Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16)

Continual Planning in Golog

Till Hofmann, Tim Niemueller, Jens Claßen, and Gerhard Lakemeyer
Knowledge-Based Systems Group
RWTH Aachen University, Germany
{hofmann, niemueller, classen, lakemeyer}@kbsg.rwth-aachen.de

Abstract

To solve ever more complex and longer tasks, mobile robots need to generate more elaborate plans and must handle dynamic environments and incomplete knowledge. We address this challenge by integrating two seemingly different approaches – PDDL-based planning for efficient plan generation and GOLOG for highly expressive behavior specification – in a coherent framework that supports continual planning. The latter allows to interleave plan generation and execution through assertions, which are placeholder actions that are dynamically expanded into conditional sub-plans (using classical planners) once a replanning condition is satisfied. We formalize and implement continual planning in GOLOG which was so far only supported in PDDL-based systems. This enables combining the execution of generated plans with regular GOLOG programs and execution monitoring. Experiments on autonomous mobile robots show that the approach supports expressive behavior specification combined with efficient sub-plan generation to handle dynamic environments and incomplete knowledge in a unified way.

Introduction

Imagine a domestic service robot with the task to clean up the dining table by fetching cups, putting clean ones back on the shelf and dirty ones into the dishwasher. This task can be solved using a GOLOG (Levesque et al. 1997) program stating that while cups are left on the table fetch one, sense whether it is clean, and put it at the appropriate place. A different approach is taken in planning where a planner such as FASTDOWNWARD (Helmert 2006) determines a sequence of actions to clean the table. GOLOG supports incomplete knowledge and sensing (e.g. whether cups are dirty), but current implementations are very inefficient when it comes to pure planning, whereas planning is efficient but needs complete knowledge. Real-world execution furthermore requires a control loop for maintaining a world model, incorporating sensing results, and monitoring action execution, which planning-based systems typically implement in an ad-hoc fashion. To alleviate some of these issues, GOLOG and PDDL planning have been integrated such that a GOLOG program could call an embedded PDDL planner for sub-tasks (Claßen et al. 2012). However, this earlier approach could cope with incomplete knowledge only in a limited fashion. While sensing could be used in other parts of the GOLOG program, planning subtasks required the agent to possess complete information about (the necessary part of) the world state. This excludes a large class of problems where it is possible to come up with a major part of a plan beforehand, but that may contain smaller parts to be filled later through sensing. In our example, the agent may generate an overall plan for cleaning the table, but the decision on where to put an individual cup depends on whether it is dirty, which is subject to sensing. *Continual planning* (Brenner and Nebel 2009) addresses this issue by interleaving planning and plan execution, allowing the agent to handle incomplete knowledge even within a planning problem.

In this paper, we extend GOLOG with continual planning, which so far is only supported in some PDDL-based systems. We define a new GOLOG action type *assertion*, which is a placeholder for conditional sub-plans that depend on yet missing information. We define Know-If fluents for representing incomplete knowledge, where a Know-If fluent F_{KIF} is true iff the value of the fluent F is known. The evaluation shows competitive and even better performance than PDDL-based continual planning alone. A particular benefit of our approach is that we can combine GOLOG control structures with efficient planning to support incomplete knowledge in both parts. Moreover, we obtain a framework that integrates all parts of a robot task in GOLOG, including the top-most execution loop that queries the planner and then executes and monitors such plans. Using an extended GOLOG execution monitoring makes the system more robust and allows to deal with dynamic environments.

In the following section, we introduce GOLOG, PDDL and continual planning. Next, we define our approach to continual planning by means of assertions and describe our representation of incomplete knowledge. Finally, we evaluate the approach for domestic and logistics service robots.

Related Work

The Situation Calculus and GOLOG

The Situation Calculus (McCarthy 1963; Reiter 2001) is a first-order logic which allows to represent world states as first-order terms called *situations*. Relations (functions)

which may change from situation to situation are called *relational (functional) fluents*. Situations are results of actions, the situation after doing action a in situation s is denoted as $do(a, s)$. The initial situation S_0 and the action preconditions and effects are defined in a *basic action theory* (BAT). *Precondition axioms* are of the form $Poss(\alpha(\vec{x}), s) \equiv \Pi_\alpha(\vec{x}, s)$ where $\Pi_\alpha(\vec{x}, s)$ is a first-order formula. *Successor state axioms* (SSA) for relational fluents F are of the form $F(\vec{x}, do(a, s)) \equiv \gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s)$ where $\gamma_F^\pm(\vec{x}, a, s)$ is a first-order formula describing whether action a causes fluent F to be true (false). As an example, $Poss(goto(l), s) \equiv \neg robot\_at(l, s)$ states that the action $goto(l)$ is executable in situation s iff the robot is not currently at l; $robot\_at(l, do(a, s)) \equiv a = goto(l) \vee robot\_at(l, s) \wedge \neg\exists l'\, a = goto(l')$ states that after executing action a, the robot is at location l if a is the action $goto(l)$ or if the robot is at l in situation s and is not going anywhere else. A BAT includes foundational axioms for situations and defines a relation $<$ on situations, where $s_1 < s_2$ means s_2 results from s_1 by executing an action sequence.

GOLOG (Levesque et al. 1997) is a high-level programming language based on the Situation Calculus. It offers imperative programming constructs such as sequences of actions and iteration as well as nondeterministic branching. The semantics of GOLOG can be specified in terms of transitions (De Giacomo et al. 2009), which describe single steps of computation between configurations of the program. In GOLOG, configurations are of the form (δ, s) where δ is the remaining program and s is the current situation. $Trans(\delta, s, \delta', s')$ describes the transition between two configurations (δ, s) and (δ', s'); the predicate $Final(\delta, s)$ specifies configurations (δ, s) where the computation is completed. As an example, if the program is a single action a and the agent is currently in situation s, the transition to the next configuration is defined as $Trans(a, s, \delta', s') \equiv Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s)$.[1]

Multiple dialects of GOLOG exist: CONGOLOG (De Giacomo, Lespérance, and Levesque 2000) adds concurrency and interrupts to GOLOG. INDIGOLOG (De Giacomo et al. 2009) executes programs on-line with a search operator for off-line execution, and adds sensing actions, which are actions that determine the value of a certain fluent. READYLOG (Ferrein and Lakemeyer 2008) extends INDIGOLOG with passive sensing.

With sensing actions, it is desirable to be able to express the agent's knowledge, which can be described using the notion of *possible worlds* (Scherl and Levesque 1993). The binary relation $K(s', s)$ describes that as far as the agent knows, it might be in situation s' when in s. Using K, one can define the predicate $Knows(P, s)$, which holds iff P is known to be true in situation s, and the predicate $Kwhether(P, s)$ which holds iff the value of P is known:

$$Knows(P, s) \stackrel{def}{=} \forall s' K(s', s) \supset P[s']$$

$$Kwhether(P, s) \stackrel{def}{=} Knows(P, s) \vee Knows(\neg P, s)$$

[1] We write $a[s]$ for the result of restoring the situation argument to any fluents mentioned by the action term a. Similar for formulas.

As a different approach, Knowledge fluents (Demolombe and del Pilar Pozos Parra 2000) extend the Situation Calculus with a modal operator K, defining fluents $KP(s)$ meaning "P is known to be true in situation s". For each ordinary relational fluent F, they define knowledge fluents KF and $K\neg F$ by successor knowledge axioms (SKA):

$$\begin{aligned} &KF(\vec{x}, do(a, s)) \equiv \\ &\quad \gamma_{KF}^+(\vec{x}, a, s) \vee KF(\vec{x}, s) \wedge \neg\gamma_{KF}^-(\vec{x}, a, s) \\ &K\neg F(\vec{x}, do(a, s)) \equiv \\ &\quad \gamma_{K\neg F}^+(\vec{x}, a, s) \vee K\neg F(\vec{x}, s) \wedge \neg\gamma_{K\neg F}^-(\vec{x}, a, s) \end{aligned}$$

Analogously to SSAs, $\gamma_{KF}^\pm(\vec{x}, a, s)$ (or $\gamma_{K\neg F}^\pm(\vec{x}, a, s)$) defines all possibilities to change the knowledge fluent KF (or $K\neg F$) to true or false. As an example, after *cup* has been sensed on *table*, $Kat(cup, table)$ holds. If the agent moves away, the cup may be removed by another agent, thus $\gamma_{Kat}^-(cup, table, goto(kitchen), s)$ holds and $Kat(cup, table)$ is false. SSAs can be translated to SKAs (Petrick and Levesque 2002). The SSA of F is translated to the SKA of KF by defining $\gamma_{KF}^\pm(\vec{x}, a, s) \equiv (\gamma_F^\pm)^K(\vec{x}, a, s) \vee \xi_F^\pm(\vec{x}, a, s)$, where $(\gamma_F^\pm)^K$ is structurally identical to $\gamma_F^\pm$ with the exception that every fluent literal P is syntactically replaced by KP and $\xi_F^\pm(\vec{x}, a, s)$ states whether a is a knowledge producing action for the fluent literal $(\neg)F$. In general, the K fluent is more expressive than knowledge fluents. However, under certain restrictions, both approaches are equivalent (Petrick and Levesque 2002).

Planning

Planning describes the problem of finding a sequence of *actions* to reach a certain world state, called the *goal state*, from an *initial state*. Actions are described by their *preconditions* and *effects*. Depending on the formalism, there are restrictions on how the world state, the preconditions, and the effects can be represented. PDDL (McDermott et al. 1998) is a family of planning formalisms which provides a standard language for planning problems. It allows STRIPS-like actions (Fikes and Nilsson 1972), but also allows features such as conditional effects and existential and universal quantification. ADL (Pednault 1989) extends STRIPS by typed objects, disjunctive preconditions, an equality predicate, quantified preconditions, and conditional effects. It can be described as a subset of PDDL. PDDL2 (Fox and Long 2003) extends PDDL with numeric fluents and expressions, durative actions, and plan metrics.

FF (Hoffmann and Nebel 2001) is a heuristic planner which uses forward state space search and a heuristic which ignores delete effects. FF was originally designed for the STRIPS subset of PDDL but has been extended to ADL. FASTDOWNWARD (Helmert 2006) is a PDDL planner which supports the ADL fragment among others. It uses the causal graph heuristic, which splits up the problem into independent sub-problems. TFD (Eyerich, Mattmüller, and Röger 2012) extends FASTDOWNWARD to PDDL2.

While planning is possible in GOLOG, it is generally not feasible (Claßen et al. 2008). Instead, the BAT and the goal can be translated to ADL, the planning problem solved with

a PDDL planner, and the solution translated back to the Situation Calculus (Claßen et al. 2007). For this purpose, an operator $Plan$ is defined, where $Trans(Plan(G, A), s, \delta', s)$ holds iff δ' is the solution for the planning problem with goal G, possible actions A, and initial situation s.

Continual Planning

Typical agent environments are dynamic and only partially observable. During planning, the agent's knowledge is incomplete and missing knowledge has to be obtained by means of sensing. Sensing results are only available after the sensing action has been executed. To overcome this, the agent could create a plan which is independent of the sensing results, which is the approach of *conformant planning* (Hoffmann and Brafman 2006), or it could plan for every possible sensing outcome, which is the approach of *contingent planning* (Hoffmann and Brafman 2005). PKS (Petrick and Bacchus 2002) is a contingent planner which represents knowledge in databases: K_f contains known facts, K_w contains formulas every instance of which the agent either knows or knows the negation, K_v contains functions whose values are known, and K_x contains strictly disjunctive knowledge of literals. PKS implements a forward-chaining planner that constructs conditional plans, which can then be linearized to allow postdiction (Petrick and Bacchus 2004).

In *continual planning*, instead of creating a complete plan before execution, planning and plan execution are interleaved: An agent may create a (partial) plan, execute parts of the plan, sense, and replan, using the sensing results.

MAPL (Brenner and Nebel 2009) is a planning language similar to PDDL which supports continual planning. It corresponds to the ADL fragment without conditional effects. It supports sensing actions and *Know-If* (KIF) fluents, where F_{KIF} holds iff the agent knows the value of F. MAPL allows the definition of *assertions*, which are placeholder actions for conditional sub-plans which depend on yet unknown fluent values. An assertion is never executed but instead it guarantees that its effects are achievable if its precondition holds. An assertion is defined as normal action extended by a *replanning condition*. If the replanning condition holds, the assertion is *expandable*. It is *permanently expandable* if it is expandable for all actions before the assertion. If an assertion is permanently expandable, it is replaced by a plan which has the assertion's effect as goal.

The PDDL-based GKI continual planner (CP) (Dornhege and Hertle 2013) runs in a loop consisting of three stages: first, it estimates the world state by observation. Then, in the monitoring stage, it checks the current plan for applicability. If the current plan is still applicable, the next action is executed. Otherwise, it replans and continues with the new plan. After executing an action, it continues with the next iteration. The GKI CP uses TFD with modules for both planning and execution monitoring and thus supports temporal domains. The GKI CP does not support MAPL and therefore neither Know-If fluents, assertions, nor sensing actions.

Continual Planning in GOLOG

In this section, we describe our approach to continual planning. We explain how we adapted the planner interface to use ensemble planning, we define Know-If fluents in the Situation Calculus, and we provide a definition of assertions as extension to GOLOG's transition semantics.

Planning in GOLOG

We adapted the PDDL-GOLOG interface introduced above: Instead of using FF as the only planner, we use an ensemble planner based on FF and FASTDOWNWARD. The adapted interface works as follows: The planning problem is translated to PDDL and both planners are started in parallel. If either of the planner finds a plan, we use that plan and stop the other planner. If a planner proves the problem to be unsolvable, we stop both planners and cancel the execution of the GOLOG program. The reason for this adaption is a practical one: While FASTDOWNWARD usually finds solutions faster than FF, we observed that it performs worse when the goal formula contains many disjunctions. Additionally, FF is faster in proving that a problem is unsolvable. Thus, using ensemble planning improves planning performance significantly. As neither planner uses multi-threading and FF has low memory requirements, this approach does not incur noticeable performance drawbacks on a multi-core system.

Representation of Incomplete Knowledge

We use the idea of *Know-If* fluents and provide a definition in the Situation Calculus. For a relational fluent F, we add a fluent F_{KIF} which holds iff the value of F is known. We assume if an action causes a fluent to change, the agent is aware of the change and knows the fluent value in the resulting situation. For each F_{KIF}, we add a SKA:

$$F_{KIF}(\vec{x}, do(a, s)) \equiv \gamma^+_F(\vec{x}, a, s) \vee \gamma^-_F(\vec{x}, a, s) \\ \vee \xi_F(\vec{x}, a, s) \vee F_{KIF}(\vec{x}, s) \wedge \neg\gamma^-_{F_{KIF}}(\vec{x}, a, s)$$

where $\gamma^+_F(\vec{x}, a, s)$ ($\gamma^-_F(\vec{x}, a, s)$) are the same as in the SSA for F, $\xi_F(\vec{x}, a, s)$ holds iff a is a knowledge producing action for F and $\gamma^-_{F_{KIF}}(\vec{x}, a, s)$ holds iff a causes the fluent F to be unknown. Note that this is different from knowledge fluents: While $KF(\vec{x}, s)$ holds if and only if F is known to be true in situation s, F_{KIF} holds if the value of F is known in situation s, independent of the value. However, we can express knowledge fluents using Know-If fluents:

$$KF(\vec{x}, s) \equiv F_{KIF}(\vec{x}, s) \wedge F(\vec{x}, s)$$
$$K\neg F(\vec{x}, s) \equiv F_{KIF}(\vec{x}, s) \wedge \neg F(\vec{x}, s)$$

Know-If fluents have several advantages compared to knowledge fluents: First, we only need to add one Know-If fluent for each ordinary fluent instead of two knowledge fluents, thus reducing the total number of fluents. Second, we avoid disjunctive preconditions such as $KF(\vec{x}, s) \vee K\neg F(\vec{x}, s)$, which generally impair planner performance. This is especially useful when defining assertions, where we typically have replanning conditions which only require a fluent to be known but which are independent of the fluent's value. Third, we can directly express that an action senses a fluent: an action which senses the fluent F has as its effect F_{KIF}; with knowledge fluents, this effect cannot be expressed as it is unclear whether KF or $K\neg F$ holds after the action. Similar to knowledge fluents, Know-If fluents are not as expressive as the K fluent.

Assertions

In order to deal with incomplete knowledge during planning, we add a new action type *assertion* to GOLOG. It is conceptually similar to assertions in MAPL and is a placeholder for an unspecified sub-plan, which guarantees that its effects are achievable when its precondition holds without specifying how the effect is achieved. An assertion is never executed but instead replanned once its replanning condition holds.

In Figure 1, the goal is to clean up the cup: if it is dirty, it belongs into the dishwasher, otherwise on the shelf. Initially, the agent does not know whether the cup is clean. The assertion *clean_up_cup* represents a conditional sub-plan. After sensing the cup state, the assertion is replaced with a plan where the goal is the assertion's conditional effect.

To determine when an assertion is replaced by a plan, we require the domain expert to devise an axiom defining the predicate *Expandable* for each assertion a, typically using conjunctions of Know-If fluents. To guarantee eventual expansion, it should furthermore be entailed that

$$Poss(a[s], s) \supset Expandable(a[s], s).$$

Example axioms for *clean_up_cup* are depicted in Figure 2. The assertion is expandable if the agent knows whether the cup is clean or dirty. Here possibility also implies expandability. Finally, the SSA of *at* depends on the cup state: If the cup is clean, it is put on the shelf, else into the dishwasher.

Next, we define the notion of *permanent expandability* of an assertion a between situations s_1 and s_2 as follows:

$$PermExpandable(a, s_1, s_2) \equiv \\ \forall s'.\, s_1 \leq s' \leq s_2 \supset Expandable(a, s')$$

Intuitively, we want the agent to avoid replanning when an assertion is only temporarily expandable and may still be rendered non-expandable by subsequent actions. Instead, we want it to wait until the assertion becomes permanently expandable from the current situation. For that purpose we

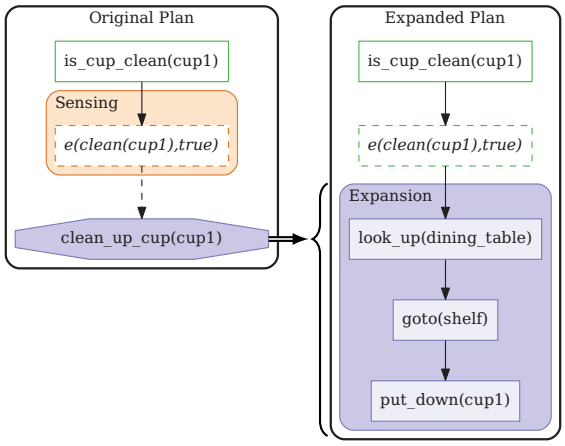

Figure 1: Example for an assertion expansion: the assertion *clean_up_cup* is expanded and replaced with a new sub-plan. $e(clean(cup1), true)$ denotes the sensing result.

introduce the special situation constant now to denote the history of actions $\langle \alpha_1, \ldots, \alpha_k \rangle$ executed so far by the agent. This is achieved by adding the axiom[2]

$$now = do(\langle \alpha_1, \ldots, \alpha_k \rangle, S_0)$$

to the BAT and updating it accordingly after the execution of further actions.

For replanning, we have an operation $Replan$ similar to *Plan* which translates the effects of assertion a to a goal and calls the planner to create a plan δ'. Given a BAT $\mathcal{D}$, a ground action α, and a ground situation term σ, let

$$Adds(\alpha, \sigma) = \{\ F(\vec{c}) \mid \mathcal{D} \models \gamma^+_F(\vec{c}, \alpha, \sigma)\ \} \\ Dels(\alpha, \sigma) = \{\ \neg F(\vec{c}) \mid \mathcal{D} \models \gamma^-_F(\vec{c}, \alpha, \sigma)\ \}$$

Then $Replan(\alpha, \sigma)$ invokes the planner with goal

$$G = \bigwedge Adds(\alpha, \sigma) \wedge \bigwedge Dels(\alpha, \sigma)$$

Theorem 1. *If* $Replan(\alpha, \sigma)$ *yields* $\langle \alpha_1, \ldots, \alpha_n \rangle$ *as solution plan for* G*, then*

$$\mathcal{D} \models G[do(\alpha, \sigma)] \text{ iff } \mathcal{D} \models G[do(\langle \alpha_1, \ldots, \alpha_n \rangle, \sigma)].$$

Note that only G is ensured to hold after executing the plan, i.e. apart from the assertion's effects, the plan may have additional side effects. To cope with problems arising from this, we use execution monitoring in order to trigger replanning in situations where a previous plan is rendered invalid through expanding an assertion. It lies in the programmer's responsibility to devise actions and assertions in a fashion that ensures the possibility of such recovery measures.

Finally, we can define the semantics of assertions in terms of *Trans* and *Final*:

$$Trans(a, s, \delta', s') \equiv now \leq s \wedge \\ \big(\ \neg PermExpandable(a, now, s) \\ \wedge Poss(a[s], s) \wedge \delta' = nil \wedge s' = do(a[s], s) \\ \vee PermExpandable(a, now, s) \\ \wedge \delta' = Replan(a, s) \wedge s' = s\ \big) \\ Final(a, s) \equiv False$$

That is, for assertions we only consider future and present situations of now. As long as the assertion is not permanently expandable, it is treated as a primitive action. Once it becomes permanently expandable, replanning is initiated.

[2]We use $do(\langle \alpha_1, \ldots, \alpha_k \rangle, S_0)$ as shorthand for situation $do(\alpha_k, do(\alpha_{k-1}, do(\ldots, do(\alpha_1, S_0))) \ldots)$.

$$Expandable(clean\_up\_cup(c), s) \equiv clean_{KIF}(c, s) \\ Poss(clean\_up\_cup(c), s) \equiv \\ holding(c, s) \wedge clean_{KIF}(c, s) \\ at(c, l, do(a, s)) \equiv a = clean\_up\_cup(c) \wedge \\ \big(clean(c, s) \wedge l = shelf \\ \vee \neg clean(c, s) \wedge l = dishwasher\big)$$

Figure 2: Example axioms for $clean\_up\_cup$

An assertion must never be used as a possible action. On the other hand, we generally want to allow the use of assertions during expansion. Therefore, we require the domain to include a (strict) finite partial order $<_A$ on assertions, where a can be used to expand b iff $a <_A b$ holds (read: "b is more abstract than a"). As an example, when expanding *clean_table* to clean up all cups on a table, the planner should be able to use the *clean_up_cup* assertion, hence we set $clean\_up\_cup <_A clean\_table$.

Theorem 2. *Let $Trans^*$ denote the reflexive and transitive closure of $Trans$ as defined in (De Giacomo, Lespérance, and Levesque 2000). If α is an assertion such that $\mathcal{D} \models Trans^*(\alpha, now, \delta', do(\beta, now))$, then β is not an assertion.*

Therefore, an assertion will never be selected for execution, but rather expansion. After at most finitely many expansions, the next action will always be a non-assertion β.

As a final remark, note that other definitions than ours for when to expand an assertion are conceivable, e.g. as soon as it is expandable (without the permanency constraint), or only if it is the next action to be executed. Both approaches would simplify the definition, but could lead to more difficult monitoring or even dead ends.

Evaluation

In this section, we evaluate our approach to continual planning in GOLOG with two applications. We present an in-depth evaluation of our approach in the household domain and compare it to the GKI CP. Additionally, we sketch how we use continual planning in the RoboCup Logistics League.

Household Domain

We evaluate our approach using the clean-up task from the household domain and compare it to the GKI CP. A comparison to MAPL was not possible because our domain relies on conditional effects, which are not supported by MAPL. To compare both continual planners, we run both systems on the same problem. To cope with discrepancies between the actual world and the agent's representation of the world, both planners use execution monitoring, which detects such discrepancies and adapts the plan accordingly. The GOLOG CP uses a monitor similar to an existing GOLOG monitor (De Giacomo, Reiter, and Soutchanski 1998), the GKI CP has a built-in monitor. In order to analyze planner performance quantitatively, we measured the elapsed real time between the invocation of the planner and its termination, excluding monitoring time. All benchmarks were done on an Intel Core i7-3770 at 3.40 GHz with 4 cores and hyper-threading, resulting in 8 parallel threads. Quantitative benchmarks were done by simulating sensing results and assuming that all actions succeeded. We ran all tasks twenty times and computed the average total time. Both systems have also run successfully on our mobile robot Caesar (Ferrein et al. 2013), which is shown in Figure 3.

Figure 3: Household robot Caesar in our robot lab

Evaluation Task Our evaluation task is a clean-up task from the household domain and modifies the setting of the TidyUp-Robot project (Dornhege and Hertle 2013): The robot is supposed to clean up all cups located on a table by putting dirty cups into the dish washer and clean cups on the shelf. There are five relevant locations: *dining_table*, *shelf*, *kitchen_entrance*, *dishwasher_side*, and *dishwasher_front* to reach the dishwasher. The dishwasher contains two spots to place cups reachable from the front, and one reachable from the side, where the robot must align before placing any cup. Initially, the robot knows all locations, how to align to all locations and all the spots in the dishwasher. The robot is able to sense the state of a cup if it is holding the cup with the explicit *is_cup_clean* sensing action. It also relies on passive sensing, e.g. when aligned to and looking onto the table, it can determine whether a cup is located there, which is done implicitly by reading cup positions directly from sensors. The goal is always the same: The robot is supposed to clean up all cups from the dining table. All dirty cups belong into the dishwasher, all clean cups belong on the shelf:

$$\begin{aligned}\neg\exists c\,(cup(c) \wedge holding(c)) \wedge looking\_at(dining\_table)\\ \wedge \forall c\,.\,\big[cup(c) \supset at_{KIF}(c, dining\_table)\\ \wedge \exists l\,.\,\big(location(l) \wedge at(c,l) \supset (clean_{KIF}(c)\\ \wedge (\neg clean(c) \supset at(c, dishwasher))\\ \wedge (clean(c) \supset at(c, shelf)))\big)\big]\end{aligned}$$

The two tasks only differ in the initial knowledge:

Task 1 The robot initially knows neither the cup positions nor whether a cup is clean. We vary the number of cups between 1 and 10; we add clean and dirty cups alternately.

Task 2 The robot does not know the cup positions, but it knows cup states initially and thus does not need to sense them. As before, every second cup is clean, the number of cups varies between 1 and 10 (simplification of Task 1).

Continual Planning in GOLOG To evaluate the performance of our approach, we compare it to the GKI CP by running Task 2 with both planners. In Figure 4, the total planning times are shown. We can see that the GOLOG CP plans much faster when using the ensemble planner. For the

problem with five cups, the GKI CP needed a total planning time of (278.1 ± 19.9) s, while the GOLOG CP only needed (0.95 ± 0.02) s. When we limit the GOLOG CP to FASTDOWNWARD, it is still faster with (183.7 ± 1.9) s. When running Task 1, the GOLOG CP could cope with the incomplete initial knowledge, but the GKI CP did not succeed even for small problems as shown in Figure 5. For the problem with two cups, the GKI CP took (1190.6 ± 141.8) s, the GOLOG CP with ensemble planning took only (2.23 ± 0.02) s, but (634.5 ± 4.1) s when limited to FASTDOWNWARD. Thus, ensemble planning clearly performs better for this problem.

To analyze plan quality, we investigate the number of actions planned by both planners when running Task 2. Both planners produced plans of similar quality; e.g. for the problem with five cups, the GKI CP needed 65.0 ± 3.4 actions, while the GOLOG CP needed 62.0 ± 0.0 actions. Thus, there is no significant difference in plan quality. However, the GKI CP could not solve problems with more than five cups, while the GOLOG CP could handle problems with ten cups.

Assertions Our continual planner relies on the use of assertions, which guarantee that certain sub-goals are reached while not having all necessary information to create an actual plan. However, continual planning is also possible without assertions: The planner creates a plan for one possible world state and revises the plan when it sensed fluents which differ from its assumptions. As we make the closed-world assumption, the planner simply assumes all fluents not known to be true to be false and plans accordingly.

To measure the performance of assertions, we ran our agent on Task 1 without assertions and with the *clean_up_cup* assertion using the ensemble planner. In Figure 5, total planning times for both cases are shown. For small problems, we can see that planning without assertions is in advantage. This seems reasonable as assertions cause an additional overhead which may surmount their advantage. However, for larger problems, the use of assertions is clearly advantageous. While the planner with assertions could solve problems with 10 cups in (10.0 ± 0.06) s, planning without assertions took (326.5 ± 4.9) s for six cups. Problems with more than six cups could not be solved without assertions. When investigating the intermittent planning results, the reason becomes clear: The plan without assertions contains actions which depend on the sensing result. If the sensing result differs from the expected results, the remaining plan becomes invalid and the agent must replan.

Both approaches with and without assertions produce plans of similar length. As an example, for the problem with six cups, the final plan consisted of 93.0 ± 0.0 actions when not using assertions and 90.0 ± 0.0 actions when using assertions. Thus, assertions do not deteriorate plan length.

RoboCup Logistics League (RCLL)

The RCLL is an industry-inspired competition under the RoboCup umbrella. The goal is to maintain and optimize the material flow in a smart factory by a group of three autonomous mobile robots. Before the production can start, the robots must explore the environment and identify the available machines. These machines are distributed in six out of twelve zones (per team). The robots travel to the zones, check whether it is occupied by a machine, and start identification if it is. For details, we refer to (Niemueller, Lakemeyer, and Ferrein 2015). For brevity, we limit our further description to the exploration phase.

In this scenario, we use planning to minimize plan duration. Our agents use local planning, i.e., each agent plans on its own. Full planning for all zones and actions is inefficient. Therefore, a GOLOG procedure as shown below is used to pick some unexplored zone and plan the necessary actions. In order to avoid resource conflicts, the robots coordinate using mutual exclusion locks for zones. Continual planning allows to recover from rejected logs (concurrently operating robots may choose the same zone at the same time).

```
while some(z, zone, explorable(z)) do
    plan(some(z,zone,and(explorable(z),explored(z))))
endWhile
```

Within the planner call (*plan*), the necessary steps to explore a specific zone are generated. An assertion is used to insert steps for identification iff a machine has been detected within a zone. GOLOG CP for the RCLL shows its useful-

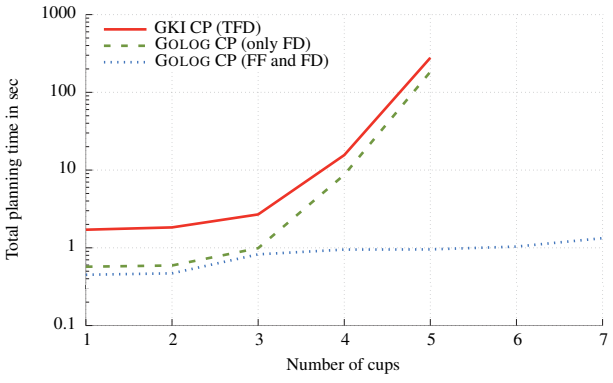

Figure 4: Planning times for continual planning with the GKI CP and the GOLOG CP running Task 2 (log. scale)

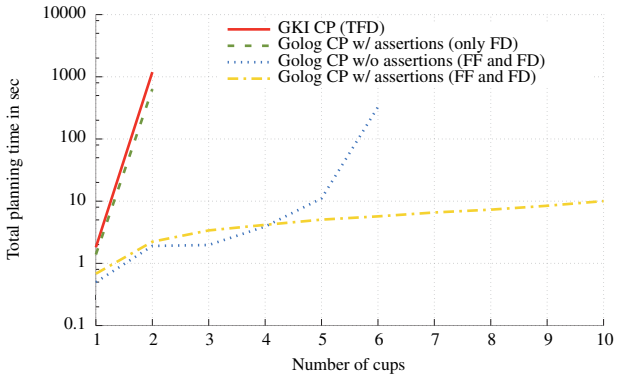

Figure 5: Planning times with the GKI CP and the GOLOG CP with and without assertions, running Task 1 (log. scale)

ness in multi-robot scenarios by combining the expressiveness of GOLOG with the efficiency of planning.

Framework Architecture and Example

In this section, we describe the architecture of our framework and we give a detailed example of a run in the household domain.

Implementation

Our GOLOG interpreter is based on the Prolog implementations of INDIGOLOG and READYLOG. We use ECLiPSe as Prolog interpreter which is integrated into the robot software framework Fawkes (Niemueller et al. 2010). Fawkes provides the functional components and simulation integration. The Lua-based Behavior Engine (BE) (Niemueller, Ferrein, and Lakemeyer 2010) provides the primitive actions. Our implementation is available on https://www.fawkesrobotics.org/p/golog-cp.

As shown in Figure 6, the GOLOG main loop consists of three stages: First, passive sensing is done by reading sensing results from Fawkes' sensing modules. Second, the current program is monitored. During monitoring, all assertions in the program are checked for expandability and are expanded if necessary. The monitor also checks if the current program achieves the goal. If not, it first tries to recover from any exogenous changes and replans if recovery fails. Assertion expansion, recovery, and replanning use the GOLOG-PDDL interface for planning. The monitor uses a *goal action* of the form `!(goal)`, which like GOLOG's *test actions* has no effects and which succeeds if the goal formula holds. It is used to determine a plan's validity. After the monitoring stage, the next action of the current program is executed. For *plan* actions, the external planner is called via the GOLOG-PDDL interface. Other actions are translated to a command for the BE, which takes care of the details of action execution. After the action has been executed, the interpreter continues with the next loop iteration.

Example

We present an example of a run of Task 1 with one clean and one dirty cup. The goal is to clean up all cups:

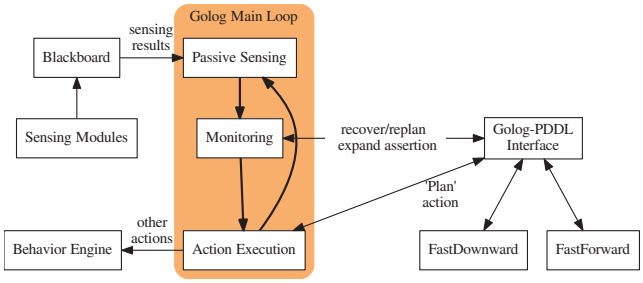

Figure 6: Interaction between GOLOG, Fawkes, and the PDDL planners

and(looking_at(livingroom_table),**all**(c,cup,
and(kif_at(c,livingroom_table),
impl(**or**(holding(c),**some**(l,location,at(c,l))),
and(kif_clean(c),
and(**impl**(**neg**(clean(c)),at(c,dishwasher)),
impl(clean(c),at(c,counter)))))))))

In the following, we abbreviate the goal formula with <*goal*>. Initially, the program consists of a single *plan* action, which achieves the main goal, and a goal check.

[plan(<goal>), !(<goal>)]

After one execution step, the *plan* call has been replaced by an actual plan. As the agent has not sensed any cups, it only needs to look at the table in order to accomplish the goal:

[goto(dining_table), look_at(dining_table), !(<goal>)]

After the agent has executed the *look_at* action, it senses *cup1* and *cup2* by means of passive sensing. The goal statement does not hold, replanning is initiated. After replanning, the program contains two *clean_up_cup* assertions:

[pick_up(cup1), is_cup_clean(cup1), clean_up_cup(cup1),
pick_up(cup2), is_cup_clean(cup2), clean_up_cup(cup2),
!(<goal>)]

After executing *is_cup_clean(cup1)*, the state of *cup1* is known and the first assertion is replanned:

[pick_up(cup1), look_up(dining_table), goto(counter),
align_to(counter), look_at(counter), put_down(cup2, counter),
pick_up(cup2), is_cup_clean(cup2), clean_up_cup(cup2),
!(<goal>)]

Note that this plan is invalid; in addition to the assertion's effect, the expanded plan also changes the robot's position. Thus, executing *pick_up(cup2)* is not possible. However, after monitoring, the new program is again valid:

[pick_up(cup1), look_up(dining_table), goto(counter),
align_to(counter), look_at(counter), put_down(cup2, counter),
goto(dining_table), look_at(dining_table),
pick_up(cup2), is_cup_clean(cup2), clean_up_cup(cup2),
!(<goal>)]

The other *clean_up_cup* assertion is expanded similarly after executing *is_cup_clean(cup2)*. After executing the resulting plan, *goal* holds and the execution terminates successfully.

Conclusion

We combined the expressiveness of GOLOG and the ability to handle incomplete knowledge with the efficiency of PDDL-based plan generation. With the introduction of assertions for planning we improve on a major drawback of previous solutions: not being able to handle incomplete knowledge in the planning part. An assertion specifies conditions when to trigger planning, which could potentially run concurrently while other parts of the GOLOG program are being executed. The conditional effect of an assertion provides the planning goal. The idea is that the planner generates an appropriate plan to achieve the desired effects. We can even run multiple different planners concurrently to benefit of their individual strengths, e.g. shorter generated plans or faster determination that a plan does not exist. By not

relying on the planner for execution, but rather translating the plan back to GOLOG, we can postpone some decisions which rely on sensing and make use of GOLOG's execution monitoring capabilities.

The presented approach provides a unified modeling framework for robot tasks based on GOLOG that can call PDDL-based planners for efficient plan generation. It allows to encode (partial) domain knowledge into the GOLOG domain and program specification and explicitly model when to generate sub-plans. The evaluation has shown that it provides comparable or even better performance than purely PDDL-based systems. Especially the integration of assertions gives a noticeable performance improvement. The system has been implemented and run on an actual mobile robot solving the described clean-up task in a real environment.

Acknowledgments. J. Claßen, T. Hofmann, and T. Niemueller were supported by the German National Science Foundation (DFG) research unit *FOR 1513* on Hybrid Reasoning for Intelligent Systems (http://www.hybrid-reasoning.org).

References

Brenner, M., and Nebel, B. 2009. Continual planning and acting in dynamic multiagent environments. *Autonomous Agents and Multi-Agent Systems* 19(3).

Claßen, J.; Eyerich, P.; Lakemeyer, G.; and Nebel, B. 2007. Towards an integration of Golog and planning. In *Proc. of the 20th Int. Joint Conference on Artificial Intelligence (IJCAI-07)*.

Claßen, J.; Engelmann, V.; Lakemeyer, G.; and Röger, G. 2008. Integrating Golog and planning: An empirical evaluation. In *Proc. of the 12th Int. Workshop on Nonmonotonic Reasoning*.

Claßen, J.; Röger, G.; Lakemeyer, G.; and Nebel, B. 2012. PLATAS — integrating planning and the action language Golog. *KI - Künstliche Intelligenz* 26(1).

De Giacomo, G.; Lespérance, Y.; Levesque, H. J.; and Sardina, S. 2009. IndiGolog: A high-level programming language for embedded reasoning agents. In *Multi-Agent Programming*. Springer.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1–2).

De Giacomo, G.; Reiter, R.; and Soutchanski, M. 1998. Execution monitoring of high-level robot programs. In *Int. Conf on Principles of Knowledge Representation and Reasoning*.

Demolombe, R., and del Pilar Pozos Parra, M. 2000. A simple and tractable extension of situation calculus to epistemic logic. In *Proc. 12th Int. Symp. on Methodologies for Intelligent Systems*. Springer.

Dornhege, C., and Hertle, A. 2013. Integrated symbolic planning in the tidyup-robot project. In *AAAI Spring Symposium - Designing Intelligent Robots: Reintegrating AI II*.

Eyerich, P.; Mattmüller, R.; and Röger, G. 2012. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Towards Service Robots for Everyday Environments*. Springer.

Ferrein, A., and Lakemeyer, G. 2008. Logic-based robot control in highly dynamic domains. *Robotics and Autonomous Systems* 56(11).

Ferrein, A.; Niemueller, T.; Schiffer, S.; and Lakemeyer, G. 2013. Lessons learnt from developing the embodied AI platform Caesar for domestic service robotics. In *AAAI Spring Symposium 2013 on Designing Intelligent Robots: Reintegrating AI II*.

Fikes, R. E., and Nilsson, N. J. 1972. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial intelligence* 2(3).

Fox, M., and Long, D. 2003. PDDL2. 1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research* 20.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26.

Hoffmann, J., and Brafman, R. 2005. Contingent planning via heuristic forward search with implicit belief states. In *Proc. of the 15th Int. Conference on Automated Planning and Scheduling*.

Hoffmann, J., and Brafman, R. 2006. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence* 170(6).

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

Levesque, H. J.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming* 31(1–3).

McCarthy, J. 1963. Situations, actions, and causal laws. Technical report, DTIC Document.

McDermott, D.; Ghallab, M.; Howe, A.; Knoblock, C.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL – The Planning Domain Definition Language. Technical report, AIPS-98 Planning Competition Committee.

Niemueller, T.; Ferrein, A.; Beck, D.; and Lakemeyer, G. 2010. Design principles of the component-based robot software framework Fawkes. In *Proceedings of the 2nd International Conference on Simulation, Modeling and Programming for Autonomous Robots*. Springer. 300–311.

Niemueller, T.; Ferrein, A.; and Lakemeyer, G. 2010. A Lua-based behavior engine for controlling the humanoid robot Nao. In *RoboCup 2009: Robot Soccer World Cup XIII*, 240–251. Springer.

Niemueller, T.; Lakemeyer, G.; and Ferrein, A. 2015. The RoboCup Logistics League as a Benchmark for Planning in Robotics. In *WS on Planning and Robotics (PlanRob) at Int. Conf. on Aut. Planning and Scheduling (ICAPS)*.

Pednault, E. P. 1989. ADL: Exploring the middle ground between STRIPS and the situation calculus. In *Proc. of the 1st Int. Conference on Principles of Knowledge Representation and Reasoning*.

Petrick, R. P., and Bacchus, F. 2002. A knowledge-based approach to planning with incomplete information and sensing. In *Proc. of the 6th International Conference on Artificial Intelligence Planning Systems*.

Petrick, R. P., and Bacchus, F. 2004. Extending the knowledge-based approach to planning with incomplete information and sensing. In *9th Int. Conf on Principles of Knowledge Representation and Reasoning*.

Petrick, R., and Levesque, H. 2002. Knowledge equivalence in combined action theories. In *Proc. of 8th Int. Conference on Principles of Knowledge Representation and Reasoning*.

Reiter, R. 2001. *Knowledge in action: logical foundations for specifying and implementing dynamical systems*. MIT Press.

Scherl, R. B., and Levesque, H. J. 1993. The frame problem and knowledge-producing actions. In *Proc. of the 11th National Conference on Artificial Intelligence*.