Flexible Plans for Adaptation by End-Users

Cyrille Martin, Humbert Fiorino and **Gaëlle Calvary**
Université Joseph Fourier, Grenoble INP, CNRS UMR 5217
Laboratory of Informatics of Grenoble
BP 53, 38041 Grenoble cedex 9, France

Abstract

Ubiquitous computing promotes flexibility for the end-user. This means that some design choices have to be shifted from design-time to run-time in order to involve the end-user into the decision process. In this paper, we study flexible plans, i.e. plans that let the end-user arrange tasks planned by an automated process seeking to achieve his needs. More precisely, we present an algorithm λ-graphplan that lets the end-user to decide the order of specific treatments (loop body) execution to a set of objects (loop variants).

λ-graphplan is based on graph planning structure. Its strength is that it does not require any problem-dependent knowledge to compute flexible plans. By relaxing mutex constraints in the planning graph, λ-graphplan discovers the loop variants and builds the macro-actions that constitute the loop bodies. We show that λ-graphplan is performant with "iterative" as well as with "linear" domains.

1 Introduction

Today's ubiquitous information technologies and computer networks supply numerous resources, namely services and data, for connected people. This wide emergent digital world is heterogeneous, highly dynamic and unpredictable: computation, communication as well as interaction resources may arrive and disappear dynamically. Thus people live in highly variable interactive spaces which in addition may trigger opportunistic user needs. The challenge we address is how to develop interactive systems capable of adaptation to dynamic user needs (Aarts and de Ruyter 2009) and resources (Weiser 1999). The problem is all the more complex because the user needs and resources cannot be comprehensively envisioned at design time (Myers, Ko, and Burnett 2006). Particularly, it is no more possible to suppose that one adequate service will be available at run-time to fulfill the user needs. More likely, the users will achieve their goals by interacting with several services.

Since most users cannot manage the composition of services themselves (Guzdial, Reppy, and Smith 1992) due to their lack of knowledge and skills, the composition has to be automated. One possibility to compose services while taking into account varying contexts of use (Rao and Su 2004) is based on automated planning. For instance, in the travel management case study, the user goal is to prepare a journey through different cities such as `Paris`, `Rome` and `Berlin`. For that purpose, the user has access to services allowing him to reserve accomodations, find restaurants, get city information, etc. Suppose that, for each city, the user wants to find a two-star *hotel* and a vegetarian *restaurant*. This requires that the user visits each service one by one and repetitively specifies his destinations and preferences. An automated composition of services handles this in a more efficient way by providing a solution plan that represents an ordered sequence of actions corresponding to hotel and restaurant reservations fitting his preferences. It is worth noting that the sequence of actions (choosing a hotel and then a restaurant) is a macro-action applied on each targeted city. However, the ordering of the cities to be visited is a decision that should be left to the user at run-time. Indeed, the user may have hidden context-dependent constraints (for instance the need to bring presents from `Paris` to parents in `Berlin` before meeting friends in `Rome`) that appear at run-time.

A *flexible plan* lets the user draw up his own arrangement at run-time over some objects subject to macro-actions. Such flexibility is key in user-centred approaches and is now crucial for ubiquitous computing. To our knowledge, automated planning either does not provide flexible plans or needs problem-dependent knowledge about the objects possibly subject to the user arrangement. In this paper, we propose a planning algorithm that computes flexible plans from STRIPS–like planning domains without problem-dependent knowledge. Gripper (Bonet, Palacios, and Geffner 2009) is used to illustrate our concepts throughout this paper: *a robot can `PICK` a ball, `DROP` it and `MOVE` from room to room. In the initial state, the `Left` room contains n balls, the `Right` room is empty, and the robot stands in this empty room.* The user needs *to move the n balls from the `Left` to the `Right` room.* The following plan (cf. plan 1) is said to be flexible as it achieves the user goal while letting him decide about the balls ordering. The *while* loop body is a *macro-action*. The balls are the *loop variant*. Thus searching for a flexible plan is equivalent to finding

Plan 1: Flexible plan for Gripper

```
1 $\mathcal{B} \leftarrow \{\texttt{Ball}_1, \ldots, \texttt{Ball}_n\}$;
2 while $\mathcal{B}$ is not empty do
3     User selects ball $\in \mathcal{B}$;
4     $\mathcal{B} \leftarrow \mathcal{B} - ball$;
5     MOVE(Right, Left);
6     PICK(ball, Left);
7     MOVE(Left, Right);
8     DROP(ball, Right);
```

a common representation for a set of solution plans (the instances of the flexible plan). To be noted, the Gripper variant that is used here only has one gripper on the robot. The other variants will be discussed later.

We propose a brief overview of the related work in section 2. The planning graph our planner is based on is presented in section 3. Our proposal is detailed in section 4 and evaluated in section 5.

2 Related Work

To our knowledge, there is no approach in planning devoted to flexibility. However generalized planning provides while-loop based plans and thus deserves attention. Generalized planning aims to plan solutions for a set of problems belonging to the same abstract class. A class of problems contains the description of *variables* which values correspond to different instances of problems.

In (Hu and Levesque 2009), the domain contains a *planning parameter* to represent the variable number of planning objects. This variable is used as a loop variant in the generated plan. For instance, in Gripper, the number of balls would be represented by this parameter. Thus, the user goal could only be to move all the balls. As a result, the user cannot aim at moving only a subset of the balls. To move this subset, another planning domain which parameter characterizes the subset of the balls has to be defined. This means that the class of problems to be solved has to be known at design time. Our proposition, λ-graphplan, solves all the problems for a given domain, not only a class of problems.

In (Srivastava, Immerman, and Zilberstein 2008), *abstraction predicates* are used to define *roles*. These roles make it possible to transform an example plan into an abstract state space (using 3-valued logic). From the transformation of the example plan, the algorithm finds loops on the objects that play the defined roles. Then, the plan can be applied to any problem that is an instance of the corresponding class. Furthermore, a set of *integrity constraints* ensures the validity of abstract states. In Gripper, if the user goal concerns only a subset of the balls that play the same role, then the balls cannot be distinguished. This limits the class of problems that can be addressed. In addition, the possible roles are elicited at design time by setting the set of abstraction predicates.

In (Winner and Veloso 2007) the proposition is slightly different of the generalized planning. Indeed, the aim is to provide domain-specific planning programs (named dsPlanner). A given example plan is observed to find repeated patterns. These patterns are transformed into while-loop structures. This requires to compute a first solution before looking for the generalized plan. λ-graphplan directly computes a flexible plan, bypassing the step of a concrete solution plan.

3 λ-graphplan Overview

λ-graphplan is based on the planning graph approach (Blum and Furst 1997). A planning graph $\mathcal{G}$ for a given planning domain and a given problem is a collapsed representation of the state space. It is composed of actions layers $\mathcal{A}_i$ and propositions layers $\mathcal{P}_i$. $\mathcal{A}_i$ (resp. $\mathcal{P}_i$) contains the actions (resp. propositions) that can be executed (resp. reached) at time step i.

Preconditions of action $a \in \mathcal{A}_i$ are propositions of $\mathcal{P}_{i-1}$ linked to a. Likewise, propositions in $\mathcal{P}_i$ are either positive or negative effects of action $a \in \mathcal{A}_i$. Furthermore, each proposition $p \in \mathcal{P}_{i-1}$ is the precondition of a dummy action *noop-p* in $\mathcal{A}_i$ which has one positive effect $p \in \mathcal{P}_i$ and no negative effect.

Two actions a_1 and a_2 in $\mathcal{A}_i$ are linked by a *mutex* relation if either a negative effect of a_1 is a precondition or a positive effect of a_2, or a precondition of a_1 is mutex with a precondition of a_2 in $\mathcal{P}_{i-1}$. Two propositions p_1 and p_2 in $\mathcal{P}_i$ are *mutex* if every action in $\mathcal{A}_i$ that has p_1 as positive effect is *mutex* with every action in $\mathcal{A}_i$ that has p_2 as positive effect, and if no action in $\mathcal{A}_i$ has both p_1 and p_2 as positive effect. $\mu\mathcal{A}_i$ and $\mu\mathcal{P}_i$ respectively denote the sets of mutex relations between actions in $\mathcal{A}_i$ and propositions in $\mathcal{P}_i$.

$\mathcal{P}_0$ represents the initial state. The planning graph is *expanded* step by step according to the following rules: an action is added into $\mathcal{A}_i$ if all its preconditions are non mutex in $\mathcal{P}_{i-1}$. A proposition p is added into $\mathcal{P}_i$ if an action in $\mathcal{A}_i$ has p as positive or negative effect. The planning graph is expanded until the current layer $\mathcal{P}_g$ contains a consistent goal state, i.e. all propositions of the goal are non mutex. When $\mathcal{P}_g$ is reached, Graphplan (Blum and Furst 1997) tries to *extract* a plan from $\mathcal{G}$: each selected proposition of layer $\mathcal{P}_i$ has to be supported by at least one action of layer $\mathcal{A}_i$. The actions that support the selected propositions must form a set that does not contain any *mutex* relation. All preconditions of this set of actions in $\mathcal{A}_i$ are then selected in $\mathcal{P}_{i-1}$ and so on. This selection process starts with the goal propositions until $\mathcal{P}_0$ is reached. If no plan is found, the planning graph is extended with an additional layer until a plan is eventually extracted or the termination condition is met.

Like Graphplan, λ-graphplan consists of an expansion and an extraction step (Figure 1). It is based on the same termination condition and mutex definition. However, in addition to these principles, we introduce the concept of λ-mutex for labelling the actions possibly subject to loop-structures. A λ-solution plan is a layered plan such that actions within each layer are either no mutex or λ-mutex: the λ-mutex relation is a relaxation of the mutex relation. When a λ-solution

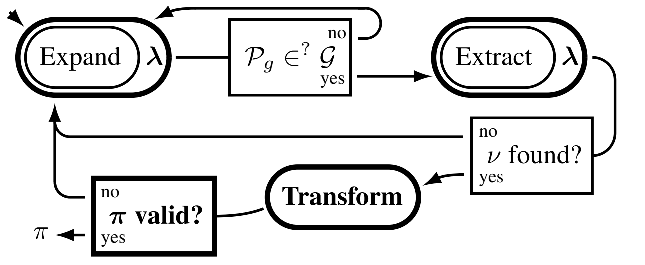

Figure 1: Illustration of λ-graphplan algorithm: extensions to Graphplan are in **bold**.

plan ν is found, λ-graphplan subsequently applies transformations on ν (because it is not consistent) to find loop-variants and include loop-structures. If the transformations succeed, the transformed plan π is returned. Otherwise, the expansion process is resumed as solution plans without λ-mutex relaxations are still possible. Indeed, by construction plans with loop-structures are shorter than plans without relaxation.

4 λ-graphplan Algorithm

In the following sections, an *operator* o is a 3-tuple $\langle pre(o), \mathit{eff}^+(o), \mathit{eff}^-(o)\rangle$ where $pre(o)$, $\mathit{eff}^+(o)$ and $\mathit{eff}^-(o)$ are sets of predicates that share a set of variables $param(o)$. They respectively represent the preconditions, the positive and negative effects of the operator. An *action* a is the application of a substitution σ on an operator o. Each predicate of $pre(o)$, $\mathit{eff}^+(o)$, $\mathit{eff}^-(o)$ becomes a proposition. That means that for each variable v in $param(o)$, a substitution $(v\backslash t)$ exists in σ and $a = \sigma(o)$. For instance, `MOVE (Left, Right)` is an action derived from the operator `MOVE`. A *partially instantiated action* results from $\sigma(o)$ if it exists v in $param(o)$ such that $(v\backslash t)$ does not exist in σ. For example, `PICK (`*ball*`, Left)` is a partially instantiated action.

4.1 Planning Graph Expansion

λ-graphplan relaxes some of the mutex constraints during the extraction process: as in Graphplan, actions in the layered plan have to be non mutex except for those that will be collapsed into macro-actions. These actions are said to be λ-mutex and are detected in the expansion process. They are mutex actions deriving from the same operator. More formally,

Definition 1 (λ-Mutex – Actions) *Two actions a_1 and a_2 in level $\mathcal{A}_i$ are* λ-mutex *if:*

- $(a_1, a_2) \in \mu\mathcal{A}_i$;
- $a_1 = \sigma_1(o)$ *and* $a_2 = \sigma_2(o)$*, where o is an operator,* σ_1 *and* σ_2 *are two substitutions.*

$\lambda\mathcal{A}_i$ is the set of λ-mutex actions: $\forall i, \lambda\mathcal{A}_i \subseteq \mu\mathcal{A}_i$. We note $\sigma_a = \sigma_1 \cap \sigma_2$, and $a_a = \sigma_a(o)$ is a partially instantiated action which can represent the two actions a_1 and a_2. Thus, a_a will be key to constructing macro-actions and then loop structures.

λ-mutex propositions are reachable propositions at time step i only if they derive from actions that can be collapsed into a loop structure. λ-mutex propositions are defined as follows:

Definition 2 (λ-Mutex – Propositions) *Two propositions p_1 and p_2 in $\mathcal{P}_i$ are* λ-mutex *if:*

- $(p_1, p_2) \in \mu\mathcal{P}_i$;
- *every action in $\mathcal{A}_i$ that has p_1 as positive effect is* λ-mutex *with every action in $\mathcal{A}_i$ that has p_2 as positive effect.*

$\lambda\mathcal{P}_i$ is the set of λ-mutex propositions in $\mathcal{P}_i$ and $\forall i, \lambda\mathcal{P}_i \subseteq \mu\mathcal{P}_i$.

At each time step i in the planning graph $\mathcal{G}$, the available information is $\mathcal{A}_i$, $\mu\mathcal{A}_i$, $\lambda\mathcal{A}_i$, $\mathcal{P}_i$, $\mu\mathcal{P}_i$ and $\lambda\mathcal{P}_i$.

4.2 Plan Extraction

The first step toward a flexible plan is to relax the mutex constraints on propositions and actions which are λ-mutex. For any layer i, $\mu\mathcal{A}_i \setminus \lambda\mathcal{A}_i$ is the set of action couples that cannot be jointly selected. Likewise, $\mu\mathcal{P}_i \setminus \lambda\mathcal{P}_i$ contains the couples of propositions that cannot be reached together in layer i. In particular, the goal is considered as reachable in layer $\mathcal{P}_g$ even if it contains λ-mutex propositions. In the Gripper domain, the goal is formed by the propositions `at Left Ball`$_i$ for each ball i. Graphplan expands the planning graph until these goal propositions without mutex relations in the propositions are found, i.e. until layer $\mathcal{P}_7$ is reached (whatever the number of balls). λ-graphplan stops the planning graph expansion at layer $\mathcal{P}_4$ because all goal propositions are strictly λ-mutex. Thus, λ-graphplan generally tries to extract a plan before Graphplan. In Gripper, the plan extracted by λ-graphplan is the following (cf. plan 2). This plan is not executable since the robot cannot pick more than one ball at the same time. The next step is to transform the plan in order to insert the loop structures.

Plan 2: Non consistent plan for Gripper, extracted by λ-graphplan

```
1 MOVE(Right, Left);
2 PICK(Ball$_1$, Left), ..., PICK(Ball$_n$, Left);
3 MOVE(Left, Right);
4 DROP(Ball$_1$, Right), ..., DROP(Ball$_n$, Right);
```

4.3 Transformation

The transformation of the plan is threefold. First of all, the possible loop variants are searched in the extracted plan. Some actions are “collapsed” into a partially instantiated action whose variables are loop variants. Then, the “internal closure” builds the loop body by aggregating collapsed actions into a macro-action. Finally, the “external closure” completes the macro-action so that its effects and preconditions are consistent with iterations.

Collapsed actions and loop variants λ-mutex actions are candidates for loops. But the first step is to search for a representation of these actions that reveals planning objects subject to loops, that is *loop variants*. For instance, *ball* $\in$ $\{$`Ball`$_1, \ldots,$ `Ball`$_n\}$ is operated by all the `Pick(Ball`$_1$, Left), Pick(Ball$_2$, Left) etc. actions. These actions are collapsed into Pick(*ball*, Left). More formally, a *collapsed-action* α (for the plan step i) is a partially instantiated action defined as follows:

- Let $\mathcal{S}_i = \{a_1, \ldots, a_n\}$ be a set of actions such that $\forall a_a, a_b \in \mathcal{S}_i, (a_a, a_b) \in \lambda\mathcal{A}_i$, i.e. $\mathcal{S}_i$ under λ-mutex relation is a clique: this ensures that the actions are derived from the same operator;
- $\forall a \in \mathcal{S}_i, \exists\sigma \mid a = \sigma(\alpha)$, i.e. α is a partially instantiated action that collapses all actions in $\mathcal{S}_i$. $var(\alpha)$, the set of α's variables are the loop variants.

This collapsed-action consists of:

- *spre*(α), the set of shared propositions in $\mathcal{S}_i$ action preconditions: *spre*$(\alpha) = \bigcap\{$*pre*$(a) \mid a \in \mathcal{S}_i\}$. Likewise, *seff*$^+(\alpha)$ and *seff*$^-(\alpha)$ are the sets of shared propositions in respectively $\mathcal{S}_i$ action positive and negative effects. The shared propositions represent the preconditions and effects that do not affect the variants: they will be key to forming the invariants of the loop;
- *upre*(α), the set of unshared propositions in $\mathcal{S}_i$ action preconditions: *upre*$(\alpha) = \bigcup\{$*pre*$(a) \mid a \in \mathcal{S}_i\} \setminus$ *spre*(α). Likewise, *ueff*$^+(\alpha)$ and *ueff*$^-(\alpha)$ are the sets of unshared propositions in respectively $\mathcal{S}_i$ action positive and negative effects. The unshared propositions represent the preconditions and effects on the variants.

Furthermore, as non collapsed-actions do not affect variants, their preconditions and effects are "shared". In the Gripper example, the transformed plan is as follows (cf. plan 3).

Plan 3: Non consistent plan for Gripper, transformed by λ-graphplan

```
1 MOVE(Right, Left);
2 PICK(ball ∈ {Ball_1, ..., Ball_n }, Left);
3 MOVE(Left, Right);
4 DROP(ball ∈ {Ball_1, ..., Ball_n }, Right);
```

Having found an appropriate collapsed-action for all actions under λ-mutex relations, the next transformation aims at finding collapsed-actions which should be in the same loop body.

Internal closing For that purpose, λ-graphplan builds a *parameterized macro-action*. We use a notion of macro-action very similar to that of macro-operator (Botea et al. 2005; Coles, Fox, and Smith 2007; Newton et al. 2007). A macro-action $\mathcal{M}$ is a sub-plan: $\mathcal{M} = \langle\pi_1, \ldots, \pi_n\rangle$ where π_i are sets of actions and collapsed-actions. The construction of a macro-action begins with an empty macro-action $\mathcal{M} = \langle\rangle$ and each set of shared and unshared propositions is empty. When a set of collapsed-actions or actions $\pi_i = \{a_1, \ldots, a_n\}$ is added to $\mathcal{M}$, the sets *spre*$(\mathcal{M})$, *upre*$(\mathcal{M})$ etc. are updated accordingly. A macro-action is an abstraction considered as a *unique action*. Therefore, the update rules are the following:

- The shared preconditions of $\mathcal{M}$ are updated with the shared preconditions of each action in π_i that are not also shared positive effects of $\mathcal{M}$. Likewise, the unshared preconditions of $\mathcal{M}$ are updated with the unshared preconditions of each action in π_i that are not also unshared positive effects of $\mathcal{M}$;
- The shared positive effects of $\mathcal{M}$ are updated with the shared positive effects of each action a in π_i and then the shared negative effects of each action a are withdrawn; the unshared positive effects are updated in the same way;
- The shared negative effects of $\mathcal{M}$ are updated with the shared negative effects of each action a in π_i and then the shared positive effects of each action a are withdrawn; the unshared negative effects are updated in the same way.

The actions to be added to the macro-actions are selected as follows:

- All collapsed-actions must be included in macro-actions that manipulate the same variants and that are linked by produced and consumed propositions (the positive and negative effects): $\mathcal{C}_k$ is a collapsed-action and $\mathcal{M}_i$ a macro-action such that *upre*$(\mathcal{C}_k) \cap$ *ueff*$^+(\mathcal{M}_i) \neq \emptyset$ and $\mathcal{C}_k$ occurs after $\mathcal{M}_i$. Then, $\mathcal{C}_k$ is included into $\mathcal{M}_i$. For example, at step 2 the collapsed-action $\mathcal{C}_{PICK}$ = PICK (*ball* $\in$ {Ball$_1$, ..., Ball$_n$}, Left) has the proposition carry Ball$_i$ as positive effect for each ball i. Thus, these propositions belong to the set *ueff*$^+(\mathcal{C}_{PICK})$, but also to the set *upre* of the collapsed-action $\mathcal{C}_{DROP}$ derived from DROP actions at step 4. Both collapsed-actions are included into the same macro-action;
- Actions are added into macro-actions in order to perform *internal closures*, i.e. to support the shared preconditions of the collapsed-actions. For instance, to trigger $\mathcal{C}_{PICK}$, the robot has to be in the left room, i.e. the proposition at-g Left belongs to *spre*$(\mathcal{C}_{PICK})$. This precondition is guaranteed to be supported, since $\mathcal{C}_{PICK}$ is the first collapsed-action of the macro-action and previous actions had this proposition as positive effect. Suppose s is the world state just before the macro-action is launched. One of the preconditions of $\mathcal{C}_{DROP}$ is the proposition at-g Right. Because this proposition is not supported by the state resulting of the application of $\mathcal{C}_{PICK}$ on s, an action has to be found between $\mathcal{C}_{PICK}$ and $\mathcal{C}_{DROP}$: it has to have at-g Right as positive effect. This action is guaranteed to exist since $\mathcal{C}_{DROP}$ has been extracted (all the actions collapsed into $\mathcal{C}_{DROP}$ are achievable): it is MOVE(Left, Right). In the general case, the internal closure is done by a set of actions between two collapsed-actions that are included into the same macro-action. Finally, the macro-action is formed by $\mathcal{C}_{PICK}$, MOVE(Left, Right) and then $\mathcal{C}_{DROP}$.

Once the robot is in the left room, this following macro-action allows to pick up one ball and to drop it in the other room.

External closing Macro-actions are not sufficient to create loops. It is also mandatory to ensure a seamless running of successive iterations, i.e. the states of the world at the beginning and at the end of a macro-action have to share propositions that represent the loop *invariant*: these

```
    parameter: $ball \in \{$Ball$_1$,..., Ball$_n$ $\}$
1   PICK(ball, Left);
2   MOVE(Left, Right);
3   DROP(ball, Right);
```

elements are the propositions that form the shared preconditions of the macro-action under consideration. They ensure the proper execution of each action in the macro-action. Thus, if the world state at the end of a macro-action supports the shared preconditions of the macro-action, the macro-action can be successfully executed.

In our example, the macro-action $\mathcal{M}$ can move a ball from the left to the right room. The shared preconditions *spre*$(\mathcal{M})$ of the macro-action contain the proposition `at-g Left`, which means that the robot must be in the left room to pick a ball. An effect of $\mathcal{M}$ is that the robot stands in the right room i.e. `at-g Left` belongs to the shared negative effects of $\mathcal{M}$ and `at-g Right` to the shared positive effects of $\mathcal{M}$. Consequently, the world state at the beginning of $\mathcal{M}$ is different from the state s reached through the application of the shared effects (*seff*$^-$ and *seff*$^+$) of $\mathcal{M}$ on that state. Therefore, a plan to support *spre*$(\mathcal{M})$ from the state s must be found: this is a new planning problem where *spre*$(\mathcal{M})$ is the planning goal and s the initial state. If solved by λ-graphplan, the solution plan is added to the macro-action. In Gripper, this plan is composed of one action, `MOVE(Right, Left)`, which is added to $\mathcal{M}$. If λ-graphplan succeeds in closing all the macro-actions, it turns them into loop structures and allows the end-user to non deterministically select the arrangement of the objects addressed by the loop variant. While processing the last object in the loop structures, the actions added to ensure the closure of the macro-action should not be executed. Indeed, the initial plan was extracted by considering a single object. Therefore the expected state at the output of the loop is the result of the application of the macro-action without the actions for loop closure. In Gripper, the returned plan is as follows (cf. plan 4). As a reminder, the steps 6 to 10 form the macro-action, the step 7 provides the internal closure of the macro-action and the step 10 its external closure.

Plan 4: Flexible plan for Gripper, returned by λ-graphplan

```
1   MOVE(Right, Left);
2   $\mathcal{B} \leftarrow \{$Ball$_1$,..., Ball$_n$ $\}$;
3   while true do
4       User selects ball in $\mathcal{B}$;
5       $\mathcal{B} \leftarrow \mathcal{B} - ball$;
6       PICK(ball, Left);
7       MOVE(Left, Right);
8       DROP(ball, Right);
9       if $\mathcal{B}$ is empty then break;
10      MOVE(Right, Left);
```

Blocks World			
n	Graphplan perf avg (ms)	λ-graphplan perf avg (ms)	extra cost (%)
2	24	25	4
4	34	39	10
6	94	134	30
8	160	263	39
10	189	295	36
12	292	396	26
14	357	514	31
16	468	644	27
18	574	828	31
20	823	1245	34

Table 1: Graphplan and λ-graphplan average performances in *ms* for 100 runs per problem for the Blocks World domain with different numbers n of blocks. The performance of λ-graphplan with respect to Graphplan is given in percent.

5 Evaluation of λ-graphplan

The computation of the λ-mutex relations is not more expensive than the computation of the mutex relations. Thus, as in Graphplan, the computation of the planning graph has a polynomial cost (Blum and Furst 1997).

We evaluate the performances[1] of λ-graphplan as an extension of Graphplan, i.e. by measuring the cost of the computation of flexibility in the plans.
First of all, we investigate the performances of λ-graphplan in the most unfavorable situation. This happens with "linear" planning domains, i.e. domains with a single sequential solution. This is for instance the case with a Blocks World in which a vertical stack of n blocks has to be unstacked. Necessarily, the solution plan unstacks the top block, drops it on the table and so on (therefore, there is no open option for the end-user). Table 1 compares the results obtained with Graphplan and λ-graphplan: λ-graphplan is between 30 and 40 percent slower than Graphplan.
The Gripper domain is favorable to λ-graphplan (cf. table 2): the performances of λ-graphplan are much better than those of Graphplan. For 3 balls, λ-graphplan already costs 51% less than Graphplan. This is due to the limited graph expansion of λ-graphplan and to the shorter plans it finds. λ-graphplan computes maximal cliques on λ-mutex actions to create collapsed-actions. This is a NP-complete problem. Although this computation can be done only once per planning graph layer, it is an expensive part of the extraction process which depends on the number of λ-mutex actions. In our tests with Gripper, the relative cost for computing cliques becomes greater than 50% beyond 8 balls.
Without actually comparing these results with those generated by work in generalized planning, we can affirm that, in the case of flexible plans, our results will be faster since the others take an example plan as input and λ-graphplan

[1]All run tests are executed on a 2.6 GHz Intel Core 2 Duo processor, with 4 Go of 667 MHz DDR2.

Gripper			
n	Graphplan perf. avg (ms)	λ-graphplan perf. avg (ms)	max-clique cost (%)
1	27	30	0.0
2	37	40	2.7
3	68	45	3.1
4	328	57	3.9
5	471	58	6.6
6	947	72	9.0
7	3507	100	27.7
8	16241	203	59.5
9	–	406	79.4
10	–	687	86.6

Table 2: Graphplan and λ-graphplan average performances in *ms* for 100 runs per problem for the Gripper domain with different numbers n of balls. The max-clique cost is the time spent to compute cliques out of the solution total computation time.

is comparable or faster than one of the fastest planner, Graphplan. The example plan has to be calculated (without flexibility), when our approach is integrated to the planning process.

The Gripper and Blocks World variants used above are sufficient to illustrate the performance of λ-graphplan, but not enough to assess its capability in terms of flexibility. To evaluate this capability, other variants of Gripper should be defined, e.g.:

- the robot has several grippers (in other words, the set of objects – the balls – can be treated in different ways);
- more than one room contains balls, or the balls have to be moved in different rooms (the loop variant becomes a tuple $\langle room, ball\rangle$);
- a specific treatment (e.g. paint each ball with different colors) has to be applied to each object (e.g. full flexibility claims for letting the user choose the colors ordering).

All these Gripper variants could be used as starting points for building a full-fledged framework devoted to flexibility. In its current version, λ-graphplan produces loop structures containing one loop variant. λ-graphplan is a first step toward flexibility that needs to be further generalized.

6 Conclusion

This paper deals with flexibility for the end-user, a key property in user centered engineering in general and now crucial for ubiquitous computing. More precisely, the paper focuses on flexible arrangement of repetitive tasks by the end-user. The approach under study is automated planning.

We define the λ-mutex relation for the planning graph approach so as to identify actions and objects candidate for loop structures at planning time. We propose the λ-graphplan algorithm to compute loop bodies and variants. We show that in addition to providing loop structures, λ-graphplan can be more efficient than Graphplan. In future work, we aim at extending λ-graphplan toward nested loops and loops with multiple variables or n-tuple variables.

Acknowledgments

Thanks to Damien Pellier for the pddl4j package.

References

Aarts, E., and de Ruyter, B. 2009. New research perspectives on ambient intelligence. *JAISE* 1:5–14.

Blum, A., and Furst, M. 1997. Fast planning through planning graph analysis. *Artificial Intelligence* 90(1):281–300.

Bonet, B.; Palacios, H.; and Geffner, H. 2009. Automatic derivation of memoryless policies and finite-state controllers using classical planners. In *ICAPS*.

Botea, B.; Enzenberger, M.; Müller, M.; and Schaeffer, J. 2005. Macro-FF: Improving AI planning with automatically learned macro-operators. *JAIR* 24:581–621.

Coles, A.; Fox, M.; and Smith, A. 2007. Online identification of useful macro-actions for planning. In *ICAPS*.

Guzdial, M.; Reppy, J.; and Smith, R. 1992. *Report of the 'User/Programmer Distinction' working group*. A. K. Peters, Ltd. 367–383.

Hu, Y., and Levesque, H. 2009. Planning with loops: Some new results. In *ICAPS Workshop on Generalized Planning*.

Myers, B.; Ko, A.; and Burnett, M. 2006. Invited research overview: End-user programming. In *CHI Extended Abstracts on Human Factors in Computing Systems*.

Newton, H.; Levine, J.; Fox, M.; and Long, D. 2007. Learning macro-actions for arbitrary planner and domains. In *ICAPS*.

Rao, J., and Su, X. 2004. A survey of automated web service composition methods. In *SWSWPC*.

Srivastava, S.; Immerman, N.; and Zilberstein, S. 2008. Learning generalized plans using abstract counting. In *AAAI Conference on Artificial intelligence*.

Weiser, M. 1999. The computer for the 21st century. *SIGMOBILE MC2R* 3:3–11.

Winner, E., and Veloso, M. M. 2007. Loopdistill: Learning domain-specific planners from example plans. In *Workshop on AI Planning and Learning, ICAPS*.