Planning Domain + Execution Semantics: A Way Towards Robust Execution?

Štefan Konečný
Örebro University
sky@aass.oru.se

Sebastian Stock
Osnabrück University
sestock@uos.de

Federico Pecora, Alessandro Saffiotti
Örebro University
{fpa, asaffio}@aass.oru.se

Abstract

Robots are expected to carry out complex plans in real world environments. This requires the robot to track the progress of plan execution and detect failures which may occur. Planners use very abstract world models to generate plans. Additional causal, temporal, categorical knowledge about the execution, which is not included in the planner's model, is often available. Can we use this knowledge to increase robustness of execution and provide early failure detection? We propose to use a dedicated Execution Model to monitor the executed plan based on runtime observations and rich execution knowledge. We show that the combined used of causal, temporal and categorical knowledge allows the robot to detect failures even when the effects of actions are not directly observable. A dedicated Execution model also introduces a degree of modularity, since the platform- and execution-specific knowledge does not need to be encoded into the planner.

Introduction

Due to advancements in robotics, robots are expected to carry out tasks in increasingly challenging domains. One such domain is illustrated by the following scenario.

A new robot waiter has just been deployed in a modern restaurant. Its first instructions are to fetch a mug from the counter and bring it to the kitchen for cleaning.

However, a robot requires more detailed instructions in order to successfully carry out the task, e.g., in the form of a detailed list of actions to execute. If the robot is too far from the counter, it should drive closer to the counter before it attempts to grasp a mug.

Planning approaches (Ghallab, Nau, and Traverso 2004) model such dependencies through preconditions and effects: preconditions specify the situation in which the robot can execute the action (e.g., being at the counter before trying to pick up the object), effects capture the expectation about the situation after execution (e.g., the drive action will change the robot's position).

To execute a plan robustly, the robot has to carry out two closely related processes: track the progress of the execution and dispatch the next action when appropriate. We shall refer to them jointly as execution monitoring.

The first point is concerned with early detection of success or failure. Typically, expectations about nominal execution are compared with sensory data obtained during execution. A major discrepancy between the two results in a failure. This matching is not done all the time, but only in few critical points during execution. The plan's structure is commonly used to identify these points, which often correspond to missing preconditions and unachieved effects.

Assuming that the expectations used in execution monitoring are simply the preconditions and effects of actions has its limits. For instance, the effect of an action modeled in the planning domain may not be observable using sensors, although there may be other indirect effects of executing the actions that could be monitored. Additionally, since a plan is typically generated before execution is started, the planner operates on initial knowledge about the environment. For example, consider the task of getting a mug from the counter. If the initial knowledge specifies that there are multiple mugs on the counter, most planners would select one of these, e.g., `mug1`, and generate a plan to grasp this specific mug — we refer to this as *early commitment*. What would happen then if that particular mug is not found on the counter? The plan would fail in the absence of `mug1` — however, note that if there were other mugs on the counter, fetching any of them would fulfill the task.

This paper studies what forms of knowledge are useful for execution monitoring, and how this knowledge is utilized. Some types of knowledge can be accommodated into the planning domain without leading to a combinatorial explosion of the planning problem. When this is possible, we obtain plans that are more robust to certain types of failures. As we show in this paper, including categorical knowledge in the planning domain can be used to avoid failures resulting from *early commitment*. Knowledge of a non-categorical nature may also be necessary for determining success or failure of execution. For example, effects may manifest themselves *post factum*, or begin to hold during action execution. The particular temporal relations that should be verified may depend on the action, or on the perception traces, or both. This additional knowledge could be modeled in the planning domain — however, this may over-burden the planning process. More importantly, including execution-specific knowledge in the model used by the planner defeats the purpose of having a general purpose planner in the first place, as execution-level knowledge is often dependent on the particular robot platform used. Keeping the planning domain decoupled from a particular robot platform allows robot developers to re-use the planning solution across different platforms.

We propose to include knowledge useful for execution monitoring into a dedicated *Execution Model* (EM). This allows us to explicitly cast the execution monitoring problem as assessing consistency of execution traces with respect to this model — what we call *Consistency Based Execution Monitoring* (CBEM). In so doing, we encapsulate platform specific, execution-related knowledge into a separate module that can be changed when the platform or other lower-level properties of the domain change. EMs express the conditions under which execution is successful with respect to one or more formal semantics, e.g., temporal, categorical, causal. In this paper, we show how EMs can be used for dispatching, fault identification, and possibly fault repair.

Consistency Based Execution Monitoring

For the purposes of this paper, we assume that plans are composed of ground actions, i.e., instantiated operators. One of the elements of an Execution Model is a set of *causal relations*, which relate the ground predicates that constitute the preconditions and effects of a given ground action. For instance, the causal domain knowledge in the planner's model may state that the operator `drive(?from, ?to)` has precondition `robotAt(?from)` and effect `robotAt(?to)`. This allows us to infer the causal relation that states that the ground action `drive(entrance, counter)` depends on the ground predicate `robotAt(entrance)` and that its expected result is `robotAt(counter)`. We shall refer to all preconditions and effects of a given ground action a jointly as *the predicates of* a.

In addition to causal semantics of the execution, an EM may also be enriched with *temporal semantics*. Thes allow us to explicitly model the temporal relations that hold between specific actions, preconditions and effects. For instance, the robot may still be at the entrance for some time after the action has started, and is expected to be at the counter at the latest just before the action terminates.

To model the temporal semantics of the execution, we employ the notion of fluents:

Definition 1 *A* fluent *is a pair* $\psi = (p, [st, ft])$*, where*

- $p = P(t_1, \ldots, t_n)$ *is a ground atom with terms* $\{t_1, \ldots, t_n\}$*, where* P *is either a predicate symbol or an action symbol;*
- $[st, ft]$ *is an interval where* $st, ft \in \mathbb{N}$ *represent the start time and finish time of the fluent, respectively.*

Fluents allow us to represent the temporal extent of facts and actions. If no information is available about the start (finish) time, it takes the value undefined ($\perp$). We say that a fluent ψ is *open* if its start time is specified and its finish time is $\perp$. For instance, the fluent $(\texttt{robotAt(entrance)}, [10, \perp])$ represents the fact that the robot is at the entrance since time 10, and we do not know until when. This can be the result of observations, or a predicted state of the robot's location decided by the planner. A fluent whose start and end times are defined is said to be *closed*.

We model temporal relations between fluents as constraints in Allen's Interval Algebra (IA) (Allen 1984). These are the thirteen possible atomic temporal relations between intervals, namely "before" (`b`), "meets" (`m`), "overlaps" (`o`), "during" (`d`),"starts" (`s`), "finishes" (`f`), their inverses (e.g., $\texttt{b}^{-1}$), and "equals" ($\equiv$). We denote the set of atomic relations B_{IA}. For example, the constraint

$$\begin{array}{r}(\texttt{robotAt(entrance)}, [10, \perp]) \\ \{\texttt{b}\} \\ (\texttt{robotAt(counter)}, [35, \perp])\end{array} \tag{1}$$

models that the robot will leave the entrance before it reaches the counter. Note also that as a result of this constraint, we can infer the latest end time of the entrance fluent, which is 34. We can maintain an upper and lower bound for finish and start time of every fluent. These bounds are updated as a result of temporal constraint reasoning, an operation which can be performed in low-order polynomial time (Dechter, Meiri, and Pearl 1991; Floyd 1962).

In general, a temporal constraint C can be a disjunction of atomic relations, denoted by $C = \{c_1, \ldots, c_n\}$ with $c_i \in B_{\text{IA}}$ for $i = 1 \ldots, n$. In order to have tractable reasoning upon temporal constraints, we use *convex* IA relations (Vilain and Kautz 1986; Krokhin, Jeavons, and Jonsson 2003). Thus, we can determine whether a set of constraints is consistent in polynomial time. As we shall see, this facilitates the process of assessing whether an execution trace is consistent with respect to the temporal semantics given in the EM. Note that, while the adopted formalism allows us to impose relations between any two fluents, in this work we focus on constraints between an action and its predicates.

An execution model with temporal semantics can be used in conjunction with the causal semantics to detect failures that would be difficult to detect with either part of the model alone. For example, consider the action to pick up `mug1`. For a given robotic platform, it may be difficult to reliably detect by visual means that the mug is held, but it may be easy to detect that something is in the gripper — i.e., the gripper cannot close completely. This sensor information will typically become available during the execution of the `pickUp` action and before its completion; correspondingly, the effect `holding(mug1)` will be observed during the execution of the action as well. We can model this temporal relation between the action and its effect with the constraint

$$\begin{array}{r}\texttt{pickUp(mug1, counter)} \\ \{\texttt{o}\} \\ \texttt{holding(mug1)}.\end{array} \tag{2}$$

In general, we say that an *execution model* EM is a prescription of expected behavior in terms of one or more semantics (in the examples above, temporal semantics). *Consistency Based Execution Monitoring* can then be seen as the process of assessing the consistency of an execution trace with respect to an EM. The feasibility of this process depends entirely on the formalisms employed to represent the different types of knowledge in the model. In this paper, we focus on CaTeGo (CAusal, TEmporal, cateGOrical) execution models:

Definition 2 *A* CaTeGo execution model *is a tuple* $(F, R_{te}, R_{ca}, R_{go})$ *, where*

- *F is a set of fluents;*
- *R_{te} is a set of convex disjunctive temporal constraints between pairs of fluents in F;*
- *R_{ca} is a set of cause-effect relations between fluents in F;*
- *R_{go} is a mapping from a set of objects $\mathcal{O}$ to a set of categories $\mathcal{C}$.*

Cause-effect constraints have the form $(\psi_p \mathcal{P} \psi_a)$, meaning that predicate fluent ψ_p is a precondition for action fluent ψ_a; or the form $(\psi_a \mathcal{E} \psi_p)$, meaning that ψ_p is an effect of ψ_a.

The following example illustrates the concept of CaTeGo.

The Robot is standing at the counter, it has already observed the counter and identified mug1. Now it should grasp mug1.

The plan consists of the single action `pickUp(mug1)`. The only precondition of this action is `robotAt(counter)` and it has the single effect `holding(mug1)`.

$$R_{te} = \{(\texttt{robotAt(counter)}\ \{\texttt{m},\texttt{o},\texttt{f}^{-1},\texttt{d}^{-1}\}\ \texttt{pickUp(mug1)}), (\texttt{pickUp(mug1)}\ \{\texttt{o}\}\ \texttt{holding(mug1)})\}.$$

$$R_{ca} = \{(\texttt{robotAt(counter)}\ \mathcal{P}\ \texttt{pickUp(mug1)}), (\texttt{pickUp(mug1)}\ \mathcal{E}\ \texttt{holding(mug1)})\}.$$

$R_{go} : \mathcal{O} \rightarrow \mathcal{C}$ where $\mathcal{O} = \{\texttt{mug1}, \texttt{mug2}, \texttt{vase1}\}$ and $\mathcal{C} = \{\texttt{Mug}, \texttt{Vase}\}$, such that $R_{go}(\texttt{mug1}) = \texttt{Mug}$, $R_{go}(\texttt{mug2}) = \texttt{Mug}$, and $R_{go}(\texttt{vase1}) = \texttt{Vase}$.

The similarity between R_{ca} and R_{te} suggests that we can attach temporal semantics to causal relations, which we do in this work. In the next section we discuss which temporal relations could be used to model causal dependencies. This choice depends on the type of action, and it influences which faults can be detected by the temporal semantics of the EM.

Temporal Semantics for Execution Monitoring

Let us discuss how we use temporal semantics attached to causal relations to detect failures. For the sake of simplicity we start with atomic IA relations. Below, we use a to denote a generic action fluent, and use p (e) to denote a precondition (effect) fluent of a. We denote the start and finish time of a (resp: p, e) by a_s, a_f (resp: $p_s, p_f; e_s, e_f$).

The relation $(p\ \{\texttt{m}\}\ a)$ can be used to model an action a whose occurrence will disturb the state modelled by the precondition p. For example the robot cannot perform some actions while driving. We could model this with the predicate `robotStandingStill()` which could be a precondition of a driving action. The precondition fluent p should be closed immediately as a driving action starts ($p_f = a_s$). If the robot remains standing after the action has started, this may indicate that the actuator is delayed, malfunctioning or that the move is not possible in the environment (it is physically blocked). Alternatively it may be the perception which is failing, by not closing the `robotStandingStill()` fluent on time. In practice it may be unrealistic to expect no delay ($p_f = a_s$) and this situation can be modeled by a delay threshold t as $((a_s - p_f) < t)$. The same reasoning applies to modeling for effects with $(a\ \{\texttt{m}\}\ e)$. Immediately after the driving action has finished, the effect `robotStandingStill()` should be produced. Note that metric bounds can be added without loosing tractability (Kautz and Ladkin 1991).

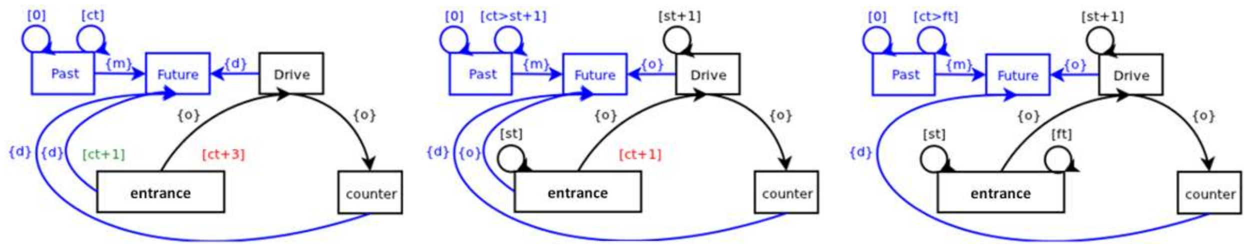
$(p\ \{\texttt{o}\}\ a)$ can be used to capture a precondition which will cease to be true during the execution of an action ($p_s < a_s < p_f < a_f$). An example would be `drive(table, counter)`, which has a precondition `robotAt(table)`. The robot is expected to leave the vicinity of the table (`robotAt(table)`) before it reaches the counter (`robotAt(counter)`). The failure to register the closing of the precondition before the action itself is closed indicates similar failures to those discussed above.

As already discussed, the usage of the relation (`pickUp(mug1)` $\{\texttt{o}\}$ `holding(mug1)`) allows us to detect whether `pickUp` has failed. The relation is violated if either `holding(mug1)` was not opened or it was opened and closed before the action has finished. As we shall discuss in the evaluation Section, the temporal semantics allows us to detect and distinguish two separate failures.

Some actions require a durative precondition in order to be executable. Execution is not supposed to alter this precondition. An example of this is `pickUp(mug1, counter)`. The robot must be close to `counter`, which is modeled by the precondition `robotAt(counter)`. Since `pickUp` does not modify the position of the robot ($p_s < a_s < a_f < p_f$), this causal relation can be modeled by (`robotAt(counter)` $\{\texttt{d}\}$ `pickUp(mug1)`). A similar case would be when the precondition is closed at the same moment as the action ($p_f = a_f$) corresponds to (`robotAt(counter)` $\{\texttt{f}\}$ `pickUp(mug1)`). The violation of either relation may indicate sensory error in detecting the closing of either the action or the precondition, or the presence of unknown environment altering activities.

Effects can be represented through $(a\ \{\texttt{b}\}\ e)$ which allows an arbitrarily long time to pass between finishing the action and observing the effect ($a_f < e_s$). On one hand this gives us an execution that is robust against delays, on the other it makes it difficult for the EM to decide whether the action has succeeded or not.

If no atomic IA relation can adequately express the causality between an action and its predicate, we may opt for a (convex) disjunction of relations (without an increase of computational complexity). Thus, Execution Monitoring will be more tolerant, as it describes multiple possible execution traces, but as a result it will detect fewer failures. A dedicated EM allows us to balance this trade off. Interestingly, the decision whether to employ a more general/specific model can be made for each action individually.



(a) Fluent before opening (b) Fluent opened (c) Fluent after closing

Figure 1: Updating the precondition fluent entrance. The predicates in the figure are simplified for clarity. ct,st,ft in the figures correspond to t_c,t_s,t_f in the text. The lower bound on t_s (green) and t_f (red) are displayed only for entrance. The circular arrows indicate the time of observation

Using Execution Models During Execution

In this work, we assume that grounded preconditions and effects of ground actions in the plan are available (they can be easily obtained from a planner and the planning domain).

Next, we construct an Execution Network (EN), which integrates: expectations encoded in the plan, i.e., the sequence of actions and their predicates; and semantics of the EM, i.e., temporal relations expressing causality between each action and its predicates. If this EN is consistent with the information observed during execution, we consider the execution successful, otherwise we detect a failure.

The nodes in the EN correspond to fluents representing actions and their monitored predicates. The temporal relations specified in the EM are reflected in the EN as edges between the nodes. Additional temporal relations expressing expectations about future events are also part of the EN and we shall discuss them shortly.

An example of EN is shown is shown in Figure 1. The concepts of Past and Future are modelled as fluents to maintain an internal representation of the execution time. The start time of Past is the start time of the execution, whereas the finish time of Past represents the current time t_c. The two fluents are connected through (Past $\{\mathtt{m}\}$ Future), which assures that both the finish time of Past and the start time of Future equals to the current time.

As we express causality through temporal relations, the resulting network consists of fluents interconnected with temporal relations. Such a network can be transformed into a Simple Temporal Problem — STP (Dechter, Meiri, and Pearl 1991). A STP maintains a lower and an upper bound for each time point (start or finish time of a fluent). The temporal relations specify constraints on these time points. Each time new temporal information is obtained from the execution, i.e., when a fluent is opened or closed, this information is updated through temporal reasoning. Next, we show how information from execution is integrated into the EN.

The Execution and Monitoring Process

The Execution Monitor is notified each time a fluent relevant to execution is produced (e.g., by the sensors). Each relevant fluent ψ which has not been observed yet, e.g., an expected future action or predicate, is connected to the Future by (ψ $\{\mathtt{d}\}$ Future). As a result the earliest start time of such fluents is $t_c + 1$ (see Figure 1a).

Let's imagine a relevant fluent ψ is opened with a specified start time ψ_s and unspecified finish time ψ_f. First the modeled current time t_c is updated to ψ_s.

The STP maintains a time bound $[s_e, s_l]$ for the start of ψ consisting of earliest possible start time s_e and latest possible start time s_l. These bounds encode the temporal knowledge in the EM, and therefore express the earliest and latest possible start and end times of a planned fluent according to the EM. If $\psi_s < s_e$ then ψ was observed too early; if $s_l < \psi_s$, it is observed too late. In either case the observation of ψ at time point ψ_s is inconsistent with our expectations about execution, and a failure is detected through constraint reasoning.

If $s_e \leq \psi_s \leq s_l$, the observed fluent ψ is consistent with our expectation. The bounds for the start time of ψ are updated to $[\psi_s, \psi_s]$. If the start time is restricted through some temporal relations, this information is propagated through these relations.

After a fluent ψ is opened the connection to Future is also changed from (ψ $\{\mathtt{d}\}$ Future) to (ψ $\{\mathtt{o}\}$ Future) (Figure 1b). If ψ is the last precondition of an action which was not open yet, the Execution Monitor dispatches this action (in time $\psi_s + 1$), as all its preconditions are fulfilled now.

An analogous process is applied when a fluent ψ is closed. This process is concerned with the finish time ψ_f of the fluent and it removes the relation (ψ $\{\mathtt{o}\}$ Future) from the EN (Figure 1c).

When an inconsistency is detected in the EN, it is a result of a discrepancy between the plan, the EM (the expectation about the execution of the given plan) and the execution trace (observation from the real execution). This signals a failure in the execution. Interestingly, departures from expectations which do not impact the execution of the current plan will not result in an inconsistency.

Planning and CBEM

Classical planners take an initial and a goal state as input and find a sequence of actions leading from the initial to the goal state. These states can be represented as sets of ground predicates and actions are defined as operators. HTN-planners (Erol, Hendler, and Nau 1994) use a different approach. They are given tasks that have to be fulfilled and the planner decomposes complex tasks into subtasks using *methods* modeled inside the planner's domain description. Methods are applied until only primitive (directly executable) tasks are left. A method consist of a task name, a set of preconditions and a set of partially ordered subtasks.

In both classical planning and HTN-planning, the planner maintains an internal state of the world and reasons with the ground predicates about how operators take effect on the world and whether they can be applied. To update the internal state, the operators have to be fully instantiated and therefore they are also fully instantiated in the resulting plan.

In our scenario the robot starts far from the counter and cannot see what is on it. Therefore, the planner has to rely on its initial world state or a memory concerning the objects on the counter. If either contains information about multiple mugs to choose from, a typical off-the-shelf planner would generate a plan to move to the counter and grasp a specific mug, e.g., `mug1`. In our scenario this plan is over-specified, since any mug from the counter would be sufficient.

In a dynamic environment this may lead to an unnecessary plan failure. If the robot arrives to the counter and `mug1` is not there (anymore), the plan will fail. This would be the case even if there were another `mug2` on the counter.

To overcome this limitation, we *lift* parts of the plan. With lifting, we abstract from the specific instantiations of objects used by the planner back to more abstract dummy instances. The Execution Monitor will decide which specific objects should be used in place of the dummy one based on information obtained during execution.

To support lifting, the planning domain must be modified to allow the planner to instantiate parameters with these dummy objects instead of specific instances. These changes can be done completely inside the domain description without changing the planner itself. For example, the operator `createLiftedMug(?liftedmug)` adds a new dummy instance to the planner's internal state and annotates it using the predicate `lifted`. The operator `liftMug(?mug, ?liftedmug)` deletes the instance of a mug. We can also include a method that lifts all the mugs which are placed on a specific area as a single lifted instance. In its decomposition it uses the two operators above to create a new instance `liftedmug1` and goes through all mugs that are on the given area and applies the `liftMug` operator.

One might argue that the planning domain could be modeled in a different way, so that the operator for grasping does not need an instance as a parameter but just a type of an object. This would have some drawbacks. First, a complex plan could involve the manipulation of several mugs, e.g., the robot could be asked to get a clean mug from the counter and put it on a tray, drive to a first table to put a dirty mug on its tray and finally drive to a second table to place the first mug onto it. With the lifted instances, the assertion of mugs to instances in the plan will be maintained, whereas this would not be possible in the alternative solution. Second, there can be other tasks where the robot has to bring one specific mug. In our approach this can be modeled in the same domain with the same kind of representations, whereas otherwise a completely different representation and therefore different domains for different tasks would be needed.

To verify at execution time which mugs are on the table and can be grasped, perception actions are planned and executed directly before the grasp action. They are important for the consistency based execution monitoring to check whether the preconditions of the `pickUp` action are fulfilled, e.g., that the chosen mug is on the table, or to re-instantiate a dummy mug to a real instance.

Evaluation

In this section we will demonstrate that an Executive Monitor equipped with our Execution Model can execute our scenario in a simulated environment (shown in Figure 2). Our setup is a one-to-one model of the physical laboratory setup used in the RACE project.[1] Although the case study reported below has been run in simulation, plans similar to the ones reported here are routinely executed in the physical setup within the project. The simulation supports our claims that: categorical knowledge enables us to overcome the *early commitment* problem, causal/temporal knowledge enables us to deduce the outcome of the action `pickUp` from indirect observation. Using an EM can therefore increase the robustness of plan execution under realistic (albeit simulated) conditions.

Implementation details

Our approach has been integrated into the architecture (Rockel et al. 2013) of the project RACE, in which a blackboard is linked to an OWL-ontology and keeps track of the internal state of the robot. This state is posted by specialized modules for proprioception, perception, and initial knowledge (e.g., poses of tables and objects). All the information in the blackboard is expressed in form of fluents. All modules communicate through the ROS (Quigley et al. 2009) middleware. As a planner we use the well known HTN-planner SHOP2 (Nau et al. 2003). The EN is implemented in the MetaCSP-framework (Pecora 2013).

Setup

All experiments have been run in the GAZEBO simulator with the standard PR2 platform. Perception actions obtain information about objects from GAZEBO. This means that we get perfect object recognition and identification.

The simulated environment consists of a counter and two tables. Three graspable objects are placed on the counter: two mugs, named `mug1` and `mug2`, and a vase `vase1`. The robot starts at the opposite side of the room, from where it cannot detect the objects on the counter. The poses of the counter, tables and three objects are provided in the robot's initial knowledge, simulating that the robot has seen the objects the last time it has been at the counter.

The PR2 has an omnidirectional base, a torso with adjustable height, a head equipped with a stereo camera and a Kinect sensor, and two arms, each equipped with a gripper. The choice of this specific platform provides some (soft) constraints on the execution: the torso should be up to provide a better view of graspable objects and to facilitate manipulation, while it should be down when the robot drives

[1]RACE is a EU FP7 project (www.project-race.eu).

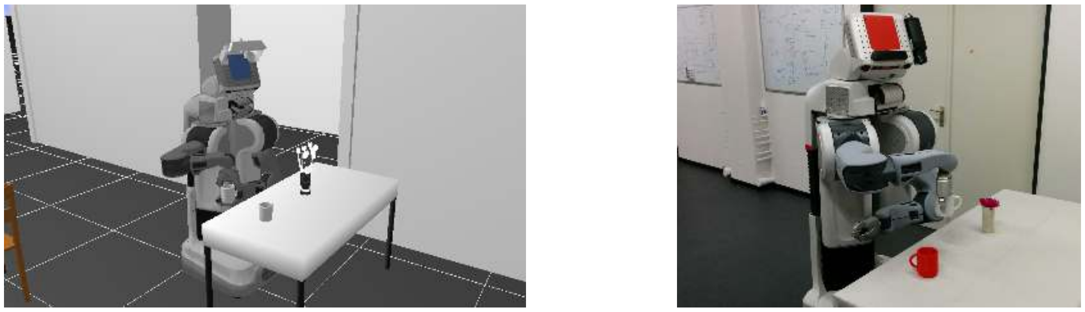


Figure 2: The simulated environment in Gazebo (left), and the corresponding physical laboratory setup (right).

in order to increase its stability. The arms should be in a specific position during driving (arms tucked) to avoid occlusion of sensors and collisions.

The modeled planning domain captures these dependencies and the generated plan consists of actions that can be executed directly on a PR2 robot. These include tucking the arms, moving the torso, driving to an area, driving close to the counter, observing the counter and grasping an object. The domain also includes a `driveBlind` action for approaching furniture without obstacle avoidance. The `drive` and `driveBlind` actions are related to their respective preconditions and effects through a $\{\texttt{o}\}$ relation; the same is true for the effects of `pickUp`. For all other actions a we have chosen a very relaxed model: their preconditions p and effects e are related through constraints $(p\ \{\texttt{m},\texttt{o},\texttt{f}^{-1},\texttt{d}^{-1}\}\ a)$ and $(a\ \{\texttt{b},\texttt{m},\texttt{o}\}\ e)$, respectively.

Case Study

In our experiments the action `pickUp(mug)` failed most often. Typically, either a feasible grasp for the mug could not be found or it could not be executed. The executed low level action would report a failure. To detect a successful grasp is more difficult and we rely on the additional knowledge in the EM for this. The same is true for detecting that the mug has slipped out of the grasp. We will discuss both cases in more detail shortly. In addition, we will describe how information in EM enables us to execute lifted plans. But first of all let us consider the base case: a ground plan.

Baseline – The ground plan At planning time, the planner chooses one of the previously known mugs on the counter, `mug1`, and commits to grasp it. The produced ground plan, shown in Listing 1, consists of tucking the arms, driving to the counter, lifting the torso, bringing the arms to appropriate position for grasping, observing the table and performing the grasp itself.

```
tuckArms(tuckedposture, tuckedposture)
drive(entrance,counter)
moveTorso(torsoupposture)
tuckArms(tuckedposture, untuckedposture)
moveArmToSide(rightarm)
driveBlind(counter)
updateObjectsOnArea(counter)
pickUp(mug1)
driveBlind(counter)
```

Listing 1: Ground plan for grasping mug1.

In the basic scenario, the ground plan is executed until the action `updateObjectsOnArea(counter)`. At this stage there are two possibilities. If `mug1` is detected, the observation action does not modify the EN and the execution continues with `pickUp(mug1)`. Note that the information acquired (real position of all objects, their graspability) increases the likelihood of a successful execution. If `mug1` is not among the objects detected on the counter (e.g., it was removed during execution), the fluent `onCounter(mug1)` representing that `mug1` is on the counter is committed to the past by adding to the EN the constraint $(\texttt{onCounter(mug1)}\ \{\texttt{d}\}\ \text{Past})$. Since this fluent is a precondition of `pickUp(mug1)`, its temporal extent is supposed to overlap with the one of `pickUp(mug1)`, but the latter cannot be started since its preconditions do not hold. This situation results in a temporal inconsistency, which shows that the plan has failed and should be aborted.

Using Categorical Knowledge for Lifted Planning The lifted plan in Listing 2 differs in adding three initial lifting operators and

In the second case, the robot generates the lifted plan given in Listing 2. Compared to the previous, this adds three initial lifting operators, `createLiftedMug` and `liftMug`, and replaces `mug1` in `pickUp` with the dummy object `liftedMug1`. Notice that the planner specifically lifts only the mugs on the counter, since only those are relevant for our task. The lifting operators are evaluated only internally in the planner and the executor.

```
createLiftedMug(liftedmug1, counter)
liftMug(mug1, liftedMug1)
liftMug(mug2, liftedMug1)
tuckArms(tuckedposture, tuckedposture)
drive(entrance,counter)
moveTorso(torsoupposture)
tuckArms(tuckedposture, untuckedposture)
moveArmToSide(rightarm)
driveBlind(counter)
updateObjectsOnArea(counter)
pickUp(liftedMug1)
driveBlind(counter)
```

Listing 2: Plan for grasping a mug with lifted mugs.

The Lifted plan is executed until the observation action, which produces a list of the objects detected on the counter. The categorical knowledge R_{ca} from the EM is used to identify which objects are eligible for grasping. Next, the pick up action planned in the EN is modified to grasp a concrete object chosen in the previous step.

To decide which information should be lifted, and left for instantiation in execution, is not trivial. This is part of the broader issue of which knowledge should be available to the planner, which to the Execution Model, and which to both.

Temporal Knowledge for Failure detection After identifying which object to grasp we should verify the outcome of the action `pickUp(mug1)`. We do this by checking the effect relation (`mug1` {o} `holding(mug1)`). If the fluent `holding(mug1)` is not opened before the action finishes, we detect a failure. This is the case when `mug1` directly slipped away from the gripper and remained on the counter.

We also detect a failure if the fluent `holding(mug1)` is closed before the action finishes. This corresponds to the case when `mug1` slipped away from the grasp (as a result gripper has closed, and so has the effect `holding(mug1)` after it has successfully been lifted from the counter.

Related Work

The first attempt to explicitly represent knowledge used for Execution Monitoring was PLANEX (Fikes 1971). PLANEX converts a STRIPS plan into a tabular structure (*triangle table*) which captures the causal dependencies (preconditions-action-effects). When an action is executed, the outcome is compared to the expectations. Therefore, PLANEX requires direct observability of effects. In addition to causal knowledge our Execution Model includes also temporal and categorical knowledge, allowing us to take into account action effects which are not directly observable.

In temporal planning and scheduling (Gallien, Ingrand, and Lemai 2004; Fratini, Pecora, and Cesta 2008; Barreiro et al. 2012; Di Rocco, Pecora, and Saffiotti 2013) temporal and resource knowledge about execution is used to detect failure. The planner reasons about typical durations and delays between actions in addition to their dependencies. Failure to adhere to these temporal expectations made in planning time results in replanning. This cycle of planning (including scheduling), execution monitoring and replanning (and rescheduling) is referred to as *continuous planning*. One well known architecture for continuous planning is T-Rex (McGann et al. 2008), which is based on a hierarchical network of so-called reactors, each of which monitors a task at a different level of abstraction. Reactors communicate with each other by posting goals and sharing information. The reactor which detects a problem in task execution will inform the reactor which posted the task. The task posting reactor has to deliberate how to proceed and post a new task.

Our work departs from the ones above in that in our approach we decouple the execution model from the planning domain. Our execution model can be richer, e.g., by incorporating also categorical knowledge. We speculate that other types of knowledge can be included in the future. Finally, we represent explicitly all the information relevant to execution into the EN prior to the execution. By contrast, in T-Rex this information emerges from the interaction between the reactors during the execution.

The integration of perception actions into plan generation, to overcome the closed world assumption presumed by most HTN planners, is presented in (Weser, Off, and Zhang 2010). If some relevant property of the environment is not known at planning time, a perception action is included into the plan to observe it directly. Predicted results of this action are used in the planning process. If the results are not observed, replanning is invoked. We also use perception actions in our scenario, to verify information obtained in the planning domain. Note that we can use categorical knowledge to avoid replanning, as long as there is a mug on the counter.

(Awaad, Kraetzschmar, and Hertzberg 2013) propose to use functional affordances and conceptual similarity to find appropriate object categories as replacements for missing objects. For example, for the task of watering plants, the task would normally fail if there is no watering can available. Functional affordances and conceptual similarities reveal that a watering can could be replaced by a tea pot. This replacement can occur at planning time or during execution. This is analogous to our use of categorical knowledge to avoid the *early commitment* problem.

(Bouguerra, Karlsson, and Saffiotti 2008; Galindo and Saffiotti 2013) also propose elaborate usage of categorical knowledge for Execution Monitoring. In (Bouguerra, Karlsson, and Saffiotti 2008) the category of the room in an apartment is identified based on the equipment therein. The reasoning behind is that a living room will typically contain a TV set but not a washing machine. This knowledge can be used for indirect verification of the effects of an action, e.g., drive to the living room. Thus, fault detection can be based on categorical knowledge, in addition to causal and temporal knowledge as considered here.

(Galindo and Saffiotti 2013) propose to use categorical knowledge to achieve goal autonomy. Consider the categorical knowledge that milk is a perishable good, and perishable goods should be stored in the fridge. If the robot finds a box of milk on the table, it could use this knowledge to generate a task to bring the milk to the fridge. This work demonstrates the usefulness of categorical (in its many forms) and temporal knowledge for execution monitoring. Our EM allowed us to combine causal, temporal, and categorical knowledge and we believe that future research in this direction has great potential.

A common way to execute and monitor a plan in ROS is by using a Finate State Machine architecture (Bohren et al. 2011). Each state can correspond to a simple action or a composition of simple states. The user has to define the states, the transitions and information flow between the states. The execution information is implicit in states and transitions. By contrast, our EN represents the causal/temporal relations explicitly. It also takes care of propagating (temporal) information between fluents and maintains the internal representation of time.

Conclusions

In this work we proposed the use of rich semantic knowledge for plan execution monitoring. To this end we have proposed a dedicated Execution Model containing the causal knowledge from the planning domain, as well as two additional forms of knowledge: temporal and categorical.

The causal knowledge is used to represent dependencies between preconditions, actions and their effects. We have also shown how to use the interplay of causal and temporal knowledge to detect effects which are difficult to observe directly, like a mug slipping away from grasp. In general, fine-grained execution monitoring can be achieved by maintaining both expectations from the execution model and fluents deriving from sensors in a common execution network, which can be tuned to match the execution semantics of the specific robotic platform. In addition, categorical knowledge allows to execute lifted plans, to avoid problems resulting from *early commitment* at planning time. In these plans the objects to grasp are abstracted away and replaced by a placeholder for their category. The Execution Model allows to instantiate this placeholder to a concrete physical objects at execution time.

The use of Execution Models opens the issue of which knowledge to consider in planning, which in execution, and which in both. Designing the interaction between Planner and Execution Model remains open. Also, the information in the Execution Network could be used to guide sensing, or to improve it by informing the perceptual processes. We will address these issues in future research.

Acknowledgment

This work is supported by the EC 7th Framework Program under Project RACE (Robustness by Autonomous Competence Enhancement, grant no. 287752). Comments by the anonymous referees greatly helped to improve this paper.

References

Allen, J. 1984. Towards a general theory of action and time. *Artificial Intelligence* 23(2):123–154.

Awaad, I.; Kraetzschmar, G. K.; and Hertzberg, J. 2013. Affordance-Based Reasoning in Robot Task Planning. In *Planning and Robotics (PlanRob) Workshop ICAPS-2013*.

Barreiro, J.; Boyce, M.; Do, M.; Frank, J.; Iatauro, M.; Kichkaylo, T.; Morris, P.; Ong, J.; Remolina, E.; Smith, T.; et al. 2012. Europa: A platform for AI planning, scheduling, constraint programming, and optimization. In *Int Competition on Knowl. Eng. for Planning and Scheduling (ICKEPS)*.

Bohren, J.; Rusu, R.; Gil Jones, E.; Marder-Eppstein, E.; Pantofaru, C.; Wise, M.; Mosenlechner, L.; Meeussen, W.; and Holzer, S. 2011. Towards autonomous robotic butlers: Lessons learned with the PR2. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, 5568–5575.

Bouguerra, A.; Karlsson, L.; and Saffiotti, A. 2008. Monitoring the execution of robot plans using semantic knowledge. *Robotics and Autonomous Systems* 56(11):942–954.

Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal constraint networks. *Artif. Intell.* 49(1-3):61–95.

Di Rocco, M.; Pecora, F.; and Saffiotti, A. 2013. When robots are late: Configuration planning for multiple robots with dynamic goals. In *IEEE/RSJ Int Conf on Intelligent Robots and Systems (IROS).*

Erol, K.; Hendler, J.; and Nau, D. 1994. HTN Planning: Complexity and expressivity. In *In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, 1123–1128. AAAI Press.

Fikes, R. E. 1971. Monitored execution of robot plans produced by STRIPS. Technical Report 55, AI Center, SRI International, 333 Ravenswood Ave, Menlo Park, CA 94025.

Floyd, R. 1962. Algorithm 97: Shortest path. *Communications of the ACM* 5(6):345–348.

Fratini, S.; Pecora, F.; and Cesta, A. 2008. Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. *Archives of Control Sciences* 18(2):231–271.

Galindo, C., and Saffiotti, A. 2013. Inferring robot goals from violations of semantic knowledge. *Robotics and Autonomous Systems* 61(10):1131–1143.

Gallien, M.; Ingrand, F.; and Lemai, S. 2004. Robot actions planning and execution control for autonomous exploration rovers. In *Plan Execution: A Reality Check Workshop at ICAPS-2004*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Kautz, H. A., and Ladkin, P. B. 1991. Integrating metric and qualitative temporal reasoning. In *Proc of the National Conf on Artificial intelligence (AAAI)*, 241–246.

Krokhin, A.; Jeavons, P.; and Jonsson, P. 2003. Reasoning about temporal relations: The tractable subalgebras of Allen's interval algebra. *J. of the ACM* 50(5):591–640.

McGann, C.; Py, F.; Rajan, K.; Ryan, J.; and Henthorn, R. 2008. Adaptive control for autonomous underwater vehicles. In *Proc. of the 23rd Nat. Conf. on Artif. Intell. (AAAI).*

Nau, D. S.; Au, T.-C.; Ilghami, O.; Kuter, U.; Murdock, J. W.; Wu, D.; and Yaman, F. 2003. SHOP2: An HTN planning system. *J. Artificial Intell. Research* 20:379–404.

Pecora, F. 2013. The Meta-CSP framework: a Java API for Meta-CSP based reasoning. http://metacsp.org.

Quigley, M.; Conley, K.; Gerkey, B. P.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; and Ng, A. Y. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.

Rockel, S.; Neumann, B.; Zhang, J.; Dubba, K. S. R.; Cohn, A. G.; Š. Konečný; Mansouri, M.; Pecora, F.; Saffiotti, A.; Günther, M.; Stock, S.; Hertzberg, J.; Tomé, A. M.; Pinho, A. J.; Lopes, L. S.; von Riegen, S.; and Hotz, L. 2013. An ontology-based multi-level robot architecture for learning from experiences. In *Designing Intelligent Robots: Reintegrating AI II, AAAI Spring Symposium*.

Vilain, M., and Kautz, H. 1986. Constraint propagation algorithms for temporal reasoning. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, 377–382.

Weser, M.; Off, D.; and Zhang, J. 2010. HTN robot planning in partially observable dynamic environments. In *Int. Conf. on Robotics and Automation (ICRA)*, 1505–1510.