Transformational Planning for Everyday Activity

Armin Müller, Alexandra Kirsch and **Michael Beetz**
Intelligent Autonomous Systems Group
Department of Informatics
Technische Universität München
Boltzmannstr. 3, D-85748 Garching
{muellear,kirsch,beetz}@cs.tum.edu

Abstract

We propose an approach to transformational planning and learning of everyday activity. This approach is targeted at autonomous robots that are to perform complex activities such as household chore. Our approach operates on flexible and reliable plans suited for long-term activity and applies plan transformations that generate competent and high-performance robot behavior. We show as a proof of concept that general transformation rules can be formulated that achieve substantially and significantly improved performance using table setting as an example.

Introduction

AI planning — fueled by the AI planning competitions — has seen tremendous successes over the last decade. Yet, the application of AI planning techniques to autonomous robots performing everyday activity, which has been one of the ultimate goals of planning research ever since its infancy, has not benefited much from these successes.

Why is this so? What do AI planners have to do for service robots? There are many situations where AI planning can help robots to solve their problems more reliably and efficiently. For example, a planner could reason about how to get a cup out of a cabinet. If the robot has to take several objects out of the cabinet it can think of an order that simplifies the reaching tasks or it could check whether temporarily moving an obstacle out of the way would help. It could reason about whether it could leave a cabinet door open until it is back or whether it would be safer to close the door in the meantime. The robot could also think about the overall structure of activities such as setting the table. Here, the question is whether to carry the tableware one by one, whether to stack the plates, or to use a tray. Which of the option is the best critically depends on the robot's dexterity, the geometry of the room furnishing, other properties of the environment, the availability of trays, etc.

Most AI planners cannot perform these planning operations for several reasons. First, the planners primarily address the problem of generating partially ordered sets of actions that achieve some desired goal state. While some of the planners reason about resources and generate resource-efficient plans they do so at an abstract level considering plan actions as black boxes. In contrast, the cases above require much more detailed consideration of resources and situation-dependent resource requirements. Also, current planners make the assumption that complex activities are sufficiently specified using the set of actions that must be carried out and a set of ordering constraints that prevent negative interferences between the plan steps. In contrast, robot activity requires sophisticated coordination using control structures much more powerful than simple action chaining. When the robot gets an object out of the way to pick up another one, the obstacle should be put back immediately after the pick up is completed and before the robot leaves its current location.

In this paper we propose TRANER (TRAnsformational PlanNER for Everyday Activity), a form of plan-based control that better matches the needs of autonomous service robots. Its planning tasks are to find out how subtasks can be performed more efficiently and reliably, and to transform default instructions into activity specifications that enable the robot to improve its performance in a specific environment.

The ultimate goal of our research is the development of

1. a library of *general* plans for everyday activities. The library contains a single plan for picking up a variety of objects in different situations: whether a pinch grasp or a wrap grasp is needed, whether one or two hands, or even a container are needed.
2. a set of transformation rules that implement *general* plan revisions such as "transport the objects using a container instead of one by one". Such revisions require substantial changes of the complex intended courses of action.

Thus, our approach assumes that we can implement plans and transformation rules of the required generality and that our approach scales to the range of activities and the plan improvements needed for a household robot. This means a plan library that contains tens or even hundreds of general plans and tens of general plan transformation rules.

This paper describes a proof of concept in which we use a plan library for picking up and placing objects and a small set of general transformation rules for optimizing transportation tasks.

We apply TRANER to a simulated robot operating in a kitchen (see Figure 1) and performing complex activities such as setting the table, cooking pasta, cleaning, etc. The simulator uses realistic physical simulations of actions and

sensing devices. This application domain requires (1) plans to include control structures that make plans reliable and flexible — they need control structures for performing carefully synchronized concurrent activity, for behavior monitoring and failure recovery at all levels of activity specification; (2) transformations to restructure the intended course of action. For example, instead of transporting a set of objects one by one a transformation might suggest the use of a container and thereby change the activity into getting the container, loading it, transferring it, and unloading it.[1]

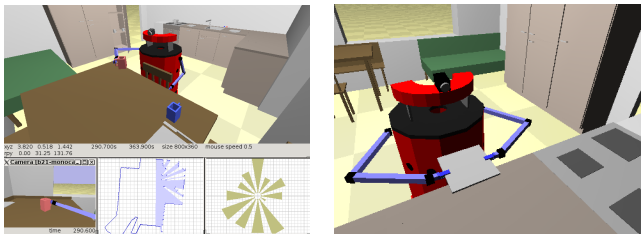

Figure 1: Kitchen scenario with a simulated B21 robot equipped with a camera, laser, sonar sensors, and two arms.

We show, in a household domain and for a table setting task that TRANER can learn flexible and reliable environment-specific default plans that improve the problem-solving behavior of the agent substantially and in a statistically significant way.

In the remainder of this paper we proceed as follows. The next section gives an overview of TRANER. After that, we describe the system in more detail and give examples of plan transformations. We conclude with presenting experimental results and related work.

Overview

TRANER operates in two modes. In the online mode TRANER retrieves plans for the given tasks and executes them. It also measures the performance of the plans with a given cost function that includes the time needed to complete tasks and the execution failures that occurred. Based on the measured performance TRANER decides whether or not it should try to improve the respective plans. In idle times, e.g. at night, TRANER generates alternative candidate plans and evaluates them by simulating them. If the candidate plans achieve better performance than the respective plan in the library, the library plan is replaced by the new one. Before explaining TRANER's software architecture shown in Figure 2, we demonstrate the working of our system with an example.

Example. Consider a robot that is to prepare a meal with a main dish and a dessert. Let's assume that some ingredients for both dishes are in the larder. The robot should realize that the same subtask — go to the larder to fetch an ingredient — occurs in both plans, the one for preparing the main dish and the one for making the dessert. In this case it should combine the two steps into one and only go to the larder

[1] A video showing our plan transformations can be found at `http://cogito.cs.tum.edu`

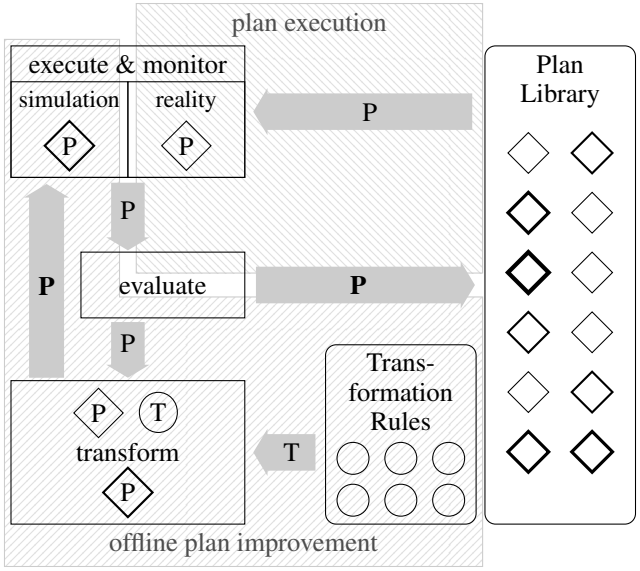

Figure 2: Overview of TRANER. Libraries are illustrated as boxes with rounded corners. Plans are depicted as diamond-shaped objects, transformation rules as circles. Plans surrounded by thicker lines have been transformed more often than those with thin lines. The boxes mark operations on data structures like plans or transformation rules.

once. But possibly the robot cannot carry all the things at once. Then it should transform the plan again in a way that it uses a container for transporting the ingredients. So the resulting plan would be to go to the larder once and get all ingredients at once, possibly by using a container.

But wait, is this really the best plan? Maybe there is no container at hand and the getting of the container needs more time than just going to the larder twice. Or the kitchen might be very small and the ingredients for the dessert are in the robot's way while preparing the main dish. So the best plan can actually not be determined without knowledge and experience about the environment. Depending on the robot's dexterity, the size of the kitchen, the way to the larder, the places where possible containers are kept, etc. different plans should be favored.

Plan Library. All plans are stored in a plan library. It contains default plans (surrounded by thin lines) that can be expected to work in any kitchen as well as plans that have already been enhanced by transformations and are adapted to the specific needs of the environment and the robot working in it (depicted with thicker lines).

The default plans must be robust and general without making assumptions about the specific situation they are executed in. For example, a plan telling the robot to get the ingredients from the larder one by one is probably suboptimal for most environments and situations, but it is quite certain to work. We have already described how this plan can be transformed to be more efficient.

Plan Execution. When the robot is to perform a job, a plan is chosen from the plan library. All the plans in the li-

brary are expected to work, so that in a known situation, the chosen plan produces the desired result with high probability. The chosen plan is then executed and monitored in the environment.

Plan Evaluation. After it has ended (either successfully or with a failure) the result of the execution is evaluated. For this purpose, some benchmark data is observed during plan execution, for example the time and resources needed or the number of occurred failures, the fraction of them that could be repaired etc. Then a cost function is applied to this data. If the execution fulfills all the quality criteria, the plan is not changed and stays in the plan library. However, if the evaluation finds that the plan doesn't work optimally, it is tried to be enhanced by plan transformation.

Plan Transformation and Rules. For making plans better, a set of transformation rules is given. A transformation rule accepts a plan with a certain structure and produces several new plans according to the specified rule. For improving plans, a possible transformation rule is chosen and applied. The resulting plans are then executed in simulation, monitoring the same quality criteria as in the real plan execution before. Some of the new plans might fail completely, possibly because they aren't even syntactically correct. In any case, all transformed plans are evaluated against the old one. If new plans fail or perform worse than their parent based on several simulation runs, they are discarded and other transformations are tried, otherwise the best plan is chosen as the new default plan for the activity.

TRANER performs exhaustive search in order to find better plans. Plan improvement as just described doesn't necessarily have to end after one transformation step, instead it is an iterative process. If the new plan is better than the old one, but still doesn't fulfill the demands of the evaluation function, it is stored and transformed further. Besides, if the original plan fits the structural conditions of several transformation rules, all possible rules are applied and the resulting plans are evaluated against each other. This procedure corresponds to an unguided search in the space of plans, where plan transformations describe the possible state transitions.

Discussion. TRANER's exhaustive search strategy waste resources. To make this search more efficient, especially when many transformation rules are available, the transformation rules provide an applicability condition, which uses the benchmark data observed during plan execution to determine if the transformation rule might be a good choice. For example, if the robot dropped objects when it had to transport several things, a rule that adds the use of a container would be more useful to make the plan stable than one that changes the order in which the objects are needed. As another way to focus the search more we envisage the following system operation: the planner is coupled to an imitation learning system, which will hypothesize that people set tables faster because they carry objects as stacks. The planner would consequently apply the stacking transformations and then debug the resulting plan candidates.

For testing transformed plans a realistic accurate simulation environment is needed. Therefore, we use the Gazebo simulator, which provides a very good physics engine and realistic, non-deterministic simulation. At the moment "real" and simulated environment are identical. This ensures that when we have a real robot also a realistic simulation is available. We describe more details about the simulation in section "Empirical Results and Discussion".

At the moment, our default plans and transformation rules are coded by hand. In the future, plans could be extracted from web sites like `eHow.com` or knowledge bases like Cyc. The implementation of transformation rules is quite straightforward for special cases, but needs experience and care if the rules are to be formulated in a general way.[2]

TRANER

In the following, we explain TRANER in more detail. The sections are organized along the main components of Figure 2. As a running example we use the task of setting the table for an arbitrary number of persons.

Plan Library

Our plan library contains plans for reaching, grasping, picking up objects, navigating, etc. — routines that can be used as building blocks of more complex activity specifications. These are the plans that are either implemented with great care or are already adapted to the specific environment, so that they are usually not changed by plan transformations.

More complex plans are filling a container with a liquid, carrying an object from one position to another, stacking objects and even more abstract plans like setting the table or cooking pasta. These are the plans that gain most by plan transformation.

The default plans, which are to be adapted to the environment, must be designed in a way that makes them general, robust, and transformable. General means that these plans are not optimized for a special environment. They should not make any assumptions about the robot's abilities, the dynamics, the spatial layout or the users.

To achieve reliability, the robot's default plans detect, monitor and recover failures during execution. Failures can be detected and recovered on any level of abstraction. For example, if the robot detects that it was unable to grip a cup, the low-level grip plan would try to grip again. If after several trials the cup still couldn't be reached, the grip plan fails and passes the failure description to the higher-level plan. This plan might try to grip with the robot's other gripper or from another location. If that still doesn't help, a higher-level plan might decide to get a different cup that can be reached more easily.

Furthermore, we want the default plans to be transformable, in order to make them better by applying plan transformations. This means that the plans must follow a structure that can be understood by TRANER, so that transformation rules fit the plan.

Both for the failure handling abilities and the explicit structure of the plans, the plan language RPL (Reactive Plan

[2]Sussman (1977) learned transformation rules automatically, but in an artificial domain. In the kitchen domain, possible transformation rules could be deduced by watching humans. This, however, still requires tremendous research effort.

```
1  (define-plan (achieve (table-set ?persons))
2    ( achieve-for-all
3      (lambda (person)
4        (with-designators
5            ( (table '(the entity (type table)
6                                  (used-for meals)))
7              (seating-location `(the location (at ,table)
8                                                (preferred-by ,?person)))
9              (plate `(an entity (type plate)
10                                (status unused)))
11             (cup `(an entity (type cup)
12                               (status unused))) )
13         (achieve (placed-on
14                   plate
15                   table
16                   `(the location (on ,table)
17                                  (matches (entity-location ,plate))
18                                  (matches ,seating-location))))
19         (achieve (placed-on
20                   cup
21                   table
22                   `(the location (on ,table)
23                                  (matches (entity-location ,cup))
24                                  (matches ,seating-location))))))
25     ?persons))
```

Figure 3: The default plan for setting the table. An example for calling the plan is (achieve (table-set '(Alvin Theodore Dave))).

Language) (McDermott 1991) together with some extensions proposed by Beetz (2001; 2002) provides an ideal basis. Unfortunately, space limitations do not allow us to introduce the language. We can only give a flavor of what the plans look like.

As the plans are to work in different environments, objects cannot be addressed by unique identifiers, which is in any event undesirable in real-world settings. Therefore, TRANER supports the concept of *designators*, logical descriptions of objects, which are resolved to object instances at run-time. For example, when setting the table, one subtask is to bring a plate to the table. The identity of the plate is irrelevant, but it should be a clean plate, not a used one. With designators, such a plate can be described declaratively and the robot tries to find one in the given situation (see lines 9 and 10 in Figure 3).

The default plan for setting the table is shown in Figure 3. It first places the plate and the cup for one person on the table, then for the next one, and so on. The low-level plan for placing an object on the table involves the two steps of picking it up and then placing it on the table. The pick-up routine searches for the object described by the appropriate designator, navigates to a location where it can grip the object and finally picks up the object. Putting down the object includes navigating to a suitable put down position and placing it on the table. These basic routines are implemented in a robust way and can recover from local failures, like losing the object during navigation.

Plan Execution

The environment our robot works in is a very realistic simulation of a real-world kitchen. In such a complex environment failures are unavoidable. Figure 4 shows a small variety of failures that occur while the robot is handling objects. The robot sometimes cannot grip the object reliably, things slip from the grippers, or the robot places an object at a position, where another object already is (either because its state estimation is inaccurate or because it has placed one of the

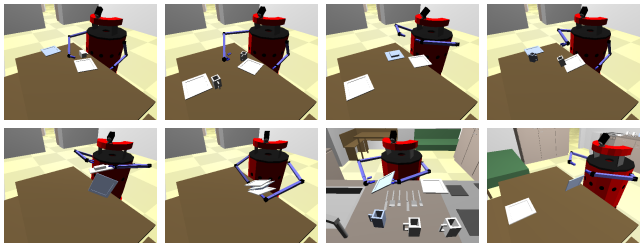

Figure 4: Different failures observed during experiments: dropping objects, not being able to grip objects, and putting objects at the wrong position.

objects at the wrong position). These failures are sometimes caused by the lacking dexterity of the robot, sometimes by the uncertainty of the environment. In any case, it is impossible to avoid them completely.

To achieve reliability and flexibility our plans monitor eight types of failures. On averages 1–4 failures are dealt with by a plan. If a failure occurred, which happened in 8% of the experimental runs, the robot achieved its goals in roughly 86% despite the failure. The failure monitors and handlers are not complete. They suffice to achieve a robustness for complete task achievement of more than 90%. Flexibility and reliability also require synchronized concurrent activity: on average 10–15 threads of activity are executed concurrently. During one run of the table setting plan approximately 700 conditions (perceptual changes, failures) are monitored.

Plan Evaluation

When evaluating plans, not only efficiency, but also stability criteria must be considered. On the other hand, not every failure causes the whole plan to fail. For assessing the importance of failures that have occurred during testing, it is necessary to test a plan several times and then decide if a failure is intrinsic in the plan (if it occurs several times) or has been observed as an instance of the uncertainty in the environment (if it is observed infrequently).

For making this testing phase efficient and safe, this step is to take place in simulation. Of course, the simulation must be near enough to reality in order to give valid results. We use the Gazebo simulation environment described below.

The system doesn't depend on the transformation rules to generate valid plans. Even syntactically invalid plans can be simulated (the controller aborts and the robot doesn't do anything afterwards) and can therefore be evaluated. Specifying good transformations only ensures that not too many garbage plans are generated.

Transformation Rules

For optimizing plans, the robot can choose from a set of predefined transformation rules of the following form:

$$\frac{\text{input schema}}{\text{output plan}}\ \text{transformation}\ \Big|\ \text{applicability}$$

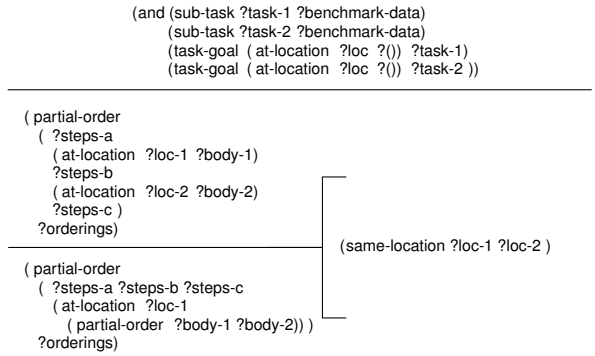

Figure 5: Rule using the at-location macro. When plans are tested, the tasks they trigger are protocolled in a feedback trace. This rule is only applicable when the benchmark data contains two tasks operating at the same location.

Each transformation rule includes an applicability condition, which decides whether or not the rule is applicable to the current plan in the current situation. The input schema determines the subplan to be transformed by specifying a pattern, which requires the plan to match certain structural conditions. When the applicability condition is fulfilled and the input schema matches, the plan is transformed according to the transformation rule, which specifies how code parts matching the input schema must be altered. The output plan states how the resulting plan is to be constructed from the transformed code pieces.

While all kinds of plan transformations can be expressed with these rules, careful design of the plan transformation rules as well as the plans is necessary in order to keep the rules general — so that they can do the same modification to many plans and are not restricted to very specific syntactic forms. We do this by introducing new classes of macro plans that syntactically represent their purpose, are modular, and work in a variety of task contexts.

For example, to realize transformation rules to sort subplans according to the locations where the actions are to be executed we introduce a new plan macro (at-location loc body). The code for the macro ensures that body can only be executed when the robot is at location loc. If the subplan is executed and the robot is not at loc the plan asks the robot to first go to loc and starts executing body after the robot's arrival at loc. The point is that if every subplan that is to be executed at particular locations is made explicit using the at-location macro, then general transformation rules that automatically group and order subplans according to the locations where they are to be executed (see Figure 5) can be specified. Besides, we have introduced macros for specifying the objects that are to be manipulated, used as containers, needed as tools or serve as ingredients.

Plan Transformations

We can now assume that the plans in the plan library and the transformation rules are implemented and designed in a way that makes them appropriate for TRANER. In the following, we show how transformation rules can be applied to the default plan from Figure 3 to make it work more efficiently. It certainly shows potential for improvement, since bringing the necessary objects to the table one by one for every person takes a lot of time. For this we wrote three transformation rules.

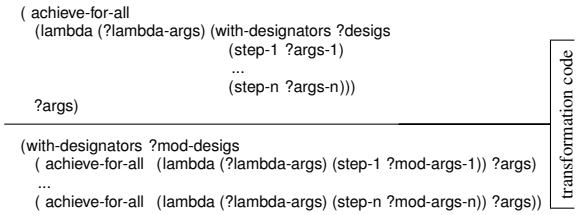

Figure 6: Transformation rule changing the order of plan steps. The rule is applicable in any situation and only the input schema and output plan are shown explicitly. The transformation code contains rules specifying how to compute values like ?mod-desigs and ?mod-args-1.

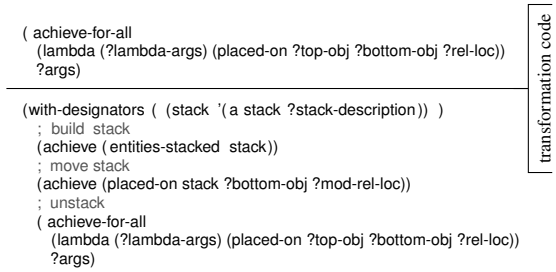

Figure 7: Transformation rule inserting a stacking plan step. The ?stack-description is generated by rules from the transformation code and includes the designators of the objects composing the stack.

1st Transformation: Regroup Plan Steps. The first transformation (see Figure 6) reorders the plan steps. This means that now first all plates and then all cups are placed on the table separately. The performance of the new plan is the same as of the old one, but now the next two transformations are possible.

This rule is an instance of a class of transformations regrouping plan steps, which is possible when a plan contains steps working on the same type of objects or performs the same task for different types of objects.

2nd Transformation: Use Containers. The second transformation (see Figure 7) inserts two new steps into the plan. Before placing each plate separately on the table the plates are stacked and the stack is placed on a suitable location on the table. A good choice is the final location of the bottom plate. Afterwards all plates are placed at their correct locations. But this time the robot needs to navigate less than in the other plan, or in the best case not at all. The stacking step is skipped if the plates are already stacked.

This rule can be regarded as a special case of using a container, where the bottommost object constitutes the container. Using containers is usually useful when many objects must be transported to similar locations or when objects can easily be lost like in the case of cutlery.

3rd Transformation: Exploit Resources. The second transformation is also applicable for the cups. But when the robot simulates the transformed plan, it will detect that it is not stable to stack cups and carry the stack. Now the third transformation is applied to the part of the plan concerned with carrying the cups. Since the robot has two arms and only one arm is used for carrying cups the third transformation always picks up two cups at once and puts down both on the table. If the table is to be set for an uneven number of persons, then in the last step only one cup is picked up and placed on the table.

Again, we can state a more general rule: Transform plans such that they use the robot's resources optimally.

Discussion. Usually one could expect the plates to be stacked in a cupboard. So why didn't we construct the default plan in a way that it assumes the plates to be stacked and takes a stack of plates out of the cupboard instead of single plates? One reason is that the default plan should be constructed in a way that it works in any situation, albeit it might not be efficient. This includes situations where the plates are in the dishwasher or are lying around in the kitchen. Another point is that sometimes people prefer specific plates, which might be located at special places.

We justified the first transformation step of reordering plan steps by referring to the later transformations of stacking the plates and the cups. If this step was not performed, the second transformation rule could still be used to stack a cup on a plate (the plate being the container for the cup).

The transformations presented here are general in that they can be applied to a variety of kitchen tasks. When dishes are taken out of the dishwasher they could be brought to their respective locations one by one or in a stack depending on the object properties and on the positions they are to be moved to. When clearing the dishes after a meal, again it could be useful to combine different plan steps. In contrast to setting the table, there might be other constraints for clearing like not to stack dirty dishes.

Empirical Results and Discussion

For our experiments we used a simulated household and robot based on the Gazebo simulator. The decision to use a simulator was made, because a kitchen is a complex environment where the robot needs sophisticated actuators, especially arms and grippers. Such equipment, together with the kitchen itself, is very expensive and hard to maintain. The simulation is much cheaper, better available and more flexible concerning different robot hardware and different environments. We state that the performance of plan transformations relies heavily on the environment. This is hardly to be tested with only one kitchen. The simulation gives us the possibility to have different testing environments at hand. Besides, we can adapt the features of the environment according to our research focus. For example, for cooking or setting the table our robot needs to open and close cupboard doors. However, for opening and closing doors in a kitchen, sophisticated motor control and plans are necessary, in which we are not interested. Therefore we added automatic doors to the kitchen, which can be remote-controlled by the robot. This is an assumption that could very well be built into a real kitchen, but the simulation could be implemented with much less effort.

On the other hand, the danger of a simulation is that interesting aspects of the environment are abstracted away from and the results gotten in simulation aren't applicable to realistic settings. To avoid this danger, we chose the Gazebo simulator, which includes the physical simulation engine ODE. All objects in the kitchen and the robot are composed of solid entities, whose interaction is simulated very realistically by ODE. The interface between our robot program and the simulator is the same as between the program and a real robot. This is possible with Player (Gerkey, Vaughan, & Howard 2003), which provides a device-layer that provides a network interface to the hardware (or simulated hardware) underneath. This makes it possible to use the same control program in simulation and on a real robot, which we are currently building.

As we aren't concerned with state estimation, we assume that the robot's position (x/y-coordinates and orientation) are given as percepts (which is quite realistic in a known environment and with a laser-equipped robot) and the position of all objects in the robot's field of view can be determined accurately. The simulation is very realistic with respect to non-determinism in the robot's actions. Because there are several processes involved (Gazebo, Player, the robot program) the execution of a robot control program in a given situation in the simulator never causes exactly the same result. This is due to the delays in normal process and network communication and makes the simulation very realistic.

Experiments. With our experiments we want to show that (1) applying fairly general transformation rules to robust and flexible plans leads to a performance improvement. (2) it depends on the environment and the dexterity of the robot, which transformed plan the robot should use.

For our experiments we used two different kitchens (Figure 8). In both kitchens there are a table and a kitchenette. For our experiments the robot had to set the table for Alvin (A), Theodore (T) and Dave (D) in four different combinations (A & T, A & D, T & D, and A, T & D). As predefined knowledge we assumed that the preferred seating locations at the table of all three persons are known, and that it doesn't matter which plate or cup they get. Also, the relative locations of the dishes (plate, cup, fork, knife, and spoon) for one cover were given. For our experiments we only used plates and cups to be fetched from the accordant kitchenette.

For setting the table we used five plans, the default plan and four transformed plans (resulting from applying only the 1st, the 1st and 2nd, the 1st and 3rd, and all three transformations respectively). Every plan was executed ten times for the eight test cases while recording the time needed for setting the table. Only if no failure occurred during one run the data was used, otherwise the run was repeated until it succeeded.

Results. Figure 9 shows a representative excerpt from the experimental results we have obtained in the way described. It shows the average time the agent needed to set the table for Alvin & Dave and Alvin, Theodore & Dave respectively

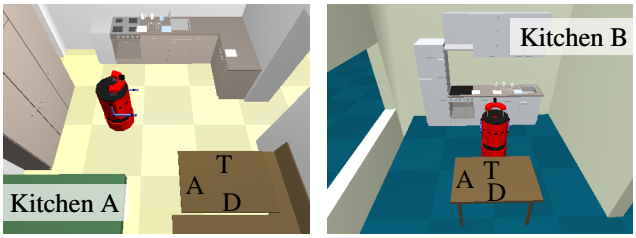

Figure 8: Top view of our two kitchens. A, T, and D denote the preferred seating locations at the tables of Alvin, Theodore, and Dave respectively.

for all of the five plans together with the maximum and minimum derivation.

As could be expected, the first transformation (grouping similar actions) has nearly no influence on the performance of the plan. But it is a necessary prerequisite for applying the other two transformations.

The second transformation (stacking the plates) improves the time needed in two of the four cases. In one case the time is nearly the same as the default plan and in the last case it takes even longer to set the table. Here the problem is that stacking and unstacking consumes more time than just carrying the plates one by one.

The third transformation (exploiting resources) always accelerated setting the table. Not surprisingly, the plan resulting from all three transformations is the sum of the improvements achieved by the second and third transformations.

These data already show that not always applying all three transformations results in the best plan. Instead, it depends on the environment and the robot which plan is the best. Figure 10 shows the best plans for each of the eight test cases. In five cases applying all three transformations is the best solution, twice not stacking plates, but using both arms for carrying cups is the choice and once only stacking.

Discussion. In our experiments we only regarded successful runs, which explains the deviation of only three seconds. Also when choosing the best plan, the time of successful runs, which we have just presented, is only one criteria, another is the success probability or how many failures occur. At first glance the time improvement achieved by the transformations isn't too impressive. The reason for this is that our simulation is relatively close to the state of the art in robotics. Thus, manipulation and dexterity are much more immature than navigation (compared to humans). Most of the plan execution time is taken up by low-level manipulations, which lessens the impact of our transformations.

More interesting than the results themselves is the fact that the results are as we expect. This is something that doesn't come for free in the robotics domain. Notably, in order to achieve the results the robot had to accomplish the tasks reliably. In some runs serious action failures (e.g. an object slipping out of the gripper) occurred. In most of these cases (86%) the top-level tasks could be achieved despite such failures. The robot never failed to achieve its tasks without recognizing these failures. This level of reliability and failure awareness is a required precondition for improving robot behavior using AI planning mechanisms.

Failure awareness is also needed to filter out experimental data that are corrupted by random failures. Only with small variances in the resulting behavior the planner can make valid inferences that one plan variant is better than another one in a statistically meaningful way (for example, passing a t-test). In our experiments the maximum difference to the average execution time was three seconds.

Of course, the most important result is that we can apply fairly general transformation rules to flexible and reliable robot plans for long-term activity and obtain substantial performance improvements. In our ongoing research we scale TRANER along two dimensions. First, we extend the task domain from setting the table to include cooking and clean up tasks. The second dimension of scaling are the aspects of activity that are handled by the transformation rules.

Our plans show that a plan representation as sequence of actions doesn't suffice by far to represent general plans for complex tasks. The original default plan for setting the table contains a loop so that the number of dishes can vary. At application time this plan can be used without any costs of planning. The optimization, too, works on the control structures so that the optimized plan is still general in the way that the number of dishes needn't be known in advance. We have shown that even on plans with complex control structures specifying robust and flexible behavior it is possible to define modular and transparent transformation rules that lead to better behavior.

As we have already argued, the best plan relies heavily on the characteristics of the underlying routines. The performance of lower-level routines depends strongly on the environment and their behavior must be adapted accordingly. For this reason it is impossible to determine the best plan for a problem analytically. Transformations only make sense in the context of a fixed environment and with empirical evidence as to the quality of a plan.

Related Work

TRANER can be viewed as a modern version of Sussman's (1977) *Hacker*. Like *Hacker*, TRANER aims at learning plan libraries by debugging the flaws of default plans. Unlike *Hacker*, which worked in the idealized Block's World domain, TRANER applies to real-world robot control.

Other transformational planners are *Chef* (Hammond 1990) and *Gordious* (Simmons 1988). The main difference between these systems and TRANER is that TRANER reasons about concurrent robot control programs while *Chef* and *Gordious* reason about plans that are sequences of plan steps. Another difference is that they try to produce correct plans while TRANER adapts plans to specific environments and corrects failures during execution. Botelho and Alami 2000 show how robots can enhance plans cooperatively by merging partially ordered plans using social rules.

TRANER is most closely related to more recent variants of transformational planning techniques. Most notably, to McDermott's (1992) XFRM planner that performs improving transformations on an idealized grid world agent. Beetz (2001) successfully applies transformational planning

	Alvin, Dave		Alvin, Theodore, Dave	
	kitchen A	kitchen B	kitchen A	kitchen B
default plan	191.7 (+1.2, -0.8)	240.3 (+0.6, -0.5)	293.4 (+1.8, -1.6)	337.8 (+1.4, -0.9)
group (1st)	190.5 (+0.9, -0.9)	241.1 (+1.2, -0.7)	295.1 (+3.4, -1.1)	336.7 (+1.4, -0.7)
stack (1st & 2nd)	191.5 (+0.8, -1.2)	233.1 (+1.7, -0.9)	285.7 (+1.4, -1.0)	373.0 (+1.4, -1.4)
resources (1st & 3rd)	184.2 (+0.8, -1.3)	210.5 (+0.9, -0.7)	268.8 (+1.2, -0.8)	330.1 (+1.4, -1.8)
stack & resources (1st, 2nd & 3rd)	185.0 (+1.7, -0.9)	202.7 (+1.7, -0.9)	259.1 (+2.6, -1.3)	363.7 (+4.0, -2.7)

Figure 9: Time in seconds needed for setting the table in both kitchens for Alvin & Dave and Alvin, Theodore & Dave respectively. The values inside the parenthesis are the maximum and minimum derivation of the time needed.

	A, T	A, D	D, T	A, T, D
kitchen A	stack & resources	resources	stack & resources	stack & resources
kitchen B	stack & resources	stack & resources	stack	resources

Figure 10: Best combination of transformation rules applied to the default plan resulting in a plan with the shortest time needed for setting the table.

mechanisms to autonomous robot control, in particular office delivery tasks for a robot without manipulators. Transformational planning, however, is particularly promising and challenging if the robots' tasks are the manipulation of objects. This is what TRANER does. Still, several of TRANER's methods for plan representation and specifying transformation rules are taken from McDermott (1992) and Beetz (2001) and have been further extended.

Discussions of specifying everyday activity incorporated and to be incorporated into TRANER can be found in Hammonds' et al. (1995) article on environment stabilization and Agre's and Horswill's (1997) work on lifeworld analysis.

Conclusions

In this paper we have proposed TRANER as a transformational planner for (still simulated) autonomous service robots performing complex everyday activity. Although the transformations TRANER performs are still quite limited we consider the results important for several reasons. First, TRANER demonstrates as a proof of concept that AI planning techniques can be successfully applied to autonomous robot control and show promise to substantially improve their performance. Second, improving the performance of service robots in our view requires planning mechanisms to reason through and revise plans that produce flexible and reliable long-term problem-solving behavior. Third, causal link planning, which is commonly considered to be general purpose planning, is only a small subset of the planning operations needed by service robots. Reasoning about activity restructuring, about the usage of partially described objects, about when and how to stabilize the environment (e.g. through cleaning up), are examples of planning techniques that are equally important. Fourth, we believe that a promising approach lies in the co-design of plan representations and plan transformations. Plan representations must make aspects of activity transparent and modular, so that we can state more general transformation rules.

The research described in this paper constitutes an important first step towards developing a computational model for plan-based control that enables service robots to substantially improve their performance doing training on the job.

References

Agre, P. E., and Horswill, I. 1997. Lifeworld analysis. *Journal of Artificial Intelligence Research* 6:111–145.

Beetz, M. 2001. Structured Reactive Controllers. *Journal of Autonomous Agents and Multi-Agent Systems. Special Issue: Best Papers of the International Conference on Autonomous Agents '99* 4:25–55.

Beetz, M. 2002. Plan representation for robotic agents. In *Proceedings of the Sixth International Conference on AI Planning and Scheduling*, 223–232. Menlo Park, CA: AAAI Press.

Bothelho, S., and Alami, R. 2000. Robots that cooperatively enhance their plans. In Parker, L. E.; Bekey, G. A.; and Barhen, J., eds., *5th International Symposium on Distributed Autonomous Robotic Systems (DARS)*, 55–68. Knoxville, Tennessee, USA: Springer.

Gerkey, B.; Vaughan, R. T.; and Howard, A. 2003. The Player/Stage Project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics (ICAR2003)*, 317–323.

Hammond, K. J.; Converse, T. M.; and Grass, J. W. 1995. The stabilization of environments. *Artificial Intelligence* 72(1-2):305–327.

Hammond, K. 1990. Explaining and repairing plans that fail. *Artificial Intelligence* 45(1):173–228.

McDermott, D. 1991. A Reactive Plan Language. Research Report YALEU/DCS/RR-864, Yale University.

McDermott, D. 1992. Transformational planning of reactive behavior. Research Report YALEU/DCS/RR-941, Yale University.

Simmons, R. 1988. A theory of debugging plans and interpretations. In *Proc. of AAAI-88*, 94–99.

Sussman, G. 1977. *A Computer Model of Skill Acquisition*, volume 1 of *Aritficial Intelligence Series*. New York, NY: American Elsevier.

Acknowledgements. The research reported in this paper is partly supported by the cluster of excellence COTESYS (Cognition for Technical Systems, `www.cotesys.org`).