algorithm that we shortly describe. The algorithm needs to solve the three dimensional bin-packing problem which, even for one bin, is already NP-hard (Martello, Pisinger, and Vigo 2000). As our main focus is implementing a correct, but not necessarily optimal solution, we therefore use a heuristic packing algorithm.

Our implementation follows a recursive approach of packing a set of rectangular packages into one rectangular container. First, the largest package that fits the container is placed in a corner. Second, the remaining space is partitioned into three new rectangular containers as shown in Figure 5. Third, the set of remaining packages is recursively packed into the remaining containers, starting with

#	Trucks	Packs	Nodes	Runtime
01	2	2	5	0.01
02	2	4	10	0.36
03	3	6	15	0.81
04	3	8	20	1.70
05	3	10	25	33.86
06	4	12	30	27.47
07	4	14	35	146.62
08	4	18	45	244.45

Table 2: Results of Experiment II (runtime in seconds).



(a) (b) (c)

Figure 6: Visualization of the grasp poses calculated by the external module during planning.

the smallest. If no unpacked packages remain, a packaging has been found and the module returns true. To make the planning algorithm complete, the exact method for three dimensional bin-packing (Martello, Pisinger, and Vigo 2000) could be used. In that case our or any other simplified solution is an obvious choice as a semantic heuristic.

The implemented attachment combined with the adapted transport-modules PDDL/M domain was run on examples based on the transport-numeric domain of the International Planning Competition 2008. Results are shown in Table 2 (as in the first example, a timeout of 30 minutes and a memory limit of 1 GB was set) and indicate the time in seconds until a valid plan was found.

Experiment III

The planning domain for our third experiment mirrors one of the real-world applications that we are investigating, plan-based robot manipulation. For this experiment, we created a geometric manipulation planning domain, focusing mainly on handling different grasp poses. Each pose is defined as a tuple $p = (rot_x(p), rot_y(p), rot_z(p))$, where for $i \in (x, y, z), rot_i(p) \in [0, 2\Pi]$ is the rotation of the robot gripper around the i axis. The module was implemented to check all possible grasping poses with a step-size of 0.1° in each rotation axis, leading to very accurate results.

The task consists of multiple cubes that have to be moved out of boxes onto a single shelf. Additionally, tables serve as temporary placement possibilities where grasp poses could be changed if necessary. To provide a means of comparison, we tested this domain setup with different numbers of cubes, boxes, and tables. Figure 6 shows a visualization of the usable grasp poses calculated by the external module during the planning process.

This experiment was run with the FF/M planning system. Results are shown in Table 3; runtimes are given in seconds.

#	Cubes	Boxes	Tables	Runtime
01	1	1	1	0.20
02	4	1	1	1.20
03	8	1	1	5.00
04	12	1	1	12.80
05	1	4	4	0.58
06	1	8	8	1.06
07	1	50	50	6.27
08	4	4	1	5.30
09	6	6	1	36.55
10	12	12	1	1444.90

Table 3: Results of Experiment III (runtime in seconds). The number of cubes, boxes, and tables gives an indication of the growing complexity of the problems.

To give a feeling for the differences between tasks the table also shows the numbers of cubes, boxes and tables available in each task. The results show that FF/M is capable of solving middle-sized problems in reasonable time, especially considering the rather small step size we chose for each rotation axis. Given that the algorithm implemented in the external module is far from optimal, this result is very promising and shows that it is reasonable to consider solving real-world applications using a symbolic, domain-independent planner with semantic attachments.

Related Work

In contrast to other planning systems that exploit domain knowledge, such as SHOP2 (Nau et al. 2003), TLPlan (Bacchus and Kabanza 2000), or TALplanner (Kvarnström and Doherty 2000), semantic attachments do not guide the search process, but provide a more precise domain semantics.

In the past, semantic attachments have already been used in some domain-specific planning systems for computing specific action preconditions (Konolige and Nilsson 1980; Orkin 2006). In this work, we have generalized this idea to domain-independent planning and have specified a suitable PDDL dialect. Moreover, we extend previous work by describing the use of semantic attachments for computing action effects.

Semantic attachments enable "outsourcing" of hard problem-specific computations during planning. In that respect, our goals resemble the ones of Fox and Long (2001), who tried to isolate optimization problems from planning problems. The work by Srivastava and Kambhampati (1999) on decomposing a general planning problem into a resource and a planning problem is also relevant here. However, they specifically investigate the relation between resource and planning problems while we propose a general framework for combining different kinds of planning.

In the area of robotic planning, the work that comes closest to our intentions is a paper by Cambon et al (2004). They also work on the integration of manipulation and symbolic planning. However, they did not try to identify a general interface between symbolic planning and domain planning,

but presented a specialized combination of a symbolic and a manipulation planner.

The mechanism we propose is similar to an undocumented feature of TLPlan (Bacchus and Kabanza 2000). This planner also permits semantic attachments to predicate symbols (Botea, Müller, and Schaeffer 2003). The main differences to our approach are that TLPlan uses domain-dependent search control, that the planning state cannot be queried via call-back functions, and that it is not possible to specify externally computed effects.

Conclusion

Planning occurs in many real-world problems. However, applying AI Planning techniques to solve them is often difficult, mainly because the planning problem cannot be isolated from other reasoning tasks which the planner is not designed to solve. Some aspects of the dynamics of an application domain may be hard or even impossible to describe declaratively, but must instead be *computed* when needed.

This is perhaps most notable in *robotics* applications where causal, symbolic reasoning must be tightly entwined with numeric computations, and where both may directly influence each other. We believe that the impossibility to interface non-symbolic reasoners (manipulation planners, path planners) *during* the planning process in most current planners has been a major hindrance for their use in robotics.

In this paper, we have presented an approach to integrating external reasoning mechanisms, so-called semantic attachments, directly into a planner. We have specified a suitable extension of PDDL to model them, and have described criteria under which soundness and completeness of planners are maintained when they are extended with semantic attachments. This allows domain designers to use domain-independent planners, and extend them with domain-specific sub-solvers where necessary. These "modules" can influence the course of the planning process directly by providing the planner with better information about action applicability and effects, thereby reducing future execution failures and the need for replanning.

In future work, we will focus on the impact of module relaxations on the efficiency and accuracy of heuristics. Additionally, we will remove an important (and somewhat arbitrary) restriction from PDDL/M and our implementations: In general, there may be many options for how to achieve a module effect. E.g., a manipulation planner may find several poses from which it could grasp an object. Currently, we only permit modules that return exactly one result. In future work, we will enable the planner to branch over an initially unknown, yet finite number of outcomes online.

Acknowledgements

This research was partially supported by DFG as part of the collaborative research center SFB/TR-8 Spatial Cognition Project R7, the German Federal Ministry of Education and Research (BMBF) under grant no. 01IME01-ALU (DESIRE) and by the EU as part of the Integrated Project CogX (FP7-ICT-2xo15181-CogX).

References

Bacchus, F., and Kabanza, F. 2000. Using temporal logics to express search control knowledge for planning. *Artif. Intell.* 116(1–2):123–191.

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS^+ planning. *Comput. Intell.* 11:625–655.

Blum, A., and Furst, M. 1995. Fast planning through planning graph analysis. In *Proc. IJCAI*, 1636–1642.

Botea, A.; Müller, M.; and Schaeffer, J. 2003. Using abstraction for planning in sokoban. In *Proc. Computers and Games*, 360–375.

Cambon, S.; Gravot, F.; and Alami, R. 2004. A robot task planer that merges symbolic and geometric reasoning. In *Proc. ECAI*, 895–899. IOS Press.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proc. ICAPS*.

Fox, M., and Long, D. 2001. Identifying and managing combinatorial optimisation subproblems in planning. In *Proc. IJCAI*, 445–452.

Fox, M., and Long, D. 2003. PDDL 2.1: an extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Helmert, M., and Geffner, H. 2008. Unifying the causal graph and additive heuristics. In *Proc. ICAPS*, 140–147.

Helmert, M. 2006. The Fast Downward planning system. *JAIR* 26:191–246.

Helmert, M. 2009. Concise finite-domain representations for PDDL planning tasks. *Artif. Intell.* 173:503–535.

Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.

Konolige, K., and Nilsson, N. J. 1980. Multiple-agent planning systems. In *AAAI*, 138–142.

Kvarnström, J., and Doherty, P. 2000. TALplanner: A temporal logic based forward chaining planner. *Ann. Math. Artif. Intell.* 30(1-4):119–169.

Martello, S.; Pisinger, D.; and Vigo, D. 2000. The three-dimensional bin packing problem. *Oper. Res.* 48:256–267.

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wau, D.; and Yaman, F. 2003. Shop2: An HTN planning system. *JAIR* 20:379–404.

Orkin, J. 2006. Three states and a plan: The A.I. of F.E.A.R. In *Proc. Game Developers Conference*.

Srivastava, B., and Kambhampati, S. 1999. Scaling up planning by teasing out resource scheduling. In *Proc. ECP*, 172–186.

Weyhrauch, R. W. 1980. Prolegomena to a theory of mechanized formal reasoning. *Artif. Intell.* 13(1-2):133–170.