Integrating Golog and Planning: An Empirical Evaluation

Jens Claßen[†] and **Viktor Engelmann**[†] and **Gerhard Lakemeyer**[†] and **Gabriele Röger**[‡]
[†] Dept. of Computer Science, RWTH Aachen University, Germany
[‡] Dept. of Computer Science, University of Freiburg, Germany

Abstract

The Golog family of action languages has proven to be a useful means for the high-level control of autonomous agents, such as mobile robots. In particular, the IndiGolog variant, where programs are executed in an online manner, is applicable in realistic scenarios where agents possess only incomplete knowledge about the state of the world, have to use sensors to gather necessary information at runtime and need to react to spontaneous, exogenous events that happen unpredictably due to a dynamic environment. Often, the specification of such an agent's program also involves that certain subgoals have to be solved by means of planning. IndiGolog supports this in principle by providing a variety of lookahead mechanisms, but when it comes to pure, sequential planning, these usually cannot compete with modern state-of-the-art planning systems, most of which being based on the Planning Domain Definition Language PDDL. Previous theoretical results provide insights on the semantical compatibility between Golog and PDDL and how they compare in terms of expressiveness. In this paper, we complement these results with an empirical evaluation that shows that equipping IndiGolog with a PDDL planner (FF in our case) pays off in terms of the runtime performance of the overall system. For that matter, we study a number of example application domains and compare the needed computation times for varying problem sizes and difficulties.

Introduction

The Golog (Levesque et al. 1997) family of action languages has already proven to be a suitable means when it comes to the high-level control of autonomous agents, such as mobile robots (Burgard et al. 1998). It provides the programmer with the flexibility to chose the right balance between deterministic, predefined behavior on the one hand and on the other hand leave certain non-deterministic parts to be solved by the system.

The IndiGolog (De Giacomo, Levesque, and Sardina 2001) variant, which is in turn based on ConGolog (De Giacomo, Lespérance, and Levesque 2000), possesses a number of features that makes it particularly suited for many practical scenarios. For one, it works in cases where agents only have incomplete knowledge about the state of the world, and where sensing actions can and most often have to be used in order to gather further information that is required to fulfill the task. Furthermore, so-called exogenous actions reflect changes in a dynamic environment that are not caused by an action of the agent.

For another, an important aspect about the language is that programs are executed in an incremental, online manner. Unlike in the original Golog, where the system searches for an action sequence that constitutes a legal execution of the entire input program, IndiGolog does not apply a general lookahead, but leaves it to the programmer to explicitly state which parts of the program ought to be solved by means of search. This helps to keep program execution tractable, in particular in the presence of incomplete knowledge and possible occurrences of exogenous actions.

However, in many application domains for which IndiGolog is suited in principle, one often encounters subproblems that are rather combinatorial in nature. Typical examples include scheduling currently pending requests to the system, finding a route through a certain topology, or a combination of these two. These are cases for classical planning, where the programmer only provides a list of available actions and a description of the goal state, and it is up to the system to search for an appropriate (and preferably short) sequence of action instances that achieves the goal. While Golog supports this in principle, it soon becomes infeasible for all but the smallest problem sizes.

However, sequential planning has received a lot of attention in recent years. PDDL, the Planning Domain Definition Language (Ghallab et al. 1998) was introduced as the common input language for the systems competing at the biennial International Planning Competition. It extends the well-known STRIPS language (Fikes and Nilsson 1971) by features from Pednault's (1989) ADL. Later extensions include (among other things) durative and concurrent actions (Fox and Long 2003; Edelkamp and Hoffmann 2004) as well as constraints on plan trajectories and preferences among goals (Gerevini and Long 2005). The language has become a de-facto standard for formulating planning benchmarks. Many efficient planners that use it have been developed by now, using a large variety of techniques and heuristics.

It suggests itself to benefit from these developments by embedding such a planner into IndiGolog. The idea is that

whenever a planning subproblem arises during the execution of a Golog program, it is translated into PDDL and the planner is called. The resulting plan is translated back and Golog resumes executing that plan. For the ADL fragment of PDDL, the theoretical foundations for such an embedding have been laid in previous work. Claßen et al. (2007) show that the state updates in PDDL can be understood as progression for a certain form of Golog action theories; Röger, Helmert and Nebel (2008; 2007) identify a maximal class of such theories that are equivalent to the ADL sub-language in terms of expressiveness.

Based on these theoretical results, we extended the current implementation of IndiGolog with the possibility of redirecting planning subgoals to a PDDL planner, in our case the FF system. In this paper, we do an empirical study that shows that such an extension is beneficial in terms of the overall computation time needed by the system. For this purpose we developed a number of example application domains, ran them in simulations and measured the corresponding runtimes for varying problem sizes.

The remainder of the paper is organized as follows. In the following section, we present details concerning the IndiGolog framework, the FF planning system, and our integration of the two. Next, we introduce the application domains that we developed, after which we discuss our experimental setup and the results obtained. We close with a brief conclusion.

Integrating FF into IndiGolog

IndiGolog

The *situation calculus* (McCarthy and Hayes 1969; Reiter 2001) is a dialect of first-order logic (with some second-order extensions) for reasoning about dynamic domains. Changes in the world are assumed to be the result of *primitive actions*, which are performed by some implicit agent and modelled by terms like $move(l_1, l_2)$. Properties that are affected by performing such actions are called fluents, which can be predicates like $Holding(obj_2, s)$ or functions like $position(robot, s)$. The last argument of a fluent is a *situation*, which should be understood as the current history of actions that have been executed. The constant S_0 is used to denote the initial situation, and when a is an action and s a situation, then $do(a, s)$ denotes the situation that results from performing a in s. For example $Holding(letter, do(pickup(letter, S_0)))$ means that the agent is holding the letter after picking it up. A particular domain is described by a *basic action theory*, which is a set of situation calculus formulas that define the fluents' values in the initial situation and preconditions and effects of actions.

Based on the situation calculus, the members of the Golog (Levesque et al. 1997) family of languages allow the definition of complex actions, also called *programs*. The ConGolog (De Giacomo, Lespérance, and Levesque 2000) variant supports the following constructs:

α	primitive action
$\phi?$	test
$\delta_1; \delta_2$	sequence
$\delta_1 \mid \delta_2$	nondeterministic choice
$\pi x.\delta(x)$	nondeterministic choice of argument
δ^*	nondeterministic iteration
if ϕ **then** δ_1 **else** δ_2 **endIf**	conditional
while ϕ **do** δ **endWhile**	loop
$\delta_1 \| \delta_2$	concurrent execution
$\delta_1 \rangle\!\rangle \delta_2$	prioritized concurrency
$\delta^{\|}$	concurrent iteration
$\langle \vec{x} : \phi(\vec{x}) \rightarrow \delta(\vec{x}) \rangle$	interrupt
$P(\vec{t})$	procedure call

Apart from conditionals, loops and recursive procedures, which are common to imperative programming languages, an important aspect is that parts of a program can also be nondeterministic. For instance, $\delta_1 \mid \delta_2$ means to do either δ_1 or δ_2, and δ^* performs δ zero or more times. The idea is that a program does not represent a complete solution to the problem, but only a sketch of it, where the nondeterministic parts constitute gaps which have to be filled by the system.

ConGolog models concurrency as (nondeterministic) interleavings of the involved processes, i.e. actions are always performed one at a time. This includes so-called *exogenous actions*, which are used to model spontaneous changes in the dynamic environment that do not constitute (direct) effects of the agent's actions. Interrupts can be used to define reactive responses that are triggered when such an event occurs or some other condition is met; the normal program execution then continues afterwards.

Programs are executed off-line in ConGolog. This means that the interpreter first analyzes the entire program to find a conforming execution trace before the first action is actually executed in the real world. This soon becomes a problem when the program is large; furthermore in many scenarios, the agent has only incomplete knowledge about its environment, making it necessary to gather information at runtime. IndiGolog (De Giacomo, Levesque, and Sardina 2001) is an extension where these issues are tackled. Programs are executed on-line, which means that there is no general lookahead; the interpreter simply executes the next possible action in each step (treating nondeterminism like random choices). A new operator $\Sigma(\delta)$ is introduced which has to be used to explicitly mark subprograms which have to be solved by means of search. This does not represent a loss of generality since one might encapsulate the entire program, but gives the programmer much more control over where the system spends its computational effort. In addition, programs may contain *sensing actions* for acquiring needed information at runtime. When such an action is executed, a sensing result is obtained (which normally is the current value of some fluent) and used to update the agent's knowledge base. Thus, a subsequent choice in the program that depends on this sensed value can be made on-line.

The features mentioned above make IndiGolog applicable in many practical scenarios, where we often are confronted with dynamically changing environments, where sensing has to be used to fill gaps in the agent's information about the world, and where computational power usually is limited. A PROLOG-based implementation of an IndiGolog agent architecture is available at Sourceforge[1]

[1] http://sourceforge.net/projects/indigolog/

and has already been successfully applied for controlling robots of the LEGO MINDSTORMS system, the ER1 EVOLUTION robot and other software agents.

Golog is therefore an appropriate means for the overall control of an agent in many application domains. However, the general task often involves certain combinatorial subproblems, like finding a route through a certain topology, scheduling currently pending requests or a combination of these two. When one is unable (or it is to tedious) to specify some (partially nondeterministic) program to restrain the space of possible execution traces, these examples correspond to classical planning tasks where only the current state, a goal, and a set of available actions are provided, and where it is the job of the system to search for an action sequence that achieves that goal.

In principle, even classical planning can be done in Golog with a completely nondeterministic program as follows:

$$\textbf{while } (\neg Goal) \textbf{ do } (\pi a)\ a \textbf{ endWhile}.$$

A (successful) execution trace of this program corresponds to a plan that reaches a situation where $Goal$ holds. However, since the Golog interpreter uses the PROLOG backtracking mechanism to resolve nondeterminism, performing planning like this basically amounts to do a blind search. The IndiGolog system further contains a number of built-in planning mechanisms, but these are merely proof of concepts for different kinds of conditional planning (Sardina et al. 2004), which use very basic, unguided search strategies. In any of these cases, planning soon becomes infeasible for all but the smallest problem sizes.

The FF Planning System

A large research community engaged in developing sophisticated techniques and domain-independent heuristics for solving classical planning problems has evolved in the last decade, where the PDDL language has become a de-facto standard for formalizing benchmarks that allow the comparison of different approaches.

The FF Planning System developed by Hoffmann (2001) is a fully automated system for classical planning. It supports the ADL fragment of PDDL which is sufficient for our purposes. Furthermore, it has proven its quality by winning the automated track of the planning competition in 2000 and still being part of state-of-the-art planning systems that support a wider fragment of PDDL.

FF performs a forward search in the state space, guided by a heuristic function that is automatically extracted from the domain description. The heuristic gets derived from the corresponding *relaxed* planning task that results from the original one by ignoring the delete effects of the actions. This relaxed task can be solved in polynomial time and the number of actions in the resulting plan provides an estimate for the goal distance in the original task. Furthermore, the relaxed task is used to identify so-called *helpful actions* that are expected to be a good choice for the next action and which hence are considered first during search.

The search method used by the FF system is a variant of hill-climbing called *enforced* hill climbing. The main difference to the standard algorithm is that in the case of a plateau it switches to best first search until a node with a strictly better evaluation is found.

Integration

We embedded the FF planning system into the IndiGolog framework to benefit from these developments. For that purpose, we introduced a new program construct `achieve_ff(G,A)`, where `G` is a goal formula and `A` is a list of actions. Whenever an `achieve_ff` statement is encountered during the execution of a program, it causes the current state, the goal and the available actions to be translated into a PDDL planning problem which is referred to FF. The resulting plan is translated back and Golog continues with executing that action sequence.

The semantical soundness of this proceeding has been laid in previous work. In a first paper (Claßen et al. 2007) it was shown that the state updates in PDDL's ADL fragment (i.e. the language we obtain when only the `:adl` requirement flag is set) can be understood as progression for a certain form of Golog action theories; in two further papers (Röger and Nebel 2007; Röger, Helmert, and Nebel 2008) a maximal class of such theories is identified that are equivalent to the ADL sub-language in terms of expressiveness. ADL extends basic STRIPS with conditional effects as well as negated, disjunctive and quantified preconditions.

We will not discuss the theoretical details here, but provide an intuition by means of an example translation, taken from one of our test applications. A PDDL planner requires two files: a *domain description*, containing types, predicates and operator definitions, and a *problem description*, which specifies the objects in the domain, the initial values of predicates and the goal formula.

Since in PDDL the closed world assumption holds, fluents and action parameters can only take values from a finite set of object constants. These may be divided into types and subtypes that are used to restrict the possible values of parameters. We use unary PROLOG predicates with finite extensions to represent types and declare the subtypes of the general supertype `object` as follows:

```
object(X) :- passenger(X) ; floor(X).
```

In general, subtypes can of course be further subdivided, using similar clauses. These Golog declarations are directly mapped to type declarations in the PDDL domain file:

```
(:types passenger floor - object)
```

A relational fluent with type restrictions on its arguments is in the following way declared in the Golog axiomatization:

```
rel_fluent(lift_at(F)) :- floor(F).
```

The compiled PDDL domain definition then contains:

```
(:predicates
   (lift_at ?f - floor) ...
```

An action is given by a declaration (with type restrictions), a precondition and a number of positive and negative effect axioms:

```
action(move(F1,F2)) :-
 floor(F1), floor(F2).
```

```
poss(move(F1,F2),
 and(lift_at(F1),
     or(above(F1,F2),above(F2,F1)))).

causes_false(move(F1,F2),
             lift_at(F1),true).
causes_true(move(F1,F2),
             lift_at(F2),true).
```

The last argument of an effect axiom may contain a condition that must hold for the effect to actually take place. In case of `true`, the translation is straightforward:

```
(:action move
 :parameters
  (?f1 - floor ?f2 - floor)
 :precondition
  (and (lift_at ?f1)
       (or (above ?f1 ?f2)
           (above ?f2 ?f1)))
 :effect
  (and (lift_at ?f2)
       (not (lift_at ?f1)))
)
```

The following example shows how an effect is translated that involves a non-trivial condition and additional variables that are not arguments of the action:

```
causes_true(stop(F),
            boarded(P), origin(P,F)).
```

Such conditions result in conditional effects in the PDDL action (the type of the quantified variable is determined on the basis of the type definition of the arguments of `origin`):

```
(forall (?p - passenger)
    (when (origin ?p ?f)(boarded ?p)))
```

To generate the actual planning instance, we need to collect all objects of the involved types, e.g.

```
passenger(p1). passenger(p2).
floor(f1). floor(f2). floor(f3).
```

and declare them in the PDDL problem file:

```
(:objects p1 p2 - passenger
          f1 f2 f3 - floor)
```

We then determine which fluent ground atoms `F` evaluate to `true` given the current action history `H`, i.e. we collect all solutions of `has_value(F,H,true)`. For instance, if `lift_at(f2)` is one such atom, then the `:init` section of the problem file contains

```
(:init  (lift_at f2)
        ... )
```

Finally, the problem description contains the translation of the goal formula `G`. For example, a goal formula `all(p,passenger,served(p))` results in

```
(:goal (forall (?p - passenger)
               (served ?p)))
```

We remark that of course the specific, restricted form of clauses in the Golog action theory is only required for those parts that are relevant for the planning problem. The axiomatization may contain additional parts that PDDL does not understand, in particular exogenous and sensing actions.

Benchmark Domains

We designed three example application domains. The first two examples are representatives of so-called transportation domains (Helmert 2008). The characterizing property of such problems is that there are portables that should be transported from their origin to their destination location using mobiles that can move between some of the locations. This type of problem is especially interesting because such tasks arise very often in practice. This is probably also the reason why a large fraction of the benchmarks used in the International Planning Competitions is among these domains. The most interesting aspect of our last example domain is that it in addition involves sensing. In the following we will briefly introduce each of these domains.

A Logistics Domain

The first domain that we studied serves as a representative for all kinds of logistics applications. The task is to transport packages to their destination locations, using a number of trucks which can only hold one package at a time. The direct connections between locations form a (not necessarily complete) graph structure.

The domain has the dynamic aspect that new packages keep arriving at runtime, represented by exogenous actions, and have to be picked up and delivered in turn.

An Elevator Domain

The second test domain has been inspired by the miconic elevator domains of the International Planning Competition in 2000. There is an elevator moving between the floors of a building. At some floors passengers are waiting and should be transported to their respective destination floor. During the program execution new passengers arrive randomly.

There are three sorts of actions that can be used to serve the passengers: Movement actions move the elevator from one floor to an adjacent floor. Since an elevator can move faster if it does not have to stop at each floor, there are also actions for fast movements that overcome two floors within one step. The third sort of actions are the stop actions which cause all passengers waiting at the current floor to enter the elevator and drop off all boarded passengers whose destination is the current floor.

A Mail Delivery Robot Domain

The third domain is a variant of a common application example (Tam et al. 1997) of a mobile robot operating in an office environment, where it has to deliver letters and parcels between the workers' mailboxes. Here, the structure of the building is assumed to consist of a number of hallways, which are connected (e.g. by an elevator) to other hallways, and where there is a certain number of offices at each hallway. Each office may contain one or multiple different mailboxes, each of which serving for both incoming and outgoing mails.

This domain involves sensing since the robot must look into a mailbox in order to find out how many and which letters it currently contains. Furthermore, before the agent actually knows where to deliver a letter, it has to pick it up and read off the addressee.

Experiments and Results

For our experiments, we further have extended the IndiGolog framework by a simple simulator that plays the role of the outside world. It runs in a separate instance of PROLOG and communicates with Golog via TCP/IP sockets. The basic idea is to keep track of the (relevant part of the) world state using PROLOG's `assert` and `retract` mechanism. When an exogenous actions occurs and what sensing result is returned after the execution of an action can then be defined as conditions wrt to the simulated state. All experiments were performed on a PC with an Intel Core 2 Duo E6750 CPU running at 2.66 GHz with 2 GB of memory.

Logistics

In the logistics domain, we defined a control program using prioritized interrupts:

proc *mainControl*
$\langle$ *undeliveredPackages* $\rightarrow$ *deliverPackages* $\rangle \rangle\!\rangle$
$\langle \neg$ *finished* $\rightarrow$ `wait` $\rangle$

The program is to be understood as follows. In each cycle of the (implicit) main loop, if there are packages that have not been delivered yet, compute a plan to deliver them and execute it. If this is not the case but execution is not yet finished, do nothing for one cycle. Otherwise terminate.

Here, *finished* is a fluent that serves as a flag for signalling when program execution is supposed to halt. This is necessary to be able to perform finite experiments for a task that is indeed open-ended: While delivering the currently pending packages, new delivery requests keep arriving, each of which being modelled by an exogenous action `new_package(p,l,d)` that sets the current location of package `p` to `l` and its destination to `d`. Since the system does not know in advance when and how many new package arrivals will occur, a special exogenous action `no_more_packages` is used to set *finished* to `TRUE` after the last `new_package` event indicating that the experiment ends at this point.

Testing whether there are still packages to be delivered is done by procedure *undeliveredPackages*:

proc *undeliveredPackages*
$\exists p$: *package* $\exists l$: *location* $\exists d$: *location*.
$at(p, l) \wedge$ *destination*$(p, d) \wedge (l \neq d)$

The *deliverPackages* procedure is the part where planning comes into play:

proc *deliverPackages*
solve($\forall p$: *package* $\forall d$: *location*.
destination$(p, d) \supset at(p, d)$,
`[load,unload,drive]`)

Here, the first argument of **solve** is the goal formula and the second one the list of actions the planner has to consider. In our experiments, we tested two different versions of **solve**: one calling the external FF planner via `achieve_ff`, the other, `achieve`, being an internal, PROLOG-implemented construct of the IndiGolog framework that basically performs an iterative deepening search. The two planners were in each case given the same amount of information: a list of available actions, the fluent predicates involved (including their initial values) and objects' as well as fluent and action parameters' types. Whereas the internal `achieve` construct directly uses the appropriate part of the Golog domain axiomatization, FF is provided with the corresponding PDDL translation as described earlier.

Pack.	Trucks	Loc.	iN	eN	iR	eR
3	2	3	9	9	10	10
3	2	4	10	7	7	10
3	2	5	10	9	7	10
3	2	6	10	10	3	9
3	2	7	9	10	3	8
5	2	3	9	10	7	10
5	2	4	10	8	3	10
5	2	5	7	10	1	10
5	2	6	7	9	0	10
5	2	7	4	10	0	10
3	3	3	10	10	9	10
3	3	4	10	10	7	10
3	3	5	10	10	6	10
3	3	6	9	10	5	10
3	3	7	7	10	4	10
5	3	3	10	10	6	10
5	3	4	9	10	5	9
5	3	5	4	10	1	10
5	3	6	7	10	1	10
5	3	7	6	9	0	10

Table 1: Logistics: Number of instances solved

For both of the planners, we considered two variants. In the first one, once a plan is found it gets executed entirely. Packages arriving during that time are ignored until plan execution finishes and the next call to the planner is made. In the other variant, the system aborts the current plan and performs a re-planning after each `new_package` event.

We performed a series of experiments where the number of locations varied between 3 to 7, the number of trucks between 2 and 3 and the number of dynamically arriving packages among 3 and 5. For each combination, we created 10 different domain instances. The initial locations of trucks and packages as well as the destinations of the packages are chosen randomly. Two locations are connected with a probability of 50%; additional random edges ensure that the roadmap graph forms a single connected component. In each instance, there is one initial package, and the intervals between the arrival times of new packages vary between 2 and 8 steps, where one step corresponds to the execution of a primitive, non-exogenous action.

For each planner variant and domain instance we measured the overall runtime of the system and the number of steps (minus the number of `wait` actions) that were taken until termination. Runs that did not terminate within 300 seconds were aborted. The runtime includes a wait interval of 0.5 seconds after each executed action which was reserved to handle the communication with and the state update of the simulator.

Pack.	Trucks	Loc.	iN	eN	iR	eR
3	2	3	14.0	14.0	16.5	15.0
3	2	4	15.0	14.5	35.0	14.0
3	2	5	16.0	13.0	23.5	15.0
3	2	6	30.5	14.5	300.0	16.5
3	2	7	25.0	14.5	300.0	17.0
5	2	3	20.0	18.0	82.5	19.0
5	2	4	35.5	20.5	300.0	21.0
5	2	5	47.5	19.0	300.0	21.0
5	2	6	75.5	19.0	300.0	21.0
5	2	7	300.0	19.5	300.0	22.5
3	3	3	17.0	14.0	28.5	15.0
3	3	4	21.0	14.0	35.5	15.0
3	3	5	27.5	14.5	174.5	15.0
3	3	6	42.5	14.0	225.0	15.0
3	3	7	127.0	14.0	300.0	15.5
5	3	3	27.5	19.0	95.5	19.5
5	3	4	58.5	21.0	298.0	23.0
5	3	5	300.0	19.5	300.0	21.5
5	3	6	216.0	20.5	300.0	22.0
5	3	7	235.5	20.5	300.0	21.0

Table 2: Logistics: Median runtimes in seconds

Table 1 summarizes how many of the 10 instances were solved within the time limit by each method. Here, "e" refers to the variant where FF was used for planning while "i" means that the internal `achieve` was used. Further "R" means the variant where instant re-planning was done after the arrival of a new request and "N" the one where the current plan was not immediately aborted. The median runtimes for each combination are given in Table 2 and shown graphically in Figure 1, using a logarithmic scale. In those cases where the run did not finish within the timeout the value is set to the maximal time of 300 seconds. Table 3 contains the median number of steps that were taken, considering only instances that were solved by all methods.

The results clearly show that using FF instead of the internal planner has a large impact on the necessary computation time of the system, letting it solve instances within seconds which otherwise would require several minutes. One might object that our comparison is not equitable because we contrast FF which is a satisficing planner (i.e. which may return suboptimal plans in terms of plan length) with the internal planner that generates optimal plans (following an iterative deepening search strategy). We argue that in many cases optimal planning is not expedient: in fact, in the presence of unpredictable exogenous events or sensing it is not possible to plan really optimal. Furthermore, the results in Table 3 show that the number of steps required by the internal system is only slightly lower. Actually, our other benchmark domains show that using the external planning system also can result in a lower number of steps (Table 9). Nevertheless, there might be applications where optimal planning is more suitable, e.g. because there are no exogenous events and sensing is not required, or because the actual execution of an action takes a lot of time. Since we use PDDL to communicate with the external planner, our approach can easily

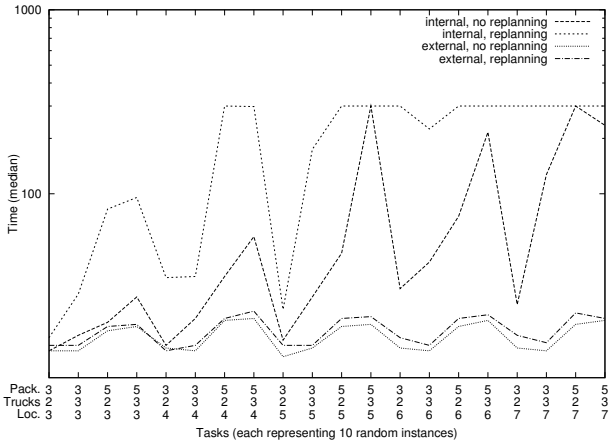

Figure 1: Logistics: Median runtimes in seconds

Pack.	Trucks	Loc.	Tasks	iN	eN	iR	eR
3	2	3	8	14.0	14.0	14.0	14.0
3	2	4	5	14.0	14.0	14.0	14.0
3	2	5	6	13.5	13.5	13.5	14.0
3	2	6	3	15.0	15.0	15.0	15.0
3	2	7	2	15.0	15.5	15.0	15.5
5	2	3	7	20.0	20.0	20.0	20.0
5	2	4	3	25.0	25.0	25.0	25.0
5	2	5	1	24.0	24.0	24.0	24.0
3	3	3	9	14.0	14.0	14.0	14.0
3	3	4	7	14.0	15.0	14.0	15.0
3	3	5	6	15.0	15.0	15.0	15.0
3	3	6	5	15.0	16.0	15.0	16.0
3	3	7	4	16.0	16.5	16.0	16.5
5	3	3	6	20.0	20.0	20.0	20.0
5	3	4	5	22.0	21.0	22.0	21.0
5	3	5	1	21.0	21.0	21.0	21.0
5	3	6	1	23.0	23.0	22.0	23.0

Table 3: Logistics: Median number of steps taken

be adapted to use any other planning system that handles PDDL, including optimal ones. As these planning systems are highly optimized, we would expect that they still would produce better results than the internal routine.

Elevator

The experimental setting for the elevator domain is analogous to the one of the logistics domain and uses the following main program.

proc *mainControl*
  $\langle$ *unservedPassengers* $\rightarrow$ *servePassengers* $\rangle\rangle$
  $\langle \neg$ *finished* $\rightarrow$ `wait` $\rangle$

Planning is used to serve the passengers that have not reached their destination yet:

proc *servePassengers*
  solve($\forall p$: *passenger*. *served*(p),
    `[move_fast,move,stop]`)

Pass..	Floors	Tasks	iN	eN	iR	eR
3	5	10	10	10	10	10
3	6	10	8	10	8	10
3	7	10	10	9	4	10
3	8	10	6	10	3	10
5	5	10	10	9	7	9
5	6	10	8	10	1	9
5	7	10	4	9	0	9
5	8	10	5	7	0	10
7	5	10	7	10	1	10
7	6	10	1	10	0	10
7	7	10	0	9	0	8
7	8	10	0	10	0	10
9	5	10	4	9	0	10
9	6	10	0	9	1	10
9	7	10	0	7	0	10
9	8	10	0	8	0	9

Table 4: Elevator: Number of instances solved

Pass..	Floors	iN	eN	iR	eR
3	5	17.0	14.0	23.5	14.5
3	6	22.0	12.0	35.5	13.5
3	7	39.5	15.0	300.0	15.0
3	8	87.0	17.0	300.0	18.0
5	5	23.0	19.0	81.5	24.0
5	6	90.0	20.5	300.0	27.0
5	7	300.0	24.0	300.0	28.0
5	8	276.5	20.5	300.0	27.5
7	5	79.5	28.0	300.0	26.5
7	6	300.0	29.5	300.0	28.0
7	7	300.0	33.0	300.0	31.5
7	8	300.0	31.5	300.0	31.5
9	5	300.0	33.0	300.0	31.5
9	6	300.0	33.5	300.0	37.0
9	7	300.0	33.5	300.0	43.0
9	8	300.0	36.5	300.0	44.5

Table 5: Elevator: Median runtimes in seconds

Again, we tested two versions of **solve**, one calling the internal planner, the other calling the FF system, each with and without re-planning on the arrival of a new passenger.

For the benchmark instances of this domain we let the number of new passengers vary among 3, 5, 7 and 9 and the number of floors between 5 and 8. As above, we created 10 different instances for each combination, choosing the passengers' origins and destinations randomly. Initially there is always one passenger request and the intervals between newly arriving passengers lie between 2 and 8 steps.

In analogy to the previous domain, table 4 summarizes how many of the 10 instances were solved within the time limit by each method. The median runtimes for each combination are stated in Table 5 and shown graphically in Figure 2. Table 6 contains the median number of taken steps, considering only instances that were solved by all methods.

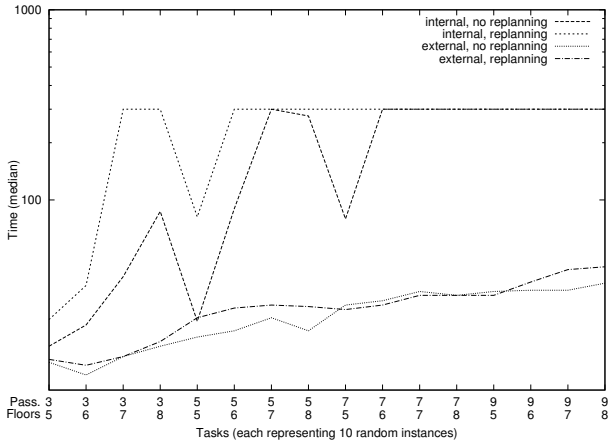

Figure 2: Elevator: Median runtimes in seconds

Pass..	Floors	Tasks	iN	eN	iR	eR
3	5	10	16.5	16.5	15.5	16.5
3	6	7	19.0	19.0	16.0	18.0
3	7	3	19.0	19.0	16.0	19.0
3	8	3	19.0	19.0	19.0	19.0
5	5	5	25.0	25.0	29.0	30.0
5	6	1	37.0	38.0	32.0	33.0
7	5	1	34.0	34.0	32.0	32.0

Table 6: Elevator: Median number of steps taken

Again, the variants with the external planner performed much better than the ones using the internal search method. The fifth row in Table 6 shows also an interesting detail: The re-planning strategy sometimes causes the overall number of steps to increase. This happens when a current plan is discarded to serve a new request, and when later another new request is made it turns out that following the original plan would have been less costly.

Mail Delivery

The mail delivery robot is controlled as follows:

proc *mainControl*
  while($\neg$*finished*) **do**
    (π_r m : *mailbox*) *getLettersFrom*(m);
    deliverLetters

Here, π_r denotes a variant of the non-deterministic choice of argument where the argument's instantiation is picked randomly. In case of the normal π construct, IndiGolog otherwise instantiates the variable always with the first applicable symbol, which would cause the program above to pick the same mailbox in each cycle of the loop. Once the next mailbox that should be visited is chosen, the path to it is determined by means of planning:

proc *getLettersFrom*(m)
  (π l : *location*)
    at(m, l)?;
    solve(*robotAt*(l), `[move]`);
    takeAllLetters(m)

Taking letters out of a mailbox requires sensing:

proc *takeAllLetters*(m)
  `look_into`(m);
  while$(\exists\, l : \textit{letter}.\ \textit{in}(l, m))$
    $\pi\ l : \textit{letter}$
      $\textit{in}(l, m)$?;
      `take_out`(l, m);
      `look_at`(l)
    `look_into`(m)

`look_into`(m) is a sensing action whose outcome is a constant l denoting one of the letters in the box (i.e. the robot can always only "see" the topmost one). Thus, the agent gets to know that fluent $\textit{in}(l, m)$ is currently true. In case the mailbox is empty, the return value is instead simply the special constant "`empty`". After picking up l, action `look_at`(l) is applicable and causes $\textit{addressee}(l, m')$ to become known to the agent for some mailbox m', which is the destination of letter l. For delivering the letters obtained like this, another call to the planner is made:

proc *deliverLetters*
  solve$(\forall\, l : \textit{letter}.$
      $(\exists\, m : \textit{mailbox}.\ \textit{addressee}(l, m))$
        $\supset \textit{delivered}(l),$
    `[put_in,move]`)

Again, we study the system's behavior for the case in which **solve** uses the internal planner and for the case where FF is called instead. Since there are no exogenous actions in this scenario, we do not consider dynamic re-planning.

In our benchmark scenarios the number of offices varies among 4, 8 and 16, the number of hallways among 2, 4 and 8, and the number of letters among 2, 4, 8 and 16. As in the other domains we created 10 instances for each combination, the offices being connected randomly to some hallway and hallways being connected to one another in a tree-like fashion. There are as many mailboxes as offices, but they are placed randomly. Therefore, it is possible that an office contains multiple mailboxes, only one, or even none at all. The origins and addressees of the letters are also chosen randomly.

Table 7 once again summarizes how many of the 10 instances were solved within the time limit by each method. The median runtimes for each combination are given in Table 8 and shown graphically in Figure 3. Table 9 contains the median number of steps that were taken, considering only instances that were solved by all methods.

The results for this domain are again quite conclusive. The controller using FF was able to solve more tasks and throughout required less computation time. In terms of steps, the two methods are comparable, but the results are somewhat erratic, which is mostly because of the randomized strategy that was used.

Lett.	Off.	Hall.	Tasks	i	e
10	4	2	10	6	10
10	4	4	10	6	10
10	4	8	10	4	10
10	8	2	10	4	10
10	8	4	10	5	10
10	8	8	10	5	10
10	16	2	10	5	10
10	16	4	10	6	10
10	16	8	10	7	10
15	4	2	10	1	10
15	4	4	10	1	10
15	4	8	10	0	10
15	8	2	10	1	10
15	8	4	10	0	10
15	8	8	10	2	10
15	16	2	10	2	10
15	16	4	10	2	10
15	16	8	10	0	10

Table 7: Mail Delivery: Number of instances solved

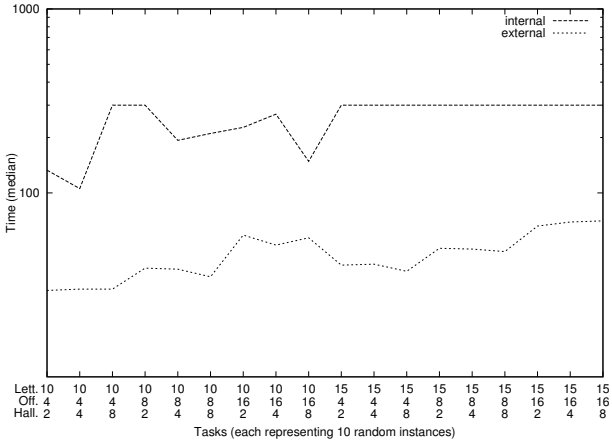

Figure 3: Mail Delivery: Median runtimes in seconds

Conclusion

We empirically evaluated a system that integrates the FF planning system into the IndiGolog agent framework. For that purpose, we developed three example application domains in which classical planning subproblems arise in the course of the execution of a high-level program. A series of experiments with different scenarios and problem sizes shows that the integration of the external planner decreases the required computation time a lot, keeping the number of executed actions similar. We ran further experiments in order to examine the effect of re-planning after exogenous events. This strategy does not pay off if planning is done by the internal mechanism because the additional computation time dominates the savings. Using the external planning system, the results are more balanced but still not clearly indicating that re-planning pays off. This may be due to the relatively simple strategy that we used. A possible direction for future work therefore is to study what effect it has on our system to use a more sophisticated method for execution monitoring and re-planning.

Lett.	Off.	Hall.	i	e
10	4	2	133.0	29.5
10	4	4	105.5	30.0
10	4	8	300.0	30.0
10	8	2	300.0	39.0
10	8	4	193.0	38.5
10	8	8	210.5	35.0
10	16	2	227.0	59.0
10	16	4	268.0	52.0
10	16	8	148.5	57.0
15	4	2	300.0	40.5
15	4	4	300.0	41.0
15	4	8	300.0	37.5
15	8	2	300.0	50.0
15	8	4	300.0	49.5
15	8	8	300.0	48.0
15	16	2	300.0	66.0
15	16	4	300.0	69.5
15	16	8	300.0	70.5

Table 8: Mail Delivery: Median runtimes in seconds

Lett.	Off.	Hall.	Tasks	i	e
10	4	2	6	50.5	51.0
10	4	4	6	49.5	51.0
10	4	8	4	54.0	50.0
10	8	2	4	68.5	63.0
10	8	4	5	61.0	67.0
10	8	8	5	71.0	59.0
10	16	2	5	92.0	93.0
10	16	4	6	92.0	93.5
10	16	8	7	91.0	95.0
15	4	2	1	69.0	72.0
15	4	4	1	69.0	70.0
15	8	2	1	101.0	92.0
15	8	8	2	76.0	95.0
15	16	2	2	134.5	114.5
15	16	4	2	112.5	105.5

Table 9: Mail Delivery: Median number of steps taken

Acknowledgments

This work was supported by the Deutsche Forschungsgemeinschaft under grants La 747/14-1 and Ne 623/10-1. We thank Sebastian Sardina and Stavros Vassos for their help with the IndiGolog framework.

References

Burgard, W.; Cremers, A. B.; Fox, D.; Hähnel, D.; Lakemeyer, G.; Schulz, D.; Steiner, W.; and Thrun, S. 1998. The interactive museum tour-guide robot. In *Proc. AAAI-98*, 11–18.

Claßen, J.; Eyerich, P.; Lakemeyer, G.; and Nebel, B. 2007. Towards an integration of Golog and planning. In *Proc. IJCAI 2007*, 1846–1851.

De Giacomo, G.; Lespérance, Y.; and Levesque, H. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artif. Intell.* 121(1–2):109–169.

De Giacomo, G.; Levesque, H. J.; and Sardina, S. 2001. Incremental execution of guarded theories. *Computational Logic* 2(4):495–525.

Edelkamp, S., and Hoffmann, J. 2004. PDDL2.2: The language for the classical part of the 4th international planning competition. report 195, Inst. f. Informatik, Univ. Freiburg.

Fikes, R., and Nilsson, N. J. 1971. STRIPS: a new approach to the application of theorem proving to problem solving. *Artif. Intell.* 2(3/4):189–208.

Fox, M., and Long, D. 2003. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *JAIR* 20:61–124.

Gerevini, A., and Long, D. 2005. Plan constraints and preferences in PDDL3. Technical report, University of Brescia.

Ghallab, M.; Howe, A.; Knoblock, C.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wilkins, D. 1998. PDDL—the planning domain definition language.

Helmert, M. 2008. *Understanding Planning Tasks – Domain Complexity and Heuristic Decomposition*, volume 4929 of *LNAI*. Springer-Verlag.

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

Levesque, H.; Reiter, R.; Lespérance, Y.; Lin, F.; and Scherl, R. B. 1997. GOLOG: A logic programming language for dynamic domains. *J. Log. Prog.* 31:59–84.

McCarthy, J., and Hayes, P. 1969. Some philosophical problems from the standpoint of artificial intelligence. In Meltzer, B., and Michie, D., eds., *Machine Intelligence 4*. New York: American Elsevier. 463–502.

Pednault, E. P. D. 1989. ADL: exploring the middle ground between STRIPS and the situation calculus. In *Proc. KR-89*, 324–332.

Reiter, R. 2001. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press.

Röger, G., and Nebel, B. 2007. Expressiveness of ADL and Golog: Functions make a difference. In *Proc. AAAI 2007*.

Röger, G.; Helmert, M.; and Nebel, B. 2008. On the relative expressiveness of ADL and Golog: The last piece in the puzzle. In *Proc. KR 2008*. to appear.

Sardina, S.; De Giacomo, G.; Lespérance, Y.; and Levesque, H. J. 2004. On the semantics of deliberation in Indigolog—from theory to implementation. *Annals of Mathematics and Artificial Intelligence* 41(2-4):259–299.

Tam, K.; Lloyd, J.; Lespérance, Y.; Levesque, H. J.; Lin, F.; Marcu, D.; Reiter, R.; and Jenkin, M. R. M. 1997. Controlling autonomous robots with GOLOG. In *Proc. IJCAI-97*, 1–12.